1 Der Bibliography Shopper

Erschienen als: Bruno Essmann, Dirk Riehle und Kai-Uwe Mätzel. "Bibliography Shopper." *Erfahrungen mit Java: Projekte aus Industrie und Hochschule.* Herausgegeben von Silvano Maffeis, Fridtjof Toenniessen und Christian Zeidler. dpunkt Verlag, 1999. Seite 173-198.

1.1 Bibliography Shopper

Der Bibliography Shopper stellt eine Infrastruktur und verschiedene Werkzeuge für das Erstellen und Verwalten bibliographischer Datenbanken bereit. Zusätzlich unterstützt er das flexible Zusammenstellen und Ausgeben von Bibliographien.

Verteilte Bibliographie-Datenbanken können mit geringem Aufwand erstellt und verwaltet werden. Mittels verschiedener Strategien können Datenbankeinträge effizient gesucht und zu Literaturreferenzlisten zusammengestellt werden. Erstellte Literaturreferenzlisten können in benutzerdefinierten Stilen formatiert und in verschiedenen Speicherformaten abgelegt werden. Stile erlauben eine einzelne Referenzliste ohne zusätzlichen Aufwand entlang den Vorschriften verschiedener Publikation wie z.B. CACM, IEEE, oder dieses Buches zu verwenden. Die verschiedenen Speicherformate erlauben es gängigen Textverarbeitungsprogrammen und Desktop Publishing Systemen erstellte Literaturreferenzlisten zu importieren.

Der Bibliography Shopper entstand im Rahmen eines Informatik-Praktikums für die ETH (Eidgenössische Technische Hochschule) Zürich. Er wurde in der Software Engineering Gruppe am Ubilab, dem Informatik-Forschungslabor der UBS AG entwickelt.

1.1.1 Motivation

Autoren von Zeitschriftenartikeln und Berichten werden im Zusammenhang mit Literaturreferenzen häufig mit den folgenden Problemen konfrontiert:

- Wie können grosse Mengen von Literaturreferenzen als Bibliographien verwaltet werden?
- Wie kann man in diesen Bibliographien einzelne Einträge wiederfinden?
- In welchem Format sollen die Bibliographien abgelegt werden?
- Wie können Referenzlisten zusammengestellt und in verschiedenen Formaten ausgegeben werden?
- Können die ausgegebenen Referenzlisten in beliebigen Textverarbeitungssystemen eingesetzt werden?

Für einige dieser Probleme existieren bereits Lösungen wie z.B. das Unix-Programm `refer` (siehe [Kernighan82]) oder das Bibliographie-Referenzsystem `BibTeX` (siehe [Knuth84]). Wir haben allerdings keine unsere Anforderungen befriedigende Lösung auf dem Markt gefunden, und uns deswegen zur Eigenentwicklung entschlossen.

Der Bibliography Shopper bietet für die oben genannten Punkte einfache Lösungen. Er stellt Anwendern folgende Funktionalität zur Verfügung

- Ein einfaches System zur verteilten Ablage grosser Mengen von Referenzdaten.
- Den Import und Export beliebiger Formate anderer Bibliographie-Referenzsysteme, wie z.B. `refer` und `BibTeX`.
- Ein flexibles, benutzerdefinierbares Format für die Referenzdaten.
- Flexible Suchmechanismen für das Wiederfinden von Referenzeinträgen.
- Die Ausgabe von Referenzlisten in beliebigen benutzerdefinierbaren Layouts und Formaten.

Die folgenden Unterkapitel geben einen Einblick in die Funktionsweise des Bibliography Shoppers. Anhand von Beispielen wird gezeigt, wie einige der oben erwähnten Punkte implementiert wurden.

1.1.2 Die Bibliography Shopper Applikationen

Der Bibliography Shopper besteht aus einer Suite von drei Applikationen:

- ❑ Benutzer-Applikation
- ❑ Bibliography-Server (+ Administrator-Applikation)
- ❑ Domain Name Server

Die Benutzer-Applikation ist das Hauptwerkzeug zur Bearbeitung von Referenzlisten. Sie bietet Werkzeuge zum Suchen und zum Bearbeiten von Referenzeinträgen, zum Zusammenstellen von Referenzlisten und zur flexiblen Ausgabe von Referenzen in benutzerdefinierten Formaten.

Ein Bibliographie-Server stellt Benutzer-Applikationen ein oder mehrere Bibliographien bereit. Eine Bibliographie enthält eine Menge von Literaturreferenzen. Jeder Bibliographie-Server wird mit Hilfe einer dazugehörigen Administrator-Applikation verwaltet, was insbesondere das Anmelden des Bibliographie-Servers beim Domain Name Server umfasst.

Der Domain Name Server vermittelt zwischen Benutzer-Applikation und Bibliography-Servern. Eine Benutzer-Applikation erfragt vom Domain Name Server, welche Bibliographie-Server verfügbar sind, und wendet sich bei der Suche nach Literaturreferenzen direkt an die von einem Bibliographie-Server bereitgestellten Bibliographien.

1.1.3 Verteilung von Bibliographien

Dem Bibliography Shopper liegt eine Client/Server-Architektur zu Grunde, die jedem Benutzer die Möglichkeit bietet, einen eigenen Bibliographie-Server zu unterhalten. Eine zentrale Administration der Bibliographien ist nicht vonnöten. Dies vereinfacht die Verwaltung und Wartung von grossen Bibliographie-Sammlung.

Abb. 1 – Ein Domain Name Server

Die Flexibilität, verschiedene Bibliographie-Server zu unterhalten wird durch den Domain Name Server ermöglicht. Abb. 1 zeigt einen Domain Name Server, bei welchem verschiedene Bibliographie-Server angemeldet sind.

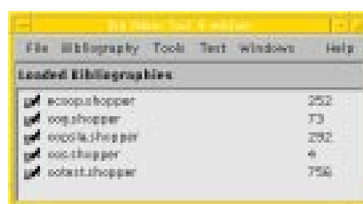


Abb. 2 – Blick auf einen Bibliographie-Server

Abb. 2 zeigt einen solchen Bibliographie-Server der Bibliographien zum Thema Objektorientierung zur Verfügung stellt.

Jede Administrator-Applikation meldet bei Programmstart ihren Bibliographie-Server bei dem ihr zugewiesenen Domain Name Server an. Alle Bibliographien, die von diesem Bibliographie-Server verwaltet werden, können dann über den zugewiesenen Domain Name Server durchsucht werden (siehe auch Kapitel 1.2.2, Client/Server-Architektur).

1.1.4 Suche und Ausgabe von Referenzen

Zur Suche nach Bibliographie-Einträgen stellt der Bibliography Shopper eine Reihe von Strategien, wie z.B. die feldweise Suche, die Suche nach Zeichenketten oder nach regulären Ausdrücken zur Verfügung. Die Benutzer des Systems können neue Suchstrategien hinzufügen.

Abb. 3 zeigt ein einfaches Suchwerkzeug in welchem feldweise nach einem Bibliographie-Eintrag gesucht werden kann. Mit den in der Abbildung angegebenen Werten werden Referenzen auf Bücher gesucht, welche als Autor „Wirth“ und im Titel das Schlagwort „Oberon“ enthalten.

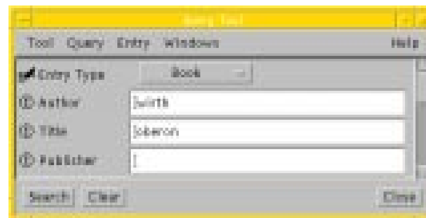


Abb. 3 – Suchwerkzeug für die feldweise Suche.

Abb. 4 zeigt das Resultat der Suche von Abb. 3. Die gefundenen Bibliographie-Referenzen wurden in Form einer Bibliographie zurückgeliefert. Die Einträge dieser Datenbank können beliebig weiterverarbeitet und in andere Datenbanken übernommen werden.

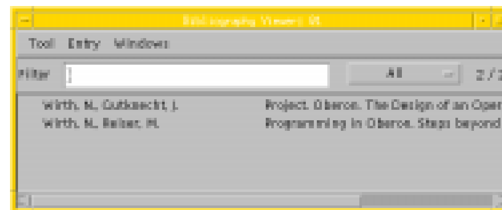


Abb. 4 – Resultat einer Suchoperation

Hat man sich in einer Bibliographie eine Liste von Bibliographie-Einträgen zusammengestellt, so kann diese als Referenzliste ausgegeben werden.

Der Bibliography Shopper stellt einen einfachen Exporter-Mechanismus für die Unterstützung von Dateiformate zur Verfügung. Exporter für ASCII-Text, HTML und RTF gehören zum Standardumfang. Zusätzliche Exporter können einfach erstellt und dynamisch dem System hinzugefügt werden.

Neben dem Dateiformat kann auch das Layout der Referenzen, also die Art und Weise wie ein Eintrag in der Referenzliste erscheint frei gewählt werden. Das Layout kann beliebige Schrifttypen und -stile enthalten. Die Exporter für die einzelnen Dateiformate versuchen die Layoutinformation möglichst genau nachzubilden.

Ähnlich dem Exporter-Mechanismus implementiert der Bibliography Shopper eine einfache Plug-in Architektur (siehe Kapitel 1.3.4) für Strategien zur Generierung von Kurzreferenzen. In der Benutzungsschnittstelle werden diese Strategien als Makros bezeichnet. Zur Zeit existieren Makros für fortlaufende Referenznummern und für Referenzkürzel welche aus den Autorennamen oder den Autorennamen und dem Erscheinungsdatum des Artikels gebildet werden.

Abb. 5 zeigt den Layout-Editor, in welchem die Layouts definiert werden können. Die Layouts können hier separat gespeichert und geladen werden, so dass Benutzer ihre Layouts untereinander austauschen können.

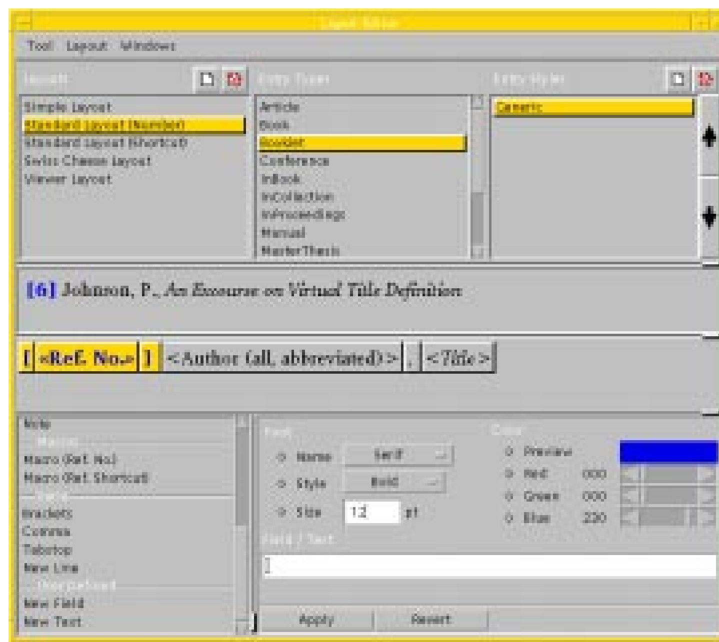


Abb. 5 – Der Layout-Editor

Abb. 6 zeigt ein Beispiel einer Referenzliste, die vom Bibliography Shopper erstellt wurde. Sie konnte ohne Anpassungen in die Textverarbeitung, welche zum Schreiben dieses Textes verwendet wurde eingesetzt werden.

[1] America, P., van der Linden, F., *A Parallel Object-Oriented Language with Inheritance and Subtyping,* in Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices

[2] Amiel, E., Dujardin, E., *Supporting Explicit Disambiguation of Multi-methods*, in Proceedings ECOOP '96

[3] Amsellem, M., *ChyPro: A Hypermedia Programming Environment for Smalltalk-80*, in Proceedings ECOOP'95

[4] Bergstein, P. L., Lieberherr, K. J., *Incremental Class Dictionary Learning and Optimization*, in Proceedings ECOOP'91

[5] Mello, P., Natali, A., *Objects as Communicating Prolog Units*, in Proceedings ECOOP'87

[6] Minsky, N., *Towards Alias-Free Pointers*

Abb. 6 – Eine Bibliography Shopper Referenzliste

1.1.5 Eingesetzte Java-Technologien

Bei der Entwicklung im Sommer/Herbst 1997 wurden die folgenden in Java 1.1 neu eingeführten Technologien verwendet:

- *Java RMI*. Die Applikationen des Bibliography Shoppers verwenden Java RMI (Remote Method Invocation, siehe [Sun96a]) zur Kommunikation untereinander.
- *Java AWT: Delegation Event Model*. Die graphische Benutzungsschnittstelle verwendet das Java AWT Delegation Event Model (siehe [Sun96b]).
- *Java AWT: Lightweight Components*. Eigene Komponenten der Benutzungsschnittstelle, wie z.B. Teile des Layout-Editors sind durchwegs als leichtgewichtige Komponenten ausgelegt (siehe [Sun96c]).

1.2 Technische Architektur

1.2.1 Kernstruktur des Bibliography Shoppers

Abb. 7 zeigt die vereinfachte Kernstruktur des Bibliography Shoppers.

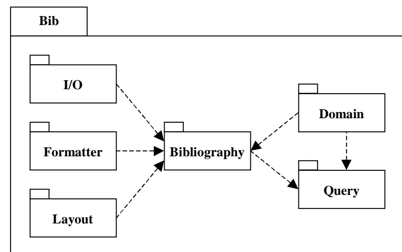


Abb. 7 – Kernstruktur des Bibliography Shoppers

Der Bibliography Shopper besteht aus den folgenden Subsystemen:

- *Bibliography*. Das Herz des Bibliography Shoppers. Hier sind die Schnittstellen und Basisklassen für Bibliographien enthalten.
- *I/O*. Enthält die Schnittstellen und Basisklassen für die Plug-ins zur Konvertierung von Bibliographie-Formaten.
- *Formatter*. Enthält die Grundklassen für die Plug-ins zur Ausgabe von Referenzlisten in verschiedenen Dateiformaten.
- *Layout*. Enthält den generischen Layouter.
- *Domain*. Enthält die Klassen des Domain Name Server.
- *Query*. Enthält die Schnittstellen und Basisklassen für die Suchwerkzeuge.

Nachfolgend werden exemplarisch die Client/Server-Architektur und die Funktionsweise des Query-Subsystems erläutert.

1.2.2 Client/Server-Architektur

Wie bereits erwähnt, bietet die Client/Server-Architektur des Bibliography Shoppers jedem Benutzer die Möglichkeit, eigene Literaturverweise mittels einem eigenen Server anderen Benutzern zur Verfügung zu stellen.

Diese Flexibilität wird durch den Domain Name Server erreicht. Die folgende Abbildung zeigt das Zusammenspiel zwischen einer Benutzer-Applikation, dem Domain Name Server und verschiedenen Bibliographie-Servern (Administrator-Applikationen).

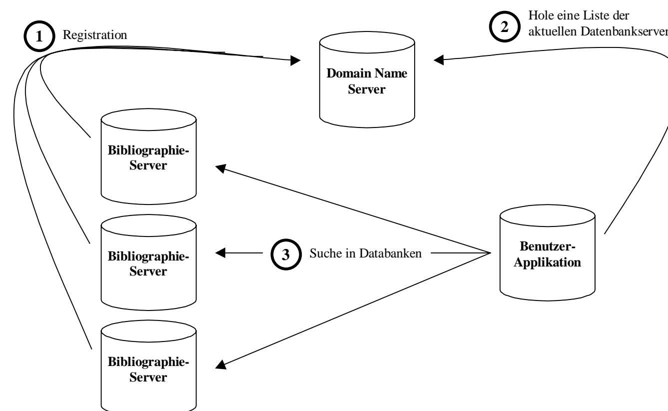


Abb. 8 – Client/Server-Architektur

Abb. 8 zeigt, dass der Domain Name Server die einzige wohlbekannte Instanz im Zusammenspiel der Server und der Klienten ist. Bibliographie-Server werden beim Hochfahren beim Domain Name Server registriert. Damit mehrere Bibliography-Server auf einem einzigen Computer gestartet werden können vergibt der Domain Name Server eine eindeutige Kennung. Ein einzelner Bibliography-Server kann also durch den Namen des Computers auf welchem er läuft und dieser Kennung immer eindeutig identifiziert werden, z.B. `//atlantis/BibDBServer3`.

Für Kleinanwendungen reicht die Verwendung eines einzelnen Domain Name Servers. Für grosse Anwendungen ist eine Strukturierung mittels mehrerer Domain Name Server sinnvoll.

Zur Kommunikation verwendet der Bibliography Shopper Java RMI (Remote Method Invocation, siehe [Sun96a]). Der Domain Name Server, sowie die einzelnen Bibliographie-Server können sich daher an einem beliebigen Ort in einem TCP/IP-Netzwerk befinden.

1.2.3 Query-Subsystem

Das Query-Subsystem kapselt die Netzwerkfunktionen der Suchwerkzeuge. Es handelt die Kommunikation mit dem Domain Name Server und den einzelnen Bibliographie-Servern, also die Punkte 2 und 3 von Abb. 8 ab.

Startet der Benutzer eine Suche, so wendet sich die Benutzer-Applikation bzw. das Suchwerkzeug mit den Suchparametern an das Query-Subsystem. Dieses handelt nun die Suche im Netzwerk komplett ab und liefert der Benutzer-Applikation die gesammelten Resultate.

Die einzelnen Schritte einer Suche werden im Ablaufdiagramm in Abb. 9 verdeutlicht. Das Query-Subsystem holt zunächst eine Liste aller beim Domain Name Server registrierten Bibliographie Server. In einem weiteren Schritt werden die Suchparameter an die einzelnen Bibliographie-Server übermittelt. Das serverlokale Suchsystem ermittelt das Resultat der Anfrage und sendet die gefundenen Bibliographie-Einträge an das Query-Subsystem. Dieses sammelt die Resultate aller Bibliographie-Server und erstellt die Bibliographie-Datenbank welche als Resultat an die Benutzer-Applikation geliefert wird.

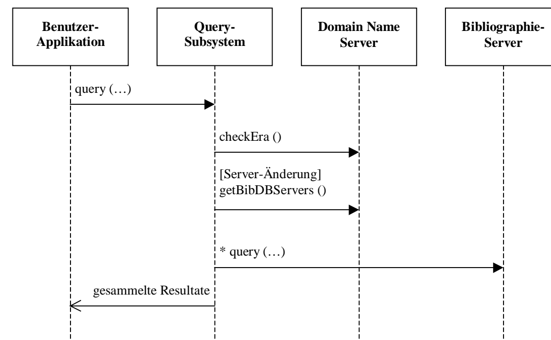


Abb. 9 – Ablaufdiagramm einer Suche

1.3 Ausgewählte Entwurfskonzepte

Bei der Entwicklung des Bibliography Shoppers wurden verschiedenen Entwurfskonzepte konsequent eingesetzt. Drei dieser Konzepte, Schnittstellen (siehe Kapitel 1.3.1), die Trennung von Werten und Objekten (siehe Kapitel 1.3.2), sowie die Anwendung der Entwurfsmuster Type Object und Product Trader (siehe Kapitel **Error! Reference source not found.**) werden nachfolgen stellvertretend beschrieben. Sie wurden ausgewählt da ihre Anwendung durch Java besonderes unterstützt bzw. im Java Umfeld besonders sinnvoll erscheint.

1.3.1 Schnittstellen

Bei der Entwicklung des Bibliography Shoppers wurde durchwegs das Java-Sprachkonzept der Schnittstellen zur Trennung von Entwurf und Implementierung verwendet (siehe [Riehle97a und Riehle97b]).

Klassen können verschiedene Schnittstellen implementieren. Ein Objekt kann, je nach Verwendungszweck unter einer anderen Schnittstelle angesprochen werden. Klienten kennen also selten die gesamte Funktionalität eines Objektes. Sie kennen nur die Schnittstelle, unter welcher sie das Objekt ansprechen. Bei konsequenten Einsatz können Systeme mit erheblich geringer Kopplung erstellt werden als dies bei der Verwendung von Klassen und sinnvollen Vererbungsstrukturen möglich wäre.

Abb. 10 zeigt ein Beispiel für eine Schnittstelle, welche aus zwei anderen Schnittstellen gebildet wurde. Objekte der Implementationsklasse `AuthorFieldImpl` können sowohl unter der Schnittstelle `Author`, als auch unter der Schnittstelle `Field` angesprochen werden.

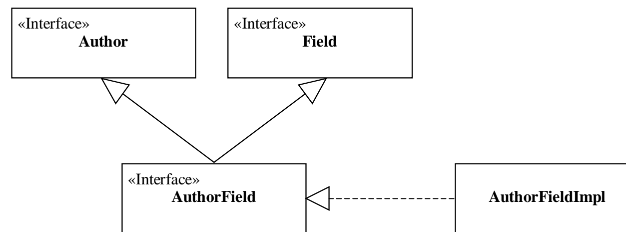


Abb. 10 – Erben von mehreren Schnittstellen

Abb. 11 zeigt, wie trotz der starken Trennung von Schnittstelle und Implementierung nicht auf die Wiederverwendung von bestehendem Code verzichtetet werden muss.

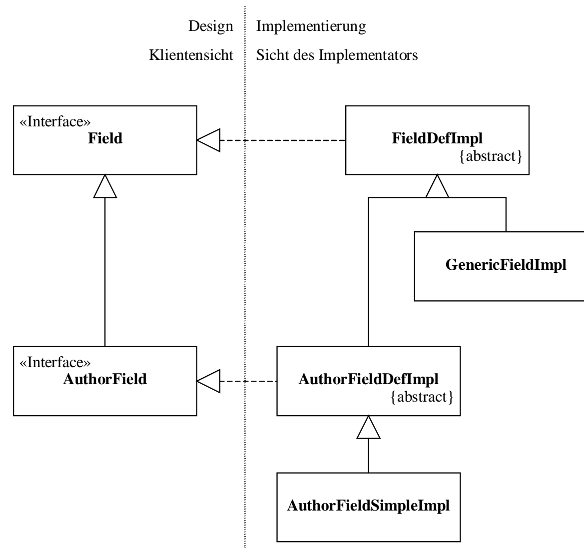


Abb. 11 – Trennung von Schnittstelle und Implementierung

Unter Verwendung oben genannter Struktur bleibt der Klientencode frei von Verweisen auf die bestehende Implementierung, während der Implementator trotzdem Code wiederverwenden kann.

Abb. 11 zeigt auch, wie sich dem Klienten ein übersichtlicher Blick auf die Schnittstellen bietet, während die Implementierung der Funktionalität beliebig komplex sein kann.

Mit der vorgestellten Struktur ist es auch möglich die Implementierung jederzeit durch eine andere zu ersetzen. Klienten bleiben von solchen Massnahmen unbetroffen, da sie nur die Schnittstellen kennen.

Verwendet man diese Struktur zur Implementierung eines Plug-in Mechanismus, so muss man darauf achten, dass man zur Objekterzeugung keine konkrete Implementierung referenziert. Kapitel 1.3.4 zeigt eine Lösung für dieses Problem mittels Product Traders.

1.3.2 Wertobjekte

Beim Kernentwurf des Bibliography Shoppers wurde der Trennung von Werten und Objekten viel Beachtung geschenkt. Obwohl in Java kein Sprachkonstrukt zur Verfügung steht, um diese Trennung zu forcieren, kann

man sie mit wenig Aufwand durch Trennung der Implementierung und der erzeugenden Instanz (z.B. mittels abstrakter Fabriken [Gamma+95], Product Traders [Bäumer+96], bzw. spezieller Creator-Objekte) erreichen.

Die Trennung von Werten und Objekten als Konzept ist nicht neu. Bereits vor fünfzehn Jahren wurden Methoden hierfür vorgestellt (siehe [MacLennan82]). Im folgenden sollen daher nur die wichtigsten Vorteile und Eigenschaften von Werten und Objekten aufgezeigt werden.

Die folgende Auflistung zeigt die wichtigsten Eigenschaften von Werten:

- Werte sind Konzepte welche eine Abstraktion für einen Problembereich modellieren.
- Werte haben keinen Lebenszyklus. Sie kennen keine Zeit, können nicht erzeugt, verändert oder gelöscht werden.
- Werte haben keinen änderbaren Zustand. Ihre Repräsentationen können nur interpretiert, nicht geändert werden.
- Durch die Verwendung von Werten entstehen keine Seiteneffekte an anderen Stellen des Systems.

Java kennt nur die üblichen Wertetypen, also ints, floats, etc. Hinzu kommt die String-Klasse, sowie die Wrapper-Klassen für die primitiven Wertetypen, wie Integer, Long, und Float.

Möchte man in Java anwendungsspezifische Wertetypen einführen, so muss man sie mittels Klassen nachbilden. Die Implementierung dieser Klassen sollte dann sicherstellen, dass ihre Instanzen Wertsemantik besitzen. Objekte mit Wertsemantik werden daher als Wertobjekte bezeichnet. Wertobjekte sind deswegen zumeist unveränderbare Objekte („immutable objects“). Somit können sie beliebig referenziert werden, ohne dass Seiteneffekte zu befürchten wären.

Da es sich bei Wertobjekten um Konstanten handelt, eignen sie sich besonders gut zur Reduzierung der Anzahl Objekte im System. Mittels dem Fliegengewicht-Entwurfsmuster und abstrakten Fabriken (siehe [Gamma+95]) oder Product Traders kann vermieden werden, dass Wertobjekte gleichen Wertes mehrfach erzeugt werden.

Beim Bibliography Shopper sind ein Grossteil der Elemente des Kernentwurfs Wertobjekte. Die Verwendung von Werten erweist sich vor allem bei verteilten Applikationen, wie dem Bibliography Shopper als grossen Vorteil.

In der aktuellen Version des Bibliography Shoppers werden auf alle Wertobjekte mittels einer abstrakten Fabrik oder über einen Product Trader zugegriffen. Auf die Reduzierung der Anzahl Objekte, wie oben beschrieben wurde jedoch verzichtet. Es wäre zwar einfach möglich, sich alle jemals erzeugten Werte zu merken und bei Bedarf zurückzuliefern. Dies würde die Anzahl der Objekte im System drastisch reduzieren (man stelle sich z.B. vor, wieviele Bibliographie-Einträge denselben Autoren haben). In der aktuellen Java-Version ist es jedoch nicht möglich, weil schwache Referenzen, d.h. Referenzen welche vom Garbage Collector nicht beachtet werden, nicht existieren. Schwache Referenzen werden jedoch benötigt, da es sonst nicht möglich ist, sich von nicht mehr benötigten Werten zu trennen. Nach einiger Zeit würde dies dann zu Speicherproblemen führen.

Ab Java Version 1.2 wird es möglich sein, schwache Referenzen zu halten (siehe [Sun98a]). Dann wird es auch möglich sein, den Speicherbedarf und die Zugriffszeit (z.B. ein Lookup in einer Hashtabelle anstelle eines Erzeugen des Objektes) auf Werte drastisch zu reduzieren.

An dieser Stelle möchte ich auch noch auf einen kleinen Nachteil von Werten hinweisen. Da Werte unveränderlich sind, muss, wenn auch nur ein kleiner Teil des Wertes anzupassen wäre, der alte Wert komplett durch einen neuen Wert ersetzt werden. Beim Bibliography Shopper bedeutet dies kein Nachteil, da beim Editieren von Einträgen der gesamte Inhalt des Eintrages in einer Eingabemaske gehalten wird. Nach Abschluss des Editierens kann mit den Daten aus der Eingabemaske einfach ein neuer Eintrag erstellt werden, welcher den ursprünglichen Eintrag ersetzt.

1.3.3 Entwurfsmuster

Um die Schnittstellen noch stärker von den Implementierungen zu trennen, wurde beim Entwurf des Bibliography Shoppers auf einige Entwurfsmuster zurückgegriffen.

Unter anderen wurden die folgenden Entwurfsmuster verwendet (siehe [Gamma+95]):

- Abstract Factory, zum erzeugen von Werten.
- Composite, z.B. bei der Definition eines Autoren.
- Command, zur Veränderung von Bibliographien.
- Observer, für die Notifikation von Veränderungen an Bibliographien.

- ❏ Strategy, für die Suchwerkzeuge.
- ❏ Type Object [Woolf96]
- ❏ Product Trader [Bäumer+96]

Die Verwendung von Type Object und Product Trader wird im folgenden Unterkapitel speziell geschildert, da diese Entwurfsmuster wenig bekannt sind und in Java elegant implementiert werden können.

1.3.4 Flexibilität durch Type Objects und Product Traders

Das Schema einer Bibliographie-Datenbank beschreibt verschiedene Eintragstypen. Für jeden Typ wird festgelegt, welche Felder ein Eintrag beinhalten muss und welche Felder nützliche Zusatzinformation wären. So existieren z.B. die Eintragstypen Bucheintrag, Dissertation, und Proceedings-Beitrag.

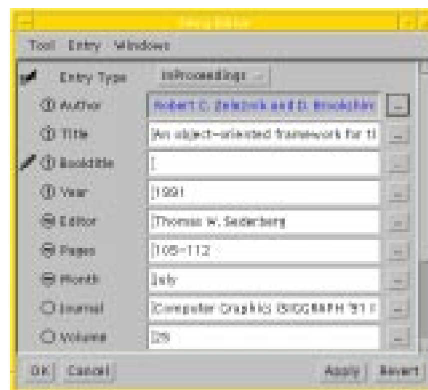
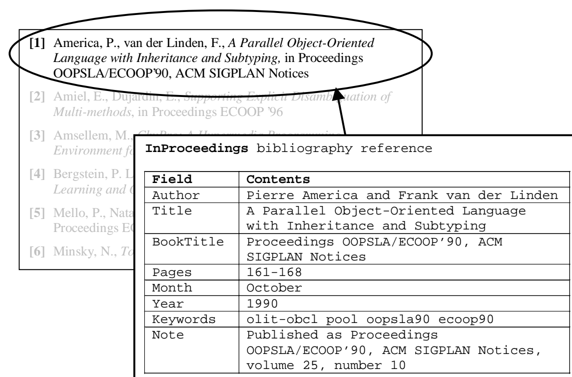


Abb. 12 – Editor für Bibliographie-Einträge

Abb. 12 zeigt den Editor für Bibliographie-Einträge. Felder, welche zwingend vorhanden sein müssen sind mit einem Ausrufungszeichen und optionale Felder mit einer Tilde versehen. Der Bibliography Shopper verwendet für strukturierte Felder spezielle Eingabemasken. So existiert z.B. für das Autor-Feld ein Feldeditor, in welchem die einzelnen Personen eines Autoren bequem eingegeben werden können.

Bei der Entwicklung des Bibliography Shoppers mussten Lösungen für die folgenden Anforderungen gefunden werden: Das System sollte einerseits ein striktes Typsystem wie oben beschrieben unterstützen, andererseits müssen die Typenmenge und jeder einzelne Typ einfach erweiterbar sein. Folgendes Beispiel verdeutlicht die Problematik:



[1] America, P., van der Linden, F., *A Parallel Object-Oriented Language with Inheritance and Subtyping*, in Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices

InProceedings bibliography reference

Field	**Contents**
Author	Pierre America and Frank van der Linden
Title	A Parallel Object-Oriented Language with Inheritance and Subtyping
BookTitle	Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices
Pages	161-168
Month	October
Year	1990
Keywords	olit-obcl pool oopsla90 ecoop90
Note	Published as Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10

Abb. 13 – Ein Bibliographie-Eintrag

Abb. 13 zeigt die Ausgangsdaten eines Bibliographie-Eintrages. Die einzelnen Felder des Eintrages enthalten die unterschiedlichsten Daten, wie z.B. eine Liste von Personen im Autor-Feld, einen Monatsnamen im Monat-Feld, etc.

Die einfachste Art diese Daten zu speichern wäre nun den Feldnamen und den -inhalt jeweils als String zu speichern. Diese Methode hat jedoch einen grossen Nachteil, alle Felder werden gleich behandelt, man könnte also den Monat eines Monats-Feldes nicht als Zahl auslesen.

Gesucht ist also eine Struktur welche es ermöglicht die Felder unter einer einheitlichen Schnittstelle anzusprechen, aber dennoch erlaubt auf die Eigenheiten eines Feldes einzugehen.

Eine Implementierung mittels Klassen kann ebenfalls nicht vorgenommen werden, da Klassen zu schwergewichtig und nicht adaptierbar sind. Ausserdem müsste man z.B. für jeden möglichen Eintragstypen eine Klasse definieren, was zu einer Explosion der Anzahl Klassen im System führen würde.

Die Lösung liegt in der Verwendung von dynamischen Typen. Zur Implementierung dieser dynamischen Typen verwendet der Bibliography Shopper das Type Object Pattern (siehe [Woolf96]) und eine Reihe von Product Traders (siehe [Bäumer+96]).

Jeder Bibliographie-Eintrag und jedes der Felder eines Eintrages verfügt über einen dynamischen Typen, der die Eigenschaften des Eintrages, bzw. Feldes beschreibt. Die Eigenschaften können vom Benutzer in einer Beschreibungsdatei festgelegt werden. Ermittelt man z.B. für einen Bibliographie-Eintrag, dass er den dynamischen Typen „Bucheintrag" besitzt, so kann man anhand der Eigenschaften des dynamischen Typen feststellen, welche Felder zwingend vorhanden sein müssen und welche optional angegeben werden können. Diese Eigenschaften kann man z.B. bei der Definition eines Ausgabelayouts verwenden um sicherzustellen, dass immer korrekte Ausgaben erfolgen.

Damit wäre geklärt, wie man spezielle Feldstrukturen in den Entwurf aufnimmt, doch wie werden nun die Felder erzeugt? Hier setzen die Product Traders an. Der Bibliography Shopper unterstützt verschiedene Importer, welche Bibliographie-Referenzen von einem anderen Format in das Format des Bibliography Shoppers überführen. Für jeden dieser Importer existiert nun ein Product Trader, der aus den Ursprungsdaten Felder im Bibliography Shopper-Format erzeugt. Sieht man sich z.B. die Struktur eines BibTeX-Eintrages an, so kann man die Funktionsweise des BibTeX-Importers leicht verstehen.

```
@inproceedings {Amer90b,
   author    = {Pierre America and Frank van der Linden},
   title     = {A Parallel Object-Oriented Language with
                Inheritance and Subtyping},
   booktitle = {Proceedings OOPSLA/ECOOP'90,
                ACM SIGPLAN Notices},
   pages     = {161-168},
   month     = oct,
   year      = {1990},
   keywords  = {olit-obcl pool oopsla90 ecoop90},
   note      = {Published as Proceedings OOPSLA/ECOOP'90,
                ACM SIGPLAN Notices, volume 25, number 10}
}
```

Abb. 14 – BibTeX-Bibliographie-Eintrag

Abb. 14 zeigt, dass man beim Lesen der einzelnen Felder eines BibTeX-Bibliographie-Eintrages zunächst den Feldtypen und dann den Inhalt des Feldes erfährt. Der BibTeX-Importer wandelt nun den Feldtypen und den Inhalt in ein Bibliography Shopper-Feld um, indem es sich an seinen Product Trader wendet und dort einen Feld-Erzeuger für Felder vom Typen „Author" erfragt. Diesem Feld-Erzeuger wird dann der Feldtyp und der Inhalt übergeben und dieser erzeugt ein Feld mit dem speziellen Type Object „Author". Abb. 15 zeigt diesen Ablauf zum Erzeugen eines Feldes.

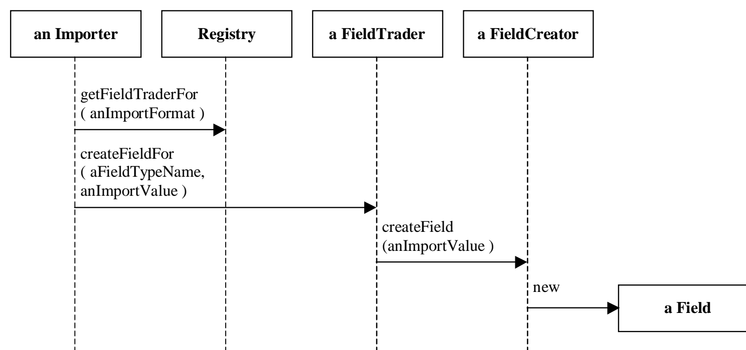


Abb. 15 – Ablaufdiagramm zur Erzeugung eines Feldes

Die gleiche Methode wird an verschiedenen Stellen des Bibliography Shoppers wie z.B. im Eintrags-Editor oder in den Suchwerkzeugen verwendet. Soll z.B. ein Bibliographie-Eintrag editiert werden, so möchte der Benutzer zur Eingabe des Autoren eine spezielle Maske erhalten. Für diese spezielle Eingabemaske wendet sich der Editor mit dem Type Object des Feldes an einen Product Trader für Eingabemasken. Dieser liefert dann eine auf den Feldtypen zugeschnittene Eingabemaske.

Bei der Vielzahl von Type Objects, Product Tradern und Objekt-Erzeugern (dynamische Instanzen) wie sie im Bibliography Shopper verwendet werden, kann es leicht passieren, dass man den Überblick verliert. Um die Beziehungen zwischen den Objekten möglichst gering zu halten und den Überblick zu wahren, wurde für alle diese Instanzen eine Registry aufgesetzt. Wird nun z.B. ein Product Trader gesucht, so wendet man sich einfach mit einer Beschreibung des Traders an die Registry und diese liefert einen passenden Trader (siehe Abb. 15 oben).

Die Registry wird beim Starten des Systems durch eine Beschreibungsdatei initialisiert. Hierzu wird das Reflection API von Java benützt (siehe [Sun96d]). Mittels `java.lang.Class.forName(aClassName)` werden die Klassen der dynamischen Instanzen geladen. Die so geladenen Klassen können dann zum Erzeugen der Instanzen selbst verwendet werden.

Durch die starke Dynamik des Systems treten jedoch auch einige Probleme auf. So ist während der Arbeit am System z.B. besonders Sorge zu tragen, dass man mit den aktuellen Versionen der dynamischen Instanzen arbeitet. Der Java Compiler kann (verständlicherweise) Beziehungen zu dynamisch geladenen Klassen nicht erkennen, so dass man die dynamischen Instanzen jeweils von Hand kompilieren, bzw. ein anderes Build-System wie z.B. „make“ verwenden muss (siehe auch Kapitel 1.5.4 Abschnitt Build Probleme). Weiter entstehen beim Debuggen Probleme wenn nicht zu Beginn des Programmes alle dynamischen Instanzen geladen werden. Der Bibliography Shopper lädt z.B. eine dynamische Instanz erst dann in den Speicher, wenn sie von der Registry das erste Mal angesprochen wird. Beim Debuggen möchte man jedoch im Quelltext einer dynamischen Instanz vor der ersten Benutzung einen Breakpoint setzen können (für weitere Informationen hierzu siehe Kapitel 1.5.4 Abschnitt Debugging Probleme).

1.4 Der Entwicklungsprozeß, die Methodik

1.4.1 Mengengerüste und Leistungsdaten

Der Bibliography Shopper besteht aus ca. 40’000 Zeilen Java Code verteilt auf etwa 250 Klassen und 100 Schnittstellen. Den grössten Teil des Code macht dabei die Benutzerschnittstelle mit ca. 20’000 Zeilen Code und etwa 150 Klassen und Schnittstellen aus. Die mitttlere Vererbungstiefe der Klassen des Kernentwurfs beträgt 4.

Zum Test wurden verschiedene BibTeX Bibliographien verwendet, welche insgesamt ca. 10’000 Bibliographie-Referenzen enthielten. Eine gespeicherte Bibliography Shopper-Datenbank benötigt nach dem Import ca. den doppelten Speicherbedarf einer BibTeX-Datenbank, da zur Kontrolle alle Originaldaten zu den konvertierten Daten behalten werden. Nachdem die Bibliography Shopper-Datenbank nachträglich validiert und von den Originaldaten befreit wurde, braucht sie in etwa gleich viel Speicherplatz wie eine BibTeX-Datenbank.

Wie in Kapitel 1.5.2, Abschnitt „Speicherprobleme“ genauer ausgeführt wird, ist Java beim Serialisieren von Bibliographien sehr speicherhungrig. So muss der Bibliography Shopper für eine Datenbank mit 5’000 Einträgen (4MB auf Disk) mit einem Heap von mindestens 32MB gestartet werden.

1.4.2 Eingesetzte Hardware/Netzwerk

Der Bibliography Shopper wurde komplett auf einer Sparc-Station unter Solaris entwickelt. Zum Test der Platformunabhängigkeit von Java und des erstellten Codes wurde zusätzlich ein PC unter Windows NT verwendet.

Zum Test der Client/Server-Architektur wurde das Ubilab-Intranet verwendet. Auch hier wurden Sparc-Stations und PCs mit Windows NT in den Test einbezogen.

1.4.3 Eingesetzte Produkte

Bei der Entwicklung um beim Test kam das Java Development Kit 1.1.3 von Sun Microsystems zum Einsatz. Weitere Java Implementierungen, wie z.B. Visual Age for Java von IBM wurden getestet, doch da sie zum

Zeitpunkt der Entwicklung (Sommer/Herbst 1997) noch nicht dem Java 1.1 Standard entsprachen, nicht verwendet.

Zusätzlich zu eigens entwickelten, wiederverwendbaren Datenstrukturen und Algorithmen wie z.B. einem „Regular Expression"-Matcher wurde die Java Generic Collection Library (JGL), Version 2.0.2 der Firma Objectspace verwendet. Die Berührungspunkte zu dieser Library wurden jedoch gut gekapselt, so dass sie in Zukunft ohne Probleme durch andere Implementierungen ersetzt werden können.

Als Entwicklungsumgebung wurde SNiFF+, Version 2.3.1 der Firma TakeFive eingesetzt. Vor allem die vielseitigen Möglichkeiten im Code zu navigieren haben bei der Entwicklung viele Vorteile gebracht. In SNiFF+ konnte mit Leichtigkeit der gesamte JDK Sourcecode zum Projekt dazu geladen werden, so dass man über den gesamten Bibliography Shopper Code inklusive JDK Code navigieren konnte. Die zahlreichen Browser und der Cross-Referenzer haben sich dabei als sehr nützlich herausgestellt, wenn es darum ging, die Funktionsweise von schlecht dokumentierten Java Klassen zu verstehen oder einfach kurz die Methoden einer Klasse nachzuschlagen. Auf eine Verwendung der JavaDoc des JDK konnte daher in den meisten Fällen verzichtet werden.

1.5 Kritische Bewertung des Einsatzes von Java

Der Einsatz von Java hat sich unseres Erachtens nach bewährt. Vor allem die klaren Konzepte in den Java-Basisklassen, wie z.B. die Verwendung von Entwurfsmustern und Schnittstellen haben sich positiv auf die Entwicklung ausgewirkt.

Ein weiterer klarer Vorteil von Java liegt im Garbage Collector, der vor allem bei sehr dynamischen, verteilten Systemen wie beim Bibliography Shopper von grossem Nutzen ist, da nicht immer klar definiert werden kann, welche Instanz für die Verwaltung, also auch das Löschen, von Objekten zuständig ist.

1.5.1 Portabilität

Ein wichtiger Faktor, der dem Bibliography Shopper zugute kommt, ist die Portabilität von Java Applikationen. In der Tat waren grosse Teile des Bibliography Shoppers ohne Änderung auf verschiedenen Plattformen lauffähig.

Leider muss jedoch auch erwähnt werden, dass gewisse Teile von Java, wie z.B. das Abstract Window Toolkit (AWT) noch einige Probleme in Bezug auf Portabilität bieten. So ist es beim Bibliography Shopper auf Grund einiger (schon seit geraumer Zeit) offiziell von Sun dokumentierten Fehlern zu Problemen gekommen. So musste z.B. die eigens entwickelte SplitPane von Grund auf neu programmiert werden, nachdem festgestellt wurde, dass unter Windows NT Mausevents an andere Komponenten verschickt wurden, als unter Solaris. Ein Umstieg auf Swing (siehe [Sun98b]) bzw. Java 1.2 dürfte die meisten dieser Probleme jedoch beseitigen. Ob jedoch die gravierendsten Probleme welche beim Mischen von plattformspezifischen und leichtgewichtigen Komponenten (v.a. in Bezug auf Events) auftreten, gelöst werden, ist jedoch unklar.

Ein weiteres Problem stellt im Moment die Betriebssystemkapslung dar. Die von Java gebotene Anbindung an das Betriebssystem ist sehr dünn und führt daher immer wieder zu Problemen. Bei Softwaresystemen, die nur wenige Funktionen des Betriebssystems gebrauchen, wie z.B. dem Bibliography Shopper, mag dies auf den ersten Blick nicht tragisch erscheinen. Doch bei näherer Betrachtung stellt man auch hier einige Probleme fest.

So ist es nicht möglich festzustellen, ob das Betriebssystem eine Unterstützung zur Registrierung von Dokumenttypen und Programmen unterstützt. Möchte man z.B. unter Windows NT aus einer Java Applikation heraus eine Textdatei in einen externen Texteditor laden, so muss man den zu verwendenden Texteditor explizit von aussen setzen, obwohl das Betriebssystem den Texteditor bereits kennt.

Ferner hat sich die plattformunabhängige Verwaltung von Benutzereinstellungen als schwierig herausgestellt. Unter Unix ist es z.B. üblich, dass man verschiedene Präferenzdateien unterhält. So würde für den Bibliography Shopper die Datei `/etc/.bibshopperrc` die systemweiten Einstellungen, wie z.B. Standardlayouts beinhalten, während die Präferenzdatei `~meier/.bibshopperrc` die angepassten Einstellungen des Benutzers „meier" beinhalten würde. Unter Windows NT würde man in ähnlicher Manier systemweite und benutzerspezifische Einstellungen trennen. Die Daten würden jedoch nicht in Dateien, sondern in die Registry geschrieben. In Java stellt man nun fest, dass Einstellungen in sog. Properties festgehalten werden können und eigentlich wäre es auch anzunehmen, dass diese Properties der Plattform entsprechend abgelegt

werden. Doch leider bietet das JDK beim sichern der Properties nur wenig Hilfe. Man muss den Ort an welchem die Property-Datei abgelegt wird selbst bestimmen. Ein Speichern in die Registry wird aus Gründen der plattformunabhängigkeit erst gar nicht unterstützt.

Als nützlichen Workaround hat sich hierbei erwiesen, dass man den Laufzeitsystem explizit via Java Properties mitteilt, wo die Einstellungsdateien liegen. Zum Start des Bibliography Shoppers wird unter Unix ein Shell-Script verwendet welches die folgende Zeile beinhaltet:

```
java  -classpath $CLASSPATH:$BIBSHOPPER_HOME
      -Dbibshopper.systemrc=/etc/.bibshopperrc
      -Dbibshopper.userrc=$HOME/.bibshopperrc
      Bib.Shopper
```

Unter Windows NT funktioniert dieser Trick zwar auch, wenn man damit leben kann, dass die Properties in eine Datei geschrieben werden. Möchte man jedoch die Daten in der Registry speichern, so kommt man um einen plattformabhängigen Codeteil oder ein externes Programm, das dies erledigt, nicht herum.

Die Portabilität einer Java Applikation ist also mit wenigen Ausnahmen im AWT und in der Betriebssystemkapslung von Hause aus schon ziemlich gross. Sie wird jedoch durch die Verwendung von Java alleine nicht garantiert. Es müssen einige Vorkehrungen getroffen werden, damit die Applikation problemlos auf mehreren Plattformen arbeitet. Das JDK 1.2 wird in dieser Hinsicht einige Neuerungen bringen, wie z.B. Swing, oder erweiterte Properties, die Portabilität erhöhen können.

1.5.2 Performance

Die Performance von Java hat sich, im Gegensatz zu oft gehörten Aussagen, als ausreichend erwiesen.

Vor allem die Performance der Kernteile, wie die Netzwerkkommunikation und das Suchen z.B. mittels regulärer Ausdrücke haben selbst bei grösseren Datenmenge nicht zu Problemen geführt.

Geschwindigkeitsprobleme

Probleme hingegen hat die schwache Performance des AWT bereitet. Vor allem wenn viele AWT Komponenten erzeugt werden sinkt die Performance des AWT drastisch.

Wir sind jedoch zuversichtlich, dass mit einigen wenigen Optimierungen an der Benutzerschnittstelle des Bibliography Shoppers eine genügende Performance erzielt werden kann. Im Moment werden z.B. Tabellen mittels ScrollablePanels erzeugt welches alle Einträge beinhalten. Dies führt dazu, dass bei grossen Tabellen sehr viele AWT Komponenten erzeugt werden. Eine erhebliche Beschleunigung könnte durch eine spezielle Tabelle erzielt werden, welche den Inhalt selbst verwaltet und nur gerade so viele AWT Komponenten erzeugt, wie zur Darstellung einer Bildschirmseite nötig sind. Wird die Tabelle gescrollt, so könnte einfach der dargestellte Inhalt der AWT Komponenten ersetzt werden.

Speicherprobleme

Die automatische Serialisierung von Objekten hat sich vor allem bei vielen, kleinen Objekten als problematisch herausgestellt. So sind z.B. zur Serialisierung einer 4MB grossen Datenbank mehr als 32MB Heap vonnöten.

Abhilfe zu diesem Problem kann auf zwei verschiedenen Wegen erreicht werden. Einerseits kann man anstelle der automatischen Serialisierung via `Serializable` Interface die Serialisierung mittels dem `Externalizable` Interface selbst durchführen. Dies ist wesentlich resourcenschonender, da die Struktur der zu speichernden Daten bestens bekannt ist und dahingehend optimiert werden kann. Der zweite Weg zur Beseitigung dieses Problems ist die komplette Serialisierung von Hand. Dieser Weg bringt deutliche Speicher- und Performancevorteile ist jedoch wesentlich aufwendiger als die Verwendung der Objektserialisierung, die von Java angeboten wird.

1.5.3 Stabilität

Die Stabilität der verwendeten Java-Implementierung von Sun Microsystems, Inc. hat sich im Betrieb der ersten Bibliography Shopper-Version als sehr hoch herausgestellt. So liefen z.B. frühe Versionen des Domain Name Servers ununterbrochen mehrere Wochen bis hin zu einigen Monaten ohne grössere Probleme.

Obwohl der Bibliography Shopper viele kurzlebige Objekte erzeugt, traten in den Tests auch nach längerer Laufzeit keine Probleme auf. Dies lässt auf einen zuverlässigen Garbage Collector schliessen.

1.5.4 Entwicklungswerkzeuge

Die Entwicklungswerkzeuge des Sun Microsystems JDK entsprechen einem hohen Standard und haben sich während der Entwicklungsphase als stabil herausgestellt. Es sind folgende Build und Debugging Probleme aufgetreten.

Build Probleme

Das Kompilieren von Applikationen birgt ab einer gewissen Grösse der Applikation einige Probleme. Das von der Unix-Welt her bekannte „make“ wird in vielen Firmen noch als *das* Build-Tool verwendet.

Der Java Compiler von Sun Microsystems braucht, so heisst es, keine weiteren Build-Tools. Er erkennt Beziehungen von alleine und kompiliert geänderte Sourcen richtig, ohne weiteres zu tun. Nun, leider funktioniert dies nur wenn einige Vorbedingungen, welche bei grösseren Projekten nur sehr schwer einzuhalten sind, stimmen.

Zu Fallstricken können die folgenden Dinge werden:

- ❏ Dynamische Beziehungen
- ❏ Rekursive Abhängigkeiten

Verwendet man, wie im Bibliography Shopper abstrakte Fabriken und Product Traders, welche via dem Java Reflection API agieren, so kann der Java Compiler diese Beziehungen nicht feststellen. Der Grund hierfür ist einfach: die verwendeten Klassen werden nicht statisch im Sourcecode referenziert, sondern dynamisch über Strings benannt.

Eine einfache Abhilfe zu diesem Problem gibt es nicht. Entweder man entschliesst sich für eine externe Buildlösung, in welcher man die Abhängigkeiten explizit angeben kann (wie z.B. bei „make“) oder man fügt die Referenzen explizit in den Sourcecode ein (was klar gegen die Grundidee der dynamischen Beziehung geht und vermieden werden sollte).

Für das Auflösen von rekursiven Abhängigkeiten bietet der Java Compiler die `-depend` Option. Diese Option ist sehr nützlich, aber die Kompilierzeiten erhöhen sich bei grösseren Projekten drastisch.

Für den Bibliography Shopper wurde aus diesem Grund der Makefile Generator von SNiFF+ verwendet. Leider war zum Zeitpunkt der Entwicklung der Java-Support von SNiFF+ noch nicht ausgereift, doch immerhin konnte man ohne Probleme mittels „make all“ den gesamten Bibliography Shopper kompilieren, bzw. mittels direktem Aufruf des Java Compilers das kompilieren einzelner Sourcen erzwingen. Alles in allem hat sich der Make-Support von SNiFF+ als sehr nützlich herausgestellt und Tests zum bauen des Bibliography Shoppers von Hand haben deutlich aufgezeigt, dass eine Build-Lösung wie die von SNiFF+ dringend zu empfehlen ist.

Debugging Probleme

Wie schon beim Build Problem hat sich die starke Dynamik des Bibliography Shoppers beim Debuggen als ein Problem herausgestellt. Java lädt auch beim Debuggen die Klassen dynamisch beim ersten Gebrauch nach. Beim Debuggen kann jedoch der Fehler schon bei der Initialisierung eines Objektes, also im Konstruktor geschehen, ohne dass man eine Möglichkeit hat vorher einen Breakpoint zu setzen.

Um diesem Problem zu umgehen wurde beim Bibliography Shopper eine pragmatische Lösung verwendet. Die für den Bootstrap notwendige Klasse, welche die abstrakten Fabriken und Product Traders initialisiert lädt auch viele wichtige Klassen via `java.lang.Class.forName (aClassName)` im Voraus. Da hierbei noch keine Objekte erzeugt werden kann man nun in aller Ruhe Breakpoints auch in den Konstruktoren der Objekte setzen.

1.6 Zusammenfassung

Der Bibliography Shopper ist ein kleineres, aber dennoch recht komplexes, verteiltes Java System. Der Einsatz von Java hat sich für dieses Projekt sehr bewährt. Wir gehen somit davon aus, dass Java für gleichartige System als geeignete Imlementierungsplattform betrachtet werden kann..

Die Performance von Java im nicht-graphischen Bereich war unseren Ansprüchen durchaus genügend. Schwierigkeiten verursacht haben aber die graphischen Bereiche. Da der Grund hierfür in dem bisherigen Implementierungen des JDK Graphikmodells gesucht werden muss, versprechen zukünftige JDK Versionen Abhilfe.

Die Einfachheit der Sprache, insbesondere die automatische Speicherverwaltung, erlaubt bei vergleichbaren Bibliotheken, erheblich kürzere Entwicklungszeiten als z. B. C++.

Eine grosse Stärke von Java liegt in ihrer Flexibilität, die durch das Reflection API und das dynamisches Laden von Klassen erreicht wird. Dynamisch konfigurierbare Systeme können so mit geringen Aufwand erstellt werden, wie am Bibliography Shopper demonstriert wurde. Java selbst bietet jedoch über die Flexibilität hinaus keine weiteren Mechanismen zum Erstellen und Verwalten solcher Systeme. So kann z.B. der Sun Java Compiler keine dynamischen Abhängigkeiten erkennen, was uns zwang, einen externen Build-Mechanismus zu benutzen. Ebenso ist es dem Debugger nicht möglich, Unterbrechungspunkte in noch nicht geladenen Klassen setzen zu lassen. Wir mussten den Code für Testphasen so modifizieren, dass alle benötigten Klassen vorab geladen wurden.

Die Java-Bibliotheken sind im Grossen und Ganzen von zufriedenstellender Qualität, im Detail aber noch nicht ausgereift. So sind verschiedene, sehr häufig benötigte Mechanismen, wie z.B. das Addieren zweier Vektoren, die eine gewisse Bequemlichkeit im Umgang mit Bibliotheken bieten, nicht zu finden. Zukünftige JDK Versionen versprechen auch hier Abhilfe zu bieten.

1.7 Danksagungen

Wir möchten uns herzlich bei Dirk Bäumer, Walter Bischofberger und Thomas Kofler für ihre Unterstützung bedanken. Durch wiederholtes Korrekturlesen und in zahlreichen Diskussionen haben sie wertvolle Hinweise zu diesem Artikel geliefert.

1.8 Kurzvorstellung der Autoren

Bruno Essmann (be@acm.org) befindet sich zur Zeit in der Endphase seines Informatik-Studiums an der ETH (Eidgenössische Technische Hochschule) Zürich. Er hat im Sommer/Herbst 1997 sein Industriepraktikum am Ubilab, dem Informatik-Forschungslabor der UBS (Union Bank of Switzerland) absolviert und arbeitet im Moment als Werkstudent bei der Firma TakeFive an einer Build-Informationsdatenbank.

Dipl.-Inform. Dirk Riehle arbeitet am Ubilab, dem Informatik-Forschungslabor der UBS (Union Bank of Switzerland). Er ist Autor zahlreicher Fachartikel zu den Themen Objektorientierung und Entwurfsmuster. Außerdem ist er Übersetzer des Buchs *Design Patterns* sowie Autor des frisch bei Addison-Wesley Deutschland erschienenen Buchs *Entwurfsmuster für Softwarewerkzeuge*.

Kai-Uwe Mätzel arbeitet seit 1998 als Designer und Entwickler bei Object Technology International (OTI). Zuvor war er am Ubilab, dem Informatik-Forschungslabor der UBS. Er entwickelte dort eine Mehrbenutzerentwicklungsumgebung für C++ und arbeitet auf dem Gebiet adaptiver Softwarearchitekturen. Er ist Autor mehrere Fachartikel zu diesen Themen.

A Referenzen

[Bäumer+96] Bäumer D., Riehle D.: „Product Trader“; *Pattern Languages of Program Design 3*. Edited by Robert C. Martin, Dirk Riehle and Frank Buschmann. Reading, MA: Addison-Wesley, 1998. Kapitel 3.

[Bischofberger96] Bischofberger W. R., Riehle D.: Global Business Objects - Requirements and Solutions. Mätzel K.-U., Frei H. P. (eds.): *Computer Science Research at Ubilab, Research Projects 1995/96; Proceedings of the Ubilab Conference '96*, Universitätsverlag Konstanz, Konstanz, November 1996, pp. 79-98.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster: Elemente Wiederverwendbaren Objekt-orientierter Software.* Übersetzung von: *Design Patterns: Elements of Reusable Design.* Reading, MA: Addison-Wesley, 1995.

[Knuth84] D. E. Knuth: *TeXBook*. Reading, MA: Addison-Wesley, 1984, pp. 140-147.

[Kernighan82] B. W. Kernighan, M. E. Lesk: UNIX document preparation. J. Nievergelt, G. Coray, J.-D. Nicoud, and A. C. Shaw (eds.): *Document Preparation Systems: A Collection of Survey Articles*"**,** Elsevier North-Holland, Inc. 1982, pp. 1-20.

[MacLennan82] MacLennan B. J.: Values and Objects in Programming Languages. *ACM SIGPLAN Notices* 17, 12 (December 1982), pp. 70-79.

[Mätzel96] Mätzel K.-U., Bischofberger W. R.: *The Any Framework - A Pragmatic Approach to Flexibility. Proc of the 2nd USENIX Conf on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Canada, June 1996, pp. 179-190.

[Mätzel96a] Mätzel K.-U., Bischofberger W. R.: Evolution of Object Systems - How to Tackle the Slippage Problem. Mätzel K.-U., Frei H. P. (eds.): *Computer Science Research at Ubilab, Research Projects 1995/96; Proceedings of the Ubilab Conference '96*, Universitätsverlag Konstanz, Konstanz, November 1996, pp. 99-119.

[Riehle96] Riehle D.: Describing and Composing Patterns Using Role Diagrams. Mätzel K.-U., Frei H. P. (eds.): *Computer Science Research at Ubilab, Research Projects 1995/96; Proceedings of the Ubilab Conference '96*, Universitätsverlag Konstanz, Konstanz, Germany, 1996, pp. 137-152.
Originally published in: WOON '96 (1st Int'l Conference on Object-Orientation in Russia), conference proceedings. Edited by A. Smolyani and A. Shestialtynov. St. Petersburg, St. Petersburg Electrotechnical University, 1996.

[Riehle97a] Riehle D.: *Arbeiten mit Java-Schnittstellen und -Klassen (Teil 1 von 2)*. Java Spektrum, 5/97 (September/Oktober 1997), September 1997, pp. 26-33.

[Riehle97b] Riehle D.: *Arbeiten mit Java-Schnittstellen und -Klassen (Teil 2 von 2)*. Java Spektrum, 6/97 (November/Dezember 1997), November 1997, pp. 35-43.

[Sun96a] Sun Microsystems, Inc.: *Remote Method Invocation Specification*. http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.

[Sun96b] Sun Microsystems, Inc.: *Java AWT: Delegation Event Model.* http://www.javasoft.com/products/jdk/1.1/docs/guide/awt/designspec/events.html.

[Sun96c] Sun Microsystems, Inc.: *Java AWT: Lightweight UI Framework*. http://www.javasoft.com/products/jdk/1.1/docs/guide/awt/designspec/lightweights.html.

[Sun96d] Sun Microsystems, Inc.: *Reflection*. http://www.javasoft.com/products/jdk/1.1/docs/guide/reflection/index.html.

[Sun98a] Sun Microsystems, Inc.: *Reference Objects*. http://www.javasoft.com/products/jdk/1.2/docs/guide/refobs/index.html.

[Sun98b] Sun Microsystems, Inc.: Sun Microsystems, Inc.: *Swing (JFC)*. http://www.javasoft.com/products/jdk/1.2/docs/guide/swing/index.html.

[Woolf96] Woolf B.: „The Type Object Pattern“. *Pattern Languages of Program Design 3*. Edited by Robert C. Martin, Dirk Riehle and Frank Buschmann. Reading, MA: Addison-Wesley, 1998. Kapitel 4.