Entwicklung von qualitätssichernden Werkzeugen für ObjectTeams/Java

Diplomarbeit

Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik
Prof. Dr. Ing. Stefan Jähnichen

Betreut von Dr. Stephan Herrmann

SS 2005

Iryna Ivanova
Matrikel: 194032

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 30. Januar 2006

_____

Iryna Ivanova

Inhaltsverzeichnis

Einleitung

Aspektorientierte Softwareentwicklung ist ein Paradigma, das die Objektorientierung um die Möglichkeit erweitert, Querschnittsmerkmale des Systems getrennt voneinander zu modellieren und zu implementieren. Dies verspricht eine erleichterte Wiederverwendung und Wartung von Software-Systemen und dadurch eine höhere Produktivität der Entwickler.
Eine Trennung von einzelnen Merkmalen des Systems kann auf unterschiedliche Weisen umgesetzt werden. *ObjectTeams* ist ein Ansatz, bei dem die Modellierung, Verfeinerung und Komposition von Objekt-Kollaborationen und ihre nachträgliche Integration in existierende Modelle im Vordergrund stehen.
Die Programmiersprache *ObjectTeams/Java* (*OT/J*) erweitert *Java* um die Konzepte von *ObjectTeams*. Eine Unterstützung der *OT/J*-Entwickler ist deswegen wichtig, weil die Technologie jung und komplex ist. Einen bedeutenden Schritt in diese Richtung stellt die Integration von *OT/J* in die Entwicklungsumgebung *Eclipse* dar.
Da die Sprache einen neuen Modularisierungsmechanismus und zusätzliche, der objektorientierten Welt fremde Konzepte realisiert, sind die herkömmlichen qualitätssichernden Werkzeuge nicht brauchbar.
Diese Arbeit beschäftigt sich mit der Entwicklung einer *Toolbox* - eines Basiskerns für Werkzeuge, der eine geeignete Grundlage für die Werkzeugimplementierung für *OT/J* darstellt und in *Eclipse* integriert ist. Anhand von Implementierungen eines Dokumentationswerkzeugs und eines Design-Kritiken-Werkzeugs werden die Vorteile einer gemeinsamen Basis exemplarisch untersucht.
Werkzeuge, die auf der qualitativen statischen Programmanalyse basieren, manipulieren die Daten, indem sie entweder durchsucht oder transformiert werden. Die Repräsentation der Daten spielt dabei eine wichtige Rolle: Sie muss leicht zu manipulieren sein und die logische Struktur des Quellcodes wiederspiegeln. Aus diesem Grund fiel die Wahl der Datenrepräsentation in dieser Arbeit auf *XML* (*Extensible Markup Language*).
Die Arbeit ist wie folgt gegliedert:

- Kapitel 1 führt in die aspektorientierten Konzepte von *ObjectTeams* ein.
- Kapitel 2 gibt einen Überblick über geeignete Programmdatenrepräsentation für die Werkzeuge. Dabei werden Vorteile einer *XML*-Repräsentation von *OT*-Programmen gezeigt und XML-verarbeitende Technologien vorgestellt.
- Kapitel 3 beschreibt die Realisierung einer Toolbox für *OT/J*.
- Im Kapitel 4 wird die Beschreibung der Implementierung des Dokumentationswerkzeugs gegeben.
- Kapitel 5 beschäftigt sich mit der Beschreibung der Design-Kritiken-Werkzeug-Implementierung.
- Im Kapitel 6 wird die Arbeit zusammenfasst.

1 Aspektorientiertes Programmiermodell *ObjectTeams*

1.1 Aspektorientierung

Um den hohen Ansprüchen der Industrie Rechnung zu tragen, bei der Erstellung, Wartung und Erweiterung von komplexen Systemen den Akzent auf die Wirtschaftlichkeit und die Zuverlässigkeit von Software zu setzen, brauchen die Softwareentwickler Methoden, die diese Zielanforderungen erfüllen. Dabei spielt ein gutes Strukturierungs- und Modularisierungskonzept eine wichtige Rolle.
Das objektorientierte Paradigma bietet eine strukturelle Modularisierung durch Klassen bzw. Objekte, in denen Daten und Verhalten einer Entität gekapselt werden. In die Hauptfunktionalität der Klassen schleichen sich oft systemübergreifende Eigenschaften ein (*crosscutting concerns*). Diese Querschnittsmerkmale können nicht mit herkömmlichen Techniken realisiert werden, ohne, dass sich der inhaltlich zusammenhängende Code einer Funktionalität über viele Klassen verteilt (*scattering*) oder der Code einer Klassenmethode verschiedene Aufgaben erledigt (*tangling*).
Das aspektorientierte Paradigma verspricht genau bei diesem Problem Abhilfe zu schaffen, indem die Objektorientierung um einen neuen Modularisierungsmechanismus - Aspekt erweitert wird. Querschnittsmerkmale des Systems werden als Aspekte (*concerns*) zusammengefasst, die unabhängig voneinander entwickelt und verfeinert werden können.
In der aspektorientierten Programmierung (AOP) werden folglich systemübergreifende Funktionalitäten in *concerns* modelliert und in bestehende Software-Systeme integriert. Dabei wird das Verhalten einer Basisanwendung, auf die eine solche Funktionalität abgebildet wird, verändert bzw. erweitert.
Was *concerns* eigentlich sind und wie ihre Konstruktion erfolgen soll, ist nicht eindeutig festgelegt. Genauso undefiniert ist die Integration der *concerns* auf bestehende Applikationen. Die Umsetzung von AOP-Konzepten wird in verschiedenen Ansätzen unterschiedlich gelöst.

In allen Ansätzen werden Stellen im Kontrollfluss einer Basisanwendung festgelegt, auf die ein Aspekt abgebildet wird und damit ein neues Verhalten hinzugefügt oder das Originalverhalten geändert wird. Diese Stellen werden *join points* genannt.
Die Abbildung eines Aspekts auf Softwareartefakte der Programmiersprache einer Basisanwendung erfolgt in den meisten realisierenden Techniken durch Weben (*Weaving*). Der Aspektcode wird dabei mit dem Code der Basisanwendung an den durch *join points* gekennzeichneten Stellen verwoben. Dies kann entweder zur Compile-Zeit (statisches Weben) oder zur Laufzeit (dynamisches Weben) geschehen. Einige Techniken erlauben zusätzlich eine dynamische Aktivierung von Aspekten zur Laufzeit.

1.2 ObjectTeams

Das Programmiermodell ObjectTeams [2] vereinigt viele Fähigkeiten von anderen existierenden Methoden für die Komposition und Verfeinerung von Objekt-Kollaborationen und ergänzt sie um die Konzepte der unabhängigen Entwicklung von Kollaborationen und der nachträglichen Integration der Kollaborationen in existierende Modelle.
Eine Kollaboration erstreckt sich über mehrere Klassen bzw. Objekte, deren Zusammenspiel ein bestimmtes Verhalten definiert, wobei jede beteiligte Klasse darin ihre spezifische Rolle spielt. Eine Kollaboration zeigt den Charakter einer weiteren Modularisierungsdimension einer objektorientierten Anwendung neben einer strukturellen Modularisierung durch Klassen.
Im Folgenden beziehen sich weitere Beschreibungen und Codefragmente von ObjectTeams auf die Syntax der Sprache ObjectTeams/Java, welche die Konzepte von ObjectTeams umsetzt und auf Java basiert.

1.2.1 Der Team-Begriff

Der neue Modularisierungsmechanismus bei ObjectTeams heißt Team und kapselt einzelne Kollaborationen mehrerer Klassen, die im Team als Rollen bezeichnet werden. Eine Funktionalität, die in einem Team unabhängig von den anderen Modulen entwickelt wird, kann einer oder mehreren Basisanwendungen nachträglich hinzugefügt werden.
Ein Team ist auf der einen Seite eine spezielle Klasse, die mit einem Schlüsselwort **team** markiert wird und eigene Attribute und Methoden enthalten kann. Auf der anderen Seite ist ein Team eine Aggregation von Rollenklassen, die in Form von inneren Klassen implementiert werden.
Ein Bespiel für ein Team ist in Listing 1.1 angegeben. Das Team `Authori-`

```
1  public abstract team class Authorization {
2    public abstract SystemUser getCurrentUser();
3    public abstract class RestrictedAccount { // role class
4        public abstract SystemUser getOwner();
5        public callin double requireOwner() {
6            SystemUser currentUser = getCurrentUser();
7            if(! currentUser.equals(getOwner()))
8              try {
9                throw new AutorizationException("no access");
10             } catch(AutorizationException e) {
11               e.printStackTrace();
12               //throw a NumberFormatException
13               return Double.parseDouble("");
14             }
15           return base.requireOwner();
16         }
17   }
18   public abstract class SystemUser { // role class
19       public boolean equals(Object another){
20           SystemUser su = (SystemUser) another;
21           return
22             getUsername().equals(su.getUsername()) &&
23             getPassword().equals(su.getPassword());
24       }
25       public abstract String getUsername();
26       private abstract String getPassword();
27   }
28 }
```

Listing 1.1: *Abstraktes Team Authorization*

`zation` ist abstrakt und besteht aus zwei Rollen `RestrictedAccount` und `SystemUser`. Die Rollen definieren das Verhalten eines Kontos bzw. eines Benutzers bei der Erteilung der Zugriffsberechtigung.
Die Methode `requireOwner()` der Rolle `RestrictedAccount` führt eine betragsverändernde Operation auf dem Konto nur dann aus, wenn sich der angemeldete Benutzer als der Eigentümer des Kontos erweist. Sonst darf kein Zugriff erteilt werden, daher wird eine *Exception* geworfen.
Eine Implementierung der Methoden `getUsername()` und `getPassword()` der Rolle `SystemUser` und `getOwner()` der Rolle `RestrictedAccount` hängen von der konkreten Basisanwendung ab. Diese Methoden bleiben deshalb abstrakt und erzwingen dadurch die Abstraktheit der Rollen und damit wiederum des Teams.
Die Team-Methode `getCurrentUser()` ist von der Implementierung des As-

pekts der Zugangsberechtigung abhängig und wird daher später in einem konkreten Team definiert.

1.2.2 Team- und Rollenvererbung

Sofern in einem Team keine explizite Superklasse nach dem Schlüsselwort **extends** angegeben ist, erbt das Team implizit von einer vordefinierten Teamklasse `org.objectteams.Team`. Die Superklasse eines Teams muss selbst wiederum ein Team sein.
Bei einer Teamspezialisierung werden Felder, Methoden aber auch Rollen des Superteams geerbt. In einem Team können die geerbten Rollenklassen seines Superteams verfeinert werden, dabei muss die Spezialisierungsbeziehung nicht extra gekennzeichnet werden, da sie implizit durch die Namensgleichheit mit der Superrolle gilt.
Die implizite Vererbung von Rollen eines Superteams stellt keine Subtypbeziehung her. Es gibt keine Konformität von Rollen eines Teams zu Rollen eines anderen Teams und daher keine polymorphe Substitution der Rollen aus verschiedenen Teams.
Eine Rolle kann neben der impliziten Superrolle auch eine explizite Superklasse besitzen. Die explizite Vererbung wird nach dem Schlüsselwort *extends* angegeben. Die Superklasse der Rolle kann selbst eine gewöhnliche Klasse oder eine Rollenklasse desselben Teams sein.
Erbt eine Rolle implizit von einer Superrolle, die wiederum eine explizite Superklasse besitzt, dann hat die Rolle auch diese Superklasse. In diesem Fall kann die explizite Vererbungsbeziehung nur durch eine spezifischere Superklasse überschrieben werden. Listing 1.2 stellt diese Regel exemplarisch dar. Demnach muss die Klasse `B` in einer Subtypbeziehung mit der Klasse `A` stehen.

```
1 class team Superteam {
2   class Role extends A { ... }
3 }
4
5 class team Subteam extends Superteam {
6   // B must be a sub-class of A
7   class Role extends B { ... }
8 }
```

Listing 1.2: *Rollenvererbung*

1.2.3 Beziehung zwischen Rollen und Basisklassen

Wie bereits erwähnt, wird die in einem Team entwickelte Funktionalität einer oder mehreren Basisanwendungen nachträglich hinzugefügt. Die Adaption verläuft für die Basisanwendung unerwartet und nicht invasiv, dabei hat sie zu keinem Zeitpunkt Wissen über Anzahl oder Inhalte der sie adaptierenden Teams.
Die Integration eines neuen Verhaltens geschieht in Rollen des Teams auf der Klassenebene durch die Bindung von Rollen an Basisklassen und auf der Methodenebene durch zwei Arten von Bindungen der Rollenmethoden an Basismethoden: *callin* und *callout*.

Konnektor Kommen in einer Teamklasse Bindungen auf der Klassen- oder Methodenebene vor, nennt man sie einen *Konnektor*. Ein Konnektor führt das abstrakte Verhalten (definiert in einem abstrakten Team) auf der einen Seite und eine konkrete Basisanwendung auf der anderen Seite zusammen. Optimal ist es, wenn ein adaptierendes Team soviel wie möglich Implementierungen aber keine Bindungen enthält, im Gegensatz zu einem Konnektor, der kaum Implementierungen und alle Arten von Bindungen enthalten soll. In Listing 1.3 ist der Konnektor `AccountAuthorization` implementiert, der das Team `Authorization` aus Listing 1.1 spezialisiert, in welchem das Verhalten einer Erteilung von Zugriffsrechten definiert ist.

playedBy-Beziehung Die Bindung einer Rolle an eine Basisklasse erfolgt durch das Schlüsselwort *playedBy*. So zum Beispiel wird in Listing 1.3 die Rolle `RestrictedAccount` an die Basisklasse `Account` gebunden. Die *playedBy* Beziehung wird von impliziten und expliziten Subklassen mitgeerbt und kann nur von einer expliziten Subrolle durch eine spezifischere Basisklasse verfeinert werden.
Die Bindung auf der Klassenebene ändert weder das Verhalten von Rollen noch von Basisklassen, sondern schafft die Rahmenbedienungen für diese Änderungen durch die oben erwähnten *callin*- und *callout*-Bindungen.

1.2.3.1 Callin-Bindung

Durch eine callin-Bindung kann das Originalverhalten einer an eine Rollenmethode gebundenen Basismethode erweitert oder überschrieben werden. Dabei fängt die Rollemethode die Basismethode ab und wird je nach Art der callin-Bindung vor dem Aufruf der Basismethode, nach dem Aufruf oder statt des Aufrufs ausgeführt. Technisch gesehen wird zusätzlicher Code zur Laufzeit in den Bytecode der Basismethode eingewebt [5].

```
1  public team class AccountAuthorization extends Authorization {
2    public SystemUser getCurrentUser() {
3      User u = Authentication.getInstance().getCurrentUser();
4      return getSystemUser(u);
5    }
6    public SystemUser getSystemUser(User as SystemUser user) {
7      return user;
8    }
9    public class RestrictedAccount playedBy Account { //role
10     getOwner -> getOwner; //callout
11     requireOwner <- replace getBalance, debit, transfer;
12   }                                                    //callins
13   public class SystemUser playedBy User { //role
14     getUsername -> getUsername;  //callout
15     getPassword -> get password; //callout to field
16   }
17 }
```

Listing 1.3: *Rollen- und Methodenbindungen im Konnektor AccountAuthorization*

Eine callin-Bindung wird in der gebundenen Rolle mit einem Pfeil, der von der Basismethode in Richtung der Rollenmethode (`<-`) zeigt, und einem der Schlüsselwörter **before**, **after** oder **replace** gekennzeichnet.
Soll eine Rollenmethode durch einen *replace callin* das Verhalten von Methoden der Basis neu definieren und damit diese durch sich selbst ersetzen, muss sie mit einem Schlüsselwort *callin* markiert werden. Dabei besteht die Möglichkeit aus der Rollenmethode mittels des Schlüsselworts **base**, die Originalbasismethode aufzurufen.
Die callin-Methode `requireOwner()` aus Listing 1.1 verwaltet den Aufruf ihrer Basismethode, welche nur unter bestimmten Bedingungen ausgeführt wird. In Zeile 15 ist der eigentliche Aufruf der Basismethode (`base.requireOwner()`) enthalten, wobei der Name der callin-Methode für den Namen der Basismethode und das Schlüsselwort `base` für die Basisreferenz stehen. In Zeile 11 von Listing 1.3 wird diese Methode durch einen *replace callin* an alle Kontostand betreffenden Methoden der Klasse `Account` gebunden.

1.2.3.2 Callout-Bindung

Methoden, die für die Definition des Verhaltens einer Rolle benötigt werden, aber noch nicht implementiert sind, werden in der Rolle als abstrakte Methoden deklariert, um aufgerufen werden zu können. Sie können entweder von den Subklassen der Rolle direkt implementiert oder durch die Definition von callout-Bindungen an Features der Basisklasse weitergeleitet werden.

Eine callout-Bindung vervollständigt eine abstrakte Rollenmethode und entspricht einer Delegation des Aufrufs der Rollenmethode an die Methode der Basisklasse. Sie wird mit einem Pfeil von der Rollenmethode in Richtung der Basismethode (`->`) gekennzeichnet.
Eine callout-Bindung ist in Listing 1.3 in Zeile 14 definiert: Die abstrakte Methode `getUsername` der Rolleklasse `SystemUser` wird an die konkrete Methode `getUsername` der Klasse `User` gebunden.
Durch eine callout-Bindung besteht auch die Möglichkeit auf die Felder der Basisklasse zuzugreifen. Dazu wird in der Rolle eine geeignete Signatur deklariert und ein callout definiert, in dem nach dem Pfeil (`->`) die Art des Zugriffs (*get* oder *set*) und der Name des Feldes angegeben werden.
In Zeile 15 in Listing 1.3 wird die Rollenmethode `getPassword` an das Feld `password` der Klasse `User` gebunden, was zur Generierung einer Implementierung von `getPassword` führt, die den Wert des Feldes der Basisklasse `password` zurückgibt.
Zudem gibt es eine Möglichkeit, eine konkrete Rollenmethode von einer Methode der Basisklasse zu überschreiben. Dabei wird in der callout-Definition ein Doppelpfeil `=>` statt des normalen Pfeils `->` verwendet.

Decapsulation Durch Methodenbindungen ist es möglich, auf die Features der Basisklasse zuzugreifen und dabei die eventuell in der Basis vereinbarten Zugriffseinschränkungen zu umgehen. Dieses Konzept wird *Decapsulation* genannt.

Signatur-Anpassung Unterscheiden sich die Signaturen einer Rollenmethode und einer Methode der Basis in einer callin- oder callout-Bindung, wird eine Signatur-Anpassung definiert. Dabei werden bei der Definition der Bindung vollständige Signaturen beider Methoden angegeben, und es kann nach dem Schlüsselwort **with** festgelegt werden, welche Parameter der Methoden in welcher Weise aufeinander abgebildet werden und wie das Ergebnis angepasst werden soll. Einzelheiten über die Syntax der Signatur-Anpassung können auf der ObjectTeams Homepage [1] bei der ObjectTeams/Java Sprachdefinition nachgelesen werden.

1.2.3.3 Translationspolymorphismus

In einem bestimmten Kontext eines Konnektors verhält sich eine gebundene Klasse wie eine von ihr gespielte Rolleklasse, eine Rolle hat wiederum das Verhalten und die Eigenschaften ihrer Basisklasse, obwohl beide Typen sich vor der Bindung in der Regel nicht kennen. Die playedBy-Beziehung schafft

die Voraussetzung, Typen der Rolle und der Basisklasse je nach Kontext gegeneinander zu ersetzen, ohne dass es sich um eine polymorphe Substitution auf der Grundlage der Vererbung handelt. Die neue Art der Überführung in den jeweils anderen Typ geschieht implizit und heißt Translationspolymorphismus.
Die Überführung einer Basisklasse in ihre Rolle wird *lifting* genannt, der umgekehrte Fall wird als *lowering* bezeichnet [3].

Lifting Ein Lifting geschieht implizit, wenn bei einem callout das Ergebnis zurücküberführt werden soll oder bei einem callin die Parameter zu ihren Rollen geliftet werden sollen. Zudem gibt es eine Möglichkeit, ein Lifting explizit für Parameter einer nicht statischen Methode eines Konnektors zu erzwingen. Dabei werden zwei Typen - Typ der Basisklasse und der Rolle, getrennt durch das Schlüsselwort *as* angegeben.
In Listing 1.3 fragt die Methode `getCurrentUser()` nach dem angemeldeten Benutzer und bekommt eine Instanz vom Typ `User`. Damit diese Instanz zu ihrer Rolle überführt werden kann, wird die Methode `getSystemUser(User as SystemUser user)` aufgerufen, die ein Liften eines Objekts vom Typ `User` zu einem Objekt vom Typ `SystemUser` zum Ziel hat.

Lowering Eine Überführung eines Rollenobjekts zu einem Objekt der Basisklasse geschieht implizit, wenn anstelle einer erwarteten Instanz der Basisklasse eine Instanz der Rollenklasse in den Zuweisungen oder als Parameter von Methoden eingesetzt wird.

1.2.3.4 Externalized roles

Wie oben erwähnt, kapselt ein Team seine Rollenklassen und stellt nach außen eine Schnittstelle für ihre Manipulation zur Verfügung. So genannte *externalized roles* sind auch außerhalb ihres Teams sichtbar. Der Typ solch einer Rolle ist abhängig von der Instanz des sie einschließenden Teams, um die Typsicherheit zu erreichen, damit eine Rolle nicht in ein fremdes Team gelangt.

1.2.4 Aktivierung von callin-Bindungen

Alle Änderungen des Verhaltens einer Basisanwendung mittels callin-Bindungen eines Teams sind nur dann wirksam, wenn das Team aktiv ist. Die Aktivierung eines Team kann explizit erfolgen oder durch den Kontrollfluss implizit durchgeführt werden.

```
1  public team class AuthorizationWithGuards {
2      public User getUser() {
3          return Authentication.getInstance().getCurrentUser();
4      }
5      public class RestrictedAccount playedBy Account
6      base when(!AuthorizationWithGuards.this.getUser().equals(
7        base.getOwner()) ) {
8          public callin double requireOwner() {
9            try {
10             throw new AutorizationException("no access");
11           } catch (AutorizationException e) {
12             e.printStackTrace();
13           }
14           //throw a NumberFormatException
15           return Double.parseDouble("");
16         }
17         requireOwner <- replace getBalance, debit, transfer;
18     }
19 }
```

Listing 1.4: *Aspekt der Zugriffsberechtigung mit einem base guard*

1.2.4.1 Wächterprädikate

Die Wirksamkeit der callin-Bindungen kann darüber hinaus durch sogenannte Wächterprädikate (*guard predicates*) kontrolliert werden [6]. Wird ein Wächterprädikat erfüllt, dann sind die von ihm gesteuerten callin-Bindungen wirksam.
In der bereits bekannten Implementierung des Aspekts der Zugriffsberechtigung aus Listing 1.1 kommt es in der Methode `requireOwner()` zum *tangling* vom Rollenmethodencode mit dem Code der Aktivierungskontrolle. Die Basismethode darf nur dann ausgeführt werden, wenn der Eigentümer des Kontos angemeldet ist. Dementsprechend gilt die Rollenmethode für übrige Benutzer, sie wirft eine *Exception* und ruft die Basismethode nicht auf. Die Abfrage, ob ein angemeldeter Benutzer der Kontoinhaber ist, gehört inhaltlich nicht zur Rollenmethode.
Um *tangling* an dieser Stelle zu umgehen, kann ein Wächterprädikat in einer callin-Bindung, Rollenmethode, Rolleklasse oder Teamklasse gesetzt werden, welches die Aktivierung der callin-Bindung prüft. Ein Wächter verwendet das Schlüsselwort **when**, nach dem in Klammern der Prädikatausdruck angegeben wird.
Es gibt je nach Kontext der Auswertung zwei Typen von Wächtern: *regular* und *base*. Reguläre Wächterprädikate werden bei einem *callin* nach dem Lifting der Basisklasse und vor dem Aufruf der Rollenmethode ausgewertet.

Wächterprädikate vom Typ *base* werden bereits vor dem Lifting ausgewertet, so dass einige durch *lifting* verursachte Seiteneffekte (4.2 in [6]) vermieden werden können.
Alle Wächter dürfen auf die Teaminstanz zugreifen, Wächter vom Typ *base* haben außerdem den Zugriff auf die Basisklasse. Wächterprädikate werden an implizite und explizite Subklassen vererbt.
In Listing 1.4 ist der Aspekt der Zugriffsberechtigung mittels eines Wächterprädikats vom Typ *base* implementiert. Die Rolle `RestrictedAccount` wird nur dann erzeugt, wenn der Eigentümer des Kontos nicht mit dem aktuellen Benutzer übereinstimmt. Die Prüfung der Rollenerzeugung übernimmt der Prädikatausdruck (Zeilen 6-7). Dabei wird zum Beispiel in Zeile 6 auf eine Teammethode zugegriffen, Zeile 7 zeigt den Aufruf einer Basismethode.

1.2.5 Entwicklung mit ObjectTeams/Java

ObjectTeams/Java ist eine Implementierung des ObjectTeams-Modells, die einen modifizierten Java-Compiler [7] und eine Laufzeitumgebung [5] beinhaltet. Der ObjectTeams/Java-Compiler wurde in die offene, erweiterbare Entwicklungsumgebung Eclipse [10] integriert [8], die einen inkrementellen Java-Compiler verwendet und viele Werkzeuge umfasst. Die neue Entwicklungsumgebung heißt Object Teams Development Tooling (OTDT) [11] und enthält viele nützliche Features für die Entwicklung mit ObjectTeams/Java.

2 XML-Repräsentation von ObjectTeams-Programmen

2.1 Werkzeuge als Qualitätssicherung

Die Software-Systeme sind oft komplex und umfangreich, sodass es für Entwickler schwierig ist, alle Abhängigkeiten ohne die Unterstützung der entsprechenden Werkzeuge auf verschiedenen Abstraktionsebenen zu erfassen.
Qualität von Software-Systemen wird unter anderem durch die Anzahl der darin enthaltenen Fehler, den Schwierigkeitsgrad ihres Verständnisses und die Flexibilität ihrer Wartung und Weiterentwicklung charakterisiert.
Auf allen Ebenen der Softwareentwicklung treten Fehler auf bzw. werden Fehlentscheidungen getroffen, die zu strukturellen Schwächen [21] führen, welche das Verständnis eines Software-Systems und seine Änderung unnötig erschweren. Der beste Weg der Qualitätssicherung ist die Anwendung von Mechanismen zur Fehlervorbeugung. Aber trotz aller Vorbeugungsverfahren bleibt ein Teil der Schwächen, die als solche nicht erkannt wurden, bestehen. Diese Mängel müssen entgedeckt und entfernt werden, um ein Maximum an Qualität zu erreichen.
Da strukturelle Schwächen sich im Quelltext manifestieren, ist es vorteilhaft, sie auf der Codeebene ausfindig zu machen. Dafür gibt es verschiedene Verfahren der Programmanalyse, die sich in statische und dynamische Analyse unterteilen. Bei einer statischen Analyse werden die Daten eines zu untersuchenden Programms - der Code vor der Programmausführung untersucht und durch qualitative oder quantitative Aussagen ausgewertet. Eine dynamische Analyse untersucht ein System zur Laufzeit und kann Fehler im Verhalten des Systems bei der Ausführung entdecken.
Im Rahmen meiner Diplomarbeit wird für die Sprache *OT/Java* eine *Toolbox* realisiert, die auf der qualitativen statischen Programmanalyse basiert.

2.2 Datenrepräsentation für die Programmanalyse

Damit eine statische automatische qualitative Programmanalyse durchgeführt werden kann, müssen Regeln definiert werden, mit deren Einsatz der Code des Programms nach spezifischen strukturellen Charakteristika durchsucht wird (s. 3.1.5). Die Suche erfordert eine gut strukturierte Form der Daten. Da der Quellcode eines Programms dafür wenig geeignet ist, werden für die Analyse meistens abstrakte Syntaxbäume (*Abstract Syntax Tree* = AST) eingesetzt. Ein AST gibt die logische Struktur des Quellcodes in einer baumförmigen Struktur wieder: Jedes Element im Quellcode wird von einem Knoten in der entsprechenden Hierarchieebene des Baums repräsentiert. So besteht zum Beispiel der Team-Knoten eines AST aus Knoten, die seine Felder, Methoden und Rollen wiederspiegeln, welche wiederum weitere Unterknoten enthalten. Für eine detailliertere Analyse wird ein AST mit zusätzlichen Assoziationen zwischen den Knoten angereichert. So hat zum Beispiel ein Feldknoten eine Assoziation zum Klassenknoten seines Typs.

2.2.1 DOM/AST

Das OTDT (*Object Teams Development Tooling*) der Eclipse-Plattform implementiert eine Entwicklungsumgebung für ObjectTeams und besteht aus einer Menge von Plugins.
Das Paket vom `OTDT Core Plugin`[1] `org.eclipse.jdt.core.dom` bietet eine Sammlung von Klassen, die mit Hilfe des Parsers (und Checkers) es möglich machen, aus dem Quellcode eines OT-Programms ein strukturiertes Dokument (DOM/AST) zu erzeugen. Zudem können alle im DOM/AST vorkommenden Typen zu *Bindings* aufgelöst werden. *Bindings* sind Datenstrukturen, die eine eindeutige Identität eines benannten Typs darstellen, dessen Daten repräsentieren und Verbindungen zu den anderen Programmteilen enthalten (siehe 3.2.1.2).

2.2.1.1 Schwierigkeiten der DOM/AST-Repräsentation

Da der OT-Compiler, dessen Parser für die DOM/AST-Generierung verwendet wird, in Java implementiert ist, müssen die Spracherweiterungen von OT auf die Java-Mittel angepasst werden. So besteht zum Beispiel eine Rolle aus einem Klassen- und einem Interfaceteil und ihre Features sind entsprechend

[1] ist eine Erweiterung vom `Java Development Tooling Core Plugin`

auf beide Teile verteilt. Solche internen Strukturen und andere Zusatzinformationen für den Compiler werden vom DOM/AST wiedergegeben.
Weil die Sprache komplex ist, erweist sich die Analyse eines OT-Programms als schwierig, wenn die Suche aller seiner Bestandteile durchgeführt werden soll. Man ermittle zum Beispiel alle Methodensignaturen einer Rolle, folglich die von den direkt implementierten Methoden, von explizit und implizit geerbten Methoden und die in den Superinterfaces angegeben Signaturen.
Die Traversierung durch eine AST-Struktur erfolgt mittels des Designmusters *Visitor*, sodass für jede Regel der Analyse ein eigener *Visitor* geschrieben werden muss.

Gewünschte Datenrepräsentation Für eine OT-Programmanalyse ist eine Datenrepräsentation geeignet, wenn sie strukturiert, einfach durchzusuchen, zu transformieren und zu erweitern ist, und mit der sich ein AST aufbauen lässt.
Im dieser Arbeit ist ein Umweg über ein anderes Format - XML gewählt. Im Folgenden wird die Sprache näher vorgestellt und ihre Vorteile als Datenrepräsentation erläutert.

2.3 XML

XML (*Extensible Markup Language*) [15] ist eine textbasierte Auszeichnungssprache. Sie wurde vom *World Wide Web Consortium*[2](W3C) [12] entwickelt.

2.3.1 Entstehung von XML

Der "Urvater" von XML ist SGML (*Standard Generalized Markup Language*), eine Metasprache, mit der die Erstellung von neuen Auszeichnungssprachen möglich ist. Der Zweck von Auszeichnungssprachen ist es, für Daten eine logische Struktur zu bilden, welche die Datenverarbeitung vereinfacht.
Die komplexen Mechanismen von SGML machen die Sprache mächtig aber auch kompliziert. XML stellt eine vereinfachte Version von SGML dar. Der Grund für die Entwicklung von XML war der Wunsch nach einem flexiblen und einfachen Mechanismus, Daten systemunabhängig zu repräsentieren und auszutauschen.
Mit XML lassen sich Auszeichnungssprachen definieren, die in einer strukturierten Form Daten beliebigen Typs beschreiben können. Darüber hinaus

[2]W3C ist eine nicht kommerzielle Organisation, deren Aufgabe darin besteht, Standards und Technologien für das *World Wide Web* zu entwickeln.

```
<?xml version="1.0" encoding="UTF-8"?>
<share_index name="DAX">
     <quotation date="1.8.05 5:51 PM">
         <name ISIN="DE0005003404">Adidas</name>
         <price>148.31</price>
         <difference value="-1.09" percent="-0.7"/>
     </quotation>
     <quotation date="1.8.05 5:51 PM">
         <name ISIN="DE0007664005">Volkswagen</name>
         <price>43.96</price>
         <difference value="-0.80" percent="-1.8"/>
     </quotation>
</share_index>
```

Listing 2.1: *Aufbau eines XML-Dokument*

kann die Gültigkeit von XML-Dokumenten bereits vor ihrer Bearbeitung überprüft werden, wenn ein Standard für diese Struktur vereinbart wurde.

2.3.2 XML-Dokument

Tags Ein XML-Dokument enthält spezielle Steueranweisungen *tags*, deren Namen nicht fest sind und vom Benutzer nach seinen Anforderungen gewählt werden können. Die Menge aller *tags* bildet eine spezielle Auszeichnungssprache. *Tags* werden syntaktisch durch spitze Klammern $< \ldots >$ gekennzeichnet.

Elemente Jedes XML-Dokument enthält ein oder mehrere Elemente, die einen durch den Namen festgelegten Typ haben und aus anderen Elementen bestehen können. Die Abgrenzung von einem Element wird vom Anfang-*tag* `<tagname>` und Ende-*tag* `</tagname>` bestimmt. Ein sogenanntes leeres Element kann keine weiteren Elemente enthalten und wird nur mit einer Markierung `<tagname/>` ausgezeichnet. Ein XML-Dokument besitzt nur ein Wurzelelement, das alle anderen Elemente in sich birgt.
Elemente können auch Attribute enthalten - zusätzliche Informationen, die als Teil des Anfang-*tags* in seinen spitzen Klammern eingeschlossen sind.

Beispiel Listing 2.1 zeigt ein XML-Dokument, das Aktienkurse eines speziellen Indexes beschreibt. Die erste Zeile der Datei legt fest, dass es sich um ein XML-Dokument handelt, sie beinhaltet die Version von XML und die Art der Kodierung und gehört zum XML-Prolog. Das Wurzelelement wird in diesem Beispiel vom tag `share_index` ausgezeichnet, es enthält zwei weitere

Elemente, die durch den *tag* `quotation` markiert werden, und besitzt außerdem ein Attribut `name` mit dem Wert `DAX`. In Zeile 11 ist ein leeres Element angegeben, dass nur Attribute beinhaltet.

Außerdem ist es möglich, ein XML-Dokument mit Kommentaren zu versehen und mit Anweisungen oder Informationen für die Applikationen (*processing instructions*) auszurüsten, welche diese XML-Daten später verarbeiten. Um Namenskonflikte der selbst definierten *tag*-Namen zu vermeiden, besteht in XML die Möglichkeit, Namensräume für *tags* (*namespaces*) zu definieren.

Wohlgeformtes Dokument Ein XML-Dokument wird als wohlgeformt bezeichnet, wenn es syntaktisch korrekt ist.

Gültiges Dokument Ein XML-Dokument ist gültig bezüglich einer konkreten Auszeichnungssprache, wenn es den Regeln der Grammatik dieser Sprache, die in einer DTD (siehe 2.3.2.1) festgelegt wurden, entspricht. Es besteht die Möglichkeit, in einem XML-Dokument eine DTD-Datei mittels Anweisung `<!DOCTYPE ...>` anzugeben, in der die Elemente aus dem XML-Dokument definiert sind.

2.3.2.1 Typ eines XML-Dokuments (DTD)

Eine DTD (Document Type Definition) definiert die Syntax einer Auszeichnungssprache und legt die Struktur eines Typs von XML-Dokumenten fest. Sie beschreibt, welche Elemente und Attribute in einem Element vorkommen dürfen und von welchem Typ sie sein sollen und verwendet die Syntax der EBNF-Grammatik. Zudem kann die Reihenfolge des Vorkommens der Elemente festgelegt und ein Hinweis auf ihre Anzahl mit +, * oder ? gegeben

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!ELEMENT share_index (quotation*)>
3  <!ATTLIST share_index name CDATA #REQUIRED>
4    <!ELEMENT quotation (name, price, difference)>
5    <!ATTLIST quotation date CDATA #IMPLIED>
6      <!ELEMENT name (#PCDATA)>
7      <!ATTLIST name ISIN CDATA #REQUIRED>
8      <!ELEMENT price (#PCDATA)>
9      <!ELEMENT difference EMPTY>
10     <!ATTLIST difference value CDATA #REQUIRED
11                          percent CDATA #REQUIRED>
```

Listing 2.2: *Beispiel einer DTD*

werden. Das Vorkommen eines Attributs wird als erforderlich oder optional definiert, sein Wert kann als konstant oder als Defaultwert festgelegt werden. Die Definition einer DTD ist nicht erforderlich aber empfehlenswert, um der Erzeugung ungültiger XML-Strukturen vorzubeugen.

Beispiel In Listing 2.2 ist eine DTD für das bereits vorgestellte XML-Dokument aus Listing 2.1 aufgestellt. Zeile 2 definiert zum Beispiel ein XML-Element: Ein Index-Element kann mehrere (keine oder viele) Notierungs-Elemente enthalten. In Zeile 3 ist die Attributliste des Elements Index angegeben, die ein erforderliches Attribut vom Typ Zeichenkette beinhaltet. Zeile 6 sagt aus, dass das Element `name` aus reinem Text (*parsed character data*) besteht. Zeile 9 legt fest, dass das zu definierende Element keine Unterelemente enthalten kann.

2.3.3 Fazit

XML hat eine generische Syntax für die Erzeugung neuer Auszeichnungssprachen. Der logische Aufbau eines XML-Dokuments kann als eine baumförmige Struktur betrachtet werden, in der Elemente dieses Dokuments Knoten des Baumes bilden und in einem Wurzelknoten enthalten sind.
Folglich lässt sich mit XML eine Sprache definieren, die den abstrakten Syntaxbaum eines OT-Programms als XML-Repräsentation beschreibt. Aus einem DOM/AST kann mit Hilfe von entsprechenden Java APIs ein XML/AST erzeugt werden, bei dem interne Zusatzinformationen für den Compiler ausgeblendet sind.
Ein Nachteil der XML-Repräsentation liegt in der Isolierung vom Compiler. Wird zum Beispiel ein einziges Team nach XML umgewandelt, wobei nur der Quellcode des Teams vom XML/AST wiedergegeben wird, so besteht keine Möglichkeit des Zugriffs auf die geerbten Features seiner Superklasse.
Für die Typen in einem AST kann zwar eine eindeutige Identifikation im XML-Format vereinbart werden, so dass sie im XML-Dokument gefunden werden können, dabei müssen sie aber mitkonvertiert werden. Da sie andere Typen enthalten, die wiederum andere Abhängigkeiten aufweisen können, müssen alle möglichen Klassen (auch Bibliotheksklassen) nach XML transformiert werden.
Eine andere Lösung besteht in der Generierung von vollständigen Informationen einer Klasse, um die Suche nach geerbten Features in anderen Klassen zu vermeiden. Dadurch wird die Suche nach Bestandteilen einer Klasse wesentlich erleichtert und eine optimale Datenrepräsentation für analysierende Werkzeuge erzeugt.

Für die Manipulation der XML-Repräsentation steht eine Vielfalt von XML verarbeitenden Technologien zur Verfügung.

2.3.4 Technologien auf der Basis von und rund um XML

Im Folgenden werden Sprachen vorgestellt, die für die Verarbeitung der mit XML strukturierten Daten vom W3C entwickelt wurden und in dieser Arbeit für die Manipulation der XML-Repräsentation eingesetzt werden.

2.3.4.1 Zugriff auf die Knoten mit XPath

Die Sprache XPath (*XML Path*) bietet eine effektive Syntax für die Adressierung von XML-Daten. XPath gibt eine Pfadbeschreibung für den Zugriff auf die Knoten einer XML-Struktur und wird von anderen XML-Technologien benutzt.
In der Sprache XPath wird ein XML-Dokument als ein Baum angesehen, dessen Knoten nach sieben verschiedenen Arten unterschieden werden: Wurzelknoten, Element, Attribut, Text, Kommentar, Verarbeitungsanweisung und Namensraum. Entlang der Baumhierarchie bewegt man sich durch die Beschreibung eines Pfades mithilfe der Dateisystemnotation von Unix(`/`, `.`, `..`).
Ein XPath-Ausdruck spezifiziert mittels Operatoren, Funktionen, Platzhaltern und anderen Mechanismen ein Muster, das eine Menge von Knoten auswählt.
Im Weiteren werden zwei Beispiele von XPath-Ausdrücken erläutert, um die Möglichkeiten von XPath näher vorzustellen. Einzelheiten über diese Sprache können in der XPath-Spezifikation [16] nachgelesen werden.

- `//quotation[./name = "Adidas"]`
  Der Ausdruck `quotation` wählt alle Kinderelemente `quotation` vom Kontextknoten aus. Mit einem Doppelschrägstrich `//` wird eine beliebige Hierarchieebene erreicht, das bedeutet, dass mit `//quotation` alle im Kontextknoten vorkommenden Notierungselemente ausgewählt werden. Um den Ausdruck zu spezifizieren, werden eckige Klammern (`[ ]`) verwendet, in denen eine Bedingung angegeben werden kann. In diesem Fall werden nur die Elemente ausgewählt, die ein direktes (`./`) Unterelement `name` besitzen, das den Text Adidas enthält.

- `count(distinct-values( /share_index[@name]/quotation/name/@ISIN) )`
  Der Pfad beginnt mit einem Schrägstrich (`/`), also mit einem absoluten Pfad zum Wurzelelement. Mit einem `@`-Zeichen werden Attribute identifiziert. Der im Beispiel spezifizierte Pfad führt zu ISIN-Attributen

von Elementen `name`, die in einem Element `quotation` enthalten sind, das wiederum in einem `share_index` Element, das ein Attribut `name` besitzt, enthalten ist. Die Funktion `distinct-values` entfernt das Mehrfachvorkommen gleicher Elemente und die Funktion `count` gibt die Anzahl der ausgewählten Knoten zurück.

2.3.4.2 Transformation von XML mit XSLT

Die Sprache XSLT (*XML Stylesheet Language Transformations*) [17] erlaubt die Umwandlung von XML-Dokumenten in andere Formate. Eine in XSLT formulierte Transformation besteht aus Regeln für die Umwandlung eines XML-Dokuments in ein Ergebnisdokument, das im XML-Format oder in einem anderen Textformat wie zum Beispiel HTML [14] erzeugt wird.
XSLT definiert Schablonen, mit deren Hilfe die Umwandlung stattfindet. Eine XSLT-Schablone enthält eine Menge von Formatierungsinstruktionen, die an den durch einen XPath-Ausdruck ausgewählten Knoten angewendet werden. Eine XSLT-Schablone könnte folgendermaßen aussehen:

```
1 <xsl:template match="share_index">
2   <html>
3     <xsl:apply-templates/>
4   </html>
5 </xsl:template>
```

Diese Schablone entspricht einem Element von Typ `share_index`. Das Ergebnis der Umwandlung dieses Elements stellt ein HTML-Dokument dar, welches die Auswertung seiner Kinderknoten enthält.
Um XSLT näher vorzustellen, ist in Listing 2.3 ein Auszug aus einer XSLT-Datei dargestellt, die eine Umwandlung der aus 2.3.2 bekannten Auszeichnungssprache nach HTML definiert. Dabei sollen Aktieninformationen, sortiert nach dem Namen der Aktie, in einer Tabelle dargestellt werden. Das Ergebnis der Umwandlung sollte in einem Browser wie folgend aussehen:

DAX

Name	ISIN	Price	+/-	%
Adidas	DE0005003404	148.31	-1.09	-0.7
Volkswagen	DE0007664005	43.96	-0.80	-1.8

Abbildung 2.1: *Darstellung der mit XSLT umgewandelten XML-Datei in einem Browser*

```
<table border = "1">
  ...
  <xsl:for-each select = "quotation">
    <xsl:sort select = "quotation/name" order="ascending"/>
    <tr>
    <td><xsl:value-of select = "name"/></td>
    <td><xsl:value-of select = "name/@ISIN"/></td>
    <td align = "right"><xsl:value-of select = "price"/></td>
    <td align = "right">
      <xsl:value-of select = "difference/@value"/></td>
    <td align = "right">
      <xsl:value-of select = "difference/@percent"/></td>
    </tr>
  </xsl:for-each>
</table>
```

Listing 2.3: *Auszug aus einer XSLT-Datei*

XSLT-Anweisungen aus Listing 2.3 Um eine bestimmte Anweisungsfolge auf eine Menge von Knoten anzuwenden, wird die XSLT-Anweisung `for-each` verwendet. Mit der XSLT-Anweisung `sort` ist es möglich, Elemente innerhalb von `for-each` anhand eines bestimmten Kriteriums zu sortieren. Um den *String*-Wert eines Knotens zu erzeugen, wird die XSLT-Anweisung `value-of` verwendet, die den Knoten spezifizierenden Ausdruck auswertet und das Ergebnis in einen *String* umwandelt.

2.3.4.3 Suche in XML mit XQuery

XML Query (XQuery) [18] ist eine Abfragesprache für XML, die auch allgemeine Verarbeitung von XML erlaubt. Sie stellt flexible Abfragemöglichkeiten zur Verfügung, Daten aus XML-Dokumenten zu extrahieren und ermöglicht damit einen schnellen Zugriff auf bestimmte Informationen einer XML-Struktur.
XQuery ist eine funktionale Sprache (keine XML-Sprache). Jede Abfrage wird als ein Ausdruck formuliert, der auszuwerten ist. Ausdrücke der XQuery können flexibel miteinander kombiniert werden.
Jede Eingabe und jede Ausgabe einer Abfrage ist eine Sequenz von atomaren, typisierten Werten[3] oder Knoten einer XML-Struktur. Die Klassifizierung von Knoten nach sieben verschiedenen Arten wurde aus XPath (siehe 2.3.4.1) übernommen[4]. Für den Zugriff auf die Knoten werden in XQuery

[3] XQuery verwendet Datentypen nach der W3C Recommendation, einem vom W3C festgelegten Standard.
[4] Der Wurzelknoten wird in XQuery als Dokumentknoten bezeichnet.

XPath-Ausdrücke eingesetzt. Sollen neue XML-Knoten mit XQuery erzeugt werden, kann dafür sowohl die XML-Syntax als auch ein für jede Knotenart eigener Konstruktor verwendet werden.
XQuery-Ausdrücke kombinieren oft Informationen von einer oder mehreren XML-Quellen und konstruieren somit eine neue Struktur. Die mächtigsten Hilfsmittel dafür sind die FLWOR-Ausdrücke[5]. FLWOR ist ein Akronym, das von den Anweisungen abgeleitet ist, die in einem Ausdruck vorkommen können.

- **for** - verbindet eine Variable mit einem Ausdruck. Dabei wird eine Liste von Tupeln von Bindungen dieser gegebenen Variable an eins der Elemente erzeugt, die beim Auswerten des ihr entsprechenden Ausdrucks entstehen
- **let** - bindet eine Variable an das Gesamtergebnis eines gegebenen Ausdrucks. Sie fügt diese Bindung zur Liste von mit `for` erzeugten Tupel hinzu oder erzeugt einen einzigen Tupel mit dieser Bindung, wenn `for` im FLWOR-Ausdruck nicht vorkommt
- **where** - filtert die mit `for` oder `let` erzeugten Tupel, indem für jeden Tupel geprüft wird, ob eine gegebene Bedingung erfüllt wird
- **order by** - sortiert die Tupel der Liste nach einem gegebenem Kriterium auf- oder absteigend
- **return** - konstruiert das Ergebnis des FLWOR-Ausdrucks für alle entstandenen Tupel

Jede Anweisung eines FLWOR-Ausdrucks wird tupelbezogen definiert, deshalb beginnt ein FLWOR-Ausdruck mit einer oder mehreren `for`- oder `let`-Anweisungen, da sie die Tupel erzeugen. Eine `where`- oder `order by`- Anweisung ist in einem FLWOR-Ausdruck optional, während eine ergebnisbildende `return`-Anweisung unbedingt erforderlich ist.

Beispiel In Listing 2.4 ist eine Abfrage formuliert, in der viele der oben erwähnten Mechanismen verwendet werden. Die Abfrage erzeugt ein XML-Element `flops`, in dem alle Notierungen aus der angegebenen XML-Datei, absteigend nach den Namen sortiert, aufgelistet sind, deren Kurs im Minus zum Vorkurs steht. Es sollen nur Namen der Aktien in einem neuen Element `stock`, das mit dem Konstruktor `element stock { }` erzeugt ist, erscheinen.

[5]FLWOR wird wie *flower* ausgesprochen.

```
1  (: Query :)
2  <flops>
3  {
4      for $q in doc("share_index.xml")//quotation
5      let $names := $q/name
6      where $q/difference/@value < 0
7      order by $names descending
8      return
9          element stock {
10             $q/name/text()
11         }
12 }
13 </flops>
14
15 (: A possible result of the query
16 <flops>
17    <stock>Volkswagen</stock>
18    <stock>Adidas</stock>
19 </flops>
20 :)
```

Listing 2.4: *Abfrage in XQuery und ihr Ergebnis als Kommentar*

Im Beispiel wird die XPath-Funktion `doc(string? $uri)` verwendet, dabei wird ein Dokument entsprechend seinem URI (*Universal Resource Identifier*) geladen. Die Beschreibung des XPath-Ausdrucks `//` ist in 2.3.4.1 zu finden. Wenn die Eingabedatei dieser Abfrage das XML-Dokument aus Listing 2.1 beinhaltet, dann ist ihre Ausgabe die XML-Struktur aus Zeilen 16 bis 19 in Listing 2.4, die in einem XQuery-Kommentar enthalten ist.

Darüber hinaus gehören zu dem Rüstzeug von XQuery arithmetische und Vergleichs-Operatoren, Quantoren, Fallunterscheidung und vielerlei eingebaute Funktionen. Außerdem besteht die Möglichkeit, eigene Funktionen zu definieren. Mehr darüber und über die bereits erwähnten Konzepte von XQuery kann in [19] nachgelesen werden.

3 Analysierende Werkzeuge für ObjectTeams

Die Werkzeugunterstützung ist in der heutigen Software-Entwicklung unverzichtbar. Für jede Phase des Software-Lebenszyklus existieren entsprechende Werkzeuge, die Methoden und Erfahrungen unterstützen, ohne deren Einsatz die Entwicklung komplexer Software nicht planmäßig und zielgerecht, sondern ad hoc stattfinden werden könnte.

Werkzeug-Begriff Eine Definition des Begriffs Werkzeug im Kontext der Software-Entwicklung lautet wie folgt: „Programme, die die Herstellung, Prüfung, Wartung und Dokumentation von Programmen vereinfachen, beschleunigen oder in ihrer Qualität verbessern" [23].
Diese allgemein gehaltene Definition schließt alle möglichen Arten der Werkzeuge zur Unterstützung der Software-Entwicklung ein. Im folgenden Abschnitt wird ein Überblick über die Klassifizierung von Werkzeugen gegeben.

3.1 Werkzeugklassifizierung

Werkzeuge lassen sich nach [22] im Wesentlichem durch die unterstützten Tätigkeiten unterscheiden, dabei werden zwei Kategorien von Werkzeugen hervorgehoben: tätigkeits- oder phasenübergreifende Werkzeuge und tätigkeits- oder phasenspezifische Werkzeuge.

3.1.1 Tätigkeitsübergreifende Werkzeuge

Die tätigkeitsübergreifenden Werkzeuge werden in verschiedenen Entwicklungsphasen, möglicherweise im gesamten Entwicklungsprozess verwendet. Ein Beispiel dafür stellt eine integrierte Entwicklungsumgebung dar, dabei handelt es sich um ein Werkzeug, welches unter anderem das Implementieren, Debuggen und Testen unterstützt.
Die tätigkeitsübergreifenden Werkzeuge können wie folgt gegliedert werden:

- Werkzeuge für die Dokumentation
- Werkzeuge für die Verwaltung
- Werkzeuge für das Projektmanagement
- Software-Entwicklungsumgebungen

Im Folgenden werden diese Kategorien näher erläutert.

3.1.1.1 Werkzeuge für die Dokumentation

Ein Dokumentationswerkzeug generiert aus vielen Software-Bestandteilen und ihrer einzelner Beschreibungen und Kommentaren ein Dokument, das eine Hilfestellung zum Umgang mit dem Gesamtprodukt bietet. Im Abschnitt 4.1 wird näher auf die Anforderungen an diese Werkzeuge eingegangen.

3.1.1.2 Werkzeuge für die Verwaltung

Zu den Aufgaben der Verwaltungswerkzeuge gehören die Grundfunktionen wie Speichern, Öffnen, Ersetzen, Archivieren und Löschen von Dokumenten. Weiterhin ermöglichen sie die Suche nach den Bestandteilen oder Typen von Dokumenten.
Zur dieser Kategorie gehören auch fortgeschrittene Werkzeuge wie *Datenlexikon*, das die Elemente von Anwendungsmodellen verwaltet, *Projektbibliothek*, die die Verwaltung von allen bei der Entwicklung entstehenden projektbezogenen Dokumenten regelt, *Versionsverwaltungssystem*, das den Umgang mit verschiedenen Versionen eines Dokuments erleichtert.

3.1.1.3 Werkzeuge für das Projektmanagement

Werkzeuge dieser Kategorie werden im Laufe des gesamten Entwicklungsprozesses eingesetzt. Dazu gehört die Gruppe der Planungswerkzeuge, die eine Schätzung von persönlichen Ressourcen (Zeit), vom Aufwand oder von der Anzahl der Fehler ermöglichen und die Einhaltung der festgelegten Zielwerte kontrollieren. Außerdem fallen darunter die Werkzeuge für Qualitätssicherungsmaßnahmen wie Reviews und Inspektionen und zur Generierung von Berichten.

3.1.1.4 Software-Entwicklungsumgebungen

Eine Software-Entwicklungsumgebung beinhaltet mehrere Werkzeuge, die über eine gemeinsame Benutzeroberfläche benutzt werden. Werkzeuge einer

Entwicklungsumgebung bauen auf einander auf bzw. ergänzen einander, so dass eine Unterstützung für mehrere Phasen der Entwicklung geboten wird.

3.1.2 Tätigkeitsspezifische Werkzeuge

Die tätigkeitsspezifischen Werkzeuge unterstützen den Entwickler bei einer spezifischen Tätigkeit. So sind Werkzeuge wie zum Beispiel ein Programmeditor und ein Compiler in der Implementierungsphase unverzichtbar und werden beim Programmieren eingesetzt.
Die tätigkeitsspezifischen Werkzeuge können je nach der entsprechenden Entwicklungsphase wie folgt unterteilt werden:

- Werkzeuge zur Analyse und Modellierung
- Werkzeuge für die Systemspezifikation
- Werkzeuge für den Software-Entwurf
- Programmier- und Implementierungswerkzeuge
- Testwerkzeuge und weitere Werkzeuge zur Qualitätssicherung

Im Folgenden wird näher auf die Werkzeuge zur Qualitätssicherung eingegangen.

3.1.3 Qualität von Software

Nach ISO[1] wird die Qualität durch die folgende Definition beschrieben: „Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Dienstleistung, die sich auf deren Eignung zur Erfüllung festgelegter oder vorausgesetzter Erfordernisse bezieht".
Nach dieser Definition ist die Qualität von Software von ihren Eigenschaften abhängig. Zu solchen Qualitätseigenschaften zählen unter anderem: Korrektheit, Performance, Stabilität, Robustheit, Lesbarkeit, Verständlichkeit, Wartbarkeit, Erweiterbarkeit und Fehlerfreiheit. Die Qualität von oben erwähnten Eigenschaften wird durch auftretende Fehler und strukturelle Schwächen negativ beeinflusst. Unter einer strukturellen Schwäche wird nach [21] ein Teil eines Softwaresystems bezeichnet, dessen Struktur einen Entwickler im Ändern des Systems hindert. Eine schwache Struktur erschwert unter anderem das Verständnis des Systems. Folglich müssen die Fehler und Schwächen vermieden und gefunden werden, um eine Qualitätssteigerung zu erreichen.

[1] *International Organization for Standardization*

Bezüglich des Zeitpunkts der Maßnahmen kann die Qualitätssicherung in zwei Arten unterteilt werden: konstruktive und analytische Qualitätssicherung. Während es bei der konstruktiven Qualitätssicherung darum handelt, wie ein System realisiert werden soll, dass kaum (keine) Probleme, Fehler auftreten, untersuchen die analytischen Qualitätssicherungsmaßnahmen ein bereits realisiertes Produkt auf die Fehlerfreiheit seiner Umsetzung.
Das Fehlerbeheben in den späteren Entwicklungsphasen insbesondere beim Testen ist schwierig und zeitaufwendig. Je früher die Fehler eliminiert sind, desto fehlerfreier wird das Produkt.

3.1.4 Vermeidung von Fehlern, Schwächen als Qualitätssicherung

Für verschiedene Entwicklungsphasen können Strategien angegeben werden, die eine systematische Vermeidung von Fehlern bzw. strukturellen Schwächen zum Ziel erklären. Solche Strategien basieren oft auf Erfahrungswerten oder auf historischen Daten.
Auf der Designebene können Fehlentscheidungen verhindert werden, indem eine spezifische, vordefinierte, erfahrungsgemäß einfache und elegante Lösung, ein sogenanntes Entwurfsmuster (*design pattern*) verwendet wird. Entwurfsmuster sind Designvorschläge für allgemein bekannte Entwurfsprobleme, die sich als gute Lösungen bewährt haben. Als Gegenteil dazu wurden auch sogenannte Antimuster definiert. Sie beschreiben schlechte Lösungen, deren Einsatz verhindert werden soll.
Auf der Code-Ebene lassen sich die Fehler bzw. Schwächen durch die Einhaltung von Codekonventionen vermeiden, welche bestimmte Regeln für die Struktur der Implementierung vorschreiben. Die Einhaltung dieser Regeln führt zu einer besseren Lesbarkeit, Wartung und Weiterentwicklung.
Es ist schwierig einen Werkzeugtyp für die Fehlervermeidung, eine konstruktive Qualitätssicherungsmaßnahme anzugeben, weil diese Tätigkeit unmöglich zu automatisieren ist. Werkzeuge, welche die Einhaltung irgendwelcher Regeln kontrollieren, gehören zur Kategorie der Fehlersuche und folglich zu analytischen Qualitätssicherungsmaßnahmen.

3.1.5 Suche nach Fehlern, Schwächen als Qualitätssicherung

Für die Fehler- und Schwächensuche lassen sich Werkzeuge vereinfacht in statische und dynamische Werkzeuge einteilen. Statische Analysewerkzeuge

betrachten den Quellcode von Programmen, ohne sie dabei auszuführen, wohingegen dynamische Werkzeuge die Programmausführung analysieren.

3.1.5.1 Werkzeugtypen

Die Werkzeuge, welche die Fehler-, Schwächensuche unterstützen, lassen sich wie folgt unterscheiden:

- Prüfwerkzeuge
- Kritik-Werkzeuge
- Metriken

Im Folgenden wird detailliert auf die einzelnen Kategorien eingegangen.

Prüfwerkzeuge Prüfwerkzeuge, die auch als Stilanalysatoren bezeichnet werden, analysieren statisch den Quellcode, wobei die Einhaltung von vordefinierten Regeln in einem Programmsystem überprüft wird. Regelverletzungen, die gefunden werden, werden dem Benutzer mit der Angabe der Position mitgeteilt oder können automatisch ausgebessert werden.
Diese Regeln müssen im Voraus definiert werden und beschreiben Eigenschaften auf der niedrigsten Abstraktionsebene, die beim Implementieren zu beachten sind. Häufig werden Codekonventionen und typische Einschränkungen von Programmiersprachen als Regeln für Prüfungswerkzeuge festgelegt.

Kritik-Werkzeuge Kritik-Werkzeuge sind auch statische Analysatoren, welche vordefinierte Regeln verwenden. Die Abstraktionsebene der verwendeten Regeln unterscheidet sich dabei von der Ebene bei Prüfungswerkzeugen: Sie betreffen meistens das Design und werden deshalb auch Design-Kritiken genannt.
Solche Kritiken stellen oft Vorschläge dar, die nicht auf einen Fehler hinweisen, sondern eine Empfehlung abgeben, wie eine komplexe, schwache oder kritische Stelle im Design verbessert werden kann. Der Entwickler entscheidet, ob er diesen Vorschlag befolgt. Kritik-Werkzeuge ermitteln normalerweise unvollständige Strukturen, unnötige Komplexitäten aber auch Designfehler oder Schwachstellen, die zu einem Problem führen können.

Metriken Messwerkzeuge basieren auf der quantitativen statischen Analyse, die strukturelle Schwächen eines Software-Produkts ermittelt. Dazu werden quantitative Merkmale oder Maße definiert, die für die Qualität von Software-Systemen entscheidend sind. Bei der Analyse wird der Messwert

von diesem Maß ermittelt. Der Benutzer muss einen Wert (Wertebereich) für dieses Maß festgelegen, anhand dem er entscheiden kann, ob eine strukturelle Schwäche vorliegt.
Maße, die von Metriken verwendet werden, sind sehr spezifische Eigenschaften, die entweder intuitiv (*Lines of Code*) gewählt werden oder statistisch begründet werden müssen.
Aufgrund des hohen Automatisierungsgrades kann die Qualität gerade größerer Software-Systeme objektiv geschätzt werden.
In einer parallelen Diplomarbeit von Michael Krüger [24] handelt es sich darum, wie Metriken für ObjectTeams definiert werden können, um die Qualität der damit geschriebenen Programme zu messen.

3.2 Toolbox für ObjectTeams

Werkzeuge bieten eine große Unterstützung für Entwickler und tragen zur Verbesserung der Software-Qualität bei. Die meisten von ihnen sind programmiersprachen- bzw. programmierparadigmaspezifisch. Wird eine Sprache um neue Konzepte erweitert, können die alten Werkzeuge nur teilweise und nicht flächendeckend im Entwicklungsprozess angewendet werden. Da ObjectTeams/Java die objektorientierte Sprache Java um die Aspektorientierung erweitert, müssen die Werkzeuge neu entwickelt werden, damit sie mit der objekt- und der aspektorientierten Welt umgehen können.
Einen großen Beitrag zur Unterstützung der OT-Entwickler leistet eine nach Eclipse [10] integrierte Entwicklungsumgebung für ObjectTeams [11]. Dort sind viele Werkzeuge wie zum Beispiel ein komfortabler Editor, ein inkrementeller Compiler, bequeme Verwaltung mit Navigationsmöglichkeiten und einer guten Suchfunktionalität vereint.
ObjectTeams ist eine ziemlich komplexe Sprache, die eine junge Technik realisiert. Deswegen muss der Umgang mit dieser Sprache besonders gefördert werden. Ein Entwickler, der auf OT setzt, möchte durch eine bessere Modularisierung die Lesbarkeit, Wartbarkeit, Wiederverwendung und damit die Qualität seiner Software erhöhen. Besonders neuen Entwicklern fehlen oft die Erfahrungswerte, welche die Qualität von OT-Programmen betreffen. Wünschenswert sind daher qualitätssichernde Werkzeuge, die die Qualität eines OT-Programms prüfen, beurteilen oder messen können.
Fehlentscheidungen, strukturelle Schwächen und andere Fehler manifestieren sich schließlich im Quellcode, deshalb ist es von Nutzen, dort nach Mängeln zu suchen. Wie oben geschildert, kommt dafür die statische Codeanalyse zum Einsatz. Von Werkzeugen, die darauf basieren, werden in der Regel die Programmdaten traversiert, um nach bestimmten Eigenschaften zu suchen,

Daten umzuwandeln oder auszuwerten.
Aus den oben geschilderten Überlegungen erscheint die Idee eines gemeinsamen Basiskerns für die Entwicklung der Analysewerkzeuge sinnvoll. Zu den Aufgaben dieser Toolbox gehört die Bildung einer Datenrepräsentation, die gut strukturiert, einfach und flexibel zu manipulieren und schnell zu verstehen ist. Außerdem können die gemeinsamen Aufgaben der Analyse zentral implementiert werden. Vorteilhaft ist auch eine Eingliederung der Toolbox in die in Eclipse integrierte Entwicklungsumgebung für ObjectTeams.
Da eine gute Datenrepräsentation für die Toolbox eine zentrale Rolle spielt, wird im Folgenden darauf eingegangen, wie die Programmdaten in Eclipse repräsentiert werden.

3.2.1 Datenrepräsentation in Eclipse

Eclipse selbst und die Entwicklungsumgebung für OT sind vollständig in Java geschrieben. Dateien mit der Endung .java werden in Eclipse von Objekten der Klasse `org.eclipse.jdt.core.ICompilationUnit` repräsentiert. Um die grammatikalische Struktur eines OT-Programms zu verstehen und seine interne Struktur abzubilden, wird ein abstrakter Syntaxbaum (AST) benötigt.

3.2.1.1 DOM/AST

Das Paket `org.eclipse.jdt.core.dom` aus dem `OTDT Core Plugin` stellt alle notwendigen Klassen zur Verfügung, um aus einer Datei mit der Endung .java einen abstrakten Syntaxbaum zu erzeugen.
Dieser AST hat einen Wurzelknoten (`CompilationUnit`), welcher die Datei repräsentiert und Unterknoten für die in der Datei deklarierten Typen enthält. Die Unterknoten eines Typknotens sind Knoten für Rollen, Methoden, Methodenbindungen u.Ä. und können selbst wiederum Unterknoten besitzen. Zudem kann der AST Informationen zu JavaDoc-Kommentaren enthalten.
In der Abbildung 3.1 wird an einem Beispiel veranschaulicht, wie ein AST aus einer OT-Datei gebildet wird.
Der AST wird mit Hilfe eines *ASTParsers* nach den Regeln der OT-Grammatik erzeugt. Dabei besteht die Möglichkeit den Parser so zu konfigurieren, dass die verschiedenen Namen und Typen, die im AST vorkommen, zu *Bindings* aufgelöst werden können.

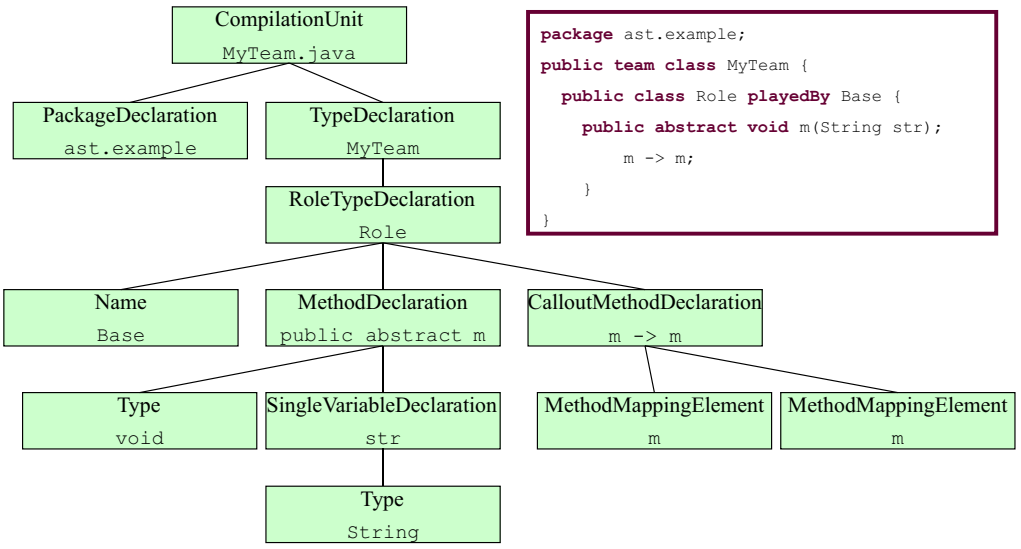


Abbildung 3.1: *Beispiel eines AST*

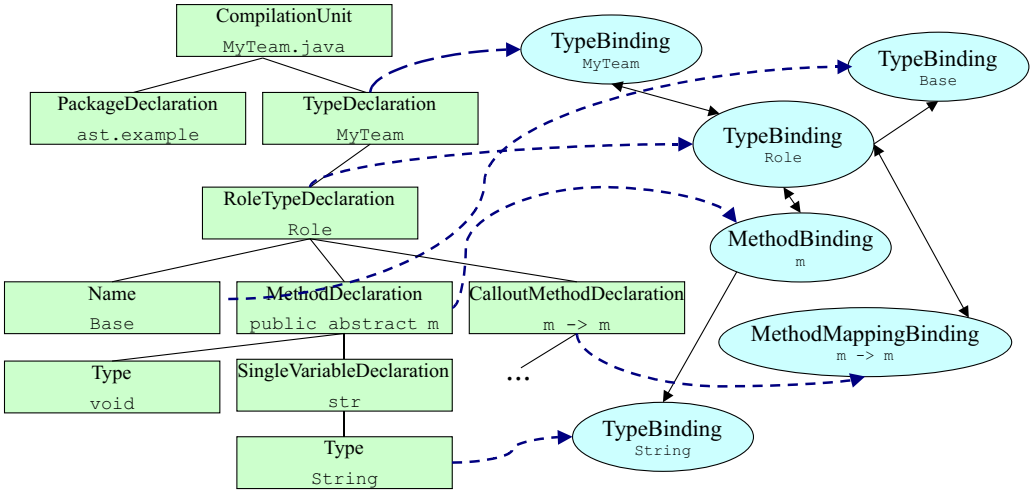


Abbildung 3.2: *AST und Binding-Graph*

3.2.1.2 Binding

Ein *Binding* repräsentiert eine durch einen Namen gekennzeichnete Entität. Beispiele für solche Entitäten in der Sprache ObjectTeams/Java sind Typen, Methoden, Methodenbindungen. *Bindings* sind mit Verbindungen zwischen den verschiedenen Teilen des Programms angereichert, sodass mit deren Hilfe eine Programmstruktur tiefer analysiert werden kann.
Bindings werden für Dateien des Quellcodes und für Dateien mit der Endung .class erzeugt. Sie werden in einen *Binding*-Graph eingebettet, der nicht zyklenfrei ist und als ein abstrakter Syntaxgraph angesehen werden kann.
Ein *Binding*-Graph enthält eine Kopie der Teilmenge des AST: Für jede Deklaration im AST existiert ein entsprechendes *Binding*. Jedoch überall dort, wo im AST ein Knoten durch einen Namen oder Typ (Objekt vom Typ `Name` oder `Type`) repräsentiert wird, enthält der *Binding*-Graph eine Instanz auf das *Binding* von diesem Typ (Objekt vom Typ `ITypeBinding`).
Die Abbildung 3.2 zeigt einen Auszug eines *Binding*-Graphen, der für die Datei aus der Abbildung 3.1 erzeugt wird. Auf der linken Seite ist der AST angegeben, dessen Elemente zu den entsprechenden *Bindings* aufgelöst werden können, was in der Abbildung durch blaue gestrichelte Pfeile markiert ist.

3.2.2 Umweg über XML

Die Java/AST-Repräsentation der Daten hat einige Nachteile.
Der AST spiegelt nicht nur den Quellcode eines OT-Programms wieder, sondern birgt in sich zusätzliche Informationen, welche die Realisierung neuer Sprachartefakte betreffen. Es handelt sich zum Beispiel um Interfaces, Methoden, die extra ins Programm eingebaut werden, um die neuen Konzepte wie Lifting, Lowering, implizite Vererbung usw. in Java abzubilden. Aus diesem Grund erfordert die Daten-Manipulation Kenntnisse über bestimmte Realisierungsdetails des Compilers.
Die bequemste Art der Traversierung eines AST ist die Verwendung des Designmusters *Visitors* (5.3.1.1). Dieses Designmuster ist in Eclipse vorhanden; die Klasse, die einen Besucher implementiert, der für jeden Knotentyp des AST eine entsprechende `visit`-Methode bereitstellt, heißt `ASTVisitor`. Für jede Operation auf dem AST, muss eine Unterklasse von ASTVisitor gebildet werden, die entsprechende `visit`-Methoden überschreibt. Die Klasse `ASTNode` im Paket `org.eclipse.jdt.core.dom`, die einen AST-Knoten beschreibt, hat 108 Unterklassen.
Gemäß den Anforderungen für eine geeignete Datenrepräsentation der Toolbox bot sich die Sprache XML als ein Zwischenformat an. Zunächst wurde

mit XML eine Auszeichnungssprache definiert, in der die Daten eines OT-Programms strukturiert gespeichert werden konnten. Anschließend wurde ein Konvertierer entwickelt, der Java/AST nach XML umwandelt, sodass die internen Zusatzinformationen ausgeblendet werden und auch eine tiefere Analyse (über den Quellcode der zu analysierenden Klasse hinaus) möglich ist. Angesichts der Unterstützung vieler XML-Techniken, kann ein nach XML umgewandeltes OT-Programm flexibel manipuliert werden.

3.2.2.1 Auszeichnungssprache für den AST

Für die XML-Repräsentation des AST wurde eine DTD (s. 2.3.2.1) erstellt, um zunächst die „AST-Auszeichnungssprache" zu definieren. Außerdem kann diese DTD dazu behilflich sein, die vom Konvertierer erzeugten XML-Strukturen und ihre (von anderen Modulen) modifizierten Varianten auf Korrektheit überprüfen zu können.
Sobald eine Klasse nach XML umgewandelt wurde, besteht keine Möglichkeit Informationen über andere Programmteile nachzuladen, sie können lediglich aus dem XML-Dokument (sofern vorhanden) durch die Eindeutigkeit der Bezeichner ermittelt werden. Die Eindeutigkeit eines Typs im XML-Dokument wird damit erreicht, dass der vollständige Name des Typs und des Pakets, in dem der Typ enthalten ist, gespeichert wird. Aus diesem Grund muss ein XML/AST auch Informationen speichern, die im Java/AST über aufgelöste *Bindings* erhalten werden können.
Da ein XML-Dokument ein Wurzelement benötigt, wurde ein festes Element definiert, das alle anderen Knoten enthalten soll. Die möglichen Unterelemente sind Elemente, welche ein Paket, ein Team, eine gewöhnliche Klasse und ein Interface beschreiben. Ein Paket kann weitere Pakete aber auch alle möglichen Arten von Klassen als Unterelemente enthalten, sein Name wird als ein Attribut gespeichert. Wie ein Team in XML dargestellt werden kann, wird anhand eines Beispiels veranschaulicht.
Listing 3.2 zeigt ein XML-Element, welches das Team `MyTeam` aus Listing 3.1 abbildet. Eine Java/AST-Darstellung von diesem Team wurde bereits in Abbildung 3.1 vorgestellt.

Listing 3.1: *Team* `MyTeam`

```
1 package ast.example;
2 public team class MyTeam {
3   public class Role playedBy Base {
4     public abstract void m(String str);
5         m -> m;
6     }
7 }
```

Listing 3.2: *XML-Repräsentation des Teams* `MyTeam`

```
<package name="ast">
 <package name="example">
  <team public="yes" name="MyTeam" qualifiedName="MyTeam">
   <extends>
    <type name="Team" qualifiedName="Team"
     qualifiedPackageName="org.objectteams" />
    <extends>
     <type name="Object" qualifiedName="Object"
      qualifiedPackageName="java.lang" />
    </extends>
   </extends>
   <constructor name="MyTeam" qualifiedName="MyTeam()"
    public="yes" />
   <role name="Role" qualifiedName="MyTeam.Role"
    public="yes" >
    <extends>
     <type name="Object" qualifiedName="Object"
      qualifiedPackageName="java.lang" />
    </extends>
    <playedBy>
     <type name="Base" qualifiedName="Base"
      qualifiedPackageName="ast.example" />
    </playedBy>
    <method name="m" qualifiedName="m(java.lang.String)"
     public="yes" abstract="yes">
     <type name="void" qualifiedName="void" />
     <parameter name="str">
      <type name="String" qualifiedName="String"
       qualifiedPackageName="java.lang" />
     </parameter>
    </method>
    <constructor name="Role" qualifiedName="Role()"
     public="yes" />
    <callout fieldAccess="no">
     <methodspec name="m" qualifiedName="m(java.lang.String)"
      basemethodspec="no" />
     <methodspec name="m" qualifiedName="m(java.lang.String)"
      basemethodspec="yes" />
    </callout>
   </role>
  </team>
 </package>
</package>
```

Die vollständige Definition der DTD, welche für diese Arbeit relevant ist, ist im Anhang A angegeben. Um einige Entwurfsentscheidungen bei ihrer

Erstellung zu begründen, wird die Definition der Konstuktor-Beschreibung aus Listing 3.3 erläutert.

Listing 3.3: *Definition des Konstuktor-Elements*

```
<!ELEMENT constructor
    (documentation?, parameter*, throws*)>
<!ATTLIST constructor
    name            CDATA    #REQUIRED
    qualifiedName   CDATA    #REQUIRED
    default         (yes|no) #REQUIRED
    public          (yes|no) #IMPLIED
    packageprivate  (yes|no) #IMPLIED
    protected       (yes|no) #IMPLIED
    private         (yes|no) #IMPLIED
    final           (yes|no) #IMPLIED>
```

Die Daten des Konstruktor-Elements werden durch weitere Elemente und zahlreiche Attribute beschrieben. Die Entscheidung, ob ein Attribut oder Element verwendet wird, wurde in Abhängigkeit vom Detailgrad der darzustellenden Information und in Hinsicht auf bessere Übersichtlichkeit getroffen. Bei der Darstellung von Attributen wurde ebenfalls auf die Übersichtlichkeit geachtet. So ist zum Beispiel die Information des Attributs `qualifiedName`, das den Konstruktornamen mit Parametertypen beschreibt, zwar im Element enthalten, kann aber für die eindeutige Identifizierung des Konstruktor-Elements durch die Verwendung des Attributs abgerufen werden, sodass keine weitere Suche durch die Unterknoten nötig ist.
Vier Attribute für Sichtbarkeits-*Modifizierer* dienen zur besseren Übersicht, wobei nur ein Attribut und nicht alle vier gespeichert werden müssen. Diese Darstellung ist vergleichbar mit der Beschreibung durch ein Attribut mit einem Bezeichner und vier möglichen Werten.

3.2.3 XML-Konvertierer

Die Hauptaufgabe eines XML-Konvertierers ist die Umwandlung einer .java Datei nach XML, sodass alle für eine tiefe statische Programmanalyse relevanten Daten der in der Datei enthaltenen Klassen in der XML-Struktur zu finden sind. Grundsätzlich gibt es drei Realisierungsmöglichkeiten, für die Vollständigkeit der Informationen zu sorgen, wenn eine Klasse[2] `C` nach XML konvertiert werden soll.

- Eine transitive Hülle wird erzeugt. Sobald eine Abhängigkeit (z.B. Vererbung) von einer anderen Klasse `A` auftritt, die für eine tiefe Analyse

[2]Im Folgenden wird unter dem Begriff Klasse folgendes zusammengefasst: eine gewöhnliche Klasse, ein Interface, eine Team- oder Rolleklasse und sich daraus ergebende Hybridformen.

von `C` relevant ist, wird auch `A` nach XML umgewandelt, um auch in `A` nach erforderlichen Daten suchen zu können. Die Umwandlung von `A` zieht eventuell eine Umwandlung von n weiteren Klassen nach sich usw.

- Alle verfügbaren Klassen werden nach XML umgewandelt.
- Alle benötigten Informationen der Klasse `C` werden selektiert und der Repräsentation von `C` hinzugefügt.

In dieser Arbeit wurde der dritte Ansatz gewählt. Die statische Analyse der Qualitätssicherungsmaßnahmen bei dieser Alternative ist gegenüber den anderen Verfahren wesentlich einfacher. Die implizite und explizite Vererbungshierarchie einer Klasse wird dabei flachgeklopft, sodass sie bereits ihre geerbten Features enthält. Dabei wird die Suche nach diesen Komponenten nur einmal zentral implementiert. Außerdem ist die Datenbasis für die Analyse eindeutig: Alle im XML-Dokument vorhandenen Klassen werden als „zu manipulierende Objekte" betrachtet.
Ein klarer Nachteil des dritten Ansatzes liegt darin, dass viele Informationen mehrfach vorkommen können. So sind zum Beispiel die von einer Superklasse geerbten Methoden in jeder Unterklasse enthalten. Ein anderer Nachteil dieser Alternative ist die Garantie der Vollständigkeit der XML-Repräsentation. Sobald ein Werkzeug eine spezielle, nicht berücksichtigte Information einer Klasse benötigt, muss diese Erweiterung direkt im XML-Konvertierer durchgeführt werden.

3.2.3.1 Implementierungsdetails

Die Umwandlung einer *compilation unit* (Datei mit der Endung .java) in XML-Format beginnt beim Wurzelknoten des vom AST-Parser erzeugten

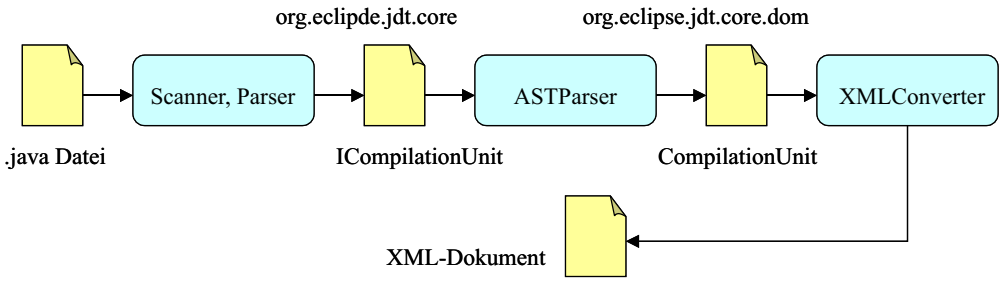


Abbildung 3.3: *Konvertierung einer compilation unit nach XML*

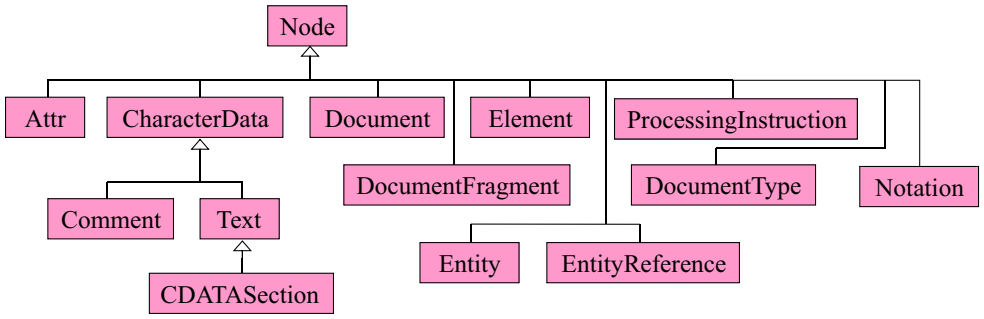


Abbildung 3.4: *Knotentypen im Paket* `org.w3c.dom`

AST, welcher für die Java-Repräsentation der Datei (Objekt der Klasse `ICompilationUnit`) gebildet wird (Abbildung 3.3). Dabei wird der AST von oben nach unter traversiert und seine Knoten jeweils nach XML umgewandelt.

Erzeugung der XML-Strukturen Für die XML-Erzeugung wurde die Java-API verwendet. Es gibt zwei wohldefinierte Verfahren, um XML-Daten zu verarbeiten: DOM (*Document Object Model*) und SAX (*Simple API for XML*). Sie wurden vom W3C entwickelt und für Java implementiert. SAX verfolgt einen ereignisorientierten Ansatz, dabei wird das XML-Dokument nicht vollständig im Speicher abgelegt. Aus diesem Grund ist SAX für den XML-Konvertierer ungeeignet.
Ein DOM-Modell ist eine Baumstruktur, wo jeder Knoten einen der Bestandteile von einer XML-Struktur enthält. DOM erzeugt eine strikte Hierarchie (Abbildung 3.4). Die zwei häufigsten Knotentypen sind Element-Knoten und Textknoten. Durch die Verwendung von DOM-Funktionen können Knoten erzeugt, entfernt, der Knoteninhalt geändert und die Knotenhierarchie traversiert werden. Das Paket `org.w3c.dom` stellt die Schnittstellen für das DOM-Modell zur Verfügung.

XMLConverter Die XML-Konvertierung wird in der Klasse `XMLConverter` durchgeführt. Sie implementiert (s. Abbildung 3.5) das Designmuster `Singleton`[3] und enthält ein Objekt der Klasse `Document`, welches die Wurzel des XML-Dokumentenbaums repräsentiert. Der Zugriff auf die Daten des Dokumentes wird durch sein Wurzelelement organisiert.
Die Konvertierung erfolgt jeweils für ein Objekt der Klasse `CompilationUnit`. Jeder Aufruf der Methode `convert(CompilationUnit)` erzeugt den dort enthaltenen Klassen entsprechende XML-Elemente, welche gemäß der

[3]Es kann nur eine Instanz dieser Klasse existieren.

festgelegten AST-Auszeichnungssprache in das jeweilige Paketelement eingefügt werden. Falls eine Klasse zu keinem Paket gehört, wird das Klassenelement dem Wurzelelement hinzugefügt.
Die Erzeugung von Paketelementen wird mit Hilfe eines *Map*-Objektes kontrolliert, wobei als Schlüssel der vollständige Name des Pakets (*qualified package name*) dient und als Wert das entsprechende Paketelement gespeichert wird. Demzufolge existiert nach der Konvertierung einer Datenbasis neben der baumförmigen Struktur des Dokuments auch ein Paket-*Map*.
Neben der Konvertierung wird auch die Suche nach geerbten Features durchgeführt. Hierbei wird die ganze Vererbungshierarchie (von spezielleren zu allgemeineren Superklassen) durchsucht, wobei wieder eine *Map*-Struktur zum Einsatz kommt. Da Features mit der gleichen Signatur in der Regel nicht doppelt vorkommen dürfen, muss dafür gesorgt werden, dass nur die fehlenden Features betrachtet werden.
Um zum Beispiel Methodenelemente einer Klasse zu erzeugen, werden folgende Schritte durchgeführt:

- Zunächst werden die in der Klasse definierten Methoden nach XML konvertiert, dabei werden sie dem XML-Element, das die Klasse beschreibt, hinzugefügt und außerdem im *Map* unter der Signatur abgelegt.

- Sofern kein Konstruktor unter diesen Methoden vorhanden war, wird ein *Default*-Konstruktor generiert.

- Danach werden Methoden der Superklassen durchsucht, wobei rekursiv von der direkten Superklasse nach oben in der Vererbungshierarchie vorgegangen wird. Eine geerbte Methode wird nur in dem Fall konvertiert und hinzugefügt, falls keine Methode mit der gleichen Signatur im *Map* enthalten ist.

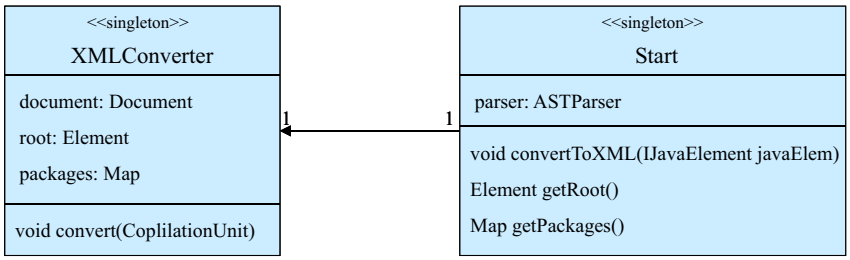


Abbildung 3.5: `XMLConverter`

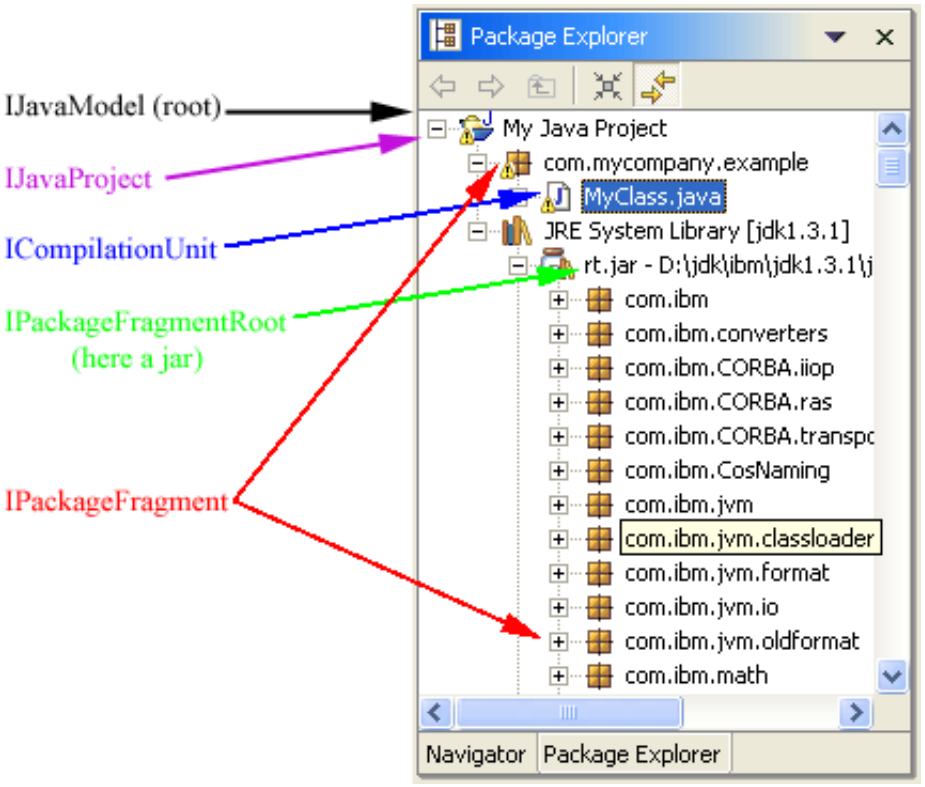


Abbildung 3.6: *Package view* (Abbildung aus der Eclipse-Dokumentation)

- Außerdem werden die geerbten Methoden aus allen (auch geerbten) zu implementierenden Interfaces ermittelt.

Aufgrund des hohen (Zeit-)Aufwandes wurde in dieser Arbeit die Konvertierung einer Methode auf die Konvertierung ihrer Signatur beschränkt, sodass der Methodenrumpf nicht weiter betrachtet wird.

3.2.3.2 Eclipse-Integration

Compilation units in Eclipse sind in Paketen organisiert, welche wiederum in Projekten verwaltet werden. Das Paket *org.eclipse.jdt.core* legt die Klassen fest, welche die Elemente modellieren, aus denen sich ein Java-Programm zusammensetzt. Das Modell ist hierarchisch, die gemeinsame Superklasse ist die Klasse `IJavaElement`.
Package Explorer aus der Abbildung 3.6 ist eine der Perspektiven von Eclipse, die das Arbeiten mit Java-Projekten erleichtern. Er stellt Java-Elemente aller Pakete aller Projekte des Java-Modells dar.
In der Klasse `Start` (Abbildung 3.5) ist eine Methode `convertToXML(IJa-`

`vaElement`) implementiert, welche eine XML-Konvertierung für Objekte der folgenden Klassen erlaubt: `IJavaProject`, `IPackageFragment`, `IPackage-FragmentRoot` und `ICompilationUnit`. Dabei werden alle *compilation units* ermittelt und in das XML-Format umgewandelt.

Im Folgenden wird auf die Entwicklung der Werkzeuge eingegangen, welche auf der oben vorgestellten XML-Repräsentation aufbauen. In dieser Arbeit wurden zwei Werkzeuge realisiert, deren Funktionalität und Implementierung in den zwei nachstehenden Kapiteln beschrieben wird.

4 Dokumentationswerkzeug OTDoc

4.1 Anforderungen

Ein Dokumentationswerkzeug kann zu den qualitätssichernden Werkzeugen gezählt werden. Zum einen liefert die generierte Dokumentation eine andere Repräsentation des Programmsystems, die Kommentare an relevanter, eventuell problematischer (Ausnahmebehandlungen) Stelle anzeigt. Zum anderen ist eine gute Dokumentation auf Grund der Langlebigkeit eines Softwareprogramms insbesondere für die Wartung und Weiterentwicklung von großem Interesse. Außerdem kann eine automatisch erstellbare Dokumentation immer wieder an Änderungen im System angepasst werden.
Ein Dokumentationswerkzeug kann von vielen verschiedenen Benutzern mit unterschiedlichen Vorlieben verwendet werden, sodass die Gestaltung der Informationsdarstellung eine wichtige Rolle spielt. Dazu gehören aus der Sicht der Software-Ergonomie unter anderem die Anordnung und Strukturierung der Informationen, die Gestaltung von Text und Graphik, der Einsatz von Farbe. Außerdem muss die bequeme Navigation durch die Dokumentation gewährleistet werden.
Darüber hinaus kann die Möglichkeit der Generierung von Hypertext-Dokumentationen zu den Anforderungen an Dokumentationswerkzeuge gezählt werden.

4.2 Beschreibung

OTDoc ist das in dieser Arbeit entwickelte Werkzeug, das aus dem Quelltext eines OT-Programms eine Dokumentation im HTML-Format generiert. In der Dokumentation sind Informationen über die OT-Elemente wie Pakete, Klassen[1], Felder, Methoden, Konstruktoren und Methodenbindungen enthalten. Dabei wird eine Beschreibung dieser Elemente erzeugt, die aus ihren De-

[1] Unter dem Begriff Klasse werden in diesem Kapitel auch Teams, Rollen und Interfaces verstanden. Eine Java-Klasse wird als gewöhnliche Klasse bezeichnet.

klarationen und Dokumentationskommentaren besteht. Zu jeder Klasse wird eine HTML-Seite erstellt, von der über verschiedene Querverweise (Links) zu den Dokumentationsseiten der mit dieser Klasse in Verbindung stehenden Klassen navigiert werden kann.
OTDoc erfüllt die Funktionalität von JavaDoc (Dokumentationswerkzeug für Java [9]) für ObjectTeams. Dabei wird lediglich die bewährte und gewohnte Anordnung, Strukturierung und Detaillierungsgrad von Informationen der Dokumentation von JavaDoc übernommen und entsprechend den Anforderungen der neuen Sprache erweitert. Die Realisierung beider Werkzeuge ist von Grund auf verschieden: Während JavaDoc eine reine Java-Lösung ist, wurden bei OTDoc überwiegend XML-Techniken eingesetzt.

4.2.1 Generierte Dateien

Für alle Klassen (innere Klassen eingeschlossen) der Datenbasis wird eine HMTL-Datei generiert. Außerdem erstellt OTDoc in Anlehnung an JavaDoc weitere Dateien:

- overview-summary.html
  Ein Überblick über alle Pakete der Datenbasis
- package-summary.html
  Dokumentation eines Pakets, welche eine Benutzer-Beschreibung aus der Datei package.html enthält
- allclasses-noframe.html Eine nicht *Frame* basierte Anzeige aller dokumentierten Klassen in allen Paketen der Datenbasis

  *-frame.html
  Dateien, die für die Darstellung mit Frames verwendete Listen mit Paketen oder Klassen anbieten
- index-*.html
  Die Übersicht aller Klassen, Felder, Methoden, Methodenbindungen in einem Index

Zur Unterstützung werden folgende Dateien zur Verfügung gestellt:

- help-doc.html
  Eine Kurzbeschreibung von OTDoc
- index.html
  Eine *Frame* basierte Ansicht, die Paket- und Klassenanzeige enthält und ausgewählte Dokumentationen anzeigt

- stylesheet.css
  Formatvorlage für alle HTML-Dateien, in der Farben- und Textgestaltung festgelegt wird

4.2.2 Dokumentationskommentare

Ein Dokumentationskommentar wird zwischen `/**` und `*/` unmittelbar vor dem zu dokumentierendem Element geschrieben. Er wird unverändert in die Dokumentation übernommen und darf HTML-Markierungen enthalten. Außerdem kann eine spezielle Markierung einem Dokumentationskommentar hinzugefügt werden, um damit gewisse Details zu beschreiben oder bestimmte Abkürzungen zu verwenden. Die Übersicht aller möglichen Markierungen und ihre kurze Beschreibung ist im Anhang B angegeben.

Darstellung von neuen Konzepten

Vor einer konkreten Implementierung musste entschieden werden, wie die neuen Konzepte von ObjectTeams dargestellt werden können. Dafür wurde ein Prototyp manuell in HTML angefertigt, an dem experimentiert werden konnte. Die wichtigsten Erweiterungen der JavaDoc-Dokumentation werden im Folgenden zusammengefasst:

- Teams, Rollen, Klassen und Interfaces müssen in der Übersicht optisch von einander unterschieden werden.
- Die implizite Vererbungshierarchie wird neben der expliziten angezeigt.
- Die Dokumentation einer Rollenklasse enthält eine Übersicht aller Methodenbindungen, von der zu ihrer Dokumentation im Detail navigiert werden kann.
- Teams enthalten eine Übersicht aller (auch geerbten) Rollen.
- Basisklassen enthalten eine Übersicht sämtlicher an sie gebundenen Rollen aus der Datenbasis.

4.3 Implementierungsdetails

Liegen die Programmdaten im XML-Format vor, dann bietet sich die Sprache XSLT (s. 2.3.4.2) an, um sie ins HTML-Format umzuwandeln.

4.3.1 Datenbasis

Die Datenbasis für die Dokumentation ist eine XML-Struktur, welche vom XML-Konvertierer erzeugt wird.
Wie bereits erwähnt, ist der Java/AST von Eclipse kein reiner abstrakter Syntaxbaum, was sich unter anderem darin äußert, dass dort Dokumentationskommentare enthalten sind. Der XML-Konvertierer wandelt neben den Deklarationen auch entsprechende Dokumentationskommentare nach XML um, sodass XML-Elemente, die eine Deklaration darstellen ein Dokumentationselement enthalten (s. Definition der Auszeichnungssprache im Anhang A).
Ein generiertes XML-Dokument wird mit Hilfe von XSLT-Dateien transformiert. Dabei werden die Informationen der Datenbasis entnommen und lediglich anders strukturiert. Folglich müssen alle Daten zum Zeitpunkt der Transformation in der XML-Struktur vorhanden sein und bevor Transformationen durchgeführt werden, muss die Vollständigkeit der Informationen geprüft werden.

Bildung von Verknüpfungen

Eine wichtige Funktionalität der Dokumentation ist die Navigation durch Verknüpfungen. Spezifiziert zum Beispiel ein Team `MyTeam` ein anderes Team `SuperTeam`, dann ist eine Verknüpfung auf der Dokumentationsseite von `MyTeam` zur Seite von `SuperTeam` gewünscht.

```
/** ... */
public team class MyTeam extends SuperTeam { ... }
```

Diese Verknüpfung ist nur dann möglich, wenn `SuperTeam` zur Datenbasis gehört.
Die Frage, ob eine Klasse in der XML-Struktur vorhanden ist, kann geklärt werden, indem die Datenbasis nach dieser Klasse durchsucht wird. Die Suchzeit ist immer dann am längsten, wenn eine Klasse nicht zur Datenbasis gehört. Wenn eine Verknüpfung an verschiedenen Stellen angezeigt werden soll, wie zum Beispiel ein Parametertyp einer Methode in der Methodenübersicht und in der Methodenbeschreibung, muss die Suche wiederholt durchgeführt werden.
Um die mehrfache Suche zu vermeiden, wurde die generierte XML-Struktur, die im Speicher als DOM-Baum vorliegt, vor Transformationen in einer Java-Anwendung mit zusätzlichen Informationen angereichert. Für alle Elemente, die einen Typ und damit eine mögliche Verknüpfung darstellen, wurde bestimmt, ob die von ihnen referenzierten Klassen zur Datenbasis gehören. Zudem wurde für gültige Verknüpfungen ein relativer Pfad zum entsprechenden

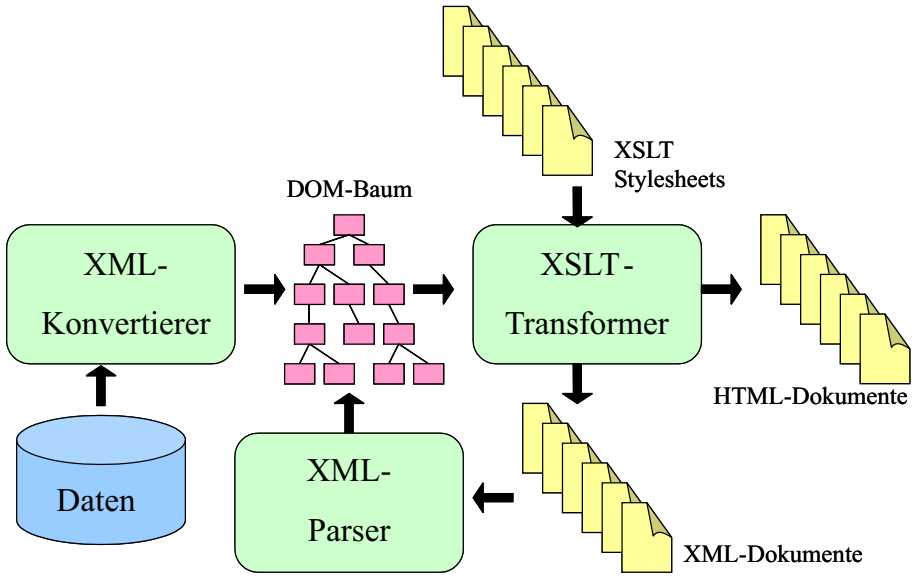


Abbildung 4.1: *Ablauf der Transformationen*

Dokumentationsteil erzeugt und den Typ-Elementen der XML-Struktur hinzugefügt. Listing 4.1 zeigt das Typ-Element mit einer entsprechenden Verknüpfung.

Listing 4.1: *Das Typ-Element referenziert die Klasse* `a.b.c.A`.

```
<type name="A" qualifiedPackageName="b.c" qualifiedName="A">
   <link href="b/c/A.html" />
</type>
```

4.3.2 Einsatz von XSLT

In der Abbildung 4.1 ist der Ablauf der Dokumentationsgenerierung veranschaulicht.
Der DOM-Baum, der vom XML-Konvertierer erzeugt wird, ist eine Objekt-Darstellung der XML-Datenbasis im Speicher und lässt sich mit Java weiterhin manipulieren. Außerdem kann diese Datenbasis durch die Verwendung von Formatierungsvorlagen (*style sheets*) manipuliert werden.
Um in 4.2.1 vorgestellte Dateien zu erzeugen, wurde für jeden Datei-Typ eine XSLT-Datei erstellt, in der eine entsprechende Formatierung nach HTML festgelegt wurde.
Die Dateigenerierung wird aus einer Java-Anwendung kontrolliert. Dabei werden Transformationen des ganzen DOM-Baums oder einer seiner Knoten ver-

anlasst und die Ergebnisse der Umwandlung in HTML-Dateien geschrieben. Wenn die Struktur der Datenbasis für das gewünschte Ergebnis ungeeignet ist, wird eine andere passende XML-Struktur mittels einer entsprechenden Formatierungsvorlage gebildet und weiter verarbeitet, indem sie mittels des XML-Parsers in ein DOM-Objekt umgewandelt wird.

4.3.2.1 Formatierungsvorlagen

Der Schwerpunkt dieser Implementierung liegt in der Definition von Formatierungsvorlagen, mit deren Hilfe XML-Daten nach HTML konvertiert werden. Im Folgenden werden einige für die Werkzeug-Realisierung spezifische Aufgabenstellungen exemplarisch skizziert und deren Lösung in XSLT vorgestellt.

Generierung von sortierten Listen

Eine wichtige Rolle in der Dokumentation spielen alphabetisch sortierte Listen, die eine Übersicht mit entsprechenden Verknüpfungen anbieten. Dank des Adressierungsmechanismus von XPath und Möglichkeiten von XSLT läst sich diese Aufgabe sehr einfach lösen.
In Listing 4.2 ist eine Formatierungsvorlage zu sehen, welche eine Tabelle mit allen Klassennamen erzeugt. Die Namen werden alphabetisch geordnet, erhalten Verknüpfungen zu ihren Klassen-Seiten und sollen in einer Tabelle angezeigt werden.
In Zeile 10 wird eine Schleife über alle Klassen-Elemente gestartet, die sie in der Reihenfolge, welche sich durch die Sortierung nach dem Namen ergibt(Zeile 11), auswertet. Wie eine Klasse angezeigt wird, wird durch die Auswertung einer entsprechenden Schablone bestimmt (Zeile 13). Als Beispiel ist eine Schablone für das Team-Element angegeben (Zeilen 21-23).

Listing 4.2: *Vorlage zur Generierung von allclasses-noframe.html*

```
1  <xsl:template match="otdoc">
2  <html>
3   <head>
4    <title>All Classes</title>
5    <link rel="stylesheet" type="text/css" href="otxx.css"/>
6   </head>
7   <body>
8    <h4>All Classes</h4>
9    <table border="0" width="100%"><tr><td>
10    <xsl:for-each select="//team|//role|//class|//interface">
11      <xsl:sort select="@qualifiedName"/>
12     <a href="{@path}/{@qualifiedName}.html">
13      <xsl:apply-templates select = "."/>
```

```
14      </a><br/>
15     </xsl:for-each>
16    </td></tr></table>
17   </body>
18 </html>
19 </xsl:template>
20
21 <xsl:template match="team">
22         <b><xsl:value-of select = "@qualifiedName"/></b>
23 </xsl:template>
24 ...
```

Suche in der Datenbasis

Eine wichtige Funktionalität, die in den Formatierungsvorlagen für die Verknüpfungsbildung verwendet wird, ist die Suche in der Datenbasis. Im Folgenden werden zwei Arten der Suche und ihre Implementierung in XSLT anhand eines Beispiels dargestellt.

- Gesucht ist eine Klasse, die durch ein Typ-Element repräsentiert wird.

Um die Klasse zu finden, muss das Paketelement in dem sie enthalten ist, gefunden werden. Das Typ-Element enthält den qualifizierten Namen des Pakets und der Klasse, sodass diese Informationen für die Suche zur Verfügung stehen. Zur Erinnerung ist zu sagen, dass Paket in der Datenbasis ein rekursives Element darstellt. Die Lösung des Problems der Suche nach einem Element mit dem qualifizierten Namen in einem Kontextknoten wird in der folgenden Funktion vorgestellt.

Listing 4.3: *Elementensuche in einem Kontextknoten*

```
1  <xsl:function name="own:f">
2    <xsl:param name="contextnode"/>
3    <xsl:param name="qname"/>
4   <xsl:choose>
5    <xsl:when test="contains($qname, '.')">
6     <xsl:sequence select=
7      "own:f(
8        $contextnode/*[@name=substring-before($name, '.')],
9        substring-after($name, '.'))"/>
10   </xsl:when>
11   <xsl:otherwise>
12    <xsl:sequence select="$contextnode/*[@name=$qname]"/>
13   </xsl:otherwise>
14  </xsl:choose>
15 </xsl:function>
```

Wird diese Funktion mit dem Wurzelknoten und dem Paketnamen aufgerufen, dann wird entweder das Paket-Element zurückgegeben oder eine leere Menge, falls das Paket in der Datenbasis nicht existiert. Nun kann die Klasse durch den Aufruf der Funktion mit dem Paketelement und Klassennamen als Argumente gefunden werden.

- Gesucht sind alle an eine Klasse gebundenen Rollenklassen

Die Klasse wird durch den qualifizierten Namen und ihren Paketnamen repräsentiert. Die Lösung ist wie folgt implementiert.

Listing 4.4: *Suche nach Rollenklassen*

```
1 <xsl:variable name="bases" select=
2   "root(.)/otdoc//playedBy/type[@qualifiedPackageName=
3             $packagename and @qualifiedName=$qname]"/>
```

Ausgehend vom Wurzelknoten werden alle in einer *playedBy*-Beziehung stehenden Basisklassen auf die Identität mit der gesuchten Klasse untersucht. Beim Übereinstimmen wird das *playedBy*-Element zurückgegeben, sodass der Ausdruck eine Menge mit Elementen liefert, die *playedBy*-Beziehung zu den sie bindenden Rollen beschreiben. Da die XML-Struktur genau bekannt ist, kann man davon durch einen XPath-Ausdruck zum Rollen-Element selbst gelangen.

Index-Erstellung

Zur komfortablen Navigation innerhalb der Dokumentation soll ein alphabetisch sortierter Index aller Deklarationen (Namen von Teams, Rollen, Interfaces, Klassen, Feldern, Konstruktoren und Methoden) erstellt werden. Dieser Index soll zur besseren Übersichtlichkeit in verschiedene Dateien (eine Datei je Anfangsbuchstabe der vorhandenen Namen) organisiert werden. Dabei soll auf jeder Indexseite Querverweise zu allen existierenden Indexseiten und jeweils zur nachfolgenden und vorherigen Seite angeboten werden.
Diese Aufgabe ist nicht mit einer Formatierungsvorlage zu realisieren. Die Datenbasis muss mit Hilfe einer XSLT-Vorlage so umstrukturiert werden, dass Elementnamen mit dazugehörigen Pfaden nach Anfangsbuchstaben gruppiert werden. Dann kann eine Transformation nach HTML für jedes nicht leere Gruppenelement anhand einer anderen Vorlage durchgeführt werden.
In Listing 4.5 ist ein Ausschnitt aus der XSLT-Datei dargestellt, in der die Umstrukturierung der Datenbasis durchgeführt wird. In Zeilen 1-6 werden zuerst alle zu dokumentierenden Elemente ausgewählt. Dann werden diese Elemente in einer Schleife über Buchstaben von A bis Z (ASCII-Code) entsprechend dem Anfangsbuchstaben einer Gruppe zugeordnet (Zeilen 12-20)

und in einer XML-Struktur (Zeilen 23-31) sortiert (Zeile 30) abgelegt. Der Detaillierungsgrad der Informationen eines Elements wird in einer Extraschablone definiert, deren Auswertung in Zeile 29 veranlasst wird.

Listing 4.5: *Umstrukturierung der Datenbasis*

```
1  <xsl:variable name="classes" select=
2    "//interface|//class|//team|//role"/>
3  <xsl:variable name="fields" select=
4    "//field[empty(@inherited)]"/>
5  <xsl:variable name="methods" select=
6    "//method[empty(@inherited)]|//constructor"/>
7  <xsl:for-each select="97 to 122">
8   <!-- small letter -->
9   <xsl:variable name="a" select="codepoints-to-string(.)"/>
10  <!-- capital letter -->
11  <xsl:variable name="A" select="codepoints-to-string(.-32)"/>
12  <xsl:variable name="v1" select=
13    "$classes[starts-with(@qualifiedName, $a)
14              or starts-with(@qualifiedName, $A)]"/>
15  <xsl:variable name="v2" select=
16    "$methods[starts-with(@qualifiedName, $a)
17              or starts-with(@qualifiedName, $A)]"/>
18  <xsl:variable name="v3" select=
19    "$fields[starts-with(@name, $a)
20            or starts-with(@name, $A)]"/>
21  <xsl:if test=
22    "not(empty($v1) and empty($v2) and empty($v3))">
23   <letter>
24    <xsl:attribute name="a">
25     <xsl:value-of select="$A"/>
26    </xsl:attribute>
27    <xsl:for-each select="$v1|$v2|$v3">
28      <xsl:sort select="own:qn(.)"/>
29     <xsl:apply-templates select="."/>
30    </xsl:for-each>
31   </letter>
32  </xsl:if>
33 </xsl:for-each>
```

4.3.2.2 Grenzen von XSLT

Wie bereits angedeutet, wird der Ablauf der Transformationen aus einer Java-Anwendung gesteuert. Die Begründung dafür liegt zum Einen darin, dass der DOM-Baum sich im Speicher befindet und einen schnellen Zugriff auf seine Elemente und deren flexible Manipulation anbietet. Zum Anderen können bestimmte Aufgaben wie Verknüpfungsbildung einmalig ausgeführt werden,

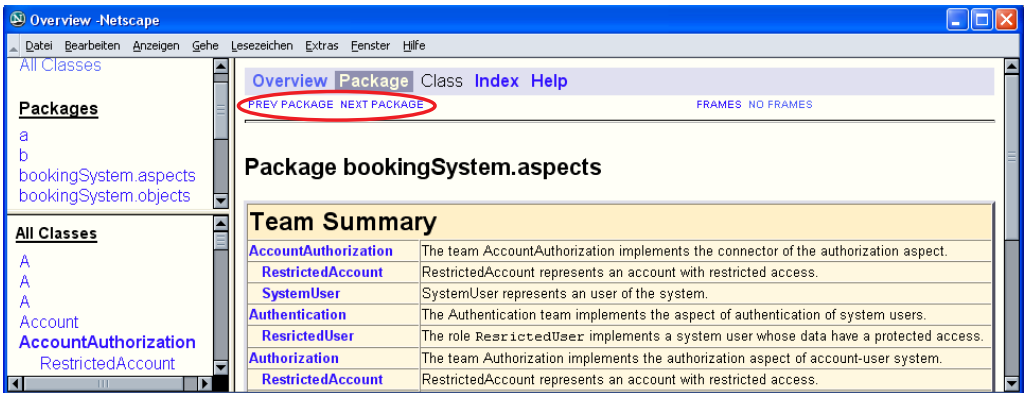


Abbildung 4.2: *Übersicht über die Pakete*

sodass der Aufwand erheblich reduziert werden kann. Die Implementierung von OTDoc verbindet zwei Techniken miteinander, indem die Vorteile der jeweiligen Sprache in den Vordergrund gestellt werden.
Dass Java-Lösungen mancher Aufgabenstellungen einfacher und intuitiver sind, wird im Folgenden anhand eines Beispiels demonstriert.
In der Dokumentation eines Pakets gibt es Verknüpfungen, die zur Beschreibung des Vorgänger- und Nachfolger-Pakets führen (s. Abbildung 4.2), wobei die Reihenfolge auf der alphabetischen Sortierung beruht. Informationen über diese Querverweise sind, wie im Folgenden gezeigt wird, zum Zeitpunkt der Paket-Transformation nur mit einem großen Aufwand zu erhalten. Zuerst müssen alle nicht leeren Pakete extrahiert und aufsteigend nach dem Namen sortiert werden. Dann muss die Position des aktuellen Pakets in dieser Menge bestimmt werden, um anschließend den Vorgänger und Nachfolger oder deren Nichtexistenz zu bestimmen. Dabei müssen diese Berechnungen für jedes Paket neu durchgeführt werden.
In OTDoc wurde dieses Problem mit Java gelöst. Da es schon ein *Map*-Objekt mit allen Paketelementen existiert, werden vor der Pakettransformation Instanzen auf alle nicht leeren Paket-Elemente in einer nach Namen sortierten Struktur gespeichert, damit der Nachfolger und Vorgänger direkt abgelesen werden kann. Dann können die gewünschten Querverweise direkt in einem Paket repräsentierenden XML-Element abgelegt und während der Transformation benutzt werden.

5 Design-Kritiken-Werkzeug

5.1 Beschreibung

Ein Design-Kritiken-Werkzeug (siehe 3.1.5.1) nutzt vordefinierte Regeln, um das Design eines Programmsystems auszuwerten. Design-Kritiken machen auf Schwächen oder potentielle Probleme im Design aufmerksam, weisen auf ungünstige Konstellationen oder unpassend verwendete Namen hin, markieren unvollständige oder komplizierte Inhalte.
Dabei zeigen Kritiken in der Regel keine Fehler an. Sie liefern Verbesserungsvorschläge, die vom Entwickler nicht umgesetzt werden müssen.

5.2 Definition von Design-Kritiken für OT

Vordefinierte Regeln, mit deren Hilfe das Design eines OT-Programms ausgewertet werden kann, spielen eine zentrale Rolle im implementierten Design-Kritiken-Werkzeug. Zuerst mussten ungünstige Szenarien im OT-Design aufgefunden werden, die im Werkzeug als Design-Kritiken eingesetzt werden konnten. Im Folgenden werden einige dieser Kritiken vorgestellt.

Team ohne Rollen Besitzt eine Teamklasse keine Rollen, dann hat die Benutzung des Teamkonstrukts keinen Sinn, weil diese Klasse als eine gewöhnliche Java-Klasse definiert werden kann.

Abgeleitetes Team nur mit impliziten Rollen Das vorherige Szenario kann wie folgt erweitert werden: Ein Team spezialisiert ein anderes Team, verfeinert jedoch die geerbten Rollen nicht und definiert keine neuen Rollenklassen. So eine Spezialisierung erhöht unnötig die Komplexität, da sie an dieser Stelle nutzlos ist.

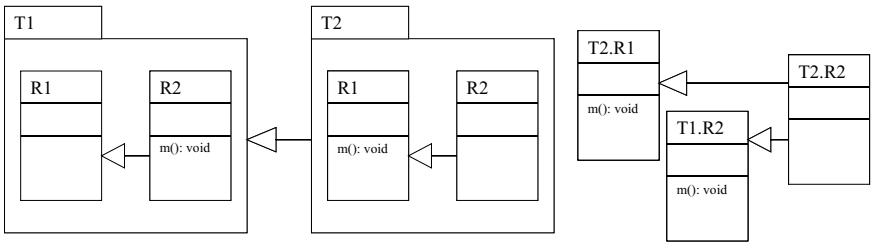
Konkretes Team ohne eine Fassade und callin-Bindungen Funktionalitäten, die in Rollenklassen eines Teams implementiert werden, können

über die öffentlichen Methoden des Teams (seine Fassade) aufgerufen werden. Wird in einem Team keine Fassade implementiert, kann der Sinn eines solchen Teams darin liegen, eine Basisanwendung über die Aktivierung von *callin*-Bindungen nachträglich zu manipulieren. Wenn allerdings keine *callin*-Bindungen in Basisklassen des Teams definiert werden, ist die Funktion eines solchen Teams fraglich.

Gemischter Konnektor Ein Konnektor soll möglichst kaum Implementierungen und dafür alle Arten von Bindungen enthalten. Wird diese Regel gebrochen, sodass in einem Team, das Bindungen enthält, zusätzliche Methoden auf der Team- oder Rollenebene definiert werden, wird die Übersichtlichkeit und damit das Verständnis des Designs unnötig erschwert.

Gebundene Rolle ohne Methodenbindungen Wenn eine Rollenklasse an eine Basisklasse gebunden wird und dabei weder die Funktionalität der Basis über *callout*-Bindungen nutzt, noch das Verhalten der Basis über *callin*-Bindungen beeinflusst, bleibt solche *playedBy*-Beziehung bedeutungslos.

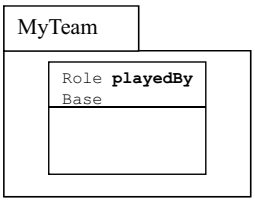
Mehrdeutigkeiten bei der multiplen Vererbung Die Mehrfachvererbung in ObjectTeams ergibt sich dann, wenn eine Rollenklasse sowohl implizit erbt als auch eine explizite Superklasse besitzt. Dabei kann es zu einer Namenskollision führen, wenn das Szenario eintritt, das in der folgenden Abbildung geschildert ist.
Diese Situation ist zwar ungefährlich, da die implizite Vererbung vom Com-



piler stärker gewichtet wird, kann aber zu Missverständnissen führen.

Rollen mit der gleichen Basisklasse Wenn Rollen eines Teams an die gleiche Basis gebunden sind aber keine Generalisierungsbeziehung miteinander haben, wird die *playedBy*-Beziehung inkonsequent genutzt.

Nichtexklusive Erwähnung von Basisklassen Wenn eine Rolle `Role` an eine Basisklasse `Base` gebunden ist, dann sollte die Funktionalität dieser Basis innerhalb des entsprechenden Teams `MyTeam` nur über ihre Rolle benutzt werden.
Um eine widersprüchliche Benutzung der `playedBy`-Beziehung zu vermeiden,



sollte `Base` als Typ außer hinter dem Schlüsselwort `playedBy` nur an folgenden Stellen erwähnt werden:

- in einer Methodensignatur von `MyTeam` als Teil eines Liftingtyps vor dem Schlüsselwort `as`
- als Rückgabetyp der Basismethode in der Signaturanpassung bei einer Methodenbindung
- als Parameter der Basismethode in der Signaturanpassung bei einer Methodenbindung

Abstrakte Rolle ohne abstrakte Methoden Hat eine Rollenklasse keine abstrakten Methoden, dann sollte sie nicht als abstrakt gekennzeichnet werden.

Abstraktes Team ohne abstrakte Rollen oder Methoden Das vorbezeichnete Szenario kann auf Teams erweitert werden. Besitzt ein Team keine abstrakten Rollen und keine zu implementierenden Methodendeklarationen, dann sollte es nicht mit dem Schlüsselwort `abstract` markiert werden.

5.3 Realisierung

Die Vorgehensweise eines Design-Kritiken-Werkzeugs beruht auf der statischen Programmanalyse, wobei die Repräsentation eines Programmsystems nach den charakteristischen Merkmalen einer Kritik durchsucht wird. Die Suche nach den Eigenschaften eines OT-Programms kann hinsichtlich der zwei verschiedenen Datenrepräsentationen (Java und XML) auf zwei unterschiedliche Weisen realisiert werden. Im Folgenden werden zwei Alternativen der

Kritikimplementierung anhand eines Beispiels vorgestellt und miteinander verglichen.

5.3.1 Java-Lösung

Der DOM/AST in Eclipse bietet eine zweckmäßige Datenrepräsentation an, OT-Programme nach bestimmten Eigenschaften zu durchsuchen. Dazu muss der AST traversiert werden, was am besten mit Hilfe eines *Visitors* realisiert werden kann.

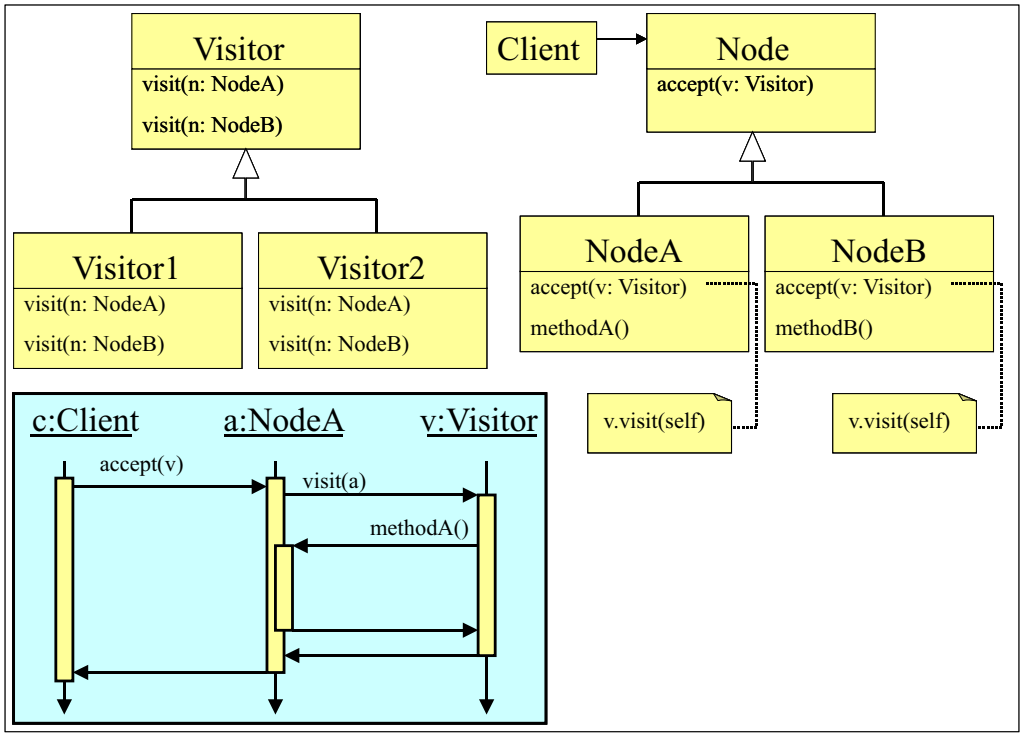


Abbildung 5.1: *Entwurfsmuster Visitor*

5.3.1.1 Entwurfsmuster *Visitor*

Visitor stellt eine Lösung dar, Datenstrukturen neue Verarbeitungsalgorithmen hinzuzufügen. Das Entwurfsmuster besteht aus einem Interface *Visitor* und einer (oft baumförmigen) Datenstruktur, deren Komponentenklassen eine gemeinsame Superklasse besitzen.
Im Interface *Visitor* werden `visit`-Methoden deklariert, welche als Parameter eine Klasse der Datenstruktur erhalten und beim Traversieren der jeweili-

gen Argumentklasse aufgerufen werden. Dazu erhalten die Klassen der Datenstruktur eine Methode `accept(Visitor)`, welche die entsprechende `visit`-Methode des `Visitors` aufruft. Ein *Container*-Objekt ruft in `accept` die `accept`-Methoden seiner Komponenten auf.
Soll eine neue Operation auf der Datenstruktur definiert werden, wird eine Klasse definiert, die das Visitor-Interface implementiert (s. Abbildung 5.1). Auf der *Client*-Seite wird dieser konkrete Visitor instanziert und an die zu traversierenden Objekte der Datenstruktur übergeben.

In Eclipse wird ein für den DOM/AST implementierter *Visitor* (`ASTVisitor`) zur Verfügung gestellt. Diese abstrakte Klasse enthält für jeden Knotentyp des AST eine entsprechende `visit`-Methode. Die *Default*-Implementierung aller `visit`-Methoden aus `ASTVisitor` gibt einen wahren Wert zurück. Durch den Rückgabewert von `visit` (`true`) wird veranlasst, dass die Unterknoten des Kontextknoten ebenfalls besucht werden.

5.3.1.2 Implementierung einer Beispielkritik

Beispiel Als Beispiel wird eine Kritik implementiert, die auf eine inkonsequente Nutzung der Funktionalität von Basisklassen hinweist und im vorherigen Abschnitt 5.2 als nichtexklusive Erwähnung von Basisklassen bezeichnet wurde. Die Stellen der inkonsequenten Erwähnung sollen gefunden und dem Benutzer angezeigt werden.

Realisierung

Die Suche wird in zwei Aufgaben aufgeteilt. Zuerst werden aus einem Team alle Basisklassen ermittelt, an welche die Rollen des Teams gebunden sind. Für jede Basisklasse wird dann geprüft, ob sie innerhalb des Teams erwähnt wird:

- als Typ eines Feldes
- in der Signatur einer Methode als Rückgabetyp oder als Parameter
- als Superklasse des Teams oder seiner Rollen
- als zu implementierendes Interface

Basisklassensuche Für diese Aufgabe wurde ein eigener *Visitor* geschrieben, der von `ASTVisitor` abgeleitet wurde (siehe Listing 5.1). Er reimplementiert die Methode `visit(RoleTypeDeclaration)` (Zeile 11-16), weil Klassenbindungen in den Rollen zu finden sind. Namen von gefundenen Basisklassen werden in einem Container abgelegt, wobei darauf geachtet wird, dass gleiche Namen nur einmal gespeichert werden (Zeile 8).

Listing 5.1: *Ermittlung aller Basisklassen innerhalb eines Teams*

```
1  public class AllBaseClassesOfATeam extends ASTVisitor {
2    private Map result;
3    public AllBaseClassesOfATeam() {
4     super();
5     this.result = new HashMap();
6    }
7    private void add(String key, Name name) {
8      if(! result.containsKey(key))
9        result.put(key, name);
10   }
11   public boolean visit(RoleTypeDeclaration node) {
12     Name baseclass = node.getBaseClass();
13     if(baseclass != null)
14       add(baseclass.getFullyQualifiedName(), baseclass);
15     return false;
16   }
17 }
```

Suche nach Erwähnungen von Basisklassen In Listing 5.2 ist ein Ausschnitt der Implementierung dargestellt. Dieser *Visitor* ist auf das Ergebnis des gerade vorgestellten *Visitors* angewiesen.
Da nun Basisklassen-Namen zur Verfügung stehen, macht es Sinn, die Methode `visit(SimpleName)` zu redefinieren (Zeilen 35-43) und dort Namen von vorkommenden Typen mit den von Basisklassen zu vergleichen. Weil aber Basisklassen an einigen Stellen erwähnt werden dürfen (siehe die Kritikdefinition), muss die Traversierung bestimmter Knoten verhindert werden.

Dies wird durch die Redefinition bestimmter visit-Methoden erreicht, deren Unterknoten eventuell nicht besucht werden sollen. Zum Beispiel veranlasst die redefinierte Methode `visit(RoleTypeDeclaration node)` mit dem Aufruf von `accept` den Besuch des Superklassenknoten (Zeile 10). Der Basisklassenknoten wird absichtlich ausgelassen.
Zur besseren Übersichtlichkeit wird die Ergebnisausgabe mit der Methode `createWarning` (Zeile 40) nur angedeutet.

Listing 5.2:

```
1  public class NotExclusiveMentionOfBase extends ASTVisitor {
2    private Map baseClasses;
3    public NotExclusiveMentionOfBase(Map baseClasses) {
4      super();
5      this.baseClasses = baseClasses;
6    }
7    public boolean visit(RoleTypeDeclaration node) {
8      Iterator it;
9      if(node.getSuperclass() != null)
10       node.getSuperclass().accept(this);
11     if(!node.superInterfaces().isEmpty()) {
12       it = node.superInterfaces().iterator();
13       while(it.hasNext()) {
14         Name n = (Name) it.next();
15         n.accept(this);
16       }
17     }
18     it = node.bodyDeclarations().iterator();
19     while(it.hasNext()) {
20       BodyDeclaration d = (BodyDeclaration) it.next();
21       d.accept(this);
22     }
23     return false;
24   }
25   public boolean visit(LiftingType node) {
26     return false;
27   }
28   public boolean visit(CalloutMappingDeclaration node) {
29     return false;
30   }
31   public boolean visit(CallinMappingDeclaration node) {
32     return false;
33   }
34   public boolean visit(SimpleName node) {
35     String name = node.getIdentifier();
36     Set keys = baseClasses.keySet();
37     for(Iterator it=keys.iterator(); it.hasNext();) {
38       Object o = it.next();
39       if(name.compareTo(o) == 0)
40         createWarning(node);
41     }
42     return false;
43   }
44 }
```

5.3.2 XML basierte Lösung

Unter Gebrauch der XML-Datenrepräsentation von OT kann die Abfragesprache XQuery (siehe 2.3.4.3) verwendet werden, um Kritiken als Abfragen zu formulieren. Um die Auswertung von definierten Abfragen zu ermöglichen, wurde Saxon [20] (eine Sammlung von offenen Werkzeugen für die Verarbeitung von XML-Dokumenten) eingesetzt. Saxon stellt unter anderen XPath 2.0 und XQuery 1.0 Prozessoren bereit. Die Abfragenauswertung kann aus einer Java-Anwendung mittels einer Java-API veranlasst werden.
Um die Suche mittels der XML-Abfragesprache zu demonstrieren, wird die oben in Java implementierte Kritik in XQuery definiert.

5.3.2.1 Implementierung der Beispielkritik

In Listing 5.3 ist eine XQuery-Abfrage definiert, die alle inkonsequenten Erwähnungen von Basisklassen liefert, die innerhalb des Kontextknoten vorkommen. Das Ergebnis der Abfrage ist ein XML-Element `NotExklusive-MentionOfBase`, das mehrere `mention`-Elemente enthält, die eine Basisklassenerwähnung beschreiben.
Die Suche geht über alle Teams des Kontextknoten (Zeile 2). In Zeile 3 werden

Listing 5.3: *XQuery-Implementierung*

```
1  element NotExklusiveMentionOfBase {
2    for $team in //team
3    for $base in ot:distinct-types($team//role/playedBy/type)
4    return
5      for $type in $team//extends/type|$team//implements/type|
6                   $team//method/type|$team//field/type|
7                   $team//method/parameter/type[empty(@as)]
8      return
9      if(ot:equalTypes($base, $type)) then
10       ot:createWarning($team, $type)
11     else ()
12 }
13 declare function ot:createWarning(
14     $team as element(), $type as element())  as element() {
15   ...
16   element mention { ... }
17 };
```

Typ-Elemente aller Basisklassen, die innerhalb eines Teams gebunden sind, ermittelt. Dabei wird das Mehrfachvorkommen der Namen in der Funktion `distinct-types` eliminiert. In Zeilen 5-7 werden alle Typen ausgewählt,

die an den zu untersuchenden Stellen im Team erwähnt werden. Der Ausdruck `$team//field/type` liefert z.B. eine Sequenz von `type`-Elementen, deren Elternknoten `field` heißt und innerhalb eines Elements aus der Sequenz `$team` enthalten ist. *Lifting*-Typen der Methodenparameter werden durch den Prädikat-Ausdruck `[empty(@as)]` ausgegrenzt (Zeile 7).
Alle ausgewählten `type`-Elemente werden mit den Basisklassentypen verglichen (Zeile 9), wobei beim Übereinstimmen ein entsprechendes `mention`-Element in der Funktion `createWarning` erzeugt wird.

5.3.3 Vergleich der alternativen Realisierungen

Im Folgenden werden zwei alternative Techniken: Java und XQuery hinsichtlich der Formulierung und Auswertung von Design-Kritiken für OT in Bezug auf verschiedene Kriterien gegenübergestellt.

Definition von Kritiken

Beim Vergleich der Beispielkritikimplementierungen aus dem vorherigen Unterabschnitt fällt als erstes der Größenunterschied der Lösungen auf. In der Tat sind XQuery-Abfragen sehr flexibel und ermöglichen nicht nur bei diesem Beispiel, Suchanweisungen kompakt zu formulieren.
Dies liegt zum Einen an der Mächtigkeit der XPath-Ausdrücke. Die Navigation durch den Baum in verschiedene Richtungen kann intuitiv durch die Angabe eines Pfads realisiert werden. Mit `//` können verschiedene Ebenen der Hierarchie in einem Schritt erreicht werden. Gerade solche Aufgabenstellungen wie „Finde alle Elemente mit der gleichen Eigenschaft" können mit einem Ausdruck definiert werden.
Zum Anderen kann XQuery vielmehr als nur abfragen, sie hat die Möglichkeiten einer funktionalen Programmiersprache, Funktionalitäten in Funktionen auszulagern und bereits definierte Funktionen zu importieren.
Außerdem können XQuery-Abfragen flexibel miteinander kombiniert werden. Dagegen muss in Java für jede neue Operation auf dem AST ein eigener Visitor geschrieben werden, der von `ASTVisitor` abgeleitet wird.

Darstellung von Ergebnissen

Die Ergebnisse einer Design-Kritik-Auswertung sollen dem Benutzer in einer geeigneten Form dargestellt werden. Während bei einer Java-Realsierung Ergebnisse sofort weiterverarbeitet werden und Möglichkeiten einer Eclipse-Ausgabe von Warnungen in Anspruch genommen werden können, stellt das Ergebnis der XQuery-Abfrage ein XML-Dokument dar. Um das Ergebnis in

Eclipse zu visualisieren, muss das erzeugte XML-Dokument nach Java konvertiert werden.
Für Ergebnisse von XQuery-Abfragen existiert auch eine andere Möglichkeit der Darstellung: Das erzeugte XML-Dokument kann mittels geeigneter XSLT-Formatierungsvorlagen in ein HTML- oder Pdf-Dokument ungewandelt werden.

Struktur der Datenbasis

Ein wichtiger Unterschied der eben vorgestellten Implementierungen ist die Tiefe der durchgeführten Suche. Da bei der XML-Darstellung die Vererbungshierarchie bereits flachgeklopft wurde, ist eine tiefere Analyse der Daten ohne weiteren Aufwand möglich.
Das Beispiel in der Abbildung 5.2 zeigt eine irreführende Konstellation in der

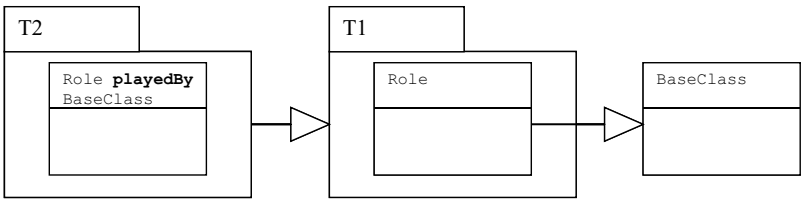


Abbildung 5.2: *BaseClass tritt als indirekte Superklasse und als Basis von T2.Role auf*

Rolle `T2.Role`. Sie erbt indirekt von der Klasse `BaseClass` und wird an sie gebunden.
Während die in XQuery implementierte Lösung vor so einem Design warnt, bleibt diese Inkonsequenz in der Java-Lösung unentdeckt. Um eine entsprechende Analysetiefe in Java zu ereichen, wäre eine viel komplexere Suche nötig, da auch Bindings traversiert werden müssten, sodass eine von `ASTVisitor` abgeleitete Klasse nicht ausreichen würde.

6 Zusammenfassung

6.1 Zusammenfassung und Fazit

Die Größe und Komplexität von modernen Software-Systemen auf der einen Seite und wirtschaftliche Aspekte wie konkurrenzfähige Kosten auf der anderen Seite stellen hohe Anforderungen an ihre Entwicklung. Der aspektorientierte Ansatz ObjectTeams bietet zusätzlich zu den objektorientierten Features, eine gute Modularisierung und nicht-invasive Adaptierung von Software-Systemen und verbessert dadurch ihre qualitativen Eigenschaften wie Verständlichkeit, Wartbarkeit und Erweiterbarkeit.
Damit die Qualität der OT-Programmsysteme und deren Entwicklung erhöht werden kann, ist die neue Sprache auf eine gute Werkzeugunterstützung angewiesen. Im Rahmen dieser Arbeit wurde eine gemeinsame Basis für die Implementierung von Werkzeugen entwickelt, welche zur Kategorie der Qualitätssicherung gehören und eine statische Programmanalyse verwenden. Bei diesen Werkzeugen werden die Daten statisch nach bestimmten Kriterien durchsucht oder in eine spezielle Darstellung umgewandelt.
Der *Toolbox* liegt eine XML-Repräsentation der Programmdaten zugrunde. Die Begründung, dass XML für die Datenrepräsentation gewählt wurde, liegt zum Einen an ihrer Fähigkeit, hierarchische Daten des Quellcodes zu repräsentieren, sodass ein abstrakter Syntaxbaum beschrieben werden kann. Zum Anderen liegt es, an der Existenz von zahlreichen Werkzeugen, die XML-verarbeitende Techniken realisieren. Außerdem enthält die Java-Repräsentation einige interne Zusatzinformationen zur Realisierung der OT-Konzepte, die in XML ausgeblendet werden können.
Die nach *Eclipse* integrierte Entwicklungsumgebung für ObjectTeams/Java bietet eine gute Benutzeroberfläche und enthält eine Reihe von Werkzeugen zur Manipulation des OT-Quellcodes. Die Toolbox verwendet die vorhandenen Funktionalitäten, sodass darauf basierende Werkzeuge in die Benutzeroberfläche integriert werden können.
Entsprechend der OT-Organisation der Programmdaten in Pakete wurde eine Auszeichnungssprache definiert, die Informationen eines OT-Programms

beschreibt. Da ein XML-Dokument nur einen Wurzelknoten zulässt, wurde ein fester Wurzelknoten definiert, der alle anderen Java-Elemente enthält und in *Eclipse* dem Java-Modell (s. Abbildung 3.6) entspricht. Der Wurzelknoten besteht aus Elementen, die Paketfragmente und Klassen beschreiben. Paketfragment-Elemente enthalten andere Paketfragmente und/oder Klassen. Um vier verschiedene Klassen-Typen (Team, Rolle, Klasse, Interface) klar voneinander zu unterscheiden, wurden Elemente mit vier verschiedenen Namen verwendet.
Die Auszeichnungssprache beschreibt nicht nur Informationen aus dem AST. Wenn nur AST-Daten einer Klasse als XML-Struktur vorliegen, besteht keine Möglichkeit, Daten aus der Vererbungshierarchie der Klasse zu erhalten, wenn ihre Superklassen nicht in der gleichen XML-Struktur vorhanden sind. Außerdem kann eine Abhängigkeit von Bibliotheksklassen bestehen. Folglich müssen bei der XML-Konvertierung auch Daten aus dem *Binding*-Graph betrachtet werden.
Um diese Abhängigkeiten zu berücksichtigen, wurde eine Entscheidung getroffen, die implizite und explizite Vererbungshierarchie für die XML-Repräsentation flach zu klopfen. Dies hat den Vorteil, dass die Suche nach geerbten Features einmal zentral implementiert wird und die Ergebnisse den Werkzeugen der *Toolbox* zur Verfügung stehen. Ein Nachteil dieser Entscheidung ist die Tatsache, dass viele Informationen im XML-Dokument mehrfach vorkommen können. Insbesondere bei größeren Projekten mit ausgeprägter Vererbungshierarchie steigt die Größe des XML-Dokuments enorm.
Eine andere Realisierungsmöglichkeit besteht in der Bildung einer transitiven Hülle, die alle für die Suche relevanten Klassen umfasst, und ihrer Konvertierung nach XML. Diese Suche wäre nach XML verlagert und müsste mit der XML-Technik XQuery realisiert werden.
Für die definierte Auszeichnungssprache wurde ein XML-Konvertierer implementiert, der für die OT-Dateien-Repräsentation aus *Eclipse* eine XML-Darstellung bildet. Unter Verwendung der XML-Repräsentation wurden ein Dokumentationswerkzeug und ein Design-Kritiken-Werkzeug implementiert. Bei der Implementierung des Dokumentationswerkzeugs wurde die Sprache XSLT eingesetzt, die Stilvorlagen verwendet, um Daten u.a. nach HTML zu formatieren. Dieses Verfahren kann dann eingesetzt werden, wenn alle für die Formatierung benötigten Informationen im XML-Dokument vorhanden sind. Folglich war die Struktur des generierten XML-Dokuments von großem Vorteil.
Die Implementierung des Design-Kritiken-Werkzeugs bestand vor allem darin, Design-Kritiken zu definieren und sie in XQuery als Abfragen zu implementieren. Dabei konnte festgestellt werden, dass es mit XQuery-Abfragen möglich ist, in prägnanter Form Suchanweisungen über die Datenbasis zu

formulieren. XQuery eignet sich gut als Informationsbeschaffungstechnik für die XML-Repräsentation der Programmdaten. Zudem können mit Hilfe von XQuery einige andere Werkzeuge der analytischen Qualitätssicherung realisiert werden, bei denen es sich darum handelt, Programmdaten nach Erfüllung bestimmter Vereinbarrungen zu untersuchen.

6.2 Offene Aufgaben

Folgende Aufgaben bleiben offen:

- Die XML-Repräsentation kann bis auf die Statements erweitert werden.
- Der XML-Konvertierer berücksichtigt die implizite Vererbungshierarchie nur bei Rollen eines echten Teams. Die implizite Vererbung der Rollen einer Team-Rolle wird nicht ausgewertet. Es ist zu empfehlen, diese Aufgabe mit XQuery zu lösen.
- Eine vernünftige Verwaltung von XQuery-Abfragen sollte organisiert werden, die u.a. dem Benutzer die Möglichkeit der Formulierung von eigenen Abfragen bietet.
- *Eclipse*-Visualisierung von Ergebnissen der Auswertung von Design-Kritiken
- Definition von noch mehr Design-Kritiken, aber auch Design-Mustern.

A Auszeichnungssprache für DOM/AST

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Document Type Definition
     This XML format is used by otdoc to store objectteams
     source code.

     Typical usage:
     <?xml version="1.0"?>
     <!DOCTYPE otdoc SYSTEM "otdoc.dtd">
     <otdoc>
     ...
     </otdoc>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
<!ELEMENT otdoc
     (package|class|interface|team)*>

<!ELEMENT package
     (package|class|interface|team)*>
<!ATTLIST package
     name CDATA #REQUIRED
     docpath CDATA #IMPLIED>

<!ELEMENT class
    (documentation?,
     extends?,
     implements*,
     (constructor|field|method|class)* )>
<!ATTLIST class
     name           CDATA    #REQUIRED
     qualifiedName  CDATA    #REQUIRED
     public         (yes|no) #IMPLIED
     packageprivate (yes|no) #IMPLIED
     protected      (yes|no) #IMPLIED
     private        (yes|no) #IMPLIED
     final          (yes|no) #IMPLIED
     abstract       (yes|no) #IMPLIED
```

```
    static          (yes|no) #IMPLIED>

<!ELEMENT interface
    (documentation?, extends*, (field|method)* )>
<!ATTLIST interface
    name           CDATA    #REQUIRED
    qualifiedName  CDATA    #REQUIRED
    public         (yes|no) #IMPLIED
    packageprivate (yes|no) #IMPLIED
    protected      (yes|no) #IMPLIED
    private        (yes|no) #IMPLIED
    final          (yes|no) #IMPLIED
    implicit       (yes|no) #IMPLIED>

<!ELEMENT team
    (documentation?, extends?, extends?, implements*,
     (constructor|field|method|role)* )>
<!ATTLIST team
    name           CDATA    #REQUIRED
    qualifiedName  CDATA    #REQUIRED
    public         (yes|no) #IMPLIED
    packageprivate (yes|no) #IMPLIED
    protected      (yes|no) #IMPLIED
    private        (yes|no) #IMPLIED
    final          (yes|no) #IMPLIED
    abstract       (yes|no) #IMPLIED
    static         (yes|no) #IMPLIED>

<!ELEMENT role
    (documentation?, extends?, extends?,
     implements*, playedBy?,
     (constructor|field|method|role|callin|callout)*)>
<!ATTLIST role
    name           CDATA    #REQUIRED
    qualifiedName  CDATA    #REQUIRED
    public         (yes|no) #IMPLIED
    packageprivate (yes|no) #IMPLIED
    protected      (yes|no) #IMPLIED
    private        (yes|no) #IMPLIED
    team           (yes|no) #IMPLIED
    final          (yes|no) #IMPLIED
    abstract       (yes|no) #IMPLIED
    static         (yes|no) #IMPLIED
    implicit       (yes|no) #IMPLIED>

<!ELEMENT playedBy (type)>

<!ELEMENT documentation (description|(tag)*)>
```

```
<!ELEMENT description (#PCDATA|tag)*>

<!ELEMENT tag
    (type|#PCDATA)>
<!ATTLIST tag
    name    CDATA    #REQUIRED
    param   CDATA    #IMPLIED
    allowed (yes|no) #REQUIRED>

<!ELEMENT extends
    (type, extends?)>
<!ATTLIST extends
    implicit (yes|no) #IMPLIED>

<!ELEMENT implements
    (type)>
<!ATTLIST implements
    implicit (yes|no) #IMPLIED>

<!ELEMENT constructor
    (documentation?, parameter*, throws*)>
<!ATTLIST constructor
    name           CDATA    #REQUIRED
    qualifiedName  CDATA    #REQUIRED
    default        (yes|no) #REQUIRED
    public         (yes|no) #IMPLIED
    packageprivate (yes|no) #IMPLIED
    protected      (yes|no) #IMPLIED
    private        (yes|no) #IMPLIED
    final          (yes|no) #IMPLIED>

<!ELEMENT method
    (documentation?,
     type, parameter*, throws*,
     from?, specifiedBy*)>
<!ATTLIST method
    name           CDATA    #REQUIRED
    qualifiedName  CDATA    #REQUIRED
    callin         (yes|no) #IMPLIED
    public         (yes|no) #IMPLIED
    packageprivate (yes|no) #IMPLIED
    protected      (yes|no) #IMPLIED
    private        (yes|no) #IMPLIED
    final          (yes|no) #IMPLIED
    abstract       (yes|no) #IMPLIED
    static         (yes|no) #IMPLIED
    synchronized   (yes|no) #IMPLIED
    native         (yes|no) #IMPLIED
    implicit       (yes|no) #IMPLIED
```

```
    inherited      (yes|no) #IMPLIED>

<!ELEMENT field
    (documentation?, type, initializer?, from?)>
<!ATTLIST field
    name           CDATA    #REQUIRED
    public         (yes|no) #IMPLIED
    packageprivate (yes|no) #IMPLIED
    protected      (yes|no) #IMPLIED
    private        (yes|no) #IMPLIED
    final          (yes|no) #IMPLIED
    abstract       (yes|no) #IMPLIED
    static         (yes|no) #IMPLIED
    transient      (yes|no) #IMPLIED
    volatile       (yes|no) #IMPLIED
    implicit       (yes|no) #IMPLIED
    inherited      (yes|no) #IMPLIED>

<!ELEMENT type (link?)>
<!ATTLIST type
    qualifiedName        CDATA #REQUIRED
    name                 CDATA #REQUIRED
    qualifiedPackageName CDATA #IMPLIED
    as                   CDATA #IMPLIED
    array                (yes|no) "no"
    kind                 (class|interface|role|team|
                          return_type)
                               #IMPLIED>

<!ELEMENT callin
    (documentation?, methodspec, methodspec, with?) >
<!ATTLIST callin
    modifier (before|after|replace) "replace">

<!ELEMENT callout
    (documentation?, methodspec, methodspec, with?)>
<!ATTLIST callout
    fieldAccess     (no|get|set) "no"
    calloutoverride (yes|no)     #IMPLIED>

<!ELEMENT methodspec
    (type, parameter*, signature?)>
<!ATTLIST methodspec
    basemethodspec (yes|no) #REQUIRED
    name           CDATA    #REQUIRED
    qualifiedName  CDATA    #IMPLIED>

<!ELEMENT signature (type, parameter*)>
```

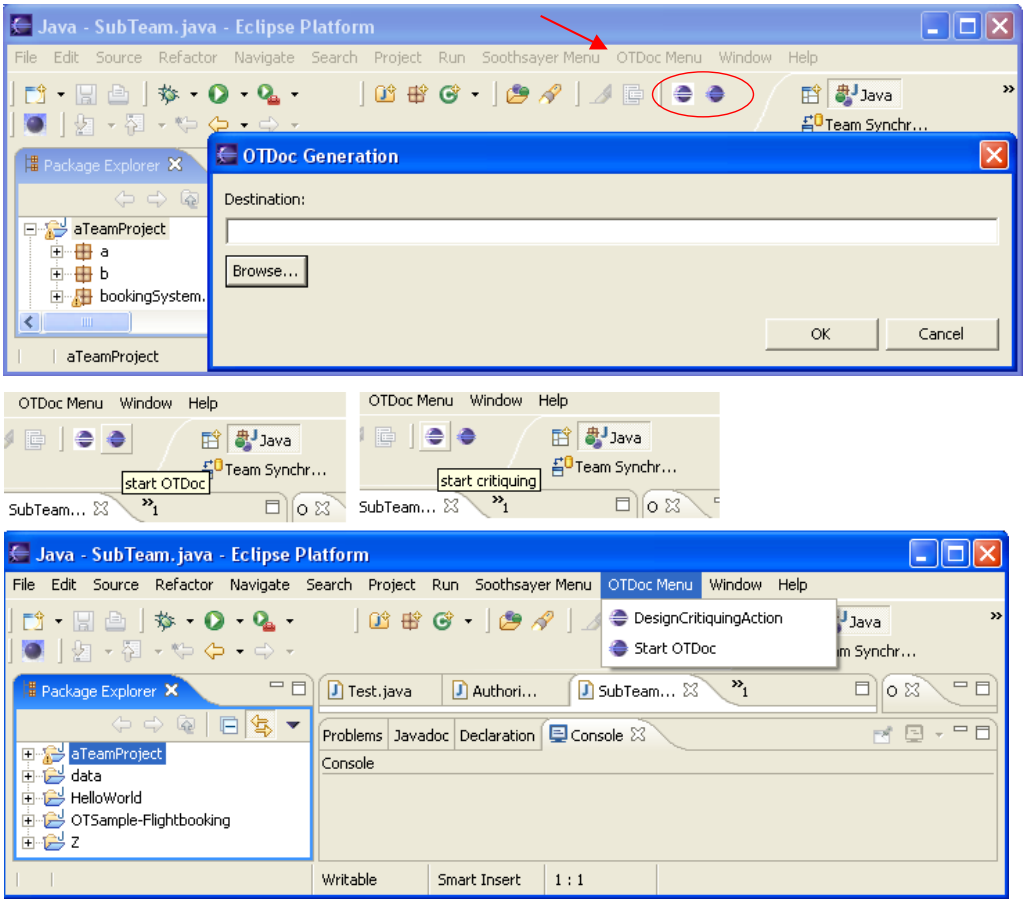
```
<!ELEMENT with (#PCDATA)>

<!ELEMENT parameter (type)>
<!ATTLIST parameter
    name CDATA #REQUIRED>

<!ELEMENT initializer (#PCDATA)>

<!ELEMENT from (type)>

<!ELEMENT specifiedBy EMPTY>
<!ATTLIST specifiedBy
    qualifiedName        CDATA             #REQUIRED
    qualifiedPackageName CDATA             #IMPLIED
    method               CDATA             #IMPLIED
    type                 (class|interface) #IMPLIED>

<!ELEMENT throws (type)>

<!ELEMENT link EMPTY>
<!ATTLIST link href CDATA #REQUIRED>
```

B Markierungen in Kommentaren bei OTDoc

Markierung	Beschreibung	Verwendung in
@author name	Erstellt einen Autoreneintrag	Klasse
@version version	Erstellt einen Versionseintrag	Klasse
@since text	Gibt an, seit wann das Feature existiert	Klasse, Feld, Methode
@see reference	Erstellt eine Verknüpfung zu einem anderen Teil der Dokumentation oder zu einer anderen Quelle im Netz bzw. erzeugt einen Texteintrag	Klasse, Feld, Methode
@deprecated text	Markiert ein veraltetes Feature	Klasse, Feld, Methode
{**@link** reference label}	Erzeugt eine Verknüpfung zu einem anderen Teil der Dokumentation aus einer Referenz der folgenden Form: *package.class#member*	Klasse, Feld, Methode
{**@linkplain** reference label}	Identisch mit {@link}, wird lediglich anders angezeigt	Klasse, Feld, Methode
{**@docRoot** }	Repräsentiert den relativen Pfad zum Wurzelverzeichnis der generierten Dokumentation	Klasse, Feld Methode
@param name text	Erzeugt eine Parameter-beschreibung	Methode
@return text	Erzeugt eine Rückgabewert-beschreibung	Methode
@throws name text	Erzeugt eine Ausnahme-behandlungsbeschreibung	Methode
@exception name text	Identisch zu @throws	Methode

Tabelle B.1: Markierungen in Dokumentationskommentaren

C Handhabung der entwickelten Werkzeuge


Java - SubTeam.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Soothsayer Menu OTDoc Menu Window Help
Java
Team Synchr...
Package Explorer
aTeamProject
a
b
bookingSystem.
aTeamProject
OTDoc Generation
Destination:
Browse...
OK
Cancel
OTDoc Menu Window Help
Java
Team Synchr...
start OTDoc
SubTeam...
OTDoc Menu Window Help
Java
Team Synchr...
start critiquing
SubTeam...
Java - SubTeam.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Soothsayer Menu OTDoc Menu Window Help
DesignCritiquingAction
Start OTDoc
Java
Team Synchr...
Package Explorer
Test.java
Authori...
SubTeam...
Problems Javadoc Declaration Console
Console
aTeamProject
data
HelloWorld
OTSample-Flightbooking
Z
Writable
Smart Insert
1 : 1

D Danksagung

An dieser Stelle danke ich allen, ohne deren Unterstützung diese Arbeit nicht zustande gekommen wäre. Gedankt sei dem lieben Gott im Himmel, meinen Eltern, dem Erfinder der automatischen Rechschreibprüfung und Thesaurus, den Herstellern der Bestandteile meines Rechners aus dem fernen China ...
Besonderer Dank gilt meinem Betreuer Dr. Stephan Herrmann. Außerdem soll den folgenden Personen namentlich gedankt werden: meinem Korrektor, der unter dem Pseudonym Francesco Adreano Miccelli erwähnt werden wollte, dem *Eclipse*-Engel Jan Wloka, meiner Notebook-Ausrüsterin Lena Ivanova, meinem stillen Mitstreiter Henry Sudhof und den INFO-Tutoren aus dem Raum FR5515 für ihre Fragen, wie weit ich schon bin.
Außerdem gilt mein Beileid meinem langjährigen Partner, dem Sternchen-Drucker, der bei dieser Arbeit den Geist aufgegeben hat. Möge er im Frieden ruhen.

Literaturverzeichnis

[1] ObjectTeams Homepage
http://www.ObjectTeams.org

[2] Stephan Herrmann
Object Teams: Improving Modularity for Crosscutting Collaborations
In: Net Object Days 2002, 2002.

[3] Stephan Herrmann
Translation polymorphism in Object Teams
Technical Report 2004/06, Technical University Berlin, 2004.

[4] Stephan Herrmann
Composable Designs with UFA
In: Workshop on Aspect-Oriented Modeling with UML, 2002.

[5] Christine Hundt
Bytecode-Transformation zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit ObjectTeams/Java
Diplomarbeit, TU Berlin, 2003.

[6] Stephan Herrmann, Christine Hundt, Katharina Mehner, Jan Wloka
Using Guard Predicates for Generalized Control of Aspects Instantiation and Activation. In: Dynamic Aspects Workshop'05, 2005.

[7] Christof Binder
Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken
Diplomarbeit, TU Berlin, 2003.

[8] Markus Witte
Portierung, Erweiterung und Integration des ObjectTeams/Java Compilers für die Entwicklungsumgebung Eclipse
Diplomarbeit, TU Berlin, 2003.

[9] JavaDoc Homepage
http://java.sun.com/j2se/javadoc

[10] Eclipse Homepage
http://www.eclipse.org

[11] Object Teams Development Tooling Homepage
http://www.ObjectTeams.org/distrib/otdt.html

[12] World Wide Web Consortium Homepage
http://www.w3.org

[13] ISO/IEC JTC1
Standard Generalized Markup Language. ISO/IEC IS 8879, 1986

[14] HTML 4.01 Specification
http://www.w3.org/html401

[15] XML Homepage
http://www.w3.org/XML

[16] XPath 2.0 Specification
http://www.w3.org/TR/xpath20

[17] XSLT 2.0 Specification
http://www.w3.org/TR/xslt20

[18] XQuery 1.0 Specification
http://www.w3.org/TR/xquery

[19] Howard Katz
XQuery from the Experts: A Guide to the W3C XML Query Language
Part 1 - Basics, Chapter 1 - XQuery: A Guided Tour, by Jonathan Robie

[20] M.H. Kay Saxon Project
http://saxon.sourceforge.net

[21] Thomas Dudziak, Jan Wloka
Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code
Diplomarbeit, TU Berlin, 2002

[22] Heinz-Peter Gumm, Manfred Sommer
Einführung in die Informatik
In: Oldenbourg-Verlag München Wien, 2002

[23] Peter Rechenberg
Editorial zu Werkzeuge der Softwaretechnik
In: Elektronische Rechenanlagen

[24] Michael Krüger
Definition und Implementierung von Metriken für die aspektorientierte Programmiersprache ObjectTeams/Java
Diplomarbeit, TU Berlin, 2006.