1 FRISCO OEF – Dokument-basiertes Editor-Framework

1.1 FRISCO Open Editor Framework

FRISCO ist ein Werkzeugentwicklungsprojekt, aufgelegt im Rahmen des Forschungsprojektes SYSLAB am Lehrstuhl Prof. Broy der Technischen Universität München, um eine Reihe von Zielen zu erreichen:

- Aufbau von Know-how im Bereich Java,
- Ausbau und Anwendung von Know-how im Bereich objektorientierter Modellierung, speziell UML [UML 97b], und
- Entwicklung eines an UML orientierten Werkzeugs zur Unterstützung von objektorientierter Softwareentwicklung mit der Zielsprache Java [Breu et al. 97].

Das vorgestellte FRISCO Open Editor Framework (kurz: OEF) ist ein Teil des Gesamtprojekts FRISCO, das einen Dokument-orientierten Ansatz zur Modellierung und Implementierung objektorientierter Softwaresysteme bietet. Es erlaubt neben der Bearbeitung von Diagrammen auch den Umgang mit textuellen Teilen, wie Java Klassen- und Interface-Definitionen, Methodenrümpfen, oder Vor- und Nachbedingungen.

Die Realisierung des Open Editor Framework wird weitgehend im Rahmen von Fortgeschrittenenpraktika, Systementwicklungsprojekten sowie mit Unterstützung durch studentische Hilfskräfte vorangetrieben. Ziel ist es, eine im wesentlichen einheitliche Plattform für die Entwicklung verschiedener Arten von Editoren zu schaffen. Vornehmlich wird dabei an die Entwicklung von CASE-Tools gedacht, die den Entwickler beim Entwurf von Softwaresystemen unterstützen und ihm später auch bei der Implementierung und beim Test zur Seite stehen. Bei der Implementierung von FRISCO konnte auf Erfahrungen aus dem im Sonderforschungsbereich 342 ebenfalls am Lehrstuhl Broy entwickelten CASE-

Werkzeug AUTOFOCUS [Huber et al. 96] zurückgegriffen werden, dem ersten größeren Java-basierten Entwicklungsprojekt am Lehrstuhl. Die theoretische Fundierung der in FRISCO realisierten Konzepte wird im Projekt SYSLAB durchgeführt [Broy et al. 97]. FRISCO bildet damit ein Werkzeug, das neue Konzepte der Softwareentwicklung, wie zum Beispiel einen Verfeinerungskalkül für Zustandsdiagramme [Rumpe 96], praktisch umsetzt. Das FRISCO OEF ist ein selbständiger Teil des Projekts, der zur einheitlichen Gestaltung der graphischen Oberfläche dient.

1.1.1 Fachlichkeit und Funktionalität

OEF ist in Anlehnung an OpenDoc von Apple [Apple 96] konzipiert worden und dient als Framework, um spezialisierte Teildokumente zu einem Gesamtdokument zu kombinieren. Dahinter steckt die Idee, daß auch in der Softwareentwicklung graphische und textuelle Anteile nebeneinander existieren, die verschiedene Aspekte eines Systems beschreiben und die nach Möglichkeit gleichberechtigt in zusammenhängender Form angezeigt und bearbeitet werden sollten. OEF bietet hierzu die Möglichkeit, textuelle, tabellarische und graphische Dokumentteile übersichtlich zu Gesamtdokumenten zu kombinieren. Abbildung 1 enthält einen Screenshot mit einem Klassendiagramm, einem erklärenden Text und einer Methodentabelle für die erwähnten Klassen. Daraus ergibt sich, daß die OEF-Architektur zweigeteilt ist. Sie besteht aus dem OEF-Kern und den Editoren, die für spezifische Diagramm- und Textarten verwendet werden.

Um eine flexible Kombination von Beschreibungstechniken zuzulassen, besteht ein Dokument aus mehreren, ggf. sehr unterschiedlichen *Parts*. Das Dokument, mit welchem das System bzw. eine Systemkomponente beschrieben wird, ist dabei nicht mehr nur einfacher Text, sondern besteht aus Grafiken, Tabellen, Animationen etc.

Für eine flexible Erweiterbarkeit des CASE-Werkzeugs um neue Dokumentarten wurde eine spezielle Komponentenarchitektur entworfen, nach der an der Darstellung eines solchen Dokuments immer mehrere sog. *Parthandler* beteiligt sind. Jeder Parthandler ist auf die Darstellung und ggf. auf das Editieren eines bestimmten Parts spezialisiert. Die Palette an Parts reicht dabei von einfachen Texten und Bildern über komplexe Graphiken bis hin zu Codegeneratoren und graphischen Simulationen. Das OEF stellt in zweifacher Hinsicht eine gemeinsame Plattform für solche Softwarekomponenten zur Verfügung:

Zum einen werden zahlreiche gemeinsame Basisklassen bereitgestellt, die bei der Editorentwicklung immer wieder benötigt werden.

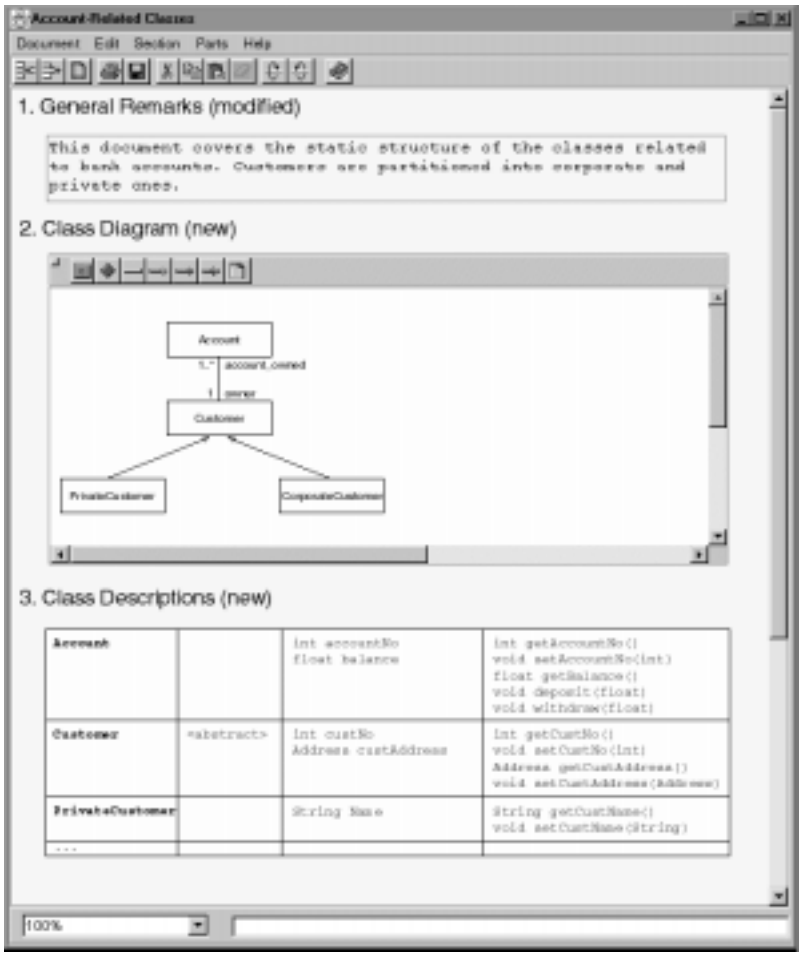

Abbildung 1. Screenshot eines FRISCO-Editors

Zum anderen beinhaltet das OEF eine Laufzeitumgebung für all die verschiedenen Parthandler. Um größtmögliche Plattformunabhängigkeit zu erreichen, ist das gesamte Projekt in Java realisiert. Die Parthandler selbst sind als Komponenten in JAR-Files gepackt und werden, sofern sie sich ein einem speziellen Verzeichnis befinden, beim Systemstart automatisch erkannt und dynamisch eingebunden. OEF muß also nicht verändert oder neu kompiliert werden, um seine Fähigkeiten zu erweitern. Das laufende System besteht also aus einem Kern und einer einzeln ladbaren Bibliothek von Plug-Ins.

Die wesentliche Aufgabe des OEF-Laufzeitsystems ist es, ein Dokument, trotz seines Aufbaus aus lauter verschiedenen Einzelteilen (Parts), als Einheit darzustellen, ein part-übergreifendes, dokumenteinheitliches Editieren zu ermöglichen sowie Interaktionen zwischen den Parts zu regeln. Hierzu gehört vor allem die Bereitstellung eines für alle Parts ge-

meinsamen Anzeigefensters, die gemeinsame Nutzung von Menü- und Werkzeugleisten, ein globales und geschachteltes Undo/Redo-Konzept, einheitliche Regelungen für die Speicherung und den Ausdruck, ein übergreifendes Management von Projekten, Dokumenten und Benutzern sowie die Integration kontextsensitiver Hilfe und ein mächtiges, global einheitliches Annotationskonzept. In der vorliegenden Version ist zusätzlich noch ein einheitliches, für jeden Part relativ leicht zu implementierendes Scripting enthalten, das einerseits als Grundlage für die Implementierung des Undo/Redo-Mechanismus dient und andererseits eine Schnittstelle zwischen den einzelnen Parthandlern sowie auch zur OEF-Umgebung darstellt.

OEF wurde entwickelt, um die Infrastruktur zur Verfügung zu stellen, die es erlaubt, flexibel verschiedene textuelle, tabellarische und graphische Dokumentteile zu kombinieren und in einer integrierten Oberfläche zu bearbeiten. Somit bietet OEF für alle Dokumentteile eine einheitliche Bearbeitung folgender Themenkomplexe:

- Laden und Speichern,
- Drucken,
- Bildschirmdarstellung,
- Menueleisten und Toolbars,
- Undo/Redo,
- Scripting und Script-Recording,
- Annotationen („Post-It-Notes“) und
- Interkomponenten-Kommunikation.

Die Dokumentteile sind hierarchisch organisiert und können, innerhalb der vom Gesamtdokument durch ein Template festgelegten Struktur, dynamisch hinzugebunden und entfernt werden. Die hierarchische Gliederung zeigt sich auch bei linearer Anzeige in der strukturierten Numerierung.

Besonders der gemeinsame Annotationsmechanismus, der es jedem Dokumentteil erlaubt, seine Elemente (z.B. ein Klassen-Icon oder eine Tabellenzelle) mit individuellen Anmerkungen zu versehen, aber eine globale Verwaltung derselben vornimmt, bietet einigen, heutige Werkzeuge deutlich übertreffenden Komfort. So sind Annotationen nicht Teil des Dokuments und können auch dann geändert werden, wenn das Dokument selbst schreibgeschützt ist. Annotationen können auch vom Werkzeug genutzt werden, um beispielsweise Syntaxfehler zu markieren.

Da die verschiedenen Dokumentteile untereinander komplexe Abhängigkeiten eingehen können, wird ein allgemeiner Mechanismus zur

Verfügung gestellt, der einzelnen Dokumentteilen erlaubt, andere über stattgefundene Änderungen zu informieren, oder sogar Kontrolle über andere Teile zu gewinnen.

Der Mechanismus des Scripting erlaubt, Änderungen nicht nur per Mausklick durchzuführen, sondern auch als Script einzugeben. Dadurch können etwa häufig wiederkehrende Tätigkeiten in Makros abgelegt werden und so wesentlich schneller ausgeführt werden. Mit Hilfe des Script-Recording kann eine Folge von stattgefundenen Tätigkeiten einfach aufgezeichnet und für eine spätere Wiederverwendung gespeichert werden.

Derzeit stehen 14 verschiedene Dokumentteile zur Verfügung, die zu Dokumenten verschiedener Arten zusammengebunden werden. Darin enthalten sind auch Dokumentteile, die externe Editoren, wie z.B. Emacs und Word benutzen. Um das OEF um neue Komponenten zu erweitern, werden diese vor dem Systemstart lediglich in einem bestimmten Verzeichnis abgelegt. Eine Modifikation von OEF selbst ist nicht erforderlich.

Beim Start des OEF wird zunächst eine hierarchische Übersicht über alle Projekte, Dokumente und Parts angeboten. Der Benutzer kann seine Projekte verwalten, neue anlegen, alte löschen oder umbenennen. Ebenso kann er auch neue Dokumente aus vordefinierten Templates erzeugen und diese mit Parts in vorgegebener Weise anreichern. Die einzelnen Editoren haben die Möglichkeit, kontextspezifische Menüs und Werkzeugleisten im Dokumentfenster anzuzeigen.

Als graphisches Frontend eines CASE-Werkzeugs dient OEF zur Visualisierung von Modelldaten aus einem Repository. OEF bietet eine 2-tier Client/Server-Architektur zwischen der zentralen Repository-Datenbank und den Clients, die sowohl technische Berechnungen, wie zum Beispiel Überprüfung von Kontextbedingungen, als auch die graphische Darstellung durchführen.

1.1.2 Projektmanagement und -verlauf

Das Projekt wird an einer Universität durchgeführt und dient daher natürlich auch der Ausbildung von Studenten. Es findet deshalb unter starker Beteiligung von Studentinnen und Studenten statt, die das Design und teilweise auch das Projektmanagement mit unterstützen. In einer ersten Phase (ein Jahr) wurde mit dem Aufwand von etwa 21 Personenmonaten, die sich ungleichmäßig auf sieben Personen verteilten, die grundlegende Architektur entwickelt und mehrere Dokumente, bestehend aus einer Reihe Dokumentteilen, geschaffen. In der zweiten Phase wird diese Architektur unter Hinzuziehung neuer Studenten verfeinert und es werden eine Reihe weiterer Editoren entwickelt und zu Dokumenten kombiniert.

Ziel ist die Fertigstellung eines gut lauffähigen und einsatzerprobten Systems bis Mitte 1999. Außerdem wird derzeit die zunächst Datei-basierte Speicherung der Dokumente auf eine über CORBA angebundene Datenbank umgestellt. Aufgrund der hohen Fluktuation der Studenten hängen Universitätsprojekte immer wieder stark davon ab, rechtzeitig neue Interessenten zu finden, die Grundkenntnisse in Java mitbringen und den stetig steigenden Aufwand zur Einarbeitung in FRISCO nicht scheuen. Durch die seit 1996 bei uns bereits im Grundstudium durchgeführte Java-Ausbildung existiert jedoch mittlerweile ein großes Potential an ausreichend vorgebildeten und interessierten Studenten. Durch die hohe Fluktuation der Studenten und den Ausbildungsauftrag der Universität entstehende besondere Situation spiegelt sich natürlich in der Projektplanung wider. So muß das Gesamtprojekt in kleine, in sich weitgehend abgeschlossene Projektteile mit einem Bearbeitungsaufwand von je ca. drei bis sechs Personenmonaten zerlegt werden. Als Entwicklungsmethode kann daher für ein Gesamtprojekt nur das evolutionäre Spiral-Modell [Boehm 94] in Frage kommen, wobei in jedem Zyklus die Mitarbeiter die Analyse und Teile des Entwurfs, die Studenten den Entwurf und die Implementierung durchführen.

Im Gegensatz zu manchen Industrieprojekten war bei diesem Forschungsprojekt auch einiges an Neuland zu betreten. Um die komplexen und anfangs nicht ganz festlegbaren Zielvorstellungen dennoch flexibel einbauen zu können, wurde OEF in einem ersten Durchgang explorativ entwickelt. Anhand des entstehenden Prototypen wurden nach und nach die Anforderungen konkretisiert und die Architektur entsprechend angepaßt. Diese Technik wurde möglich, da in Java Prototypen schnell und einfach implementiert werden können. Leider wurde der Projektverlauf auch verstärkt von den Bug-Fixes und der Weiterentwicklung von der Programmiersprache Java, ihrer Werkzeuge und ihrer Bibliotheken beeinflußt. Ein einschneidendes Beispiel hierfür war etwa die Umstellung der GUI von Java AWT [Chan et al. 97] auf die Swing Klassen der JFC [JavaSoft 98]. Bug-Fixes und vorübergehende Workarounds wurden vor allem notwendig, weil auf den unterschiedlichen, von uns genutzten Plattformen die Klassenbibliotheken nicht vollständig kompatibel waren. Die Projektplanung wurde dadurch in zweierlei Hinsicht beeinflußt: Zum einen mußte mehr Zeit als beabsichtigt in die Portierung zwischen den einzelnen Entwicklungsplattformen investiert werden. Zum anderen war die Beobachtung aktueller Kommunikations-Foren in Bezug auf uns betreffende Fehler, Patches und neue Releases der Klassenbibliotheken einzuplanen.

1.1.3 Technische Architektur

Das OEF-System bietet eine klare, erweiterbare Architektur mit schmalen und abgestuften Schnittstellen zu seinen Parthandlern.

Im System ist ein eigener Class-Loader enthalten, der Klassen aus dem aktuellen CLASSPATH sowie insbesondere Parthandler-Klassen aus speziellen Java Archiven laden kann. Die Fähigkeit, auch über ein Netzwerk zu agieren, wird angestrebt. Dieses Vorgehen des dynamischen Nachladens spezifischer Parthandler erlaubt eine prinzipiell unbegrenzte Vielfalt an Parts in einem Dokument. Dadurch ist der Inhalt eines Dokuments in seiner Art fast unbeschränkt. Was die Dokumentgröße anbelangt, so wird davon ausgegangen, daß ein Dokument typischerweise 5 bis 20 Parts enthält.

Ein Projekt ist im OEF-Kontext definiert als eine Organisationseinheit, bestehend aus Dokumenten und ggf. weiteren Unterprojekten. Gemäß einem hierarchischen Benutzerkonzept sind einem Projekt auch diverse Mitglieder zugeordnet. Dadurch lassen sich die Zugriffe auf Dokumente und Projekte systematisch regeln. Bisher ist zwar noch keine Zugriffsregelung im System implementiert, es ist aber an einigen Stellen bereits an diese Anforderungen gedacht.

Zur persistenten Speicherung von Dokumenten wird derzeit das zugrundeliegende Dateisystem verwendet. Eine effektive Versionskontrolle ist derzeit noch nicht implementiert. Als weitere Features sind in der Systemarchitektur ein Undo-Mechanismus sowie ein Scripting-Konzept enthalten und funktionsfähig implementiert. Um diese Features konsequent nutzen zu können, müssen die Parthandler die entsprechende Unterstützung implementieren.

Interaktionen der Parts untereinander arbeiten über sog. *Connections*. Über diese ist es möglich, unmittelbar Methoden eines anderen Parthandlers, der dann ein entsprechendes Interface bereitstellen muß, aufzurufen. Der zweite, prinzipiell mächtigere Weg, mit seiner Außenwelt Kontakt aufzunehmen, besteht darin, sich einfach an den Skript-Interpreter des Systems zu wenden. Dadurch ist es sogar möglich, mit Parts außerhalb des eigenen Dokuments zu interagieren oder auch neue Dokumente anzulegen.

Eine Besonderheit an OEF ist der für alle Dokumentteile einheitliche Annotationsmechanismus. Mit ihm werden Anmerkungen der Anwender verwaltet. Anmerkungen zu beliebigen annotierbaren Elementen des Parts werden vom OEF und dem Parthandler gemeinsam erstellt. Anmerkungen, die sich auf den ganzen Part beziehen, können unabhängig vom Parthandler erstellt werden. Über einen Schlüssel werden die Annotationen dann auf die zugehörigen Objekte abgebildet, wobei mehrere Anmerkun-

gen zum selben Objekt möglich sind. Die Annotationen werden durch das System editiert und verwaltet.

Eine direkte Ausgabe der Dokumente beispielsweise auf einen Drukker ist nicht vorgesehen. Statt dessen generiert OEF LATEX-Code, der unter Nutzung externer Programme in PostScript umgewandelt und an ein entsprechendes Ausgabegerät geschickt werden kann. Eingebettete Graphiken werden als EPS-Dateien gespeichert. Auf diese Weise werden dem System anspruchsvolle Layoutaufgaben erspart und der Anwender hat bis zu Schluß die volle Kontrolle über das Druckergebnis, das er durch geeignete Modifikation der LATEX-Quellen weiter bearbeiten kann. Voraussetzung hierfür ist allerdings eine meist nicht gerade kompakte Installation von TEX und LATEX, die jedoch für alle Plattformen durchführbar ist.

OEF bietet eine komponentenbasierte Entwicklungsumgebung für den Entwurf von Softwaresystemen. Intern werden Komponenten durch sauber definierte Schnittstellen voneinander getrennt. Für die präzise Definition der Schnittstellen, über einzelne Objekt-Interfaces hinaus, wurde eine eigene Beschreibungstechnik entwickelt [Huber et al. 98], die zusätzlich zu bestehenden UML-Notationen genutzt wird, um die technische Architektur zu beschreiben. In Abbildung 2 ist die Schnittstelle eines OEF-Parthandlers als Component Interface Diagram (CID) dargestellt, die Kästen bedeuten Klassen, die Pfeile sind für den Client nutzbare Navigationspfade. Letztere sind weder Aufrufbeziehungen, noch Assoziationen, können aber durch beides u. a. implementiert werden.

Entwickelt wird in Pure Java. Es werden keinen impliziten Annahmen über das eingesetzte Betriebssystem gemacht. Das Projekt wird auf drei verschiedenen Plattformen (Linux, Solaris, Windows32) entwickelt.

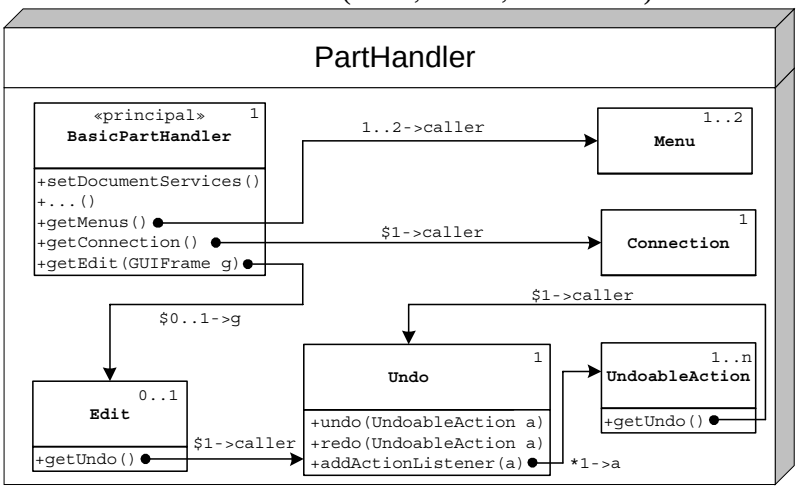

Abbildung 2 Vereinfachtes Component Interface Diagram (CID) für OEF Parthandler

Die Komponentenarchitektur spiegelt sich auf Ebene der eingebundenen Dokumentteile wieder, die jeweils in einzelne JAR-Files verpackt werden und auf klassische Weise mit Hilfe der Java Reflection-Mechanismen eingebunden werden. Die Interfaces genügen den Namenskonventionen der JavaBeans-Spezifikation [Sun 97] und erlauben so eine weitere Standardisierung der internen Struktur. Das System bedient sich der neuesten Java Foundation Classes und befindet sich damit auf dem Stand der Technik. Für das Repository ist eine Datenbankanbindung zu einer objektorientierten Datenbank (Versant) unter Benutzung von CORBA (OrbixWeb) derzeit in Arbeit.

Das Open Editor Framework setzt sich im wesentlichen aus den in Abbildung 3 skizzierten Komponenten zusammen, die im folgenden genauer beschrieben sind.

Core-System

Das Core-System ist hauptsächlich für das Laden und Binden der verschiedenen Parthandler (Editor-Komponenten) zuständig. Die Klassendefinitionen für die Parthandler werden zur Zeit in einem einzigen Class-Loader verwaltet. Dieser ist auch für das Laden von weiteren Ressourcen, die zu den Klassen gehören (z.B. Icons, ...) zuständig. Der verwendete Class-Loader sucht die benötigten Klassen zuerst im eingestellten CLASSPATH und dann in allen Java-Archiven, die sich in einem speziellen, durch Properties festgelegten, Verzeichnis befinden. Die Archive der Komponenten werden dabei zuerst alle eingelesen und die Klassen dann erst im eigentlichen Class-Loader definiert. Auf diese Weise wird ein Sharing von gemeinsamen Klassen verschiedener Dokumentteile ermöglicht. Dies gilt insbesondere für Datenstrukturen, auf die evtl. verschiedene Parthandler zugreifen müssen.

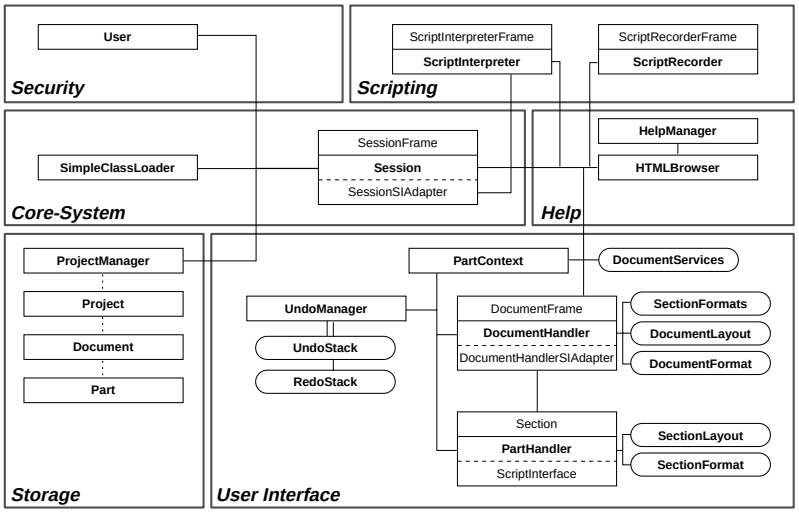

Abbildung 3. Die OEF-Systemarchitektur

Es obliegt dem Core-System, in Zusammenarbeit mit der Storage-Komponente, Projektdaten zu beschaffen und angemessen darzustellen. Baumartig werden in einem eigenen Fenster die verschiedenen, einem Projekt zugeordneten Dokumente, in ihrer Abhängigkeit angezeigt. Von hier aus läßt sich ein vorhandenes Dokument öffnen oder ein neues erstellen. Das neue Dokument ist entweder vollkommen leer oder leitet sich von einer Dokumentvorlage, einem sogenannten Template ab.

Storage

In der aktuellen Implementierung werden Projekte und Dokumente zunächst auf das zugrundeliegende Datei-System abgebildet. Projekte und Dokumente werden dabei durch Verzeichnisse dargestellt, die einzelnen Parts eines Dokuments entsprechen Dateien im Dokumentenverzeichnis. Zusätzlich wird im Dokumentenverzeichnis noch eine weitere Datei angelegt, die die Struktur des Dokuments (Formate, Anordnung und Verknüpfungen der Parts) enthält. Gemäß den Anforderungen ist auf diese Weise eine beliebige Projekthierarchie denkbar: Projekte können weitere Projekte und Dokumente enthalten. Dokumente enthalten Parts und Anmerkungen. Wie bereits erwähnt, ist derzeit eine Erweiterung in Arbeit, die einen Datenbank-Anschluß unter Benutzung von CORBA-Technologie für das OEF realisiert. Mangels bereits heute zur Verfügung stehender ausgereifter Produkte hat sich herausgestellt, daß hier mehr Arbeit zu investieren ist als ursprünglich geplant.

Security

Das Security Subsystem verwaltet Daten über Benutzer und Gruppen, Paßwörter, Informationen für Verschlüsselungsverfahren (Private und Public Keys, Certificates) und regelt den Zugriff auf diese Daten sowie den Zugriff auf das System an sich. Weiterhin ist auch daran gedacht, eine geeignete ACL (Access Control List) zu implementieren, mit der ganze Projekte, Dokumente oder nur einzelne Parts versehen und somit vor unberechtigtem Zugriff geschützt werden können.

User Interface

Die Komponente User Interface stellt den bisher umfangreichsten Teil des Projekts dar. Hier liegt die eigentliche Verantwortung, ein Dokument auf dem Bildschirm anzuzeigen und die Interaktion mit dem Systembenutzer durchzuführen. Jedes Dokument wird in genau einem Fenster angezeigt.

Jedes Dokument hat neben dem Titel, auch einen Namen. Dieser wird u.a. für das Scripting genutzt. Weiterhin besteht ein Dokument aus einer Reihe von Sektionen, die ihrerseits wieder genau einen Part enthalten. Zur Formatierung des Dokuments dient eine Datenstruktur, die Angaben über die Seitengröße, Seitenränder und die Formatierung der Überschriften

enthält. Die Sektionen selbst werden sequentiell untereinander innerhalb der seitlichen Seitenränder angeordnet. Bei der Anzeige werden die Seitenränder oben und unten ignoriert, so dass sich ein Endlosformular ähnlich der Normalansicht in MS WinWord ergibt. Das Format, mit dem die Sektionen formatiert werden können, hat Ähnlichkeiten mit einem Absatzformat und gibt Aufschluß über einen Vor- und Nachabstand, die Überschriftenformatierung, den Abstand zwischen Überschrift und Part sowie Einzüge von rechts und links und für die Überschrift (Erstzeileneinzug, gemessen vom linken Einzug). Außerdem bezeichnet es einen Level, durch welchen die Sektionen in einer Hierarchie angeordnet werden können, die sich unter anderem in der automatischen Überschriftennumerierung niederschlägt.

Ein sehr zentrales Element der komfortablen Benutzeroberfläche ist die Undo-Funktionalität. Sie ist vollständig in das Event-Modell von JDK 1.1 integriert und arbeitet nach folgendem Schema: Bei jeder Interaktion des Benutzers mit der Komponente feuert diese ein Event an alle angemeldeten Listener, das alle notwendigen Informationen für ein Undo enthält. Der Undo-Manager des Systems verwaltet diese Events auf einem Stack. Wird ein Undo gewünscht, kann der Undo-Manager die Undo-Information an die Komponente zurückschicken, so daß diese schließlich die letzte Aktion zurücknimmt. Ein Redo ist in diesem Sinne nichts anderes als ein Undo des Undo. Prinzipiell ist es auch möglich, nebenläufige Aktionen, wie zum Beispiel eine Formatierung im Hintergrund, in den Undo-Mechanismus einzubinden, jedoch ist in diesem Fall zu sichern, daß derartige Aktionen auch nach einer Teilausführung unterbrochen und zurückgenommen werden können.

Scripting

Der Skriptingmechanismus des OEF-Systems basiert auf dem Reflection API des JDK 1.1. Um eine Komponente dem Scripting zugänglich zu machen, ist lediglich ein sehr schmales Java-Interface zu implementieren und zur Verfügung zu stellen. Mit Hilfe des Java Reflection API werden textuelle Befehle des Scripts als Java-Methodenaufrufe an die Parthandler weitergegeben. Dabei werden auch einfache Parameterübergabemechanismen, wie z.B. Strings, unterstützt. Damit ist der Scriptinterpreter unabhängig von den aktuell existenten Parts. Jedes ScriptInterface hat einen eindeutigen Namen und ist entsprechend der visuellen Anzeige der Parts in einer Hierarchie geordnet. Durch entsprechende Scriptbefehle kann der Fokus zwischen den Parts verschoben werden, um so effiziente Scripts zu erstellen.

Help

Die Online-Hilfe zum System liegt, wie heute verstärkt üblich, in Form von HTML-Dateien vor. Für die Anzeige wird derzeit eine Komponente aus der Swing-Bibliothek verwendet.

Technische Einbettung

OEF ist in 100%-pure Java entwickelt und besitzt mit Ausnahmen des Aufrufs externer Editoren und der Ausgabe von LaTeX-Quellcode keine Anbindung an weitere Systeme. Aufgrund des explorativen Charakters von OEF wurden zunächst Aspekte der GUI-Programmierung näher untersucht, während erst jetzt eine Client/Server-Architektur für eine Datenbankanbindung entwickelt wird. Zum Einsatz kommen dabei Versant [Versant 98] als objektorientierte Datenbank, OrbixWeb [IONA 98] von IONA als ORB, Java Beans und die Java Foundation Classes.

1.1.4 Mengengerüste und Leistungsdaten

In der aktuellen Version besteht OEF aus (Angaben näherungsweise) 23200 Lines of Code (LoC) in 160 Dateien, zu dem insgesamt 15 Dokumentteile weitere 48600 LoC in 300 Dateien beisteuern. Ein Parser und Precompiler für eine Erweiterung von Java und ein Parser für eine OCL-ähnliche Bedingungssprache [UML 97a] liefern weitere 15000 LoC in 100 Dateien bzw. 34000 LoC in 220 Dateien. Der momentane Gesamtumfang des Projekts umfaßt also damit etwa 120800 LoC in 780 Dateien.

Geringe „Echtzeitanforderungen“ sind durch die Antwortzeit des interaktiv editierenden Benutzers definiert. Die Systemträgheit ist stark abhängig vom verwendeten Java-Interpreter und dem Betriebssystem (JIT-Compiler oder nicht, Linux/Solaris/Windows32, Pentium oder Sparc, Speichergröße,...) und läßt derzeit noch stark zu wünschen übrig. Dieses Problem ist weitgehend auf den Einsatz mächtiger und komfortabler Bibliotheken zurückzuführen. Insbesondere führt die bevorstehende Einführung des Java 2D API mit dem JDK 1.2 zu einer beträchtlichen Verlangsamung der Verarbeitung.

1.2 Der Entwicklungsprozeß, die Methodik

Die eingesetzte Entwicklungsumgebung bestand im wesentlichen aus der klassischen Kombination von XEmacs als Editor und dem Basis-Entwicklerwerkzeug JDK. Teilweise wurde auch Sniff+ verwendet. Obwohl UML als Modellierungstechnik während des Designs für die Be-

schreibung essentieller Teile eingesetzt wurde, wurden keine Analyse- und Design-Werkzeuge verwendet. Zur Dokumentation diente JavaDoc, und zum Reengineering (Aufbereiten vorhandener Implementierung) „Rational Rose“. Spezielle Debugging-Werkzeuge wurden nicht verwendet. Zur Koordination der Programmierarbeiten mehrerer gleichzeitig am Projekt arbeitender Entwickler und zur Versionsverwaltung kam CVS („Concurrent Versions System“) zum Einsatz. Die Nicht-Verwendung weiterer Werkzeuge gründet sich zu einem guten Teil darin, daß zum Zeitpunkt des Starts und auch der Durchführung die von uns evaluierten Werkzeuge keinen ausreichenden Reifegrad besaßen und praktisch nie mit der aktuellen Version der von uns benutzten Klassenbibliotheken zusammenarbeiteten. Diese Situation ist mittlerweile erheblich besser geworden.

OEF wurde weitgehend explorativ entwickelt. Ausgehend von einem ersten Anforderungskatalog wurde ein Prototyp erstellt, der eine Präzisierung und Erweiterung der Anforderungen erlaubte. Java und speziell seine Klassenbibliotheken haben diese Vorgehensweise massiv unterstützt. Natürlich waren Programmbeispiele von Sun (im JDK, BDK, oder Swing) eine große Orientierungshilfe. In weiteren Entwicklungszyklen werden nun OEF-Parthandler entwickelt, und das OEF auf eine Datenbankanbindung umgestellt. Das OEF selbst dürfte sich jedoch als weitgehend stabil erweisen, so daß hier keine weiteren Entwicklungszyklen zu erwarten sind.

1.3 Kritische Bewertung des Einsatzes von Java

Java als Ganzes besteht aus vielen unterschiedlichen Aspekten, die sicherlich eine unabhängige Bewertung verdienen. Wir bewerten deshalb die zwei Bereiche: die Programmiersprache und die Programmierumgebung, bestehend aus Werkzeugen, Klassenbibliotheken u.ä.m.

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

Zunächst wird begründet, warum die Sprache Java als Implementierungssprache ausgewählt wurde. Die nachfolgenden Gründe sind teilweise aus der Literatur bekannt, und haben sich im Projekt auch bewahrheitet:

- Die Plattformunabhängigkeit („Write once, run anywhere“) ist für ein Universitätsprojekt, dessen Ziel einerseits eine gewisse Verbreitung des Ergebnisses ist, und das andererseits eine historisch gewachsene sehr heterogene Rechnerlandschaft besitzt, sehr wichtig.

- Java bietet ein solides, gut definiertes und beschriebenes Sprachkonzept, das (speziell nach den Erfahrungen mit C++) als leicht zu erlernen gilt. Die Sprache Java bietet einerseits viel Komfort, da für alle wesentlichen Probleme ein Konzept zur Darstellung existiert, andererseits aber auch eine gewisse Benutzerfreundlichkeit, da sich im Gegensatz zu C++ nicht jeweils sehr viele verschiedene Konzept-Varianten anbieten.
- Die in den Java-Bibliotheken als Standards zur Verfügung stehenden Klassen zur Unterstützung von abstrakten Datentypen wie Strings und den Container-Klassen sind für eine kompakte und transparente Realisierung sehr hilfreich.
- Kurze Turnaround-Zeiten im Entwicklungprozeß durch Wegfall der Link-Phase erlauben eine schnellere Entwicklung.
- Konzepte wie eine komfortable Speicherverwaltung (Garbage Collector), ein starkes aber dennoch komfortables Typsystem und eine in die Sprache integrierte Fehlerbehandlung bieten weitere Vorzüge von Java.
- Mittlerweile gibt es eine umfangreiche, laufend erweiterte Auswahl an Klassenbibliotheken, die für viele Anwendungsgebiete ausgereifte Lösungen zur Verfügung stellt. Unter den nicht kommerziellen Klassenbibliotheken spielen die JFC (Java Foundation Classes) [JavaSoft 98], die letztlich in das JDK 1.2 eingehen sollen, eine wesentliche Rolle. Der GUI-Anteil (Swing) ist bereits jetzt in einem separaten Archiv verfügbar.
- Das Konzept der Class-Loader in Java macht das Erstellen einer eigenen Komponentenarchitektur fast zum Kinderspiel.
- In diesem Zusammenhang ist auch das Reflection-API, mit dem das Public Interface einer fremden Klasse gelesen werden kann, von großer Bedeutung. Es erwies sich in vielen Fällen als sehr hilfreich, Datentypen wie Objekte zu handhaben. Der gesamte Scripting-Mechanismus in OEF basiert auf dieser Technologie. Des weiteren ist natürlich die Komponentenarchitektur stark davon abhängig. Dennoch empfehlen wir nur sehr maßvollen und begrenzen Einsatz dieser Technik, da Reflection leicht zu unerwarteten Resultaten führt.
- Nicht zuletzt ist Java eine zukunftsorientierte Technologie, die gerade deshalb eine gewisse Beliebtheit bei Entwicklern im universitären Bereich sowie auch in Firmen erreicht hat und daher eine besondere Motivation hervorruft.

Wie bei jedem an sich guten, aber in seiner Gesamtheit komplexen Produkt gibt es auch einige negative Punkte über Java zu nennen. So macht uns insbesondere die Performance, die noch fehlende Stabilität der Bibliotheken, die – sich derzeit stark verbessernde – Unausgereiftheit der Werkzeuge Sorgen. Während die Sprache wirklich als gelungen bezeichnet werden kann und sicherlich in den nächsten Jahren eine starke Dominanz erreichen wird, ist davon auszugehen, daß Bibliotheken und Werkzeuge noch einige Reifezyklen benötigen werden.

Da in OEF im wesentlichen mit fremden Klassenbibliotheken, nicht aber kommerziellen Werkzeugen, gearbeitet wurde, wurden folgende weitere Schlußfolgerungen gezogen:

- Java und die diversen Klassenbibliotheken (besonders JFC) waren und sind z.T. noch sehr in Veränderung begriffen. Die große Frage lautet oft: Abwarten, ob bald eine passende Komponente erscheint, oder selbst entwickeln? Gerade was die JFC angeht, hat sich das Warten gelohnt. Wer hat schon soviel Zeit, sich einen Tree-View oder eine Tabelle selber zu bauen.
- Es ist ratsam, aufmerksam und regelmäßig die Java-Homepage und die dortigen Ankündigungen zu beachten. Dieser zusätzliche Aufwand kann sich sehr schnell amortisieren.
- Die Entscheidung für oder gegen eine Klassenbibliothek ist sehr schwer zu fällen. Zur Zeit scheint es am besten, den Produkten von Sun wesentliche Priorität einzuräumen. Dadurch ist eine gewisse Kompatibilität gesichert. Andererseits geben andere, insbesondere auch kommerzielle Produkte explizit ihre Integrierbarkeit mit Sun-Produkten an. Gerade auch duch die Etablierung der JavaBeans-Spezifikation ist in naher Zukunft mit zahlreichen Komponenten von Drittanbietern zu rechnen.
- Zum Zeitpunkt der Implementierungsphase hinkten die Entwicklungsumgebungen für Java dem Stand der Technik oft unerträglich hinterher. Das original JDK von Sun ist nicht gerade schnell und komfortabel. Native-Code-Compiler beherrschten oft nicht den gesamten Sprachumfang von Java 1.1. Die meisten kommerziellen Produkte (Symantec Café, Borland Jbuilder) waren beispielsweise erst knapp 9 Monate nach dem ersten Final Release des JDK 1.1 in der Lage, mit Inner Classes umzugehen.
- Die Programmentwicklung mit Java setzt einen leistungsfähigen Rechner sowie viel Speicher voraus, was allerdings für jegliche Entwicklertätigkeit zutrifft.

Die Verwendbarkeit von CORBA-Produkten und die Anbindung an Datenbanken ist derzeit noch in Erprobung. Hinreichend gesicherte Kommentare lassen sich dazu noch nicht abgeben. Ein erster Versuch hat jedoch ergeben, daß äußerst sorgfältig auf die Performanz geachtet werden muß.

Nicht zuletzt aufgrund von detaillierten Untersuchungen zahlreicher objektorientierter Sprachen, wie C++, Objective-C, Eiffel, Sather, Smalltalk, Oberon, Modula-3, Self, Beta und Simula, und der Entwicklung einer eigenen Variante einer objektorientierten Sprache [Rumpe 95], war es eines der Anliegen des OEF-Projekts, die Sprache Java auf Sprachkonzepte und praktische Anwendbarkeit genau zu evaluieren. Eine detaillierte Evaluierung würde hier allerdings den Rahmen sprengen, so daß wir uns auf wesentliche Punkte beschränken wollen.

Den Eindruck einer hervorragenden soliden Sprache, den Java bei ersten kleinen Testentwicklungen gemacht hat, hat sich voll und ganz bestätigt. Java bietet eine konsolidierte Menge an sprachtechnischen Konzepten an, die einerseits für jede Problematik eine Lösung anbieten, andererseits aber z.B. im Gegensatz zu C++ keine unüberschaubare Lösungsvielfalt erlauben. So bleiben Java-Programme wesentlich lesbarer als es C++-Programme sind. Syntaktisch eher mit C++ verwandt, ist jedoch geistig eine starke Verwandtschaft mit Smalltalk zu erkennen. Konzepte wie ein Reflection-API, Garbage Collection, ein sehr flexibles Konzept zur Behandlung von Fehlern, automatische Prüfung von Arraygrenzen und ein starkes Typsystem helfen einerseits bei der Vermeidung von Programmierfehlern und andererseits beim robusten Umgang mit diesen, und tragen so massiv zur Steigerung der Produktivität der Entwickler und zur Qualität des Produkts bei.

Java ist als Sprache deutlich komplexer als Smalltalk, aber auch wesentlich einfacher als C++. Durch eine intensive Prüfung von Kontextbedingungen, etwa dem komplexen Typcheck, der Prüfung, ob Subklassen-Konstruktoren jeweils Superklassen-Konstruktoren aufrufen oder alle Variablen vor Benutzung besetzt werden, wird dem Benutzer weiterer Komfort angeboten.

Die weitgehende Definition von Klassenbibliotheken für Datentypen, graphische Oberflächen, Internet-Programmierung, etc. innerhalb der Sprachdefinition stellt ein deutliches Plus dar.

Nachteilig erscheint nur, daß das Thread-Konzept noch etwas besser sein könnte. Deadlock-sicheres Programmieren erfordert eine sehr sorgfältige Vorgehensweise. Der Einsatz von Klassenbibliotheken, die vielleicht Thread-sicher, aber deswegen keineswegs Deadlock-frei sind, ist hier, auch wegen der schlechten Reproduzierbarkeit von Fehlern, mit Vorsicht zu genießen.

Zusammenfassend läßt sich sagen, daß die Sprache Java einen Reifegrad erlangt hat, den die Programmierumgebungen erst noch erreichen müssen. Eine Anwendbarkeit in großen, insbesondere geschäftskritischen Projekten ist deshalb noch mit Vorsicht zu genießen. Wir gehen davon aus, daß die Wartung von Java-Programmen schon alleine wegen der besseren Lesbarkeit (kleinere Sprachkonzept-Auswahl) einfacher als bei C++ ist. Darüber hinaus ist davon auszugehen, daß die weiter anwachsende Anzahl von Programmier- und CASE-Werkzeugen auch die Wartung bzw. das bei Weiterentwicklung oft stattfindende Reengineering gut unterstützen wird.

1.3.2 Nützliche Workarounds für identifizierte Java-Schwächen

Die oben erwähnten Schwächen der benutzen Java Programmierumgebung und Bibliotheken werden mit hoher Wahrscheinlichkeit sehr schnell behoben sein, so daß sich eine ausführliche Beschreibung der benutzten Workarounds erübrigt. Dennoch ist eine Klassifikation der Workarounds interessant:

- Defizite der Sprache konnten keine nennenswerten gefunden werden.
- Unterschiedliches Verhalten auf verschiedenen Plattformen hat dazu geführt, daß wir nur die – allerdings große – Schnittmenge der Klassen und Methoden aus Bibliotheken verwendet haben, die auf allen Plattformen gleich funktioniert.
- Fehlende Funktionalität von Klassenbibliotheken wurde durch eigene Klassen implementiert. Häufiger wurden diese Implementierungen bei Upgrades auf neuere Versionen durch die dann vorhandene Funktionaltät ersetzt.

Zwei spezielle Lösungen seien hier genannt:

- In Ermangelung bereits existenter CORBA-Anbindung wurde in OEF zunächst eine einfache Datei-basierte Speicherlösung verwendet, aber auf eine Austauschbarkeit geachtet.
- Die Verwendung eines systemweiten ClassLoaders führt dazu, daß die Klassen der verschiedenen Parthandler sich alle im gleichen Namensraum befinden. Namenskonflikte müssen also auf jeden Fall vermieden werden. Deshalb sind die Klassen jedes Parthandlers in sein eigenes Package zu stecken und so Namenskonfikte auf der Ebene der notwendigerweise eindeutigen Package-Namen aufzulösen. Klassen, die von mehreren Parthandlern genutzt werden, müssen wieder in eigene Subpackages gesteckt und separat geladen werden.

1.3.3 Was haben wir gelernt? (Lessons learned)

1. Die Verwendung von Java hat auf das Projektmanagement keinen prinzipiellen Einfluß. Einen gewissen Einfluß haben jedoch die ständig verbesserten Werkzeuge und die sich u.a. auf Web-Servern wie Gamelan [Gamelan 98] langsam entwickelnden Bibliotheken von Java-Klassen. Sie erlauben es dem Projektmanagement, mit höherer Produktivität und Produktqualität zu rechnen. Andererseits ist die Suche nach brauchbaren Komponenten als feste Größe in den Projektplan einzubauen.
2. Auf die Analyse und zumindest die frühe Entwurfsphase darf die Auswahl der Programmiersprache keinen Einfluß haben. Zwar hat in OEF die Möglichkeit des dynamischen Nachladens bei Java bereits sehr früh zur Definition einer entsprechenden Architektur geführt, jedoch wäre diese Architektur in ähnlicher Form auch entstanden, wenn in C++ oder einer anderen Sprache ohne die Möglichkeit dynamischen Ladens entwickelt worden wäre.
3. Der Einsatz von Java wird bis zur Ankunft von Programmiersprachen, die die Definition einer Komponenten- oder Software-Architektur durch sprachliche Mittel direkter unterstützt, sicherlich noch stark anwachsen. Bei Neuentwicklungen mit nicht allzu großen Performance-Anforderungen ist Java mit Sicherheit C++ vorzuziehen. Smalltalk wird weiterhin eine Alternative bleiben. Es ist aber damit rechnen, daß die für Smalltalk bereits ausgreiften Werkzeuge bald durch bessere Java-Werkzeuge übertroffen werden.
4. Die Verfügbarkeit standardisierter Bibliotheken ist für eine Sprache heute mindestens so wichtig, wie die gute Sprachdefinition selbst. Sun hat bei Java hier von Anfang an mir der Verfügbarkeit elementarer Basisbibliotheken (Containerklassen, AWT u.ä.) einen wichtigen Grundstock gelegt und damit auch dem Wildwuchs an Bibliotheken, der beispielsweise bei C++ anzutreffen ist, vorgebeugt. Die JFC stellen eine konsequente Fortführung dieser Strategie dar. Weitergehende Funktionalität, insbesondere anwendungs- und fachspezifische Klassenbibliotheken, werden derzeit von verschiedenen Seiten entwickelt. Leider wird es wohl noch etwas Zeit in Anspruch nehmen, bis diese Bibliotheken bzw. Frameworks einen ausreichenden Reifegrad erreicht haben, aber es ist davon auszugehen, daß hier in absehbarer Zeit gute Lösungen angeboten werden.
5. Nicht zuletzt werden Projekte von Menschen gemacht. Java ist eine leicht zu erlernende Sprache und bietet sowohl dem C++- als auch dem Smalltalk-Programmierer viele ihm bekannte Konzepte, die eine leichte Migration erlauben. Java ist in dieser Hinsicht eine sehr integrative Sprache.

1.4 Zusammenfassung, Ausblick in die Zukunft

1.4.1 Ausblick in die Zukunft des OEF:

Benutzerkonfiguration

Für den späteren Einsatz ist es unumgänglich, Möglichkeiten zu schaffen, die es einem Benutzer im Rahmen seiner Rechte erlauben, seine OEF-Umgebung zu konfigurieren. Außerdem müssen auch sinnvolle Rechte und ACLs (Access Control Lists) erarbeitet werden, die eine restriktive Unterscheidung von Benutzern zulassen. Eventuell ist auch eine Arbeit mit dem System in verschiedenen Modi (Anfänger, Experte, o.ä.) denkbar. Wichtig wird in diesem Zusammenhang auch ein Werkzeug zur Benutzerverwaltung, mit dem neue Benutzer eingerichtet und administriert werden können. Dieses muß u.a. auch die Möglichkeit bieten, Private und Public Keys sowie Certificates zu im- und exportieren.

Weiterentwicklung der Speicherfähigkeiten

In der aktuellen Implementierung der Speicherung werden Projekte und Dokumente einfach auf Verzeichnisse des zugrundeliegenden Dateisystems abgebildet, die Parts auf konkrete Dateien. Wie bereits mehrfach angesprochen, ist eine CORBA-basierte Datenbankanbindung in Arbeit. Dabei muß auch an eine komfortable Versionsverwaltung gedacht werden.

1.4.2 Allgemeiner Ausblick

Java ist eine faszinierende Sprache. Sie bietet neben einer wohlüberlegten Konsolidierung verschiedener Sprachkonzepte auch gute Ansätze im Hinblick auf Plattformunabhängigkeit und Internet-Anbindung.

Deshalb ist es nur zu verständlich, daß eine große Zahl von Entwicklern derzeit Java-Technologien an allen Fronten vorantreibt. Java wird deshalb ihren Kindertagen sehr schnell entspringen und zu einer ausgereiften Technologie heranwachsen, die die Möglichkeiten der heute zur Verfügung stehenden Sprachen beträchtlich überbieten wird.

Es ist deshalb für jede Softwarefirma ratsam, sich zumindest über die Möglichkeiten mit Java auf dem Laufenden zu halten und erste Gehversuche an für Geschäftsziele zunächst unkritischen Projekten mit Java zu starten, um an dem Tag, an dem Java kommerziell für große Systeme einsetzbar wird, vorne dabei zu sein. Nach unserer Einschätzung wird das schon sehr bald sein.

1.5 Kurzvorstellung der Autoren

Franz Huber studierte an der Technischen Universität München Informatik und Wirtschaftswissenschaften. Seit 1995 arbeitet er am Lehrstuhl von Prof. Broy auf dem Gebiet der Softwareentwicklungswerkzeuge. In diesem Rahmen gilt sein besonderes Interesse der Integration formaler und pragmatischer Entwurfstechniken sowie der methodischen Unterstützung von Entwicklern. Er leitet das Projekt „AUTOFOCUS“ zur Entwicklung eines CASE-Werkzeugprototyps für den komponentenbasierten Entwurf verteilter bzw. eingebetteter Systeme, das diese Ziele vorrangig verfolgt. Er ist Co-Autor verschiedener Konferenzbeiträge zu diesem Thema, die u.a. bei der FTRTFT'96, der FME'97, der PDSE'98 und den GI-Fachgesprächen FBT '96-'98 erschienen sind.

Olav Rabe absolviert derzeit sein Studium an der Fakultät für Informatik der Technischen Universität München. Durch zahlreiche Werkstudententätigkeiten bei Siemens verfügt er über umfangreiche praktische Projekterfahrung insbesondere im Bereich objektorientierter Systeme. Seine Kenntnisse der Komponentenarchitektur OpenDoc sowie Erfahrungen im Umgang mit unterschiedlichsten Entwicklungsplattformen und Betriebssystemen waren für das FRISCO-Projekt von großer Bedeutung.

Bernhard Rumpe hat an der Technischen Universität München Mathematik und Informatik studiert. Er arbeitet in München in dem von der Deutschen Forschungsgemeinschaft (DFG) und Siemens finanzierten Projekt SYSLAB an der Fundierung objektorientierter Modellierungstechniken und Programmiersprachen. In seiner Doktorarbeit hat er u.a. einen Ansatz zur dokumentorientierten Softwareentwicklung vorgestellt und anhand von Klassen- und Zustandsdiagrammen genauer untersucht. Er hat zahlreiche Veröffentlichungen in diesem Gebiet, ist Organisator mehrerer Workshops, wurde zu Panels u.a. über die UML eingeladen. Seine Interessen liegen im Bereich Objektorientierung, Software Architekturen, Modellierungstechniken und Formale Methoden.

1.6 Literatur

[Apple 96] Apple Computer Inc. OpenDoc Programmer's Guide for the MacOS. Addison-Wesley, 1996

[Boehm 94] B. W. Boehm. A Spiral Model of Software Development and Enhancement. In Software Engineering Notes, 11(4), 1994

[Breu et al. 97] R. Breu, R. Grosu, F. Huber, B. Rumpe, W. Schwerin. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In J. Bosch, S. Mitchell (Hrsg.) Object-Oriented Technology, ECOOP'97 Workshop Reader. Springer Verlag, LNCS 1357, 1997

[Broy et al. 97] M. Broy, R. Breu, R. Grosu, F. Huber, B. Rumpe, W. Schwerin. SYSLAB – Projektbeschreibung. TU München, 1997

[Chan et al. 97] P. Chan, R. Lee. The Java Class Libraries: Java.Applet, Java.Awt, Java.Beans (Vol 2). Addison-Wesley, 1997

[Flanagan 97] D. Flanagan. Java in a Nutshell. 2nd Edition, O'Reilly & Associates, 1997

[Gamelan 98] Gamelan Website, http://www.gamelan.com/, 1998

[Huber et al. 96] F. Huber, B. Schätz, A. Schmidt, K. Spies. AUTOFOCUS - A Tool for Distributed Systems Specification. In B. Jonsson, J. Parrow (Hrsg.) Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems. Springer Verlag, LNCS 1135, 1996

[Huber et al. 98] F. Huber, A. Rausch, B. Rumpe. Modeling Dynamic Component Interfaces. TOOLS-USA'98, St. Barbara, CA, 1998

[IONA 98] IONA Technologies. http://www.orbix.com/, 1998

[JavaSoft 98] JavaSoft. Java Foundation Classes website, http://java.sun.com/products/jfc/, 1998

[Rumpe 95] B. Rumpe. Gofer Objekt-System – Imperativ Objektorientierte und Funktionale Programmierung in einer Sprache vereint. In: Tiziana Margaria (Hrsg.): Kolloquium Programmiersprachen und Grundlagen der Programmierung. Technischer Bericht der Universität Passau, MIP-9519,1995

[Rumpe 96] B. Rumpe. Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Disserta-

tion, Technische Universität München. Herbert Utz Verlag Wissenschaft, 1996

[Sun 97] Sun Microsystems. Java Beans. Version 1.01, Sun Microsystems, July 1997

[UML 97a] UML Group. UML Object Constraint Language Specification. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997

[UML 97b] UML Group. Unified Modeling Language. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997

[Versant 98] Versant Object Technology. http://www.versant.com/, 1998