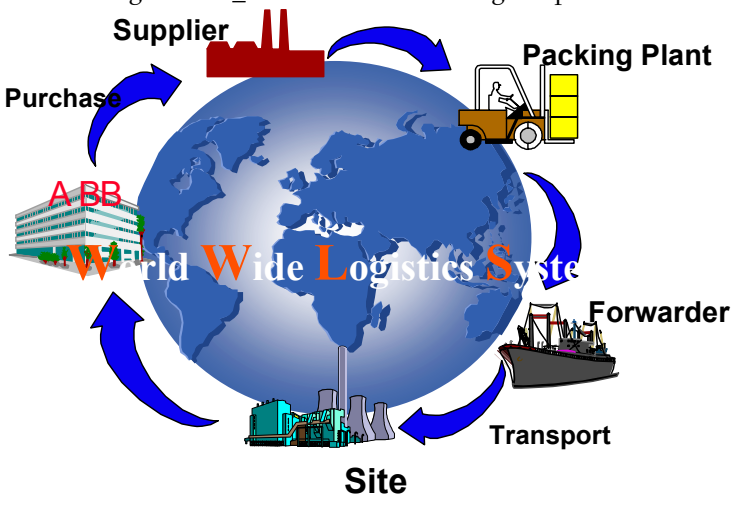
1 Projektberichte

1.1 WWLS – Das World Wide Logistics System der ABB Kraftwerke AG

ABB ist ein weltweit operierender Konzern im Bereich Elektrotechnik und zählt zu den Markführern in dieser Branche. Der ABB Konzern ist stark dezentral organisiert, etwa 1400 eigenständige ABB Gesellschaften sind in mehr als 130 Ländern angesiedelt und beschäftigen zur Zeit zusammen ca. 220.000 Mitarbeiter. Die Zentrale des Konzerns ist in Zürich. Die wesentlichen Unternehmensfelder des ABB Konzerns sind Stromerzeugung, Stromübertragung und –verteilung sowie Industrieautomation. Als innovatives Unternehmen unterhält der ABB Konzern neun weltweit verteilte Forschungszentren. Das Ziel dieser Forschungseinrichtungen ist, neueste Technologien zu entwickeln, zu erproben und in die einzelnen ABB Gesellschaften zu transferieren. Zu diesem Zweck existieren dedizierte Forschungsprogramme, die gezielt für die ABB relevante Technologiefelder bearbeiten. Eines dieser Forschungsprogramme widmet sich dem Thema „Software-Engineering“.

Software ist in den letzten Jahren verstärkt zu einem marktrelvanten Faktor für viele ABB Produkte geworden. In diesem Sinne ist bei vielen ABB Gesellschaften die Software-Entwicklung eine wesentliche Tätigkeit, wenn ein neues Produkt entwickelt werden soll . Aber auch im Umfeld der Produktentwicklung werden Softwaresysteme immer wichtiger, um Kosten- und Kundenanforderungen erfüllen zu können. Beispiele dafür sind Software-Systeme im Bereich Simulation und Service von Anlagen oder im Bereich Logistik.

Zu den traditionellen Geschäftsfeldern der ABB zählt der Bau von Kraftwerksanlagen. Das in diesem Beitrag vorgestellte Java-Projekt und die dabei entwickelte Software ist Teil des Logistiksystems der ABB Kraftwerke AG. Kraftwerke sind sehr große und komplexe Anlagen, an deren Planung und Bau viele verschiedene Unternehmen beteiligt sind, die in der Regel weltweit verteilt sind. Damit der Bau eines Kraftwerks möglichst reibungslos durchgeführt werden kann, bedarf es einer präzisen Logistik. Diese muß sicherstellen, daß alle zum Bau eines Kraftwerks benötigten Teile in sinnvoller Reihenfolge und rechtzeitig vor Ort auf der Baustelle eintreffen. Am gesamten Logistikprozeß sind viele unterschiedliche Rollen beteiligt; dazu zählen die Hersteller der einzelnen Teile, die Verpacker der Teile, die Transporteure und letztlich auch die Kraftwerksbaustelle. Abbildung WWLS_1 verdeutlicht den Logistikprozeß beim Kraftwerksbau.



Maffeis, Toenniessen, Zeidler (Hrsg.): Erfahrungen mit Java - Projekte aus Industrie und Hochschule, dpunkt.verlag, 141-170.

Abbildung WWLS:1: Logistikprozeß beim Kraftwerksbau

Um die komplexe Aufgabe der Logistik erledigen zu können, betreibt die ABB Kraftwerke AG seit Jahren ein eigens dafür entwickeltes Logistiksystem, dessen Herz eine Datenbank ist, die die logistischen Daten speichert und den am Logistikprozeß beteiligten Stellen zur Verfügung stellt. Informationen wie Verpackungsaufträge, Lieferscheine etc. wurden in diesem Prozeß bisher per Papier an die entsprechenden Stellen, die weltweit verteilt sind, geschickt. Die Eingabe der Informationen in das Logistik-System geschah im wesentlichen zentral. Diese Vorgehensweise war in einigen Bereichen fehleranfällig. So konnten sich durchaus relevante Informationen verändern, nachdem die entsprechenden Teilelisten bereits gedruckt und verschickt waren. Dies führte zu Inkonsistenzen und zu Informationsdefiziten bei allen am Logistikprozeß beteiligten Stellen. Dieses war einer der fachlichen Gründe, um das existierende Logistiksystem und dabei insbesondere die Client-Anwendungen zu erneuern. Dabei sollten die folgenden Ziele erreicht werden:

- ❏ Beschleunigung der Kommunikation zwischen allen beteiligten Partnern durch Online-Anbindung
- ❏ Verbesserung der Datenqualität
- ❏ Online-Anbindung externer Logistiksysteme, die beispielsweise von Zulieferfirmen betrieben werden.

Um diese Ziele zu erreichen, wurde Mitte 1996 ein Projekt mit dem Titel „World Wide Logistics System“ kurz WWLS gestartet.

1.1.1 Fachlichkeit und Funktionalität

Es wurde entschieden, im Rahmen des WWLS-Projekts als erstes die Anwendung APGM (*Application for Packaging Management*) zu entwickeln, die bei den Verpackerfirmen eingesetzt werden sollte. Als nächstes sollte eine Anwendung entwickelt werden, die direkt auf der Kraftwerksbaustelle installiert werden kann, damit auch dort die aktuellen Informationen über den Zustand der zu liefernden Teile verfügbar sind (SiLog = *Site Logistics*).

Nachfolgend beschreiben wir kurz die zentrale Funktionalität der ersten im Projekt WWLS erstellen Anwendung APGM. Diese als Client zur zentralen Logistikdatenbank konzipierte Anwendung, sollte die folgenden zentralen Funktionen für die Verpackerfirmen zur Verfügung stellen

- ❏ Abrufen von Verpackungsaufträgen aus der Datenbank
- ❏ Erfassen von Ablieferteilen und Versandpositionen
- ❏ Erfassen von Verpackungseinheiten (Kisten, Fässer, etc.)
- ❏ Zuordnung der Versandpositionen zu den Verpackungseinheiten
- ❏ Übertragung der erfaßten Teile und Zuordnungen zur Datenbank

Weiterhin sollte gewährleistet werden, daß die Daten nicht im Klartext übertragen werden und daß es eine genügend sichere Autorisierungsüberprüfung gibt.

Ein ganz zentrale Anforderung bestand darin, daß mit APGM auch offline gearbeitet werden können sollte. Ein Grund dafür ist darin zu sehen, daß man weltweit betrachtet nicht immer von ausreichend schnellen und stabilen Netzwerkverbindungen ausgehen kann. Da APGM – wie auch die anderen WWLS-Anwendungen – speziell auch in Ländern mit eher schwacher Netz-Infrastruktur eingesetzt werden sollte, hatte diese Anforderung höchste Priorität. Für APGM bedeutet die Offline-Fähigkeit, daß ein Verpacker sich seine relevanten Daten aus der zentralen Logistikdatenkank in seine private Umgebung laden und dort offline manipulieren kann. Anschließend müssen die manipulierten Daten wieder in die zentrale Datenbank transferiert und dort abgeglichen werden können.

1.1.2 Projektmanagement und –verlauf

Nachdem sich die ABB Kraftwerke AG entschieden hatte, Teile d.h. im wesentlichen die Client-Anwendungen ihres Logistiksystems zu erneuern, stellte sich die Frage, welche Technologie ver-

wendet werden konnte und wer ein geeigneter Entwicklungspartner war. Nach einer ersten Konzeptionsstudie wurden drei Technologien in die engere Wahl gezogen:

- Lotus Notes
- Microsoft Access
- Java und Web-Technologien

Nachdem diese Technologien bewertet und untersucht waren, entschied sich die ABB Kraftwerke AG dafür, auf der Basis von Java und in Kombination mit den vorhandenen Web-Technologien die neuen Client-Anwendungen zu entwickeln. Als Partner für dieses ehrgeizigen Projekt konnte das deutsche Forschungszentrum der ABB in Heidelberg gewonnen werden. Mitarbeiter des Forschungszentrums hatten zum Zwecke der Evaluierung einen Prototyp in Java entwickelt, der zeigte, daß die gewählte Technologie geeignet war, um diese Art von Anwendungen zu erstellen. Weitere Gründe für die getroffene Technologieentscheidung waren:

- Die *gute Portabilität* von Java-Anwendungen. Dies war wichtig, weil die zu entwickelnden Anwendungen weltweit auch auf unterschiedlichen Hardware- und Betriebssystemplattformen installiert werden sollten.
- Die softwaretechnisch gesehen *vorteilhaften Konzepte der Sprache* wie beispielsweise Ausnahmefallbehandlung, Paket-Konzept, strenge Typisierung, oder Garbage Collection. Wenn diese Konzepte sinnvoll genutzt werden, kann qualitativ hochwertiger Code entwickelt werden, der insbesondere bei Weiterentwicklungs- und Wartungstätigkeiten von großer Bedeutung ist.
- Die durchaus *gute Entwicklungseffizienz*. Dies hatte sich bereits bei der Implementierung des Prototyps gezeigt. Gründe dafür sind die gute Bibliotheksunterstützung, das inkrementelle Übersetzen oder das auf den ersten Blick leichte Einpassen einer Java-Anwendung in das WWW (mittels Browers und Web-Server).

Teamzusammensetzung

Nachdem die Technologieentscheidung gefallen war und feststand, welche Anwendung zuerst entwickelt werden sollte, wurde ein Projektteam gebildet. Dabei wurde versucht, sowohl technologische Kompetenz als auch Fachkompetenz aus dem Bereich Logistik im Team zu integrieren. Die technologische Kompetenz wurde von Mitarbeitern des ABB Forschungszentrums verstärkt durch Mitarbeiter des FZI Karlsruhe, der Universität Stuttgart, und des Software-Hauses PTA in das Projekt eingebracht. Diese Teammitglieder verfügten über zum Teil tiefes Know-how im Bereich der Objektorientierung, der Entwicklung von verteilten Systemen und auf den Gebieten Java und Internet-Technologien. Auf Seiten der ABB Kraftwerke AG lag die Projektleitung. Weiterhin waren zwei Mitarbeiter der ABB Kraftwerke AG im Team, die Know-how über den Logistikprozeß und über die Realisierung des vorhandenen Logistiksystems einbrachten. Diese Mitarbeiter hatte keine oder nur wenig Kenntnisse in der ausgewählten Technologie. Die Teamgröße variierte während der Projektdauer zwischen 4 und 11 Mitarbeitern. Nachteilig war, daß keine Endbenutzer also Mitarbeiter einer Verpackerfirma im Projektteam waren.

Projektdaten

Das Projekt WWLS wurde im Juli 1996 gestartet und ist zur Zeit noch nicht abgeschlossen. In den ersten vier Monaten des Projekts wurde ein Prototyp des APGM-Clients entwickelt, der neben der Oberfläche auch soviel Funktionalität enthielt, daß Daten aus der zentralen Logistikdatenbank via Internet zwischen Client und Server transferiert werden konnten. Dieser Prototyp wurde evaluiert. Danach wurde entschieden, das Projekt fortzusetzen. Für die Weiterentwicklung von APGM und die Produktisierung wurden anschließend 14 Monate benötigt. Danach wurde das System bei einem Pilotanwender installiert und erprobt. Zur Zeit wird es weltweit bei den Verpackungsfirmen installiert und eingesetzt. Zehn Installationen werden bereits genutzt; ca. 50 Installationen dieses Systems werden für Verpackungfirmen benötigt. Weitere Installationen für Zulieferer, insbesondere in der Schweiz, werden erwogen. APGM wurde vollständig in Java implementiert. Die Entwicklungs-, Einführungs- und bereits geleisteten Wartungskosten für APGM betragen etwa 1.2 Mio DM.

Probleme

Während die Aufwandsabschätzung für den ersten Prototyp recht präzise gemacht werden konnte, wurden die Kosten für die Weiterentwicklung und Produktisierung erheblich unterschätzt (Faktor 3-4). Ein wesentlicher Grund dafür ist darin zu sehen, daß sich die Anforderungen an das System während der Entwicklung häufig und intensiv änderten. Dies hätte zu einem großen Teil vermieden werden können, wenn Endbenutzer von Anfang an im Projektteam mitgearbeitet hätten. Die instabilen und sich ständig ändernden Anforderungen wirkten sich ebenfalls auf die Motivation des Entwicklungsteams aus. War die Motivation am Anfang sehr gut (Grund: Entwicklung eines innovativen Systems mit neuester Technologie), so ließ sie im Verlauf des Projekts bedingt durch die geschilderten Probleme auf allen Seiten deutlich nach. Dies scheint jedoch für diese Art von Technologie-getriebenen Projekten typisch zu sein. Ein weiterer Aspekt, der in diesem Kontext von Interesse ist, ist die Tatsache, daß der Versuch fehlschlug, die Mitarbeiter der ABB Kraftwerke AG, die intensiv bei der Spezifikation des APGM-Clients und beim Testen der Anwendung beteiligt waren, auf die Java- und Internet-Technologie zu schulen und auch in der Implementierung des Systems einzusetzen. Die Gründe dafür sind mannigfaltig.

1.1.3 Technische Architektur

Die neu entwickelte Anwendung APGM, die den Verpackungsprozeß unterstützen soll, basiert auf einer bestehenden Datenbankanwendung namens LOGISMA. Letztere setzt sich aus einer Reihe von *Oracle Forms* Programmen zusammen, die zur Zeit unter Windows 3.11 zum Einsatz kommen. Als Server für LOGISMA dient ein AIX-System mit einer Oracle-Datenbank. Alle logistischen Informationen für den aus der Sicht der ABB Kraftwerke AG relevanten Teil des Prozesses werden in LOGISMA abgebildet. Die existierenden *Forms*-Anwendungen umfassen neben dem Verpackungsprozeß beispielsweise die Montagevorbereitung sowie die Versandplanung.

Dreischichtenmodell

Als Architekturgrundlage wurde das Dreischichtenmodell gewählt. Generell lassen sich dadurch die Funktionalitäten für Darstellung, Anwendungs- und Prozeßlogik und die Datenhaltung sauber voneinander trennen. Dabei wird der Darstellungsteil zu den Benutzern (*Clients*) ausgeliefert, die beiden anderen Schichten laufen auf Servern, die normalerweise physisch unabhängig von den Client-Rechnern sind. Eine solche Architektur ermöglicht außerdem, den Darstellungsteil sehr klein zu halten, so daß er auch für auf sogenannten *Netzwerkcomputern* (NCs) ausgefrührt werden kann, die mit wenig Speicher und ohne lokale Festplatte, dafür mit einem guten Netzwerkanschluß auskommen.

In unserem Fall ließ sich diese klare Architektur jedoch nicht konsequent einhalten. Zu wichtig war die Anforderung, die Anwendung auf Client-Seite auch ohne bestehende Online-Verbindung betreiben zu können. Gründe hierfür waren neben der in Frage gestellten Verläßlichkeit von Internet-Verbindungen auch die Kosten, die bei einer lange andauernden Telefonverbindung zum Internet in Deutschland entstehen. Denn im Gegensatz zu den USA, wo in der Regel Ortsgespräche keinem Zeittakt unterliegen, oftmals sogar kostenlos sind, fallen hier vergleichsweise hohe Gebühren für diese Verbindungen an.

Aufgrund dieser Anforderung mußten große Teile der Prozeßlogik in den Client-seitigen Anwendungsteil verlagert werden, wodurch dieser wesentlich umfangreicher als die reine Darstellungslogik wurde. Nur so jedoch war es möglich, erfaßte Daten auch offline auf Konsistenz zu prüfen.

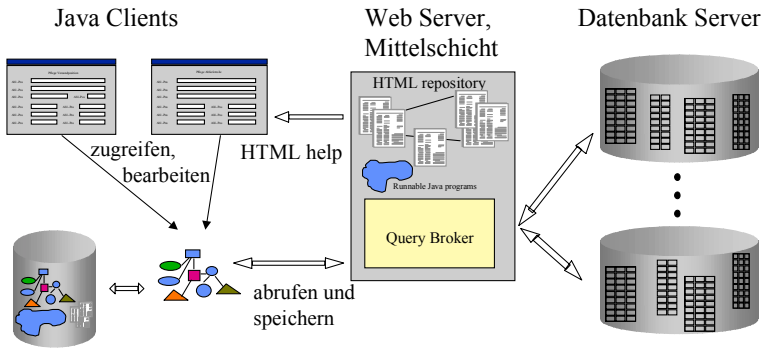


Abbildung: WWLS_2: Adaptierte Dreischichten-Architektur

Java Client: Applet oder Application?

Als Programmiersprache für die Client-Funktionalität wählten wir im Herbst 1996 Java. Bereits zu diesem Zeitpunkt boten die mit der Sprache in Form des *Java Development Kits* (JDK) kommenden Klassenbibliotheken sehr gute und einfache Einbindung von Online-Kommunikation und grafischen Ein-/Ausgabeschnittstellen (Graphical User Interfaces, GUIs). Die Sprache setzte objektorientierte Konzepte überzeugend um, Parallelität wurde in Form von Multithreading elegant eingebunden, und die Portabilität der Anwendungen war vorbildlich (siehe auch Abschnitt 1.3). Letzteres war für uns besonders wichtig, da wir über die Client-Plattformen sehr wenig Information zur Verfügung hatten und außerdem wenig Einfluß darauf nehmen konnten.

Wir entschieden uns zunächst, die Darstellungsschicht als ein *Applet*, also eine in einem Java-fähigen Browser laufende Java-Anwendung, zu implementieren. Wir hielten es für vorteilhaft, dadurch dem Anwender eine gewohnte Umgebung zu präsentieren und die neue Anwendung möglichst gut mit der Standardanwendung „Browser“ zu integrieren. Desweiteren bot sich der Browser auf diese Weise als Medium an, um die Online-Hilfe und Berichte in Form von HTML-Seiten darzustellen.

Wird eine Anwendung als Applet in einen Browser eingebettet, so hat das jedoch auch Nachteile. Aus den zuvor beschriebenen Gründen im Zusammenhang mit dem Offline-Betrieb mußten beispielsweise Daten persistent auf dem Client-Rechner persistent abgespeichert werden können. Ein Applet hat jedoch grundsätzlich keinen Zugriff auf das Dateisystem des Rechners, auf dem es läuft[1]. Indem wir eine sogenannte *native dynamic link library* (native DLL) programmierten und in den Browser integrierten, wurde ein Dateizugriff, eingeschränkt auf die Plattform, für die diese DLL geschrieben war, möglich, jedoch mit sehr mangelhafter Effizienz. Ein weiterer Nachteil der Browser-Integration besteht darin, daß die verschiedenen Browser wie *Microsoft Internet Explorer* und *Netscape Navigator / Communicator* Java unterschiedlich unterstützen. So unterstützt beispielsweise der Browser von Microsoft nur eine Teilmenge des Java-Standards[2]. Auch hinken die Java-Versionen der Browser dem Standard immer um ein bis zwei Versionen hinterher. Zwar würde mittlerweile hier das *Java Plug-In* von Sun Microsystems Abhilfe schaffen, welches die komplette Java-Umgebung in seiner aktuellen Version automatisch vom Sun-Server lädt und als ein Plug-In im Browser installiert. Jedoch war dieses Produkt zu dem Zeitpunkt, als wir die Browser-Integration implementierten, noch nicht verfügbar. Außerdem unterstützt es nicht die Ansteuerung des Browsers aus dem Java-Applet heraus, so daß ein weiterer Vorteil der Browserintegration verlorengegangen wäre.

Ein wesentlicher Vorteil eines Applets ist normalerweise, daß immer seine aktuelle Version ausgeführt wird, da es vor jedem Start vom Server geladen wird. Da in unserem Fall Anwendungslogik und Darstellung im Client-Teil zusammengefaßt werden mußte, wurde die Anwendung so groß, daß es unzumutbar gewesen wäre, sie jedesmal neu vom Server zu laden. Außerdem durfte

[1] Ausnahmen bilden hier die *signed applets*, die mit einer Signatur ausgestattet sind. Vertraut der Benutzer der Signatur, so darf er Applets, die mit dieser Signatur versehen sind, besondere Rechte wie z.B. Zugriff auf das Dateisystem einräumen. Signed Applets sind aber für den Browser-Markt noch nicht einheitlich standardisiert und schieden daher als Implementierungsvariante aus.

[2] Insbesondere verzichtet Microsoft bewußt auf die Implementierung des *Remote Method Invocation* (RMI) APIs, da es in direkter Konkurrenz zu Microsofts eigener Middleware DCOM (*Distributed Component Object Model*) steht.

dies nicht die einzige Möglichkeit sein, die Anwendung zu starten, da wie bereits erwähnt ein Offline-Betrieb möglich sein mußte. Wir lösten dieses Problem dadurch, daß der Programmcode auf den Client-Rechnern installiert wurde.

Natürlich bringt auch eine solche Installation Schwierigkeiten mit sich, besonders in einem heterogenen Umfeld, wie wir es vorfinden. Die Installationsprozedur muß für die verschiedenen Zielplattformen angepaßt werden, und es muß erheblicher Aufwand getrieben werden, um sicherzustellen, daß auf den Clients nur Versionen zum Einsatz kommen, die mit denen auf den Servern installierten verträglich sind.

Wir stellten schließlich fest, daß die vermeintliche „Integration" in den Browser abgesehen von der Hilfe- und Report-Darstellung nur darin bestand, daß das Applet in einem Browser-Fenster startete, danach jedoch eigenständige Fenster öffnete, die mit dem Browser selbst nur den Betriebssystemprozeß gemeinsam hatten. Aufgrund der zuvor erwähnten Probleme, die sich durch die Browser-Integration ergaben, beschlossen wir, statt eines Applets eine eigenständige Java-Anwendung (*Application*) auszuliefern. Die Hilfedarstellung im Browser ersetzten wir durch eine Java-Komponente zur Darstellung von HTML, die Berichte bereiteten wir in einem in Java geschriebenen Druck-Manager auf, der auch einen Preview anbietet. Damit wurde die Anwendung vollständig unabhängig vom Browser.

Mittelschicht

Die Mittelschicht, die in einer reinen dreischichtigen Architektur üblicherweise die Anwendungs- oder Geschäftslogik enthält, fiel aufgrund der Offline-Anforderung sehr schlank aus. Im wesentlichen beschränkt sich die Funktionalität des mittleren Servers darauf, Datenbanktransaktionen zu verwalten, Datenbankanfragen der Java-Clients entgegenzunehmen, zu bearbeiten und die Ergebnisse an die Java-Anwendung zurückzuliefern.

Der erste Implementierungsansatz für die Mittelschicht basierte auf dem in den Netscape Webserver integrierten Produkt *LiveWire*. Dabei handelt es sich um einen JavaScript-Interpreter mit Anbindungen an Datenbanktreiber. Die se Skripten werden über HTTP CGI-BIN Aufrufe angesteuert, die Ergebnisse werden in Form von HTML-Seiten zurückgeliefert.

Dieser Ansatz brachte jedoch erhebliche Nachteile mit sich. Neben der schwach strukturierten, schlecht skalierenden Sprache *JavaScript* bereitete das Übertragen von komplexen Datenbankanfragen, die in URLs codiert werden mußten, und das Verpacken / Entpacken der Ergebnisse in HTML Mühe. HTTP schreibt eine Maximallänge für URLs vor, die ein Server bzw. ein HTTP-Proxy verarbeiten können muß. Komplexere Anfragen sprengen diesen Rahmen jedoch leicht. Das Serialisieren der Anfrageergebnisse in HTML erwies sich zudem als umständlich; HTML wurde als binärer Übertragungskanal mißbraucht.

Wir versuchten, die Mittelschicht auf ein Produkt der Firma Weblogic mit dem Namen *T3* umzustellen. Uns gelang es zwar, die Anwendung damit lauffähig zu machen, jedoch gewannen wir nicht viel. T3 erwies sich zu diesem Zeitpunkt als recht instabil, was sich in Speicherzuordnungsfehlern und gelegentlichen Totalabstürzen äußerte. Auch mußten damals die Datenbankpaßwörter auf dem Client-Rechner abgelegt werden, was letztendlich als inakzeptabel angesehen wurde.

Wir stellten daraufhin die Mittelschicht auf *Java Remote Method Invocation* (RMI) um, indem wir einen kleinen Service in Java erstellten, der im wesentlichen die gleiche Funktionalität wie die vormalige JavaScript-Lösung aufwies, jedoch die Kommunikation nicht per HTTP sondern per RMI abwickelte. Dadurch waren lange Anfragen oder die Übertragung der Anfrageergebnisse, ebenso wir die Verwaltung von Transaktioen transparent in Java möglich. Die Kommunikation verbirgt sich also bei dieser Lösung vollständig hinter der objektorientierten Middleware.

Die Mittelschicht ist jetzt mit dem JDBC-Standard an die Datenbank angeschlossen. Dadurch ist eine gute Verfügbarkeit von Treibern gesichert. In unserem Fall liefert Oracle diese Treiber sogar kostenlos für ihre Kunden aus. Weiterhin sollte sich im Idealfall dadurch sogar die Datenbank transparent für den Client austauschen lassen, da JDBC-Aufrufe unabhängig von der zugrundeliegenden Datenbank sein sollten.

Als Betriebssystemplattform für die Mittelschicht haben wir bisher Windows-NT eingesetzt. Hier wurden jedoch mehrere Probleme offensichtlich. Die Fernwartung eines NT-Systems erweist sich momentan als aufwendig, besonders da ein *rlogin*-Daemon fehlt und Prozesse nicht über

Kommandoeingabe verwaltet werden können. Eine „Fernbedienung“ des Servers ist nur über Werkzeuge möglich, die versuchen, die komplette grafische Benutzungsoberfläche über das Netzwerk zu replizieren, z.B. PC-DUO. Weitere Schwächen, die besonders beim Auslieferungsprozeß offensichtlich werden, sind das Fehlen von *rshd*- und *rexecd*-Implementierungen. Dadurch kann zum Beispiel nicht einmal eine Archivdatei Server-seitig entpackt werden, sondern muß über langsame Netzwerklaufwerke gesendet werden.

Ein weiteres Problem stellt die gegenwärtige Implementierung des Java Development Kits von Sun für Windows-NT in seiner Version 1.1.5 dar. Gemäß unserer Erfahrungen werden Betriebssystem-Threads und Objekt-Handles teilweise nicht korrekt freigegeben, so daß es im Laufe der Zeit zu Speicherproblemen kommt.

Aus diesem Grund stellten wir den Server auf eine Sun Workstation mit Solaris als Betriebssystem um. Diese Umgebung eignet sich nach unseren bisherigen Erfahrungen deutlich besser als fernwartbarer Server, da alle wesentlichen Funktionalitäten per *rlogin* und Kommandozeilen verfügbar sind. Auch die entfernte Ausführung von Kommandos während der Auslieferung mittels *rsh* und *rexec* sind so möglich. In unseren bisherigen Tests waren wir auch nicht in der Lage, die unter NT aufgetretenen Probleme mit der Sun Java Implementierung unter Solaris zu reproduzieren.

Auslieferung, Installation und Updates

Immer häufiger werden Anwendungen nicht mehr über physikalische Datenträger, sondern über das Internet oder ein Intranet ausgeliefert. Die elektronische Auslieferung erfolgt in der Regel schneller. Außerdem fallen Vervielfältigung und Versand weg. Auch wir liefern unsere Anwendung über das Internet aus und plazieren daher alle Dateien auf einem öffentlichen Webserver, so daß von dort die Anwendung heruntergeladen werden kann.

Da wir uns für eine Realisierung der Client-Seite als *Application* und nicht als *Applet* entschieden hatten, mußte geklärt werden, wie diese auf dem Client-Rechner installiert werden konnte.[3] Es genügt nicht, die Dateien aus einem Archiv auszupacken. Es muß darüber hinaus ein einfacher Weg gefunden werden, um die Anwendung zu starten, unter Windows zum Beispiel, indem die Anwendung im Startmenü eingetragen wird. Weiterhin können Einträge in Umgebungsvariablen (z.B. *CLASSPATH*) nötig werden. Außerdem sollen alle notwendigen Schritte im Auslieferungszyklus, beginnend beim Zusammenstellen der auszuliefernden Dateien und Archive bis hin zur vollständigen Installation auf den Client-Rechnern möglichst automatisiert ablaufen.

Wir sind die geschilderte Problematik von zwei Seiten angegangen. Zum einen haben wir eine Skriptbasierte Lösung auf der Entwicklungsseite erstellt, die zu einer Auslieferung die benötigten Dateien zusammenfaßt und zum Auslieferungsserver transferiert. Dabei wird unterschieden, ob Komponenten (Dateien uns Archive) auf dem Client-Rechner oder auf dem Server installiert werden müssen. Der zweite Teil der Lösung wird durch das Produkt *InstallShield* abgedeckt. InstallShield arbeitet ein spezielles Skript ab, welches die beim Client zu installierenden Komponenten in ein einziges Archiv bündelt und mit einem Programmrahmen umgibt, der dann die Anwendung auf dem Client-Rechner installiert. Dabei kann dann InstallShield auch betriebssystemspezifische Dinge wie das Setzen von Umgebungsvariablen, Anlegen von Einträgen in das Startmenü oder das Einrichten von Verzeichnissen übernehmen. Ein wesentlicher Vorteil dieses Vorgehens liegt darin, daß die Anwendung in der Regel vom Anwender installiert werden kann. Eine Vor-Ort-Installation durch einen Mitarbeiter der ABB entfällt also weitestgehend.

Ein entscheidender Nachteil von InstallShield in diesem Zusammenhang ist, daß in der uns vorliegenden Version 5.0 das Werkzeug nicht vollständig automatisiert gesteuert werden kann. Es ist also immer noch Interaktion mit einem Benutzer nötig, um das Erstellen des auszuliefernden Archivs anzustoßen. Bis auf diesen Punkt jedoch konnten wir den Rest des Auslieferungsprozesses durch Skripten vollständig automatisieren.

Der zuvor genannte Vorteil, daß Installationen nicht vor Ort durch sachkundiges Personal betreut werden müssen, kommt nur dann voll zum tragen, wenn auch Updates der einmal installierten Anwendung vom Benutzer selbst durchgeführt werden können. Man kennt diese Technologie

[3] Diese Frage hätte sich auch gestellt, wenn wir bei der *Applet*-Architektur geblieben wären, das Applet aber lokal beim Client „gecached“ hätten.

beispielsweise aus dem *Netscape Communicator*, für den man vom Netscape-Webserver Updates laden kann, die sich selbständig in den installierten Browser integrieren.

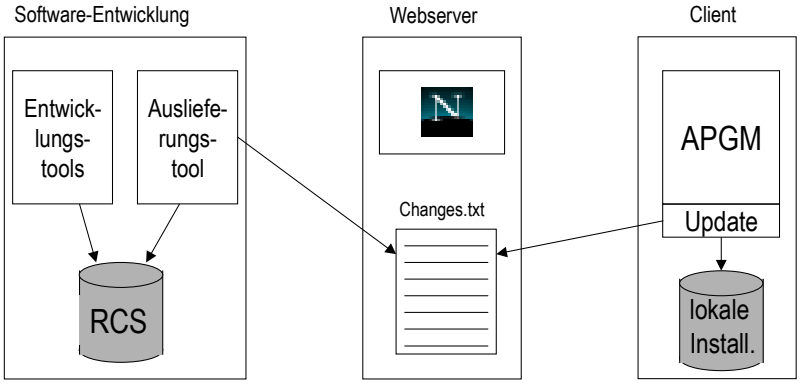


Abbildung WWLS_3: Integration von automatischem Update und Softwareentwicklung

Auch diese Problemstellung hat wiederum zwei Seiten: Zum einen müssen auf der Anwenderseite — sprich, dem Client — Vorbereitungen getroffen werden, um Updates nachzuladen und zu integrieren. Zum anderen muß der Server die entsprechenden Updates zur Verfügung stellen. Für die Client-Seite wurde ein kleines Java-Programm entwickelt, welches mit der Anwendung zusammen ausgeliefert wird und in der Lage ist, Updates vom Server zu laden und lokal zu installieren. Der Prozeß, um Updates auf dem Server bereitzustellen und zu kennzeichnen, war deutlich aufwendiger.

Wir wollten die automatischen Updates möglichst gut mit dem Entwicklungsprozeß (siehe auch Abschnitt 1.2) integrieren, so daß die Entwickler bei der Modifikation des Programmsystems nicht explizit Rücksicht auf die Verfügbarkeit der automatischen Updates nehmen müßten. Dazu war es notwendig, die automatischen Updates an das Versionsverwaltungssystem zu koppeln. Wir extrahieren dazu aus den Logfiles unseres Versionsverwaltungssystems die Änderungen seit der letzten Auslieferung. Da nur Quelldateien, nicht aber die Objektdateien von der Versionsverwaltung kontrolliert werden, wurde ein kleines Werkzeug erstellt, welches die Quelldateien auf die übersetzten Objektdateien abbildet. So erhalten wir die Menge der beim Client zu aktualisierenden Dateien seit der letzten Auslieferung. Dieses Ergebnis wird auf dem Server publiziert, so daß es vom Update-Modul des Clients zugreifbar ist. Zusammen mit der Versionsnummer des aktuell bei ihm installierten Anwendung kann dann der Client automatisch ermitteln, welche Dateien nachgeladen werden müssen, um die Anwendung auf den aktuellen Stand zu bringen.

Aus der Benutzersicht stellt sich dann das Update so dar, daß beim Anmelden an den Server gegebenenfalls eine Meldung erscheint, daß ein neues Update verfügbar ist. Der Anwender kann dann per Mausklick den Update-Vorgang anstoßen. Eine vollständige Automatisierung ohne Benutzerabfrage wurde nicht gewählt, da momentan noch Änderungen an den lokal gespeicherten Daten zurück in die Datenbank repliziert werden müssen, bevor das Update durchgeführt wird. Das liegt daran, daß die Migration von persistenten Objekten auf der Client-Seite zur Zeit noch nicht implementiert ist.

Verwendete Muster

Beim Entwurf und in Bß der Implementierung der Anwendung haben wir einige Entwurfsmuster verwendet, die hier kurz beschrieben werden sollen.

Eine der Anforderungen war, daß die Sprache der Benutzungsoberfläche zur Laufzeit umgestellt werden kann. In diesem Fall sollten dann alle GUI-Elemente ihre Beschriftungen in die dann gewählte Sprache ändern. Um diese Anforderung umzusetzen, wählten wir das *Observer*-Entwurfsmuster (siehe Gamma, 1995). Alle als separate Fenster dargestellten Anzeigeelemente registrieren sich als Beobachter des Objektes, welches für die Sprachumschaltung verantwortlich ist. Wird dann die Sprache umgeschaltet, werden alle sprachabhängigen Fenster benachrichtigt und aktualisieren ihre Titel und Inhalte, so daß diese in der neu gewählten Sprache erscheinen.

Ein weiteres Muster kam bei der Modellierung der Datenbankzugriffe zum Einsatz. Wir nennen es eine *Schattenhierarchie*. Wesentliches Merkmal dieses Musters ist, daß es zu einem existierenden Teil einer Klassenhierarchie eine spiegelbildliche Kopie (den *Schatten*) verwendet, wobei

dann in dieser Schattenhierarchie gewisse Funktionalität ausgelagert wird, die an sich in die Originalhierarchie gehört. In unserem konkreten Fall lag eine Hierarchie von domänenspezifischen Klassen vor, die etwa Dinge wie Ablieferteile, Verpackungseinheiten oder Mengeneinheiten beschreiben. Die Regeln, wie diese Objekte in die Datenbank abzubilden und von dort wieder zu materialisieren sind, gehören zwar in gewisser Weise zu diesen Klassen, sollten aber aus Gründen der Flexibilität dennoch nicht direkt in ihnen implementiert werden. Das Muster erlaubte uns also in diesem Fall, den Persistenzaspekt zwar durch die Schattenhierarchie eindeutig den verschiedenen Domänenklassen zuzuordnen, ließ uns aber die Freiheit, die Domänenklassen selbst vom Aspekt der Persistenz vollständig zu entkoppeln.

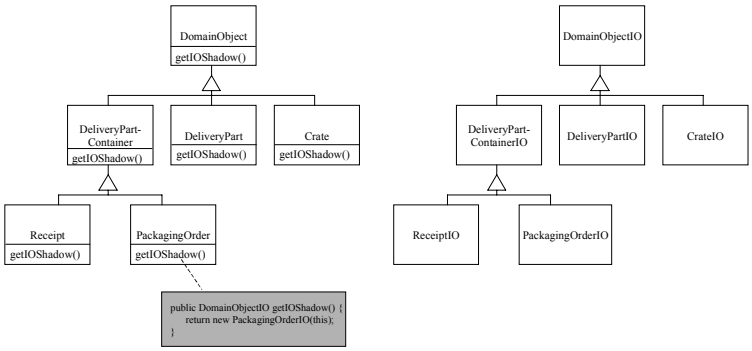


Abbildung WWLS_4: Schattenhierarchie-Muster

1.1.4 Mengengerüste und Leistungsdaten

APGM wird in Deutschland und einigen Nachbarländern an etwa 50 Vertragsverpacker ausgeliefert. Zusätzlich wird zur Zeit in Erwägung gezogen, den Zulieferfirmen der ABB Kraftwerke AG in der Schweiz ebenfalls APGM auszuliefern, so daß mit etwa weiteren 200 Installationen zu rechnen wäre.

Pro Jahr bearbeitet die ABB etwa 30 Kraftwerksprojekte, wobei jedes von ihnen sich aus etwa 10 Millionen Einzelteilen zusammensetzt. Ein Projekt läuft für gewöhnlich zwischen zwei und drei Jahren, so daß grob gesprochen pro Jahr 100 Millionen Teile in die Datenbanken der ABB eingetragen werden. Bei etwa 250 Arbeitstagen und neun Stunden pro Tag, ergibt sich ein grobes Mittel von 14 Teilen pro Sekunde, die in die Datenbank einzufügen sind.

Auf der anderen Seite erwarten wir von den Verpackern und Zulieferfirmen zwischen zwei und zehn Abgleichschritten pro Tag (im Mittel ca. fünf), wobei jeder Abgleich einige hundert Teile mitsamt ihrer Verpackungsinformation überträgt. Nehmen wir an, es handle sich jedesmal im Mittel um etwa 500 Transaktionen (ca. eine Transaktion pro verpacktem Teil), so erhalten wir 625.000 Transaktionen pro Tag. Gegeben, daß die Verpacker sich in ähnlichen Zeitzonen befinden und sich dadurch diese Transaktionen über einen Zeitraum von nicht mehr als neun Stunden verteilen dürften, ergeben sich so etwa 20 Transaktionen pro Sekunde.

Spiegelt man diesen Wert an den zuvor ermittelten 14 Teilen pro Sekunde, so stellt man fest, daß die Größenordnungen gut übereinstimmen und die Schätzungen somit plausibel sind.

Es seien noch einige Daten zur Anwendung selbst erwähnt. Zuletzt bestand APGM aus ca. 95.000 Zeilen Java Code, verteilt auf 530 Klassen. Davon sind 18.000 Zeilen und 83 Klassen in einem Paket zur Verschlüsselung und Authentisierung, welches nicht von uns selbst erstellt, sondern nur verwendet wurde. Im Durchschnitt weisen unsere Klassen vier Attribute und fünf Methoden mit jeweils etwa sechs Zeilen Code pro Methode auf. Auf die 95.000 Codezeilen entfallen 30.000 Zeilen Kommentare. Die Verteilung des Funktionsumfangs zwischen Client und Mittelschicht ist wie folgt: 515 Klassen bilden den Client, 100 Klassen (inklusive der Verschlüsselungsklassen) bilden die Mittelschicht.

1.2 Der Entwicklungsprozeß, die Methodik

1.2.1 Die Vorgehensweise

APGM wurde nicht entlang eines definierten expliziten Vorgehensmodells entwickelt. Richtlinien für die Entwicklung existierten nur im Bereich der Programmierung mit Java.

Prototyping

Aufgrund der zu lösenden unbekannten Aufgabenstellung wurden verstärkt Prototypen entwickelt. Im Sinne der in Lichter (1994) enthaltenen Klassifikation von Prototypen wurden sowohl Demonstrationsprotoypen als auch Labormuster und ein Pilotsystem entwickelt. Diese Prototypen dienten häufig mehreren Zwecken. Auf der einen Seite dienten sie dazu, den Projektfortschritt gegenüber der Fachabteilung und dem Management zu dokumentieren. Auf der anderen Seite wurden immer dann, wenn technologisch neue Ansätze verwendet werden sollten, Prototypen im Sinne von Machbarkeitsstudien (Labormuster) entwickelt. Ein Beispiel dazu war ein Prototyp, der eine verschlüsselte und geschützte Anbindung zwischen der Logistikdatenbank und des Client-Teils mittels SSL realisierte. Wenn die mit den Labormustern erzielten Erfahrungen und Ergebnisse positiv waren, wurden Teile davon in die Produktentwicklung integriert. Aufgrund der guten Entwicklungseffizienz von Java konnten diese Prototypen entsprechend schnell realisiert werden. Weiterhin wurde versucht, diese Prototypen zu benutzen, um die Anforderungen zu fixieren. Dies gelang jedoch nicht ausreichend, da keine Endbenutzer in den Bewertungsprozeß der Prototypen integriert waren.

Test

Während der Entwicklung wurde frühzeitig versucht, das entstehende System zu testen. Testfälle im Sinne eines funktionalen Black-box Tests wurden von den Mitarbeitern der ABB Kraftwerke AG entwickelt und in einer speziell dafür installierten Testumgebung ausgeführt. Dabei zeigten sich sehr bald Schwächen im Bereich des Domänenmodells, die als Konsequenz der mangelhaften Anforderungsspezifikation erst beim Test zu Tage traten. Notwendige Veränderungen im Bereich der Domänenklassen führten zu erheblichen Aufwänden bei den nachgeschalteten Regressionstests. Aufgrund dieser Erfahrungen wurden bei der Entwicklung der zweiten Anwendung folgende Maßnahmen ergriffen:

- ❑ In der Startphase wurden die zentralen Anforderungen mittels zwei-tägiger Workshops ermittelt, an denen Vertreter aller relevanten Personengruppen beteiligt waren.
- ❑ Um den Testaufwand für die Domänenklassen zu reduzieren, wurde ein spezielles Test-Framework entwickelt (siehe auch Abschnitt „Testunterstützung“).

1.2.2 Entwicklungswerkzeuge

Im Laufe des WWLS-Projekts wurden unterschiedliche Entwicklungswerkzeuge für Java erprobt, da zu Projektstart keine stabile und den Anforderungen der Teammitglieder entsprechende Entwicklungsumgebung installiert werden konnte. Im folgenden beschreiben wir tabellarisch die untersuchten und im Projekt eingesetzten Werkzeuge (diese sind kursiv gesetzt). Die Aspekte Konfigurationsmanagement, Dokumentation und Test behandeln wir anschließend getrennt.

Werkzeug	Vorteile	Probleme
Symantec Café	• einfache Installation • preisgünstig • grundlegendes Sourcecode-Browsing möglich	• nicht offen • keine Kopplung mit Versionsverwaltung • wenig mächtiges Cross-Referencing

Sniff+J	• sehr gutes Cross-Referencing • offen • einfache Einbindung verschiedenster Versionierungstools	• umständliche Bedienung • schlechte Performance • Java-Support unreif
Borland JBuilder	• sehr guter GUI-Builder • schnelle Kompilierung • guter Debugger	• benutzt JBuilder-spezifische Klassen für GUIs • Probleme mit Editor
Rational Rose	• „quasi“ ABB Standard • sehr gutes reverse engineering • flexible Diagramm-Darstellungen	• Round-Trip Engineering schwierig • schlechte Integration mit anderen Tools
java_g –prof	• enthalten im Java-Standard • keine Instrumentierung des Codes nötig	• starker Einfluß auf Laufzeit • Meßgenauigkeit schwankt
Hyperprof	• gelungene Visualisierung • Pure Java	• extrem langsam bei größeren Profiles
JavaStar	• automatische Kombination verschiedener Eingabedaten möglich	• gewöhnungsbedürftiges Interface • lange Compile-Zyklen
JTest	• vollautomatische Testumgebung für GUIs	• Probleme bei Verwendung von JFC • unhandliche Darstellung der Ergebnisse

1.2.3 Konfigurationsmanagement

Um Versionen und Konfigurationen zu verwalten, wurde die frei verfügbare NT-Version von RCS eingesetzt. Bereits sehr bald stellte sich heraus, daß die von RCS zur Verfügung gestellte Bedienerschnittstelle nicht geeignet war, damit auch neue Teammitglieder schnell und effektiv damit arbeiten konnten. Um dieses zu verbessern, wurde eine Menge an semantischen Operationen für das Versions- und Konfigurationsmanagement definiert (z.B. Installieren einer neuen Umgebung, Ein- und Auslagern, eine Baseline definieren, etc). Diese Operationen wurden als NT-Batchdateien auf Basis RCS implementiert und den Entwicklern zur Verfügung gestellt. Diese Implementierung bewährte sich zwar, war allerdings fest auf die Plattform NT zugeschnitten und nur schwer erweiterbar. Damit sie auch auf anderen Plattformen verwendet werden konnte und Erweiterungen besser zuließ, wurde eine Neuimplementierung in Java durchgeführt. Dabei wurden auch funktionelle Erweiterungen möglich, z.B. eine grafische Oberfläche, die die Baselines und Releases zeigte. Ein weiterer Vorteil war, daß die doch teilweise prototypische Implementierung der NT-Batch-Version nun durch eine software-technisch saubere und damit auch besser wartbare Implementierung ersetzt wurde. Desweiteren wurde durch den Einsatz von Java die Aufspaltung in einen Client- und einen Server-Teil erleichtert, so daß Server-lastige Tätigkeiten wie etwa Recherchen oder Statistiken über das gesamte Repository auf dem Server durchgeführt werden konnten. Eine

Replikation der Benutzerkopien auf beliebige Rechner ist in Arbeit, um die Laufzeiten von Übersetzer und Debugger zu verbessern.

Die entstandenen Java-Ergänzungen zu RCS wurden anschließend in Emacs und SNiFF+*J* integriert, so daß das Versionsverwaltungssystem per Mausklick verwendet werden kann.

1.2.4 Testunterstützung

Wie bereits erwähnt, wurde aufgrund der negativen Erfahrungen, die beim Test der Anwendung APGM gemacht wurden, konstruktiv versucht, die Testbarkeit der wesentlichen Domänenklassen der Anwendung SiLog zu verbessern. Zu diesem Zweck wurde der in Beck (1994) beschriebene Ansatz für das Testen von Smalltalk-Klassen adaptiert. Die zentrale Idee dabei ist, für jede Domänenklasse eine Testklasse zu entwickeln, die alle relevanten Tests definiert und Funktionalität zur Verfügung stellt, damit diese Tests auch durchgeführt werden können. Die folgende Abbildung zeigt schematisch, wie die Testklassen und die wesentlichen Klassen des „Test-Frameworks“ zusammenspielen.

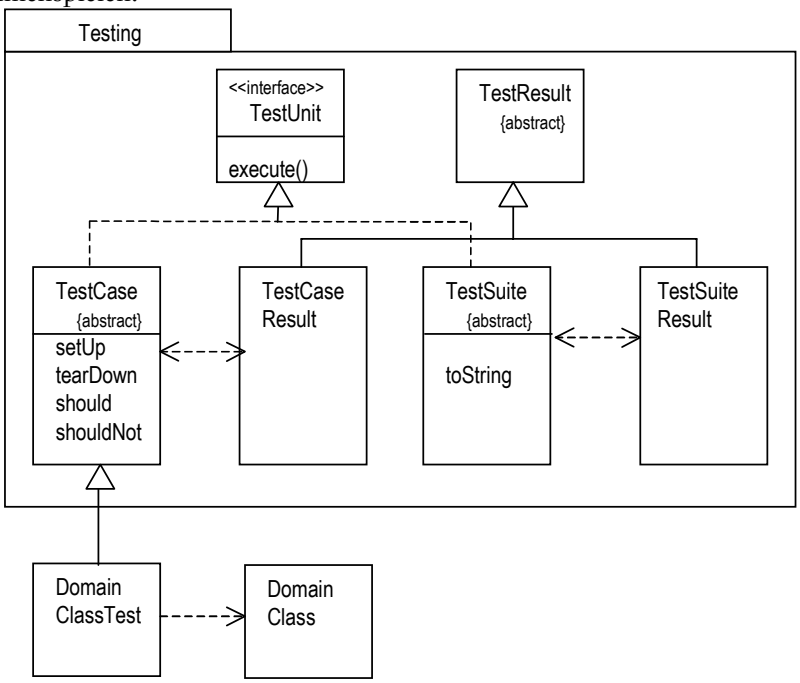


Abbildung WWLS_5: Aufbau des Test-Frameworks

Die Klasse `TestUnit` definiert eine Schnittstelle, die von den Klassen `TestCase` und `TestSuite` implementiert werden muß. Während die Klasse `TestCase` Operationen definiert, die gebraucht werden, um einen Testfall zu spezifizieren, auszuführen und Soll- und Ist-Ergebnis zu vergleichen, realisiert die Klasse `TestSuite` einen Behälter, der eine Menge von `TestCase`s aber auch `TestSuite`s aufnehmen kann. Dadurch kann eine ganze Sequenz von einzelnen Testfällen ausgeführt werden. Um in diesem Kontext einen Testfall für eine Klasse zu formulieren, muß eine entsprechende Testklasse erzeugt werden, die von der Klasse `TestCase` erben muß. Mittels der geerbten Operation `execute` kann dieser dann ausgeführt werden. Um die Klassen `TestCase` und `TestSuite` allgemein formulieren zu können, mußte die Implementierung auf Metainformationen basieren. Diese wurden mit Hilfe der Operationen des Reflection-Interface von Java gewonnen (`java.lang.reflect`).

1.2.5 Dokumentation

Die Erfahrung aus anderen Projekte zeigt, daß es nicht sinnvoll ist, die Entwicklungsdokumentation mit verschiedenen Werkzeugen und Medien zu erstellen. Da mit *JavaDoc* standardmäßig ein Werkzeug zur Verfügung stand, mit dem einfach die Paket- und Klassendokumentation in HTML

erstellt werden konnte, entschied man sich, auch die anderen relevanten Entwicklerdokumente in HTML zu verfassen. Dies hat den Vorteil, daß Querverweise zwischen allen diesen Dokumenten erstellt werden können. So war es möglich, aus Entwurfsmodellen, die z.B. mit dem Werkzeug Rational Rose erstellt und in UML notiert wurden, direkt auf die Paket- oder Klassendokumentation zu verweisen, die mittels JavaDoc immer up-to-date gehalten werden konnte. Damit die während des Projektverlaufs inkrementell entstehenden HTML-Dokumente übersichtlich strukturiert werden konnten, wurde eine mehrschichtige Dokumentenarchitektur entworfen, die schubladenmäßig für alle Texte einen Platz zur Verfügung stellte. Diese organisierte die Dokumentation auf oberster Ebene durch die folgenden sogenannten „books“:

- *Concept Book*: Es enthält Beschreibungen der wesentlichen Konzepte, die für das Verständnis der Implementierung notwendig sind; diese kann man auch als semantische Enturfsentscheidungen bezeichnen.
- *Design Book*: Es enthält beginnend bei einer Beschreibung des Systemdesigns den Entwurf aller Pakete bis auf die Klassenebene sowie kurze Designbeschreibungen.
- *Classes Book*: An dieser Stelle werden die von JavaDoc aus den Quelldateien erzeugten Klassen und Paket-Beschreibungen zusammengefaßt.
- *Development Book*: Es enthält alle Richtlinien, die im Projekt verwendet werden.

Weiterhin wurde ein Glossar vorgesehen, das alle projektrelevanten Begriffe definiert. Abbildung WWLS_6 zeigt die Einstiegsseite zu diesem Dokumentationssystem.

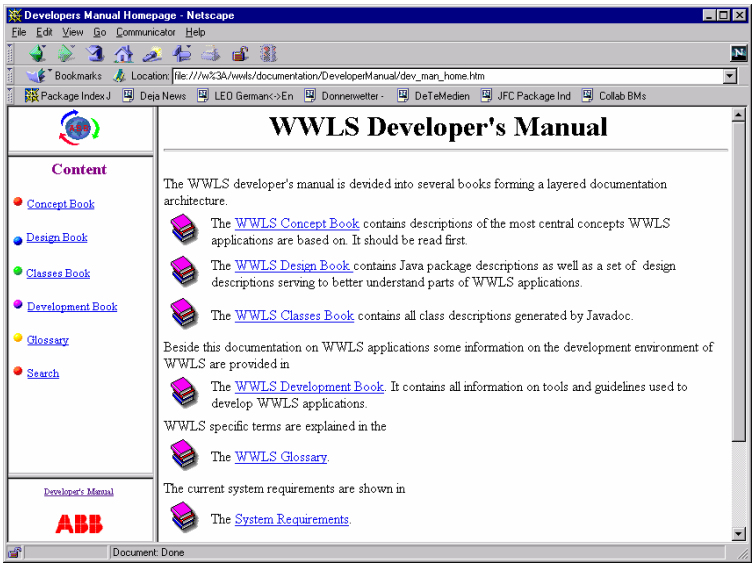


Abbildung WWLS_6: Das WWLS-Dokumentationsystem

Ein weiterer Vorteil der vollständig in HTML abgelegten Dokumentation lag darin, daß vorhandene HTML-Editoren verwendet werden konnten, um die Texte zu erstellen, und daß mit einer Suchmaschine zu jeder Zeit alle Dokumente in der bekannten Art und Weise durchsucht werden konnten.

1.3 Kritische Bewertung des Einsatzes von Java

1.3.1 Vor- und Nachteile bzw. Stärken/Schwächen von Java

Sprachmodell und JDK

Die Programmiersprache Java präsentierte sich uns bereits in der Ende 1996 vorliegenden Version (damals zusammen mit dem JDK 1.0.2) als eine elegante, einfache und gleichzeitig mächtige Sprache. Als wesentliche und offensichtliche Vorteile erschienen uns:

- *Automatische Speicherbereinigung*
  Hierdurch entfällt die Fehlerklasse, die bei einer typischen C++-Entwicklung einen beträchtlichen Anteil des Aufwandes für die Fehlersuche verursacht.
- *Umfangreiche Bibliotheken (JDK)*
  Besonders im Online-Bereich überzeugt die Einfachheit, mit der ganze Webpages gelesen, Socket-Verbindungen initiiert oder komplexe Objekte serialisiert werden können.
- *Gute Integration des Exception Handlings*
  Das JDK macht intensiven Gebrauch von Exceptions, wodurch der Entwickler gezwungen wird, diese in der einen oder anderen Form abzufangen und zu behandeln.
- *Einbindung von Multithreading und Synchronisation*
  Gerade Server-seitig ist es wichtig, Anfragen parallel abarbeiten zu können. Synchronisation auf gemeinsam genutzten Objekten ist jedoch essentiell und sehr gut in Java integriert.
- *Die Sprachumgebung enthält Middleware (RMI im JDK)*
  Dadurch wird die Konstruktion verteilter Applikationen, die vollständig in Java entwickelt wurden, erheblich erleichtert. Eine Anbindung an CORBA erfolgt mit JDK Version 1.2.
- *Gute Portabilität*
  Besonders die Möglichkeit, eine grafisches Client-Anwendung auf einfache Weise portabel gestalten zu können, verleiht Java einen erheblichen Vorteil gegenüber anderen Entwicklungsumgebungen.

Schwächen im Umfeld der Sprache sehen wir in folgenden Punkten:

- *Fehlen von Multiple Inheritance*
  Gerade bei der Implementierung bestimmter Entwurfsmuster, wie etwa dem *Observer*-Muster, werden oftmals Teile der Implementierung der verschiedenen Rollen durch Vererbung wiederverwendet. Ein Verhalten im Sinne mehrerer Rollen wird durch einfache Implementierungsvererbung weitgehend ausgeschlossen, und mehrfache Interface-Vererbung hilft nicht immer, diesen Mangel zu beheben. Es erscheint fraglich, ob der Gewinn an Einfachheit den Verlust an Ausdrucksstärke überwiegt
- *Keine Templates*
  Besonders schmerzhaft erweist sich dieser Mangel im Bereich der Kollektionen, die dadurch ihre Schnittstellen immer auf die allgemeinste Oberklasse *Object* typisieren. Dadurch werden häufige Typkonversionen (*Casts*) nötig.

Zwischen den JDK-Versionen 1.0.2 und 1.1 wurde die Sprachspezifikation geändert. Wesentliche Änderungen waren die Einführung von *Inner Classes*, *Anonymous Classes* und der Wegfall des *private protected* Sichtbarkeitsbereichs. Dazu einige Anmerkungen.

Inner Classes erlauben, Klassen in anderen Klassen zu schachteln, optional auch Exemplare der inneren Klasse fest an ein Exemplar der äußeren Klasse zu binden (*non-static inner class*). Was zunächst wie ein mächtiges Sprachmittel aussieht, erschwert letztendlich die Lesbarkeit des Codes und verletzt damit eines der ursprünglichen Entwurfsprinzipen der Sprache, nämlich die Einfachheit.

Ähnliches gilt für die sogenannten *Anonymous Classes*. Hierbei handelt es sich um Klassen, die bei der Erzeugung eines Objektes deklariert werden. Der offensichtliche Grund für die Einführung dieses Sprachmittels war, die Schreibweise bei der Verwendung des mit JDK 1.1 eingeführten Event Handling Modells zu verkürzen. Die Syntax dieses Konstrukts ist jedoch sehr gewöhnungs-

bedürftig und macht den Code unleserlich. Es werden dadurch die Objekterzeugung und die Beschreibung von Funktionalität miteinander vermischt.

Der bis zu JDK 1.0.2 vorhandene Sichtbarkeitsbereich *private protected* bildete das nach, was C++-Entwicklern als *protected*-Sichtbarkeit geläufig war, nämlich die Sichtbarkeit des deklarierten Elements auch in Unterklassen. Java belegte jedoch den *protected*-Scope mit einer leicht geänderten Semantik, indem zusätzlich zu den Unterklassen auch noch alle Klassen im selben Paket (*package*) Zugriff auf diese Elemente erhalten. Unglücklicherweise wies die Implementierung des JDK 1.0.2 einen Fehler bezüglich *private protected* auf. So wurden entsprechend deklarierte Methoden nicht polymorph aufgerufen, so daß praktisch immer der statische Typ eines Objektes entscheidend dafür war, welche Implementierung der Methode ausgeführt wurde, ungeachtet einer eventuell existierenden Redefinition in der konkreten Unterklasse. Statt diesen Fehler zu beheben, wurde dieser Sichtbarkeitsbereich entfernt. Dadurch hat die Sprache ein wichtiges Mittel verloren, um die Sichtbarkeit einzuschränken.

Effizienz

Ein häufig zitierter Nachteil von Java ist die mangelnde Effizienz der entwickelten Anwendungen, besonders im Vergleich zu C++. Wir können diese Erfahrung nur zum Teil bestätigen. Durch die *Just-in-Time* (JIT) Compiler, die den normalerweise interpretierten Java-Bytecode zur Laufzeit in Maschinencode übersetzen, steigt die Ausführungsgeschwindigkeit erheblich. Weitere Verbesserungen sind von Sun Microsystems in Form der *HotSpot*-Technologie angekündigt. Dabei handelt es sich um einen JIT-Compiler, der besonders viel Laufzeitwissen wie Variablenbelegungen und Datenflußanalysen in die Übersetzung einfließen läßt und dadurch sogar besser optimieren kann als ein statischer Compiler. Demonstrationen dieser Technologie auf der *JavaOne*-Konferenz im März 1998 haben bereits erste beeindruckende Zahlen geliefert. Demnach erreichen Java-Anwendungen etwa die gleiche Effizienz wie Anwednungen, die in C++ implementiert sind.

Für unsere Art der Anwendung zeigte sich bereits zu Beginn, daß die Antwortzeiten auch mit Java-Technologie ausreichend kurz waren. Wichtiger als Laufzeiteffizienz erwies sich jedoch die Effizienz bei der Entwicklung. Denn durch reduzierten Portierungsaufwand, durch die JDK-Bibliotheken, die automatische Speicherbereinigung und stark verkürzte Übersetzungszeiten lassen sich Applikationen in Java deutlich rascher entwickeln als in vielen anderen Sprachumgebungen, erreicht höchstens durch Smalltalk-Umgebungen.

Es hängt auch von der Art der zu entwickelnden Anwendung ab, wie man zwischen Laufzeit- und Entwicklungseffizienz einer Sprache abwägt. Für ein höchstperformantes Simulationssystem beispielsweise kann es durchaus die Mühe wert sein, handoptimierten Assemblercode zu erstellen. Für transaktionelle Anwendungen mit Benutzerinteraktion ist jedoch aus unserer Sicht die Hardware heutzutage leistungsfähig genug, um den heute noch verbliebenen Faktor von etwa 2 zwischen Java- und C++ -Effizienz irrelevant zu machen.

Werkzeuge und Komponenten

Wichtig für die Produktivität, die mit einer Programmiersprache erreicht werden kann, ist die Unterstützung durch Werkzeuge wie etwa integrierte Entwicklungsumgebungen. Als wir Ende 1996 begannen, Java zu benutzen, waren außer dem JDK, welches nur die Basiswerkzeuge wie Compiler und Debugger für die Kommandozeile zur Verfügung stellt, kaum Werkzeuge verfügbar. Symantec brachte dann mit Café die erste Umgebung auf den Markt, die Java unterstützte. Danach folgten rasch weitere Werkzeuge von anderen Herstellern, und die Auswahl wuchs. Jedoch wirken selbst jetzt, Mitte 1998, manche der für Java verfügbaren Werkzeuge noch unreif. Dies äußert sich dann beispielsweise in Abstürzen der Entwicklungsumgebung, Problemen beim Debuggen, Schwierigkeiten der Parser, die aus dem Sourcecode Browsing-Information zu extrahieren versuchen, oder unvollständigem Erkennen von Abhängigkeiten beim Übersetzen eines Projekts. Ohne es gemessen zu haben sind wir jedoch der Ansicht, daß diese Probleme durch die ansonsten sehr hohe Produktivität von Java mehr als aufgewogen werden. Außerdem verspricht die Zukunft stabilere und vollständigere Unterstützung der Sprache durch Werkzeuge.

Es ist auch zu erwähnen, daß der Java-Markt wie kein anderer mit kleinen, nützlichen Utilities und wiederverwendbaren Komponenten gefüllt ist, die in vielen Bereichen der Entwicklung hilfreich sein können. Beispielsweise enthält das Gamelan-Archiv

(http://www.developer.com/directories/pages/dir.java.html) über 10.000 Java-Komponenten (Stand Juli 1998), die als Free- oder Shareware ladbar sind und zumeist in eigenen Anwendungen wiederverwendet werden können. An dieser Stelle findet eine interessante Synergie statt: Durch die gute Verfügbarkeit von wiederverwendbaren Komponenten entscheiden sich immer mehr Entwickler für Java. Da sich die Sprache durch die klaren Konzepte gut für die Erstellung solcher Komponenten eignet, entstehen somit wiederum neue Komponenten, die im Gegenzug Java wieder attraktiver machen.

Portierung

Anwendungen müssen im allgemeinen an die jeweilige Plattform, auf der sie ausgeführt werden sollen, angepaßt werden. Das gilt sowohl für verschiedene Hardwareplattformen wie für Betriebssysteme oder andere Softwareplattformen wie etwa Compiler, Bibliotheken o.Ä. Umstellungen zwischen verschiedenen Compilern können beispielsweise erforderlich werden, da die Unterstützung des „Sprachstandards“ verschieden ist[4]. Anpassungen an Bibliotheken müssen z.B. dann erfolgen, wenn eine andere Version einer Bibliothek oder eine völlig andere Bibliothek in einer Anwendung zum Einsatz kommen soll.

Dabei bereiten die verschiedenen Anpassungen unterschiedlich viel Aufwand. Erfahrungsgemäß sehr aufwendig waren bislang die Portierung von Anwendungen, die Grafikausgabe verwenden. Auch die Portierung des nicht-grafischen funktionalen Kerns einer Anwendung von einem Betriebssystem auf ein anderes bereitet bei vielen Sprachen Mühe, da oft direkte Aufrufe in das Betriebssystem möglich sind (z.B. die sogenannten *system-calls* in C/C++ auf Unix), die dann entsprechend anzupassen sind.

Java bildet eine gute Abtraktion über die verschiedenen Hardware- und Betriebssystemplattformen, so daß direkte Systemaufrufe entfallen und selbst grafische Ausgaben plattformunabhängig formuliert werden können. Auf der anderen Seite sind zur Zeit sowohl der Sprachstandard als auch die zur Sprache gehörenden JDK-Bibliotheken noch starken Änderungen unterworfen. So ändern sich z.B. Namen von Bibliothekspaketen, die Vorgehensweise für die Behandlung von GUI-Eingaben wurde komplett überarbeitet und Teile der I/O-Bibliotheken wurden bezüglich des Unicode-Standards angepaßt. Außerdem änderte sich zwischen Version 1.0.2 und Version 1.1 des JDKs die Sprachspezifikation (siehe auch Abschnitt 0) teilweise inkompatibel. Viele dieser Änderungen stellen vernünftige Redesigns und gute Erweiterungen dar. Sie haben jedoch die Auswirkung, daß trotz der erleichterten Cross-Plattform-Portierung zur Zeit noch die „Portierung“ zwischen verschiedenen Java-Versionen einigen Aufwand verursacht.

In diesem Kontext ist auch die Lauffähigkeit von Java-Applets in verschiedenen Browsern zu bewerten. Neben der Tatsache, daß die in den Browsern integrierten Java-Versionen dem aktuellen Standard stets um einige Versionen nachhinken, fällt hier störend auf, daß die Browser-Versionen nicht einmal vollständig kompatibel zu dem ausgewiesenen Java-Standard und dadurch auch nicht untereinander kompatibel sind. Beispiele hierfür sind das Fehlen der RMI-Unterstützung in Microsoft Internet Explorer oder das proprietäre *Permissions*-Management in Netscape.

Diese Unterschiede in den verschiedenen Browsern lösen einen Trend zu Standalone-Java-Anwendungen aus, die wiederum den Markt für die Installationstools (wie z.B. *InstallShield* oder *InstallAnywhere*) florieren lassen und die Auslieferung von Applikationen generell schwieriger machen.

Nur vor diesem Hintergrund ist dann auch die Anmerkung gerechtfertigt, daß Java auf mehreren Plattformen getestet werden muß, was etliche Entwickler als einen Nachteil von Java anführen. Jede Anwendung muß, nachdem sie portiert wurde, getestet werden, unabhängig von der verwendeten Programmiersprache. Java vereinfacht allerdings die Portabilitätstests, da gewisse Basisfunktionalitäten der Sprache sehr verläßlich und konsistent auf die verschiedenen Plattformen abgebildet wurden.

[4] In C++ könnte das etwa die Unterstützung von Templates oder Exception Handling sein.

Auslieferung

Wie jedes andere ausführbare Programm ist natürlich auch ein Java-Programm über das Internet oder ein Intranet auslieferbar. Ein wesentlicher Unterschied ergibt sich jedoch beim Aktualisieren einer Installation, so daß sie einer neuen Version entspricht. Hier bietet Java durch die feine Granularität der ausgelieferten Dateien einen Vorteil.

Während die durchschnittliche Größe einer ausführbaren Datei oder einer dynamisch nachladbaren Bibliothek (DLL) in der Regel mehrere hundert Kilobyte bis zu einigen Megabyte beträgt, liegt die durchschnittliche Größe einer *.class*-Datei, dem ausführbaren Bytecode einer Java-Klasse, bei einigen wenigen Kilobyte. Wird eine Anwendung basierend auf den geänderten class-Files aktualisiert, dann ist das in der Regel günstiger als die vollständige Anwendung zu übertragen.

Etwas schwieriger wird dieses Update-Verfahren, falls das seit JDK 1.1 verfügbare Archivformat *JAR* (Java Archive) zur Bündelung der auszuliefernden Files verwendet wird. Einzelne Dateien in einem solchen Archiv auszutauschen ist grundsätzlich möglich. Zu berücksichtigen ist dabei jedoch eine eventuell vorhandene Signatur, die das Archiv vor Verfälschung schützt und seine Herkunft erkennen läßt. Diese Signatur muß, falls vorhanden, bei allen Veränderungen am Archiv entsprechend aktualisiert werden. Auf die dabei zu beachtenden Besonderheiten soll aber an dieser Stelle nicht näher eingegangen werden.

1.3.2 Nützliche Workarounds für identifizierte Java-Schwächen

Bevor wir den Entschluß gefaßt hatten, statt eines Applets eine Standalone-Anwendung auszuliefern, hatten wir das Problem, aus einem Applet heraus auf das Dateisystem des Client-Rechners zuzugreifen. Durch die unterschiedlichen Wege, wie die Browser Signaturen auf Applets behandeln, sahen wir zu dieser Zeit als einzigen funktionierenden Workaround die Imlementierung einer *native DLL* an.

Dabei handelt es sich um eine für die entsprechende Zielplattform übersetzten Bibliothek, welche die Teile der Anwendung implementiert, die ein Applet im Browser nicht ausführen dürfte, in unserem Fall also Routinen zum Dateizugriff. Diese Bibliothek wird dann vom Browser in dem Moment geladen, wo das Applet sie anfordert.

Dieses Vorgehen erfordert jedoch, daß die DLL lokal beim Client installiert ist und läßt damit eine Auslieferung als ein Applet fragwürdig erscheinen. Der konsequente und richtige Weg wäre ein *Signed Applet* gewesen, aber es kam seinerzeit aus genannten Gründen nicht in Frage.

Einige andere Probleme stellten wir in der Klasse *java.awt. GridBagLayout* fest, und zwar besonders beim Versuch, größenveränderliche Layouts zu programmieren. Eine Abhilfe für Elemente, die stets mit Größe 0 „angezeigt“ wurden, war das Setzen eines der zugehörigen *weight*-Constraints auf einen positiven, aber sehr kleinen Wert, z.B. 0.00001.

Ein Effizienz-bezogenes Problem mußten wir im Zusammenhang mit der Klasse *java.awt.List* feststellen. Bei jedem Einfügen eines neuen Elementes in eine solche Liste berechnet die Klasse die Breite sämtlicher Listeneinträge erneut unter Verwendung der jeweiligen Fontgrößen. Das heißt, der Aufwand für das Einfügen eines Elementes steigt linear mit der Anzahl der bereits in der Liste enthaltenen Elemente, so daß das Einfügen von n Elementen den Aufwand $O(n^2)$ hat. Steigt die Anzahl der Einträge auf mehr als einige hundert an, so dauern die Einfügevorgänge auf einem 200MHz-Pentium mehrere Sekunden.

Abhilfe schaffte an dieser Stelle der Einsatz einer der zahlreichen GUI-Komponenten, die über das Internet als Share- oder Freeware zu beziehen sind. In diesem Fall haben wir uns für eine Komponente von *Taligent* entschieden, die mehrspaltige Listen mit veränderlichen Spaltenbreiten und verschiedensten Sortierkriterien anzeigen kann. Neben diesen Annehmlichkeiten offeriert diese Komponente konstanten Aufwand beim Einfügen von Elementen, wodurch das ursprüngliche Problem behoben werden konnte.

1.3.3 Was haben wir gelernt?

Projektmanagement

Wir haben den Aufwand, um aus dem entwickelten Prototypen ein Produkt zu machen, erheblich falsch eingeschätzt. Trotz entsprechender Mahnungen in der Literatur und durch Berater wurde geschätzt, daß die Produktisierung des Prototypen nur etwa den doppelten Aufwand im Vergleich zur Erstellung des Prototypen benötigen würde. Tatsächlich war der erste demonstrationsfähige Prototyp bereits nach vier Monaten fertiggestellt, wohingegen die breite Auslieferung des Produktes erst 15 Monate später erfolgte.

Diese Diskrepanz wurde noch erhöht durch Probleme im Zusammenhang mit sich verändernden Anforderungen. Bedingt durch die neue unbekannte Technologie und die dadurch erst möglich gewordenen Funktionalitäten hatten die Kunden Schwierigkeiten, frühzeitig einen vollständigen Anforderungskatalog aufzustellen. An dieser Stelle haben wir den Fehler begangen, zu spät die Phase einzuleiten, wo die Anforderungen dann tatsächlich festgeschrieben werden müssen. Dadurch haben sich noch sehr spät im Entwicklungsprozeß unnötig viele Iterationen eingeschlichen, die durch die notwendigen Regressions- und Integrationstests den Gesamtaufwand unverhältnismäßig erhöht haben.

Es fehlte auch ein konsequentes Change-Request-Manangement. Stattdessen wurden informell Änderungswünsche und Fehler per EMail oder Lotus Notes gemeldet und von den Entwicklern auf eine Liste offener Punkte gestellt. Nach Erledigung einer Anfrage wurde der Punkt an das Ende der Liste verschoben. Wenn eine Version der Anwendung ausgeliefert wurde, dann wurde die Versionsnummer an das Ende dieser „to-do"-Liste angehängt, so daß zumindest erkennbar war, welche Änderungen zwischen welchen Versionen stattfanden. Als entscheidender Nachteil eines fehlenden Change-Request-Managements stellte sich auch heraus, daß wir nur unzureichend in der Lage waren, Aufwände auf einzelne Anfragen abzubilden. Auch haben sich durch den informellen Prozeß Inkonsistenzen beim Einbringen neuer Anfragen in die Liste offener Punkte ergeben.

Es hat sich auch gezeigt, daß der fachliche Prozeß, der durch die entwickelte Anwendung unterstützt werden sollte, nur unzureichend dokumentiert und standardisiert war. Durch die notwendig gewordenen Rückfragen und Klärungsprozesse verzögerte sich die Entwicklung weiter.

Wir haben festgestellt, daß es außerordentlich wichtig ist, bei den verschiedenen an der Entwicklung beteiligten Partnern (Kunde, Benutzer, Prozeßspezialisten) feste Ansprechpartner zu haben, die ein Budget für die Unterstützung der Entwickler zur Verfügung haben. Ist dies nicht der Fall, schieben sich verschiedene Mitarbeiter die Verantwortlichkeiten gegenseitig zu, so daß dem Entwicklungsteam letztendlich gar keine Unterstützung zu Teil wird.

Im Falle unseres Projekts wurde die Situation zusätzlich dadurch erschwert, daß die Benutzer (die Verpackerfirmen) nicht gleichzeitig unsere Kunden waren und außerdem keinen direkten finanziellen Vorteil durch den Einsatz der neuen Anwendung hatten. Entsprechend schwierig war es, Mitarbeiter von Verpackerfirmen zu motivieren, dem Entwicklungsteam zur Seite zu stehen. Gleichzeitig aber erscheint und die direkte Einbeziehung des Benutzers als wesentliche Voraussetzung für die Benutzbarkeit und damit den Erfolg einer solchen Anwendung.

Anwendung von Java

Wir haben uns bei der Entwicklung eng an den Java-Standard gehalten und auf die sogenannte *Pureness* geachtet. Das heißt, wir haben konsequent keine plattformabhängigen Elemente wie das Verzeichnistrennzeichen oder Bildschirmauflösungen fest in die Anwendung codiert, sondern portable Konstrukte wie *File.separatorChar* und das Konzept der *LayoutManager* verwendet. Die Belohnung dafür war, daß die Anwendung nun auch mühelos unter OS/2, Solaris und Linux einsetzbar ist.

Ein Fehler, den wir anfänglich — mehr aus Bequemlichkeit — gemacht haben, ist das Abfangen zu allgemeiner Ausnahmen (*catch (Exception e)*). Dadurch wird es schwieriger, Fehler zu lokalisieren, insbesondere, wenn nicht einmal ein Hinweis in die Fehlerausgabe generiert wird, wo diese Ausnahmen aufgetreten ist. Nachdem wir dies eingesehen hatten, haben wir konsequent nur die Ausnahmen abgefangen, die auch tatsächlich auftreten können. Außerdem geben wir in jedem

Fall, in dem die aufgetretene Ausnahme wirklich einen Fehler darstellt, eine entsprechende Meldung auf die Fehlerausgabe aus. Unserer Erfahrung nach hat sich dadurch das Finden und Beheben von Fehlern deutlich beschleunigt.

Die Auslieferung eines Applets in einem Browser kann Probleme mit sich bringen (siehe dazu auch Abschnitt 0). Besondere Schwierigkeiten bereiten die Inkompatibilitäten der Java-Implementierungen der verschiedenen Browser und das sehr restriktive Sicherheitskonzept. Die Verwendung signierter Applets würde hier Abhilfe schaffen, jedoch sind die Signaturen zur Zeit immer noch nicht zwischen den verschiedenen Browsern kompatibel.

Benutzung von Bibliotheken

Wir mußten feststellen, daß im Laufe der Entwicklung viele Komponenten, die wir anfangs mühsam zusammentragen und in unsere Auslieferungen integrieren mußten, mittlerweile seitens Sun in das JDK integriert wurden. Beispiele hierfür sind die mächtigen *Java Foundation Classes* (JFC) im Bereich der grafischen Benutzungsoberfläche, das *Mail API* zum Versenden von EMail oder die Integration von CORBA durch *JavaIDL*.

Die Auslieferung einer Anwendung wird erheblich vereinfacht, wenn sie direkt und ausschließlich auf dem Umfang des JDKs beruht. Dies erspart den Aufwand der Installation und des Aktualisierens zusätzlicher Komponenten auf den Clients.

Es hat sich daher als sinnvoll erwiesen, im Rahmen der geschätzten Entwicklungsdauer die zukünftige Weiterentwicklung des JDKs zu antizipieren und dadurch nicht unnötig externe Komponenten einzubringen oder gar selbst zu entwickeln. Diese Vorgehensweise wird auch dadurch gerechtfertigt, daß durch die *Java Plug-In* Technologie und die gute Verfügbarkeit des *Java Runtime Environments* für verschiedene Plattformen stets die aktuelle JDK-Version beim Benutzer verfügbar gemacht werden kann. Wäre das nicht der Fall, würde das Setzen auf neueste JDK-Eigenschaften aus Kompatibilitätsbetrachtungen eher ein Risiko darstellen.

Einsatz in gewachsenen Umgebungen

Java ließ sich sehr gut mit den bereits vorhandenen Systemen integrieren. Die Anbindung an relationale Datenbanken stellt mittels JDBC (*Java Database Connectivity*) kein Problem dar. Auch bieten immer mehr Hersteller objektorientierter Datenbanken Java-Anbindungen an.

Eine Integration in CORBA-basierte Umgebungen, bei denen sich die existierenden Anwendungen als CORBA-Objekte präsentieren, wird — auch wenn es in unserem Projekt bisher nicht aufgetreten ist — durch *JavaIDL* unterstützt.

1.4 Zusammenfassung, Ausblick in die Zukunft

Als Fortsetzung des Projekts ist nach der erfolgreichen Auslieferung von APGM die Entwicklung einer Anwendung für die Kraftwerksbaustellen, basierend auf der gleichen Technologie, geplant. Es herrscht eine positive Einstellung gegenüber der Java-Technologie im ABB Konzern. Nicht nur im Kraftwerksbereich tauchen daher immer mehr Neuentwicklungen auf, bei denen Java als Entwicklungssprache verwendet wird.

Durch die vielen Werkzeuge im Java- und Internet-Bereich wird der Wunsch nach einer besseren Integration dieser Tools stärker. Das scheint jedoch auch an den Herstellern zu scheitern, da für diese kein Anreiz existiert, einen Standard in diesem Bereich zu schaffen oder nach der eventuellen Schaffung eines Standards diesem zu folgen. So muß der Entwickler auch heute noch oft zwischen Designtool, Versionierungstool, Codingtool, Projektverwaltungswerkzeug und Auslieferungshilfe hin und her wechseln, wobei nicht immer alle Daten reibungslos zwischen den verschiedenen Werkzeugen übernommen werden.

Die Java-Plattform muß auch das Tempo drosseln, mit dem Veränderungen eingebracht werden. Entwickler wie Kunden werden durch die raschen Versionswechsel oft mehr irritiert, als daß wirklich wesentliche Verbesserungen ihr Leben erleichtern würden. Sun hat dies bereits erkannt und erklärt, daß mit der JDK-Version 1.2 dieses gesunde Maß an Stabilität erreicht werden soll.

Ein interessanter Meilenstein für sämtliche Online-Technologien dürfte der Ausgang der juristischen Debatten zum Thema Verschlüsselungstechniken werden. Sollte ein Verbot von Verschlüsselung ergehen, ähnlich wie in Frankreich passiert, dann wird das einen Einfluß auf den Einsatz von Web und Java besonders in empfindlichen Geschäftsfeldern wie im Bank- und Versicherungswesen, aber auch in anderen zentralen Bereichen wie Logistik und Transport haben. Es bleibt zu hoffen, daß die Politik hier wirtschaftlich vernünftige Entscheidungen trifft.

1.5 Kurzvorstellung der Autoren

Horst Lichter: Informatik-Studium in Kaiserslautern. Wissenschaftlicher Mitarbeiter an der ETH Zürich und an der Universität Stuttgart, 1993 Promotion. Zwei Jahre Mitarbeiter bei der SBG Zürich; drei Jahre Mitarbeiter des ABB Forschungszentrums Heidelberg. Seit 1998 Professor für Informatik und Leiter der Lehr- und Forschungsgruppe Informatik III der RWTH Aachen.

Axel Uhl erhielt 1995 ein Diplom im Fach Informatik von der Universität Karlsruhe, wobei er sich in den Bereichen der Parallelverarbeitung, Objektorientierung und kognitiven Systemen vertiefte. Seitdem arbeitet er im Forschungszentrum der ABB in Heidelberg, wo er der Gruppe *Softwareengineering* angehört. Seine Aufgabenschwerpunkte dort liegen in der Untersuchung von verteilter Objekttechnologie und deren Anwendung auf Geschäftsprozesse. Seit Ende 1996 leitet er die technische Entwicklung von WWLS.

1.6 Literatur

Lichter, H., M. Schneider-Hufschmidt, H. Züllighoven (1994): Prototyping in Industrial Software Projects - Bridging the Gap Between Theory and Practice. IEEE Trans. on SE, vol. 20, no. 11, pp 825 - 832.

Gamma, E. et al. (1995): Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley.

Beck, K. (1994): Simple Smalltalk Testing The Smalltalk Report, Vol. 4, No. 2, October, pp 16-18.