Entwicklung einer generischen Schnittstelle zur Instrumentierung objektorientierter Software und Testdurchführung

Diplomarbeit im Studiengang Informatik
Technische Universität Berlin

Fakultät IV, Fachgebiet Softwaretechnik,
Institut Softwaretechnik und Theoretische Informatik

Franklinstraße 28/29, 10587 Berlin, Germany

24. September 2004

Alexander Winter
Matrikelnummer 189503

Gutachter:

Professor S. Jähnichen
Dipl.-Inform. D. Sokenou

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den

_____

Inhaltsverzeichnis

Einleitung

Die Entwicklung von Software ist in den letzten Jahrzehnten rasant fortgeschritten. Immer leistungsfähigere Hard- und Software sowie neue Verfahren, die in Forschung und Industrie entwickelt werden, lassen immer größere und komplexere Softwarepakete zu. Gleichzeitig steigen auch die Anforderungen an die entwickelte Software, vor allem in Hinblick auf Qualität und Leistungsfähigkeit[1]. In Zeiten stärkerer Globalisierung spielen außerdem wirtschaftliche Gesichtspunkte eine Rolle, so daß in der Softwareentwicklung zwei Fragen berücksichtigt werden müssen: Wie entwickelt man *gute* Software (also Software, die den Anforderungen genügt) und wie entwickelt man Software *gut* (d. h. möglichst schnell, effektiv und kostensparend)?

Um diese beiden teilweise in Konflikt stehenden Ziele möglichst gut sicherzustellen, wurden in der Softwaretechnik verschiedene Modelle entwickelt, die den Prozeß der Softwareentwicklung systematisieren und genauer festlegen[2]. Jedes Modell definiert dabei verschiedene Tätigkeiten, die während der Entwicklung von Software durchzuführen sind, und zwar unabhängig von eingesetzten Entwicklungswerkzeugen, Programmiersprachen oder der verwendeten Infrastruktur.

Mit dem Zuwachs des Umfangs von Softwarepaketen sorgten die steigenden Anforderungen dafür, daß zwei Punkte immer wichtiger wurden:

- Viele Tätigkeiten konnten aufgrund des Umfangs und der zur Verfügung stehenden Zeit nicht mehr von Hand erledigt werden. Es mußten Werkzeuge zur Unterstützung des Entwicklungsprozesses zur Verfügung stehen.

- Bei der Zusammenstellung größerer Softwarepakete konnte man sich nicht darauf verlassen, daß die Software nach Auslieferung fehlerfrei funktioniert. Idealerweise sollte die Software schon **vor** der Auslieferung und dem praktischen Einsatz getestet werden. Testen von Software wurde zu einem wesentlichen Bestandteil des Entwicklungsprozesses.

Für das Testen von Software sind im Laufe der Zeit ebenfalls Methoden und passende Werkzeuge entwickelt worden. Diese Werkzeuge unterstützen in den meisten Fällen eine spezielle Programmiersprache oder eine spezielle Testmethode. Häufig soll jedoch die Software auf verschiedenen Systemen aufsetzen, die unterschiedliche Anforderungen stellen. In vielen Fällen wird für eine Umsetzung auf verschiedene Systeme die Programmiersprache gewechselt oder die benutzte Sprache im Umfang und den Fähigkeiten angepaßt. Um aufwendige Tests auf allen Plattformen zu vermeiden und den Entwurf möglichst früh

[1] Anschaulich beschrieben wird diese Entwicklung in [CK03] (unter wirtschaftlichen und teilweise softwaretechnischen Aspekten). Ein bekanntes Beispiel zeigt diese Entwicklung ebenfalls: Laut [Whe04] umfaßt Windows XP 40 000, der Linux-Kernel von RedHat 30 000 Code-Zeilen!

[2] Methoden und Modelle werden z. B. in [Som02] oder [Dou99] beschrieben.

zu prüfen, wird, sofern es möglich ist, eine *Validierung* der Software durchgeführt. Dabei erstellt man ein abstraktes Modell der Software, anhand dessen mit Hilfe mathematischer Verfahren geprüft werden kann, ob eine bestimmte Anforderung zutrifft oder nicht. Auch hier existieren Werkzeuge zur Unterstützung[3]. Beide Verfahren werden in der Softwareentwicklung häufig nebeneinander eingesetzt und ergänzen sich gegenseitig (eine gute Einführung in die Kombination verschiedener Verfahren liefert [HNSS00] – hier werden weitere Ansätze genannt und kurz erläutert).

Eine Erleichterung des Verfahrens wäre gegeben, wenn die Werkzeugunterstützung zum Testen plattform- bzw. sprachunabhängig wäre. Der Prozeß würde sich dadurch an folgenden Stellen vereinfachen:

- Ändert sich die Sprache für die Umsetzung der Software, ist kein neues Werkzeug mehr für die Testdurchführung notwendig. Auf diese Weise spart sich der Tester die Einarbeitung in ein neues Testwerkzeug.

- Ein Werkzeug zur Validierung könnte die auf dem unvollständigen Modell geprüften Anforderungen mit geringerem Aufwand an das Testwerkzeug weitergeben. Definiert jedes Testwerkzeug sein eigenes Format für die Darstellung durchzuführender Tests, ist die Übertragung umständlich. Zudem übernimmt jetzt das Testwerkzeug die Anbindung an die konkrete Programmiersprache, die für die Realisierung des vorgegebenen Modells verwendet wurde. Die Darstellung für die Tests kann dann so gewählt werden, daß sie näher an der Darstellung des abstrakten Modells liegt, weil die Transformation der Darstellung nicht mehr „von Hand“ erfolgen muß.

Im folgenden soll eine Umsetzbarkeit dieses Ansatzes geprüft werden. Dabei wird die oben genannte Idee an einigen Stellen eingeschränkt und vereinfacht, um den Rahmen nicht zu sprengen. Der nächste Abschnitt beschreibt die Umsetzungsidee genauer.

Zielsetzung und Gliederung der Arbeit

Da ein Testwerkzeug je nach Zweck und Einsatzbereich vollkommen unterschiedliche Anforderungen erfüllen muß (auf die später noch genauer eingegangen wird), ist es sinnvoll, die konkrete Realisierung eines solchen Werkzeugs offenzuhalten. Aus diesem Grund wird lediglich eine Schnittstelle beschrieben, deren Aufgabe es ist, die notwendige Vorbereitung der Tests durchzuführen und den Testablauf zu unterstützen. Die Schnittstelle soll dabei in zwei Richtungen offen gehalten werden:

- Mit Hilfe der Schnittstelle soll es möglich sein, ein funktionsfähiges Testwerkzeug aufzubauen. Die Anbindung an die konkrete Software ist dabei Aufgabe der Schnittstelle, der Programmierer legt alle anderen Details fest (z. B. Format der Datenspeicherung, konkrete Durchführung der Tests o. ä.).

- Gleichzeitig soll die Schnittstelle erweiterbar sein, so daß auch andere Funktionalitäten aufgenommen werden können, wenn es nötig wird.

Eine vollkommen generische Schnittstelle ist allerdings schwierig realisierbar. Für die Entwicklung eines Ansatzes sollen daher drei konkrete Programmiersprachen betrachtet werden:

[3] Ein weit verbreitetes Verfahren ist das *Model Checking* – da hier die Modelle recht groß sind, kommt man ohne Werkzeugunterstützung nicht aus. Zwei bekannte Werkzeuge sind z. B. SPIN (*http://www.lucenttls.com*) und SMV (*http://www-2.cs.cmu.edu/∼modelcheck/smv.html*).

- **Eiffel**: Diese Sprache bietet von sich aus bereits gute Unterstützung beim Entwurf, da spezielle Konstrukte zur Formulierung von Anforderungen bereits integriert werden. Eiffel wurde 1986 von Bertrand Meyer entwickelt und ist bei den objektorientierten Sprachen bisher die einzige, die Programmierung und Spezifikation innerhalb der Sprache integriert. Zudem haben sich die Entwickler von Eiffel bemüht, eine stimmiges objektorientiertes Konzept zu entwickeln, das Inkonsistenzen zu vermeiden sucht. Für die Schnittstelle bietet die starke Strukturierung einen guten Ausgangspunkt für die Ermittlung grundlegender Anforderungen[4].

- **C++**: C++ ist aus der Sprache C hervorgegangen und sehr hardwarenah, was eine gute Geschwindigkeit der Programme bewirkt. Gleichzeitig bietet die Sprache sehr mächtige Mittel, um Algorithmen elegant und effizient umzusetzen. C++ wurde 1983-1985 von Bjarne Stroustrup als Erweiterung von C geschrieben. C selbst wurde bei IBM 1969 von Dennis Ritchie entwickelt und war Nachfolger der Sprache B, die Ken Thompson entwarf. C entstand unter anderem im Zuge der Unix-Entwicklung[5]. Neben der weiten Verbreitung ist C++ im Hinblick auf seine Mächtigkeit ein guter Indikator, welche Konzepte sich für objektorientierte Sprachen innerhalb der Schnittstelle realisieren lassen.

- **Java**: Die von Sun Microsystems entwickelte Programmiersprache ist heute weit verbreitet. Java ist eine der jüngeren Programmiersprachen und wurde bei SUN von Patrick Naughton, Mike Sheridan und James Gosling 1995 entwickelt. Das Konzept wurde nach einer ersten Konsolenversion von Sun weiterentwickelt und um eine umfangreiche Programmierbibliothek erweitert. Damit war die Sprache auch auf anderen Systemen einsetzbar[6]. Java ist im Hinblick auf Mächtigkeit, Effizienz und Strukturierung ein guter Kompromiß zwischen C++ und Eiffel. Von den drei genannten Sprachen benutzt nur Java zur Ausführung einen Interpreter und soll damit als Stellvertreter für interpretierte Sprachen in Betracht gezogen werden.

Die Umsetzung der Schnittstelle wurde auf dem Betriebssytem Linux vorgenommen. Als Programmiersprache zur Realisierung diente C++.

Im folgenden Teil werden einzelne Aspekte erläutert und im Hinblick auf diese Aspekte die Anforderungen genauer festgelegt, während der zweite Teil die Umsetzung detailliert beschreibt. Die beiden Teile gliedern sich dabei in sechs Kapitel, die wie folgt aufgeteilt sind:

- Im **ersten Kapitel** werden kurz die allgemeinen Merkmale objektorientierter Sprachen zusammengefaßt und C++, Eiffel und Java in Bezug auf ihre speziellen Eigenschaften verglichen. Weiterhin wird der Begriff des Testens genauer gefaßt und darüber hinaus beschrieben, wo die Besonderheiten beim Testen objektorientierter Programme liegen. Schließlich gibt das Kapitel einen kurzen Überblick über einige Werkzeuge, die zum Testen objektorientierter Sprachen eingesetzt werden.

- Im **zweiten Kapitel** wird der Begriff der Instrumentierung genauer erläutert und verschiedene Ansätze verglichen. Da moderne Sprachen meist nicht in einem Schritt vom Quellcode in ein lauffähiges Programm umgesetzt werden, kann die Instrumentierung an verschiedenen Stellen ansetzen. Aus diesem Grund werden in diesem Kapitel zusätzlich einige Bereiche des Compilerbaus erläutert. Gleichzeitig macht

[4] Historische Informationen findet man unter *http://archive.eiffel.com/general/people/meyer/*.
[5] Diese Entwicklung ist nachzulesen unter *http://www.hitmill.com/programming/cpp/cppHistory.asp*.
[6] Genaueres findet man unter *http://java.sun.com/features/1998/05/birthday.html*.

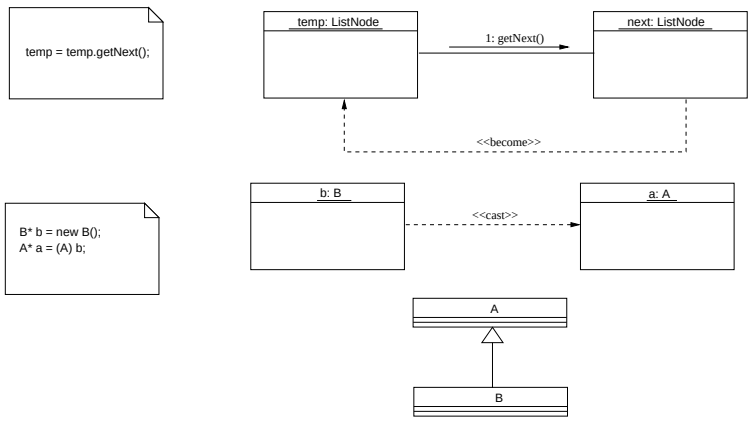

Abbildung 1: Bedeutung der zusätzlich eingeführten Tags.

dieser Abschnitt auch deutlich, wie eine konkrete Programmiersprache aus einzelnen atomaren Bestandteilen aufgebaut wird. Dies wird für die spätere Umsetzung wichtig.

- Der erste Teil wird durch das **dritte Kapitel** abgeschlossen. Hier werden vorhandene Mechanismen und Möglichkeiten betrachtet, durch die unterschiedliche Sprachen miteinander kommunizieren können. Zwei weit verbreitete Standards, .NET und CORBA, stellen dabei die bisher umfangreichsten Ansätze dar und werden separat erläutert.

- Der zweite Teil beginnt mit einer allgemeinen Festlegung und Zusammenfassung der Anforderungen. Anschließend wird der erste Teil der Umsetzung im **vierten Kapitel** beschrieben. Hier wird dargestellt, auf welche Weise mit Hilfe der Schnittstelle ein Einlesen von Informationen eines objektorientierten Programms möglich ist.

- Das folgende Kapitel, **Kapitel 5**, beschreibt die Generierung der Informationen, die später zur Anbindung der einzelnen Sprachen notwendig sind.

- Den Abschluß des zweiten Teils bildet **Kapitel 6**, wo die eigentliche Instrumentierung und die Benutzung der Schnittstelle erläutert wird. Zugleich wird auf Besonderheiten und Probleme bei der Instrumentierung eingegangen.

Abschließend bewertet eine Zusammenfassung rückblickend die Vor- und Nachteile des Ansatzes und gibt einen Ausblick auf die Erweiterungs- und Nutzungsmöglichkeiten der Schnittstelle.

Zur Darstellung der Schnittstellenfunktionalität und zur Angabe von Beispielen wird die UML-Notation verwendet, die in der Softwareentwicklung weit verbreitet und gebräuchlich ist. Genaueres findet man unter [OMG04]. Dabei werden zwei Erweiterungen vorgenommen, die für die Darstellung dynamischer Prozesse notwendig sind : Zum einen wird der Stereotype `<<become>>` dort verwendet, wo ausgedrückt werden soll, daß ein Objekt mit einer Bezeichnung später unter einer anderen wiederverwedendet wird, zum anderen beschreibt das Stereotype `<<cast>>` die Tatsache, daß für ein Objekt zu einem geeigneten Zeitpunkt ein Cast in eine andere Klasse vorgenommen wird. Abbildung 1 zeigt zwei Zeilen Programmcode und die entsprechende Notation.

Teil I

Anforderungen und zugrundeliegende Konzepte

Kapitel 1

Objektorientierung und objektorientiertes Testen

1.1 Objektorientierte Sprachen

Erste objektorientierte Sprachen entstanden bereits in den sechziger Jahren. Die bisherigen Programmierparadigmen gründeten entweder auf mathematischen Formalismen oder auf der Idee, daß ein Programm eine Abfolge von einzelnen Schritten ist. Der zweite Ansatz mündete in die iterative Programmierung[1], während aus dem ersten funktionale oder funktional-logische Programmiersprachen wie LISP oder Prolog entstanden. Der objektorientierte Ansatz unterschied sich von anderen Ansätzen durch folgende Merkmale[2]:

- Kleinste atomare Einheit eines Programms ist ein *Objekt*. Jedes Objekt zeichnet sich durch einen *Zustand* und eine *Identität* aus, durch die es innerhalb des Programms eindeutig identifizierbar ist. Der Zustand wird durch verschiedene *Attribute* beschrieben, die verschiedene Werte eines zugeordneten Wertebereichs annehmen können. Da Attribute auch selbst wieder Objekte sein können, sind *Kompositionen* von Objekten möglich. In den meisten Fällen definieren objektorientierte Sprachen außerdem *Basistypen*, die keine Objekte sind, wie z. B. ganze Zahlen oder Zeichenketten, die dann zum Aufbau anderer Objekte verwendet werden können.
- Jedes Objekt reagiert auf eine Anzahl von *Nachrichten* und verändert je nach Art dieser Nachricht seinen Zustand. Da diese Nachrichten zu einem Objekt gehören, gibt man sie meist in Form von *Methoden* an, die zum Objekt gehören und durch einen entsprechenden Methodenaufruf aktiviert werden. Um das Objekt anzusprechen, ist dabei ein Verweis notwendig, der häufig in Form einer *Referenz* angegeben wird. Die Referenz legt dabei die Sichtweise auf das Objekt und die Methoden fest, die über sie aufgerufen werden können. Da jedes Objekt eine Identität besitzt, kann es zudem auch eine Referenz auf sich selbst an andere Objekte weiterreichen.
- Jedes Objekt wird in der Regel genau einer *Klasse* zugeordnet[3]. Objekte einer Klasse besitzen dieselben Attribute und Methoden und dienen als Abstraktionsmittel innerhalb der Programmierung. Innerhalb des Quellcodes wird durch die Beschreibung

[1] Typische Vertreter dieser Idee sind z. B. C, Basic, und FORTRAN.

[2] Einige dieser Punkte sind aus [JH02] übernommen worden. Der Artikel spricht außerdem weitergehende Aspekte objektorientierter Programmierung und Modellierung an.

[3] Ausnahmen sind z. B. klassenlose objektorientierte Programmiersprachen, wie z. B. Self.

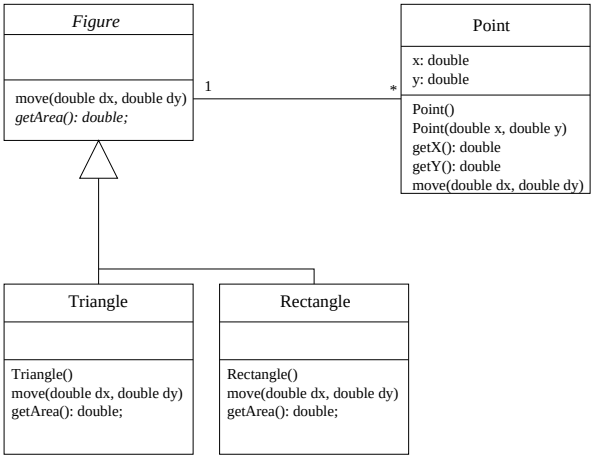

Abbildung 1.1: Beispiel eines objektorientierten Modells.

der Klasse dargestellt, welche Methoden und Attribute ein zugehöriges Objekt besitzt, die Klasse ist also eine Schablone für zugehörige Objekte.

- Eine weitere wesentliche Eigenschaft von Objektorientierung ist die *Vererbung*. Mit Hilfe der Vererbung läßt sich eine „ist ein"-Beziehung darstellen. Ist ein Objekt in Bezug auf Attribute und Methoden spezieller als ein anderes, so erzeugt man eine neue Klasse, die die Eigenschaften der alten erbt. Die speziellen Eigenschaften werden dann in der neuen Klasse implementiert, die allgemeinen Eigenschaften der alten werden in diesem Fall wiederverwendet.

Das Beispiel in Abbildung 1.1 soll die beschriebenen Punkte deutlicher machen. Das Klassendiagramm zeigt die vier Klassen *Figure*, *Triangle*, *Rectangle* und *Point*. Ein Figurenobjekt besteht aus mehreren Punkten bzw. Punktobjekten. Gleichzeitig stellen *Rectangle* und *Triangle* spezielle Ausprägungen einer Figur oder *Subklassen* von *Figure* dar, weil jedes Rechteck oder Dreieck auch eine Figur ist (und damit mehrere Punkte besitzt). Jedes Figurenobjekt besitzt die Methoden *move()* und *getArea()*, d. h. eine Figur läßt sich (innerhalb eines Koordinatensystems) bewegen und die Fläche ist berechenbar. Die Verbung stellt darüber hinaus sicher, daß an jeder Stelle, an der eine Figur einsetzbar ist, auch ein Rechteck oder ein Dreieck benutzt werden kann. In diesem Fall ist *Figure* also *Basis*- oder *Superklasse* von *Triangle* und *Rectangle*). Die „ist ein"-Beziehung ist hier deutlich erkennbar: Ein Rechteck ist eine Figur mit speziellen Eigenschaften, ebenso ein Dreieck (wobei die speziellen Eigenschaften unterschiedlich ausfallen).

Da bei einer allgemeinen Figur nicht festgelegt ist, wie die Flächenberechnung vorgenommen wird, bieten die heutigen Programmiersprachen Möglichkeiten an, Methoden oder Klassen als *abstrakt* zu kennzeichnen. Auf diese Weise läßt sich angeben, wie eine Methode aufgerufen wird, wobei die konkrete Umsetzung der Methode in einer Subklasse vorgenommen werden muß. Die abstrakte Klasse stellt also nur die *Signatur* bzw. das *Interface* einer Methode zur Verfügung. In der Abbildung ist dies durch die schräggestellte Schrift der Methode *getArea()* zu erkennen.

Zwei besondere Methoden jeder Klasse sind *Konstruktoren* und *Destruktoren*, die nach der Erzeugung bzw. vor der Zerstörung eines Objekts aufgerufen werden. Der Konstruktor (in der Abbildung durch eine Methode mit dem Namen der jeweiligen Klasse bezeichnet) setzt ein Objekt einer Klasse nach der Erzeugung in einen konsistenten Zustand. Der

Destruktor erledigt wichtige Arbeiten, die vor der Zerstörung des Objekts notwendig sind (z. B. das Schließen einer offenen Datei). Wird kein eigener Konstruktor für die Klasse vorgegeben, erzeugen die meisten Sprachen einen *Standardkonstruktor*, der technische Maßnahmen zur Einrichtung des Objekts durchführt und unter Umständen die Attribute eines Objekts mit Standardwerten belegt. Einige Sprachen verzichten außerdem auf einen expliziten Destruktor und nehmen dem Programmierer die Entscheidung ab, wann ein Objekt entfernt werden muß. Hier kümmert sich ein automatischer *Garbage Collector* darum, alle Objekte zu beseitigen, die nicht mehr benötigt werden. Für vorher zu erledigende Aufgaben muß dann eine eigene Methode implementiert werden, die man explizit aufruft.

Im Hinblick auf Methoden und im besonderen auch bei Konstruktoren spielt das *Überladen* und *Überschreiben* von Methoden eine wichtige Rolle. In einer Klasse können mehrere Methoden gleichen Namens existieren, solange sie sich durch Argumentanzahl oder Argumenttypen unterscheiden. Auf diese Weise kann beim Aufruf einer solchen überladenen Methode eindeutig zugeordnet werden, welche Version einer Methode benutzt wird. Im obigen Beispiel wäre eine weitere Methode *move()* ohne Argumente denkbar, die ein Objekt z. B. auf den Koordinatenursprung verschiebt. Bei der Vererbung können außerdem Methoden, die bereits in der Superklasse vorhanden waren, durch Überschreiben neu definiert werden, um das Verhalten anzupassen. Hätte im obigen Programm die Klasse *Rectangle* eine Subklasse *Square*, wäre es sinnvoll, die Methode *getArea()* zu überschreiben, weil sich die Berechnung der Fläche erheblich vereinfacht.

Baut man die Klassenhierarchie weiter aus, so wäre es denkbar, daß weitere Klassen wie z. B. Trapez, Drachen oder Raute hinzukommen. Da ein Quadrat die Eigenschaften eines Rechtecks und einer Raute besitzt, wäre es wünschenswert, wenn man dies auch innerhalb des Modells und bei der Implementierung ausdrücken könnte. Einige Sprachen führen für solche Fälle die *Mehrfachvererbung* ein. In diesem Fall kann eine Klasse nicht nur Subklasse eines, sondern mehrerer anderer Klassen sein. Daraus ergeben sich allerdings weitere Probleme: Wenn *Square* von *Rhomb* (Raute) und *Rectangle* erbt, erhebt sich die Frage: Welche Methode zur Flächenberechnung soll übernommen werden, wenn *Square* diese Methode nicht überschreibt? Hier treffen einzelne Sprachen unterschiedliche Festlegungen, so daß bei der Übertragung von einer Sprache in eine andere Klassen mit Mehrfachvererbung sorgfältig geprüft werden müssen.

Tabelle 1.1 zeigt die Festlegungen für einige Merkmale von C++, Java und Eiffel im Vergleich.

Merkmal	C++	Java	Eiffel
Typen	Unterscheidung zwischen Objekttypen und Basistypen (Zeichen, Ganzzahlen, ...), die keine Objekte sind.	Auch hier gibt es eine Unterscheidung zwischen Basistypen und Objekttypen. Es existieren geringe Abweichungen zu C++.	In Eiffel gibt es einen kleinen Satz vordefinierter Basistypen, die ebenfalls Objekte sind, aber besonders gehandhabt werden.
Verweise auf Objekte	Es existieren zwei unterschiedliche Verweismöglichkeiten: Pointer (ein manipulierbarer Verweis) und Referenz (ein fester Verweis auf ein Objekt). *this* bezeichnet den Verweis eines Objekts auf sich selbst, *NULL* den Verweis auf kein Objekt.	Verweise auf Objekte sind immer Referenzen. *this* erfüllt die gleiche Aufgabe wie bei C++, *null* bezeichnet den Verweis auf kein Objekt.	Verweise sind immer Referenzen. Das Schlüsselwort *Current* bezeichnet den Verweis eines Objekts auf sich selbst, *void* bezeichnet den Verweis auf kein Objekt.
Konstruktor und Destruktor	Konstruktor und Destruktor werden explizit angegeben. Zu einer Klasse *A* lautet der Konstruktor *A()*, der Destruktor *~A()*. Der Konstruktor kann überladen werden, so daß eine Argumentübergabe zur Initialisierung möglich ist.	Der Konstruktor wird mit dem Namen der Klasse angegeben, ein Destruktor wie in C++ ist nicht verfügbar, weil Java einen Garbage Collector besitzt. Für Ausnahmefälle kann die Methode *finalize()* implementiert werden, die vor Zerstörung des Objekts aufgerufen wird.	In Eiffel können beliebige Methoden nach dem Schlüsselwort `creation` aufgezählt und damit als möglicher Konstruktor gekennzeichnet werden. Eiffel besitzt einen Garbage Collector, so daß keine expliziten Destruktoren angegeben werden.
Abstrakte Klassen	Abstrakte Klassen können teilweise implementiert werden, alle abstrakten Methoden werden durch das Schlüsselwort `virtual` gekennzeichnet und mit Null initialisiert.	In Java werden die abstrakten Klassen durch das vorgestellte Schlüsselwort `abstract` markiert. Als Alternative können vollständig abstrakte Interfaces (Schnittstellen) geschrieben werden, die durch andere Klassen implementiert werden.	In Eiffel wird ähnlich wie bei Java in allen abstrakten Klassen innerhalb des Rumpfes das Schlüsselwort `deferred` deklariert, ebenso in allen Methoden, die in diesen Klassen noch nicht implementiert sind.
Mehrfachvererbung	Mehrfachvererbung ist möglich. Konflikte muß man durch eigene Festlegung auflösen, prinzipiell sind aus Sicht der Subklasse alle gleichlautenden Methoden der Superklasse zugreifbar.	Mehrfachvererbung wird in Java nicht unterstützt. Der Programmierer hat die Möglichkeit, eine Klasse mehrere Interfaces implementieren zu lassen und dadurch Mehrfachvererbung nachzubilden.	Mehrfachvererbung ist in Eiffel möglich. Konflikte werden durch Umbenennung eines Bezeichners aufgelöst, der betroffen ist.

Tabelle 1.1: Vergleich der Eigenschaften von C++, Java und Eiffel nach [Mey97], [Lou03] und [Fla02]

1.2 Testen und objektorientierte Testverfahren

Neben dem Verständnis der Objektorientierung muß man sich für die Umsetzung der Schnittstelle vor Augen halten, wie Testen funktioniert und systematisch angewandt wird. Da sich die Testmethoden für objektorientierte Software stark an allgemein anwendbaren Testverfahren orientiert, wird zuerst das Testen im allgemeinen, danach Testverfahren für objekorientierte Software betrachtet.

Testen von Software im Sinne durchgeführter systematischer Tests gewann in den 70er Jahren allmählich an Bedeutung. Der in Abbildung 1.2 gezeigte Graph demonstriert, wie sich die Anzahl der wissenschaftlichen Publikationen zu dynamischen Tests von 1975 bis 1982 veränderte. Aus dem Verlauf wird deutlich, daß der Themenbereich in der Forschung mehr Aufmerksamkeit gewann. Die Erfahrung mit verschiedenen Fehlerbildern und Entwicklungsprozessen innerhalb der Softwareentwicklung gab Anlaß zu unterschiedlichen Ansätzen, bei denen jeweils ein anderer Schwerpunkt gesetzt wurde (genauere Beschreibungen finden sich im nächsten Kapitel).

Falls die Umsetzung eines Softwareprodukts sorgfältig geplant und durchgeführt wird, reduziert dies die Zahl der möglichen Fehlerquellen. Bei größeren Programmen wird häufig eine horizontale oder vertikale Zerlegung des Gesamtprodukts vorgenommen, die Teile auf verschiedene Entwickler verteilt und später wieder integriert. Durch dieses Verfahren reduziert man die Komplexität des Gesamtproblems und damit die Wahrscheinlichkeit von Fehlern.

Zahlreiche Fälle zeigen jedoch, daß sorgfältige Entwicklung nicht alle Fehler vermeidet. Gerade in sicherheitskritischen Systemen wirken sich diese Fehler oft verheerend aus und sorgen in einigen Fällen für finanzielle Verluste oder die Gefährdung von Menschen. In [Dou99], Kapitel 3, werden einige reale Beispiele für diese Vorkommen angegeben[4]; gleichzeitig erfährt man, welche Fehler in der Praxis vermieden werden sollten.

Will man also eine Software testen, so ist für das Finden von weniger „offensichtlichen“ Fehlern zielstrebiges und systematisches Testen notwendig. In [Bin00] heißt es in der Einleitung des Autors dazu:

> *I view software testing as a problem in systems engineering. It is the design of a special software system: one that exercises another software system with the intent of finding bugs.*

[4] Douglass führt Software in Röntgengeräten und Zielsystemen für Abwehrraketen an – es reicht allerdings, sich einfache Beispiele wie Ampelschaltungen oder Airbag-Automatiken zu vergegenwärtigen.

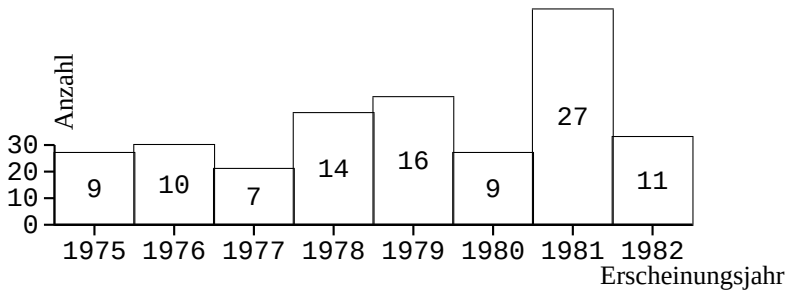

Abbildung 1.2: Anzahl der wissenschaftlichen Veröffentlichungen zu dynamischen Testverfahren (aus [Lig90]).

In [Lig90] findet man folgende Aussage:

> *Die Analyse der Kosten von Software-Entwicklungen . . . führt zu dem Ergebnis, daß der weitaus größte Anteil der Aufwendungen während der Wartungsphase eines im Markt befindlichen Produkts entsteht. . . . Die Ursache sind Fehler, die während der Softwareerstellung entstanden sind und die erst bei der Produktnutzung entdeckt werden.*

Um zu wissen, wann eine Software korrekt ist, muß eine *Spezifikation* erstellt werden, die definiert, was korrektes Verhalten bedeutet. Grob unterscheidet man zwischen zwei Klassen von Anforderungen (nach [Som02]):

- *funktionale Anforderungen*, d. h. alle Anforderungen, die das reine Ein-Ausgabe-Verhalten einer Software beschreiben. Diese Anforderungen lassen sich häufig als „wenn-dann"-Beziehung ausdrücken.
- *nicht-funktionale Anforderungen*, z. B. Stabilität, Fehlertoleranz, Geschwindigkeit, Speicherverbrauch.

Eine Abweichung vom geforderten Verhalten läßt sich dann als Fehler definieren. In [Bin00] wird zwischen drei Begriffen unterschieden:

- *failure* (in etwa: Versagen der Software), die Tatsache, daß ein Softwaresystem, die spezifizierten Anforderungen während eines Laufs nicht erfüllen kann,
- *fault*, die Ursache für das Versagen, z. B. eine fehlende Funktion und
- *error*, das Fehlverhalten des Programmierers, durch das diese Ursache zustandekommt (z. B. eine falsche Interpretation der Anforderungen oder ein Tippfehler).

Sofern ein Fehlverhalten (failure) durch einen Test entdeckt wird, kann durch genaues Prüfen des Programmablaufs die Ursache (fault) festgestellt und korrigiert werden. Kann außerdem festgestellt werden, welches Fehlverhalten des Programmierers (error) dafür verantwortlich war, lassen sich unter Umständen weitere Fehler in der Software ohne erneuten Test korrigieren.

Aus diesem Ablauf läßt sich auch der Umkehrschluß ziehen: Kennt man im voraus die Ursachen, die besonders häufig zu einem Fehlverhalten des Programms führen, kann man gezielt nach den entsprechenden Fehlern suchen. Die meisten gängigen Testverfahren definieren daher ein *Fehlermodell*, mit dem gezielt nach einer bestimmten Art von Fehlern gesucht wird. In [Bei90] und [Lig90] werden eine Reihe von Fehlerklassen definiert, zu denen entsprechende Testverfahren existieren. Durch Anwendung eines Testverfahrens auf die zu testende Software oder ein passendes Modell erhält man eine Reihe von *Testfällen*. Diese Testfälle beschreiben für gewöhnlich einen abstrakten Zustand, in dem sich die Software befinden muß, einen durchzuführenden Ablauf und ein erwartetes Ergebnis, ebenfalls in abstrakter Form, d.h. nicht in Form von konkreten Datenwerten. In den oben genannten Werken werden die gängigen Testverfahren grob klassifiziert:

- *Funktionale Testverfahren* prüfen das Ein- und Ausgabeverhalten eines Programms, also die oben schon erwähnten funktionalen Anforderungen. Innerhalb dieser Verfahren existiert zu den Testfällen meist eine Vorhersage des erwarteten Ergebnisses zu einer Eingabe. Durch Vergleich des tatsächlichen Ergebnisses mit dem erwarteten lassen sich entsprechende Fehler entdecken.

- *Strukturelle Testverfahren* versuchen, Fehler anhand der Struktur eines Programms zu finden. Um einen umfassenden und systematischen Test zu erreichen, versucht man, möglichst viele Ablaufmöglichkeiten eines Programms durch die Testfälle zu erzwingen. In diesem Fall spielt die Anzahl der angesprochenen Instruktionen, also die *Codeüberdeckung*, eine Rolle. Je mehr Code im Test überdeckt wird, desto größer ist die Wahrscheinlichkeit, Fehler zu finden.

Diese beiden Testverfahren spielen immer eine Rolle, wenn eine zusammenhängende Einheit getestet wird, wie z. B. ein Verbund aus mehreren zusammengehörenden Klassen. Unter Umständen werden auch beide Verfahren kombiniert, indem z. B. für bestimmte funktionale Verfahren eine Angabe für die Mindestüberdeckung des Codes gefordert wird.

Geht es dagegen darum, mehrere Teile zusammenzuführen und deren Wechselwirkungen zu untersuchen, werden *Integrationstests* durchgeführt. Auch wenn einzelne Teile einer Software für sich korrekt funktionieren, entstehen Fehler bei der Zusammenführung dadurch, daß wechselseitig falsche Anforderungen an den Teil der Software gestellt werden, mit dem kommuniziert werden soll. Die Größe des zu testenden Integrationsprodukts kann dabei von einigen Funktionen bis zu einem ganzen System reichen. Da die integrierten Teile in der Regel voneinander abhängig sind, spielt die Integrationsreihenfolge der einzelnen Teile eine Rolle. Zu diesem Zweck bieten einzelne Integrationstests unterschiedliche Zerlegungen, um möglichst gute Tests durchzuführen, aber gleichzeitig den Aufwand in Grenzen zu halten. Dabei schreiben einige Integrationsverfahren vor, Teile außen vor zu lassen, die eigentlich zur Durchführung eines Tests benötigt werden. An Stelle dieser Teile verwendet man *Stubs*, also Einheiten, die das Verhalten der nicht beteiligten Module bzw. Klassen simulieren und in diesem Sinne einfacher gehalten sind. Der eigentliche Test wird dann wieder mittels funktionaler oder struktureller Tests durchgeführt, so daß Integrationstests ein anderes Abstraktionsniveau besitzen.

Im Normalfall können die Testverfahren außerdem nach *Blackbox*-, *Whitebox*- und *Greybox*-Verfahren eingeteilt werden. Sind für das Testverfahren detaillierte Kenntnisse des Codes notwendig, handelt es sich um ein *Whitebox*-Verfahren. Viele strukturelle Testverfahren sind *Whitebox*-Verfahren, da hier der Code bekannt sein muß. Kann dagegen allein durch von außen beobachtbares Verhalten entschieden werden, ob die Testanforderungen erfüllt werden, handelt es sich um *Blackbox*-Verfahren (die Software ist eine schwarze Schachtel, deren Inhalt nicht bekannt ist). Aus diesem Grund sind funktionale Testverfahren häufig *Blackbox*-Verfahren. Der Test erfolgt bei diesen Verfahren gegen die Spezifikation. Einige Testverfahren erfordern die Kenntnis der Struktur der Software (z. B. die Aufgabenzuordnung zu bestimmten Codebereichen) und gehören damit zu den *Greybox*-Verfahren. Auch Verfahren, die sowohl als *Whitebox*- wie auch als *Blackbox*-Verfahren anwendbar sind, lassen sich in diese Kategorie einordnen. Zum Durchführen eines solchen Verfahrens kann dann Wissen aus der Spezifikation oder der Codestruktur herangezogen werden.

Denkt man an das in Abschnitt 1.1 beschriebene Figurenbeispiel, wird schnell deutlich, daß ein Test aller möglichen Fälle zu umfangreich und aufwendig ist (Beispiel: Man prüfe, ob die Methode *move()* für alle möglichen Drei- und Rechtecke und für alle möglichen Verschiebungen korrekt arbeitet, wenn der Code nicht bekannt ist)[5]. Aus diesem Grund ist es notwendig, durch geschickte Wahl einiger Stichproben wichtige Testfälle abzudecken und die Daten repräsentativ zu wählen (Beispiel: *Eine* Zahlenkombination für die Verschiebung eines Dreiecks nach rechts unten, eine weitere für die Verschiebung nach links oben).

[5] Eine abstraktere Betrachtung zu diesem Thema findet man ebenfalls in [Bei90], ein ähnliches Beispiel in [Bin00].

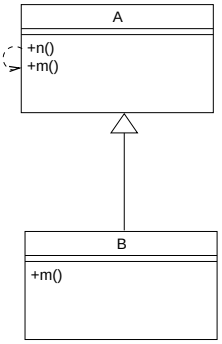

Abbildung 1.3: Eine mögliche Fehlersituation bei Überschreiben einer Methode.

Die Wahl der konkreten *Testwerte* (hier z. B. der Koordinaten) hängt von der zu testenden Software, dem gewählten Fehlermodell und einer guten Einschätzung des Testers ab.

Die Durchführung eines Tests verläuft also folgendermaßen:

1. Der Tester wählt je nach Fehlermodell und nach zu testender Software ein passendes Testverfahren.

2. Durch Anwendung des Testverfahrens und Analyse des Programms oder der Spezifikation erhält der Tester die Testfälle, die durchzuführen sind. Sofern möglich, sollten für die Testfälle bereits die erwarteten Resultate aufgestellt werden.

3. Für diese Testfälle entwickelt der Tester konkrete Testdaten und faßt die so erstellten Testfolgen zu einer sogenannten *Test Suite* zusammen. Es kann dabei sein, daß ein Testfall mehrere Aufrufe umfaßt und damit eine *Testsequenz* repräsentiert (am konkreten Beispiel: Man verschiebe ein Dreieck an eine Position und wieder zurück - ist die gespeicherte Position die ursprüngliche?)

4. Diese Test Suite wird dann implementiert oder mit Hilfe von Werkzeugen eingelesen und ausgeführt. Der Tester kann nun die Ergebnisse des Testdurchlaufs mit den erwarteten Ergebnissen vergleichen und entscheiden, wo eine Abweichung von der Spezifikation aufgetreten ist. Zur Implementierung der Testsuite werden unter anderem *Testtreiber* erstellt, die die Aufrufe an die zu testende Software durchführen.

5. Die Ergebnisse des Tests werden an den Programmierer als Feedback zurückgeben, der die Ursache des Fehlverhaltens suchen und entfernen kann. Sobald der Fehler korrigiert ist, sollte der Test erneut durchlaufen werden, um zu prüfen, ob die Korrektur erfolgreich war. In jedem Fall hat sich das Programm geändert, so daß der Tester seine Testfälle an den aktuellen Stand des Programms anpassen sollte.

Objektorientierte Testverfahren unterscheiden sich laut [Bin00] nicht wesentlich von den für andere Sprachen angewandten Testverfahren. Es kommen jedoch drei wesentliche Einschränkungen hinzu, mit denen der Tester umgehen muß:

- Der Zustand von Objekten ist (im Gegensatz zu globalen Variablen) häufig von außen nicht zugreifbar. Dies macht es bei funktionalen Tests schwierig festzustellen, ob sich die Software in einem korrekten Zustand befindet (was letztendlich auch nur ein erwartetes Resultat ist).

- Da Klassen über Vererbung implementiert werden können, erhöht sich der Testaufwand enorm. Da einer Referenz einer Klasse auch ein Objekt einer Subklasse zugewiesen werden kann, entscheidet die Sprache normalerweise bei Aufruf einer Methode erst zur Laufzeit, welche Variante dieser Methode aufgerufen wird: Die der Subklasse oder die der Superklasse. Diesen Vorgang nennt man *dynamisches Binden*. Wird eine Methode in einer Subklasse überschrieben, kann es sein, daß ein Objekt einer Subklasse plötzlich ein anderes, vollkommen falsches Verhalten annimmt, weil die Subklassenmethode die Anforderungen nicht mehr erfüllt, die für die Superklasse gültig sind. Abbildung 1.3 zeigt, daß eine Fehlersituation dabei nicht unbedingt in der überschriebenen, sondern auch in aufrufenden Methoden auftreten kann: Hier ruft die Methoden *n()* die Methode *m()* auf. Erfüllt *m()* in der Klasse *B* nicht mehr die Voraussetzungen, von denen *n()* ausgeht, kommt es in *n()* eventuell zu falschen Ergebnissen.

- Vererbung läßt zudem die *Polymorphie* von Objekten zu: Verweist ein Objekt A auf ein Objekt B und benutzt dessen Methoden, kann an Stelle von B auch ein Objekt einer Subklasse von B stehen, das sich vollkommen anders verhält als A dies erwartet. Der Fehler tritt dann in A auf, Ursache ist aber eine falsch implementierte Subklasse von B. In diesem Fall muß B systematisch durch Objekte der entsprechenden Subklasse ausgetauscht werden und der entsprechende Test erneut durchgeführt werden.

Für den ersten Punkt nennt Binder in [Bin00] Methoden und Möglichkeiten, den Zustand eines Objekts nach außen sichtbar zu machen oder den Testdurchlauf zu unterstützen. In den meisten Fällen geschieht dies durch Hinzufügen von zusätzlichem Code. Für die anderen beiden Probleme sind zusätzliche Testverfahren entwickelt worden, die durch systematische Modelle und Verfahren versuchen, einen Großteil der Möglichkeiten bei Polymorphie und dynamischem Binden abzudecken (s. [Bin00], S. 438ff und S. 513ff, *Polymorphic Server Test* und *Ploymorphic Message Test*).

Als Beispiel für einen objektorientierten Test zeigt Abbildung 1.4 den *Modal Class Test*, der in [Bin00] ausführlich erläutert wird. Dieses Verfahren dient zum Test von Klassen, deren Reaktion auf einen Methodenaufruf vom Zustand des instantiierten Objekts abhängt. Die Klasse muß dabei bereits lauffähig sein, d. h. kleinere Tests sollten schon vorher durchgeführt worden sein. Eine konkrete Anwendung dieses Verfahrens auf das bisher verwendete Figurenbeispiel wird in Abschnitt 2.2 durchgeführt.

Das genannte Beispiel zeigt bereits wesentliche Eigenschaften vieler Testverfahren: Der Tester prüft zum einen das Ein-/Ausgabeverhalten der Software, zum anderen den korrekten Zustand nach einer Ein- und Ausgabe. Eine Schnittstelle, die Tests und Testwerkzeuge unterstützt, sollte also nach Möglichkeit ebenfalls Zugriff auf die Attribute von Objekten und Ein- bzw. Ausgabewerte erlauben. Zusätzlich wäre eine Analyse der Codeüberdeckung hilfreich.

1.3 Vorhandene Testwerkzeuge

Im letzten Abschnitt wurde eine grobe Klassifikation unterschiedlicher Testverfahren vorgenommen. Da eine Testdurchführung möglichst automatisiert durchgeführt werden soll, existieren eine Reihe unterschiedlicher Werkzeuge, die in Industrie und Forschung entwickelt worden sind. Die nachfolgenden Absätze sollen einen kurzen Einblick in einige dieser existierenden Werkzeuge geben. Die Werkzeuge wurden dabei nicht im Hinblick auf Qualität oder Verbreitung ausgewählt, sondern sollen als Ergänzung zum letzten Abschnitt unterschiedliche Verfahren und Ansätze demonstrieren, die möglich sind.

1. Für die Klasse wird ein spezielles Zustandsmodell generiert, das alle (abstrakten) Zustände der Klasse und alle möglichen Übergänge zwischen diesen Zuständen mittels Methodenaufrufen erfaßt.

2. Aus diesem Zustandsmodell wird ausgehend von einem initialen Zustand ein sog. *Transition Tree* erstellt, ein Baum, der mögliche Pfade für Zustandsübergänge aufzeigt.

3. Um daraus Testfälle zu ermitteln, werden für jeden möglichen Pfad innerhalb des Baumes die notwendigen Bedingungen (in Bezug auf den Zustand und geforderte Übergänge) ermittelt. Jede separate Kombination aus Bedingungen bildet einen Testfall.

4. Für jeden dieser Testfälle werden nun konkrete Testwerte ermittelt und damit der Test durchgeführt, um sicherzustellen, daß sich die Klasse korrekt in Bezug auf das Zustandsmodell verhält (*Conformance Test*).

5. Anschließend versucht der Tester, illegale Zustandsübergänge zu finden. Hier werden in einer Tabelle alle Übergänge erfaßt, die im Modell nicht vorgesehen sind. Durch Ermittlung entsprechender Testfälle und Testwerte (sowie der erwarteten Fehlermeldung) erhält man einen zusätzlichen Testlauf (sog. *Sneak Path Test*).

6. Der Test ist beendet, wenn alle Testfälle erfolgreich abgeschlossen wurden und für jede Methode der getesten Klasse mindestens jeder Pfad einer Verzweigung eines Methodenrumpfs einmal durchlaufen wurde.

Abbildung 1.4: Schematischer Ablauf des Modal Class Test.

Zwei Werkzeuge, die funktionales Testen unterstützen, sind *Rhapsody* von I-Logix[6] und *WinRunner* (bzw. *XRunner* für Unix/Linux) von Mercury[7]. Beide Werkzeuge unterstützen vorwiegend Blackbox-Tests.

Rhapsody erlaubt dem Benutzer den Entwurf eines Systems in UML und erstellt daraus passende Codegerüste, die durch den Systementwickler zu ergänzen sind. Beim Test oder Debugging animiert Rhapsody die entsprechenden Teile des Modells auf dem Bildschirm, so daß die Zustände eines Objekts leicht erkennbar sind, ohne daß Wissen über den Code der Routinen existieren muß. Darüber hinaus kann diese Darstellung für das Debugging von Code benutzt werden. Vorteil ist außerdem, daß die Spezifikation (also das Modell) leicht konsistent zur existierenden Software gehalten werden kann. Auch die einzelnen unterstützten Modelle, z. B. Klassendiagramm, Sequenzdiagramm, Use Cases und State Charts, lassen sich untereinander leicht koordinieren, da alle Daten gleichzeitig gehalten werden. Rhapsody eignet sich mit diesem Ansatz für funktionale Tests unterschiedlicher Größenordnung, z. B. Klassentests, Tests von Klassenpaketen oder Subsystemtests.

WinRunner/XRunner bieten dagegen Unterstützung beim Simulieren von Benutzerbedienung. Die vorwiegende Anwendung ist daher ein Applikations- oder Abnahmetest. Mit Hilfe einer mausgesteuerten Eingabe oder der Benutzung einer Skriptsprache können Skripte für individuelle Appliktionstests entworfen werden. Die Steuerung erfolgt dabei durch die Angabe zu klickender Buttons, Felder, einzugebender Informationen usw. Durch Aufrufen eines Skripts zu einer Software läßt sich dann ein Testlauf durchführen. Beide Werkzeuge bieten dabei Unterstützung beim Auswerten der Ergebnisse und der Anzeige

[6] Weitere Informationen befinden sich unter *http://www.ilogix.com*.
[7] Dieses Werkzeug ist unter *http://www.mercury.com/us/* verfügbar.

betroffener Klassen. Zur Durchführung eines Lasttests kann dabei auch ein Skript mehrfach gestartet werden, wobei durch vorher eingelesene Daten z. B. verschiedene Benutzer durch verschiedene Instanzen des gleichen Skripts simuliert werden können. Eine weitere Anwendung ist der Regressionstest eines Systems, das verändert wurde, z. B. zur Entwicklung einer neuen Version. In diesem Fall kann mit Hilfe der Werkzeuge geprüft werden, ob sich das veränderte System konform zur alten Version verhält.

Ein weiteres Werkzeug speziell zur Durchführung von Monitoring, Überdeckungstests und Lasttests ist *AQTime* von AutomatedQA[8]. Hier beschränkt sich die Anwendung größtenteils auf Applikationstests wie z. B. Performance Tests und Lasttests oder spezielle Überprüfungen, wie z. B. Konformanz zu einer bestimmten Plattform. Das Werkzeug unterstützt dabei C++, .NET und Ada, wobei der Code mit Debugging-Informationen kompiliert werden muß (zur Arbeitsweise s. auch die Beschreibung im Abschnitt *Ansätze zur Instrumentierung*). Die Messung von Codeüberdeckung läßt auch Whitebox-Testen zu, so daß das Werkzeug in mehreren Phasen der Entwicklung eingesetzt werden kann (zum Beispiel bei Entwicklung eines Testtreibers für funktionale Tests mit Analyse der Überdeckung).

Ausschließlich zur Messung von Codeüberdeckung bei C++-Programmen ist *Pure Coverage* von Dynamic Memory Solutions (*www.dynamic-memory.com*) gedacht. In diesem Werkzeug wird die Überdeckungsinformation für verwendete Funktionen verwendet, allerdings muß das zu testende Programm im Gegensatz zu AQTime nicht neu kompiliert werden. Das Instrumentieren der Software geschieht hier dynamisch. Der Benutzer erhält nach dem Lauf des Programms eine Textdatei mit statistischen Informationen über erreichte Codestellen des Quellcodes, die dann zur Weiterverarbeitung und Auswertung genutzt werden können. Das Werkzeug ist dabei Teil einer von der Firma herausgebrachten Werkzeugreihe, die vorwiegend dazu gedacht sind, schwer zu findende Fehler in C++ zu ermitteln. Die Testtreiber muß der Benutzer allerdings selbst schreiben. Die Überdeckungsmessung kann auch zur Unterstützung von Applikationstests genutzt werden, da Pure Coverage nicht in den Programmablauf eingreift.

Ein weitergehender Ansatz kommt von der Firma Parasoft (zu finden unter *http://www.parasoft.com*). Zwei Werkzeuge, *TestC++* und *JTest*, unterstützen dabei die automatische Generierung von Standardtestfällen für Methodenaufrufe einer Klasse. Das Hauptanliegen liegt allerdings nicht im Testen, sondern bereits in der Fehlervermeidung (*Automated Error Prevention*), wobei Fehlervermeidung im Sinne des oben genannten *error* gemeint ist. Die Werkzeuge analysieren auf Knopfdruck den eingegebenen Quellcode des Benutzers im Hinblick auf sogenannte *Code Conventions*, also Regeln, die vorschreiben, wie der Code aussehen muß, um möglichst übersichtlich, strukturiert und wenig fehleranfällig zu sein. Der Benutzer kann bei Durchführung eines Tests und dem Auftreten eines Fehlers selbst Regeln hinzufügen, damit ihn das Werkzeug zukünftig bei der Vermeidung desselben Fehlers untestützt. Im größeren Rahmen eine Projekts können die Regeln auch zwischen mehreren Mitgliedern einer Projektgruppe z. B. über ein Netzwerk ausgetauscht werden, um eine konsistente Behandlung zu gewährleisten.

Diese kurze Auswahl verdeutlicht das in der Einleitung bereits angedeutete Problem: Je nach Anforderungen und Zielsetzung des Tests ist der Tester bereits in der Auswahl des Werkzeugs eingeschränkt. Unter Umständen ist das Werkzeug kein reines Testwerkzeug, oder die Sprache, in der die Software implementiert wird, wird durch das passende Werkzeug nicht unterstützt. Eine Unterstützung mehrerer Sprachen wird durch einige Werkzeuge zwar geboten, allerdings ist diese Zuordnung zu den Sprachen nicht änderbar. Auf der anderen Seite wird aber auch deutlich, daß für alle gezeigten Werkzeuge der Code

[8] Dieses Tool ist unter *http://www.automatedqa.com* auffindbar.

instrumentiert werden muß, um daraus Informationen zu gewinnen. Dies zeigt bereits, daß Instrumentierung für teilweise sehr unterschiedliche Anwendungen zum Tragen kommt. Ein generischer Ansatz ist in dieser Richtung allerdings nur mit .NET gemacht worden, weil der für dieses System eingesetzte Debugger auf der Repräsentation eines Programms aufbauen kann (s. Kapitel 3.1).

Aus der Auswahl lassen sich folgende Anforderungen formulieren:

- Die Schnittstelle sollte wie beispielsweise bei Pure Coverage Tests zulassen, die als Methodenaufrufe von Hand geschrieben wurden, aber genauso eine automatische Durchführung einer Testsuite erlauben, wie beispielsweise bei JTest.
- Analog zum Verfahren von WinRunner ist es sinnvoll, bestimmte Dateien zu definieren, die das Verhalten der Schnittstelle steuern. Auf diese Weise ist die Schnittstelle durch den Benutzer konfigurierbar. Die Konfiguration beeinflußt hier nicht den Testlauf, sondern die Anpassung an die Sprache bei der Instrumentation.
- Die Beschreibung der verwendeten Sprache kann in ähnlicher Form geschehen wie in den Regeln bei TestC++ oder JTest (die Definitionen über die Struktur des Quellcodes sind). Tatsächlich gibt es im Compilerbau verwendete Formalismen, deren Ausdrucksmöglichkeiten eine gute Basis bieten (siehe auch Abschnitt 2.2).

Vergleicht man Pure Coverage und AQTime, besteht offensichtlich ein Unterschied im Instrumentierungsverfahren, da das eine Werkzeug eine erneute Kompilierung erfordert, das andere nicht. Um genauer festzustellen, wo Stärken und Schwächen verschiedener Verfahren liegen, ist eine eingehendere Betrachtung notwendig. Die feste Zuordnung mehrerer Sprachen zu einem Werkzeug zeigt außerdem, daß anscheinend unterschiedliche Anbindungsmöglichkeiten für unterschiedliche Sprachen existieren. Auch hier muß eine Abwägung zwischen verschiedenen Möglichkeiten für die Schnittstelle getroffen werden. Aus diesem Grund beschäftigt sich das nächste Kapitel mit diesen beiden Punkten genauer, wobei für die Instrumentierung weitere Anwendungsmöglichkeiten aufgezeigt werden. Zusätzlich erläutert das Kapitel die bereits angesprochenen Modelle im Compilerbau.

Kapitel 2

Instrumentierung von Software

Was ist Instrumentierung?

Der moderne Begriff „Instrumentierung“ hat drei verschiedene Bedeutungen:

- In der **Musik** bezeichnet Instrumentierung die unterschiedliche Besetzung verschiedener Stimmen eines Stücks oder Orchesterwerks, was Anzahl und Art der Instrumente angeht (unter anderem auch aus historischen Gründen, weil alte Instrumente nicht mehr verfügbar sind). Je nach Interpretation des Stückes fällt unter Umständen auch die Instrumentierung anders aus, was die Klangfarbe und die Stimmung eines Stückes beeinflußt.

- In der **Medizin** instrumentiert eine Krankenschwester einen operierenden Arzt, indem sie ihm die benötigten Operationsinstrumente reicht.

- Innerhalb dieser Arbeit soll der **technische Begriff** verwendet werden. In dieser Bedeutung bezeichnet Instrumentierung den Sachverhalt, daß ein Objekt instrumentiert worden ist, die Tatsache, daß dieses Objekt mit Meßgeräten (lat. Instrument = Werkzeug) ausgestattet worden ist, mit dessen Hilfe man bestimmte Beobachtungen anstellen kann. Im Laufe der Zeit ist man außerdem dazu übergegegangen, mit Instrumentierung auch den Vorgang zu bezeichnen, der dazu führt, daß ein Objekt „meßbar“ wird.

Speziell für Software heißt das: Software wird dadurch instrumentiert, daß zusätzlicher Code zu dem bisher vorhandenen Code hinzugefügt wird. Dieser Code sorgt dann dafür, daß der Benutzer als Beobachter Rückschlüsse auf das Verhalten der Software ziehen kann. Im folgenden Abschnitt sollen mögliche Verfahren zur Instrumentierung genauer beleuchtet werden.

2.1 Ansätze zur Instrumentierung von Software

Instrumentierung von Software ist auf unterschiedliche Arten möglich. Der Effekt der Instrumentierung macht sich erst bei einem Lauf des Programms bemerkbar, weil hier die Rückmeldung der eingefügten Codeteile über dessen Verhalten erfolgt. Grundsätzlich kann man jedoch nach dem Zeitpunkt der Instrumentierung unterscheiden:

- Bei der *Quellcode-Instrumentierung* wird der Code bereits vor der Umsetzung in ein lauffähiges Programm mit den zusätzlichen Einfügungen ausgestattet. Die zusätzlichen Teile müssen also in der gleichen Programmiersprache formuliert werden wie der Originalcode.

- Die *Laufzeitinstrumentierung* verfolgt den entgegengesetzten Ansatz: Der Code wird hier erst zur Laufzeit eingefügt, d. h. nachdem aus dem Quellcode ein lauffähiges Programm erstellt wurde, das bereits ausgeführt wird. Allgemeiner spricht man von *Objektcode-Instrumentierung*, wenn das Programm nach der Umsetzung in ein lauffähiges Programm instrumentiert wird.

Beide Ansätze besitzen Vor- und Nachteile:

- Wird ein Programm erst zur Laufzeit oder vor dem Start eines Programms instrumentiert, wird für bestimmte Messungen, z. B. zeitliches Verhalten, kein Quellcode mehr benötigt. Die Messung kann also auch bei einer Software vorgenommen werden, zu der man den Quellcode nicht besitzt. Gleichzeitig ist die Instrumentierung unabhängig von der Sprache, in der das Programm ursprünglich geschrieben wurde. Innerhalb bestimmter Grenzen läßt sich die Instrumentierung dann auch ein- und ausschalten, was bei einer Quellcode-Instrumentierung nur durch erneutes Kompilieren erreicht werden kann.

- Die Quellcode-Instrumentierung erlaubt es dagegen eher und einfacher, den Bezug zu den einzelnen Programmzeilen eines Programms wieder herzustellen. Da ein Entwickler durch Instrumentierung, Messung und Tests während der Entwicklung Rückschlüsse auf die Stellen ziehen möchte, an denen die Software noch Schwächen aufweist, macht es Sinn, diesen Rückbezug zur Verfügung zu stellen. Innerhalb gewisser Grenzen geht dies auch bei der Objektcode-Instrumentierung, allerdings nur mit größerem Aufwand. Darüber hinaus ist eine Instrumentierung des Quellcodes auch dann durchführbar, wenn der Quellcode nicht direkt in ein lauffähiges Produkt umgesetzt, sondern beispielweise von einem Interpreter weiterverebrbeitet wird (s. Kapitel 2.2).

In den meisten Fällen wird die Instrumentierung automatisiert vorgenommen. Einige Beispiele für Werkzeuge und Methoden, die Instrumentierung unterstützen, sollen konkrete Ansätze aufzeigen und deutlich machen, in welchen Bereichen der Softwareentwicklung Instrumentierung zum Einsatz kommt.

Beim *Debugging* tritt Instrumentierung eher im Hintergrund auf, wird dafür aber recht intensiv genutzt. Viele gängige Compiler bieten Einstellungen an, mit denen im Code zusätzliche Informationen hinzugefügt werden. In den meisten Fällen bieten das Betriebssystem oder die Hardware eines Rechners, auf dem ein Programm ausgeführt wird, einen Mechanismus an, der dafür sorgt, daß die instrumentierten Stellen ein besonderes Ereignis (Interrupt) auslösen. Das System ermittelt bei Auftreten eines Ereignisses aus einer Tabelle im Speicher, wo die Ausführung fortgesetzt werden soll, und sorgt zudem dafür, daß die Stelle, an der das Ereignis aufgetreten ist, gespeichert wird.

Debuggingwerkzeuge nutzen diesen Mechanismus aus und tragen in der Tabelle eine ihrer eigenen Routinen ein. Der Programmierer kann dann an bestimmten Stellen *Breakpoints* setzen, an denen der Debugger das Programm anhält und der Programmierer dessen Zustand genauer untersuchen kann. Abbildung 2.1 zeigt das grobe Ablaufschema. Bekannte Werkzeuge in dieser Richtung sind beispielsweise *gdb* (ein freier C++ - Debugger) und

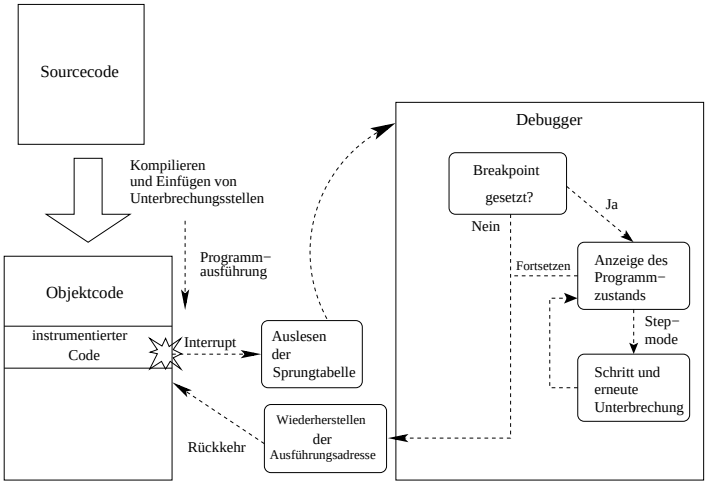

Abbildung 2.1: Schematische Darstellung des Debbugings.

jdb (das analoge Gegenstück für Java). Auch viele Entwicklungsumgebungen wie Eclipse und Microsoft Visual Studio unterstützen Debugging.

Die Instrumentierung erlaubt hier umfassenden Einblick in den Zustand der Software, allerdings ist der Ansatz für das Testen von Software nicht geeignet (und auch nicht dafür gedacht), weil der Benutzer für jeden Programmpunkt selbst entscheiden muß, ob er das Programm anhalten möchte oder nicht. Für einen Testlauf muß ein Testwerkzeug eine automatisierte Instrumentierung der geeigneten Stellen vornehmen können und die Auswahl passend beschränken.

Ein zweiter Bereich, bei dem Instrumentierung angewendet wird und dessen Anwendung eng mit dem Testen verknüpft ist, ist das *Profiling*. In diesem Fall wird die Software an bestimmten Stellen instrumentiert, um Messungen von Laufzeitgrößen wie Zeitbedarf, Stabilität, Belastbarkeit usw. zu erhalten. Aus den Messungen läßt sich ersehen, an welchen Stellen ein Programm noch verbessert werden kann und wo der Code eventuell umstrukturiert werden muß. Aus diesem Grund sind die Instrumentierungspunkte oft auch weiter gestreut (z. B. Ein- und Austrittspunkt einer Methode zur Messung des Zeitbedarfs). In speziellen Fällen bezieht sich das Profiling auch auf die Codestruktur, z. B. die Erreichbarkeit oder die Häufigkeit der Benutzung eines Codeabschnitts, um nicht benutzten Code zu entfernen oder häufig benutzen Code als Engpaß zu identifizeren. Beispiele für Profiling sind die *dyninst*-API in C++ und daraus entwickelte Werkzeuge, z.B. *TAU*[1] oder *Dyjit*[2] für Java. In beiden Entwicklungen sind zudem Ansätze aufgezeigt, Software auch zur Laufzeit zu instrumentieren (s. [TH02] und [Cor03]).

Ein Ansatz, der Instrumentierung unterstützt, ist *aspektorientierte Programmierung*. Das Anliegen der aspektorientierten Programmierung und Modellierung liegt eigentlich in der Trennung verschiedener Aspekte (sog. *Concerns*) der Software, z. B. Fehlerbehandlung, Logging oder Dateizugriff. Dabei werden die eigentliche Software und die Aspekte separat geschrieben und dann durch einen *Weaver* miteinander verknüpft. Der Programmierer gibt dabei mittels einfach gehaltener Angaben dem Weaver vor, an welcher Stelle der Software die Aspekte eingefügt werden sollen. Definiert man Ausgaben auf den Bildschirm oder Logging als Aspekt, läßt sich damit die restliche Software innerhalb gewisser

[1] Dieses Werkzeug ist unter *http://www.dyninst.org* zu finden.

[2] Für genauere Informationen sollte man unter *http://www.doc.ic.ac.uk/∼jlc01/dyjit-ng/* nachsehen.

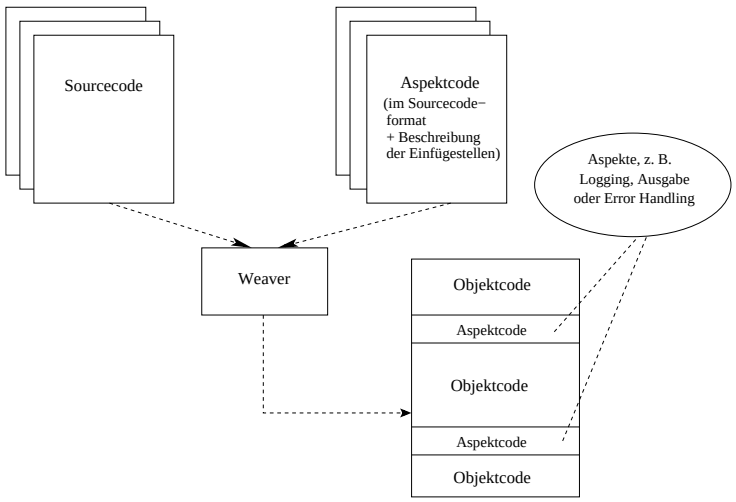

Abbildung 2.2: Schematischer Ablauf in der aspektorientierten Programmierung.

Grenzen beobachtbar machen. Auch hier zeigt Abbildung 2.2 den Vorgang schematisch. In diesem Fall sind jedoch die Möglichkeiten der Intrumentierung eingeschränkt – die kleinste atomare Einheit, in die Code eingefügt werden kann, ist die Methode einer Klasse (die Einfügung kann nur vor oder hinter einer Methode vorgenommen werden; einzige Ausnahme ist bei einigen Ansätzen ein Einfügen von zusätzlichem Code an einer Stelle, an der Exceptions geworfen werden). Dies erschwert die Überdeckungsmessung. Eine konkrete Implementierung des aspektorientierten Ansatzes ist z. B. *AspectJ*[3].

Abschließend bleibt der Einsatz bei Instrumentierung für den Test von Software. Da bereits Werkzeuge vorgestellt wurden, die objektorientierte Testverfahren unterstützen, soll an dieser Stelle auf Beispiele verzichtet werden. Die Instrumentierung ist je nach Art des Tests eher an die Instrumentierung des Debuggers oder die des Profilers angelehnt. Da allerdings in den meisten Tests Auskunft über den Zustand der Software gegeben werden soll, muß durch die Instrumentierung erreicht werden, daß entsprechende Informationen an das Testwerkzeug zurückgegeben werden. Im Debugger fehlt dazu die passende Aufbereitung, beim Profiler wird die Information in den meisten Fällen indirekt aus den erreichten instrumentierten Stellen errechnet.

Im Hinblick auf die Schnittstelle sind damit folgende Probleme zu lösen:

- Die Schnittstelle muß Unterstützung beim Auffinden der passenden Stellen bieten (im doppelten Sinne: passend zum Test und passend zur Programmiersprache), die zu instrumentieren sind, damit die Instrumentierung durchgeführt werden kann.

- Die Instrumentierung muß dafür sorgen, daß die benötigten Informationen nach außen gegeben werden.

Der nächste Abschnitt soll Lösungsmöglichkeiten dieser beiden Probleme genauer untersuchen und gleichzeitig weitere Unterschiede zwischen einzelnen Programmiersprachen aufzeigen.

[3] AspectJ befindet sich unter *http://www.aspectj.org*, Methoden zur Instrumentierung und zum Testen beschreiben z. B. [MH02] oder [DC01].

2.2 Beziehungen zum Compilerbau

In diesem Abschnitt sollen einige im Compilerbau gängige Verfahren betrachtet werden, um bereits einen Ansatz für die spätere Instrumentierung zu entwickeln. Dabei wird zuerst das Figurenbeispiel herangezogen, um zu zeigen, wie der Code für die drei Sprachen aussieht, um anschließend abzuleiten, wie korrekte Stellen für die Instrumentierung gefunden werden können.

Da ein Objekt der Klasse *Point* aus Abbildung 1.1 selbst eine Methode *move()* besitzt, soll eine Figur um eine Methode *movePoint()* erweitert werden, die einen Index für die Angabe des Punktes und die Differenz (in x- und y-Richtung) erhält, der verschoben werden soll. Auf diese Weise lassen sich sehr viel freiere Transformationen durchführen. In der Klasse *Triangle* ist dann jedoch nur eine Flächenberechnung möglich, wenn es sich um ein echtes Dreieck handelt. Liegen alle Punkte auf einer Linie, muß eine entsprechende Fehlermeldung erfolgen, oder eine entsprechende Konstruktion darf nicht zugelassen werden. In diesem Fall soll die Fehlerbehandlung darin bestehen, daß bei der Flächenberechnung -1 zurückgegeben wird, wenn alle Punkte auf einer Linie liegen. In der Klasse *Rectangle* sollen analoge Änderungen vorgenommen werden (d. h. die Berechnungsmethode meldet mit -1 einen Fehler, wenn es sich bei der Figur nicht um ein Rechteck handelt). Da jetzt das Ergebnis der Berechnungsmethode vom Zustand der Figur abhängt, sind damit beide Klassen Kanditaten für den bereits vorgestellten *Modal Class Test*.

Im Test interessiert man sich jetzt dafür, ob eine Figurenklasse das richtige Verhalten zeigt. Wie bereits angedeutet, kann jetzt geprüft werden, ob eine Berechnung innerhalb verschiedener Quadranten des Koordinatensystems richtige Ergebnis liefert, auch wenn vorher mehrere Verschiebungen der Figur oder einzelner Punkte vorgenommen wurden (Conformation Test). Für den Sneak Path Test wäre zu überprüfen, ob eine Gesamtverschiebung der Figur eventuell dazu führt, daß alle drei Punkte auf einer Linie liegen. Instrumentiert werden müßten beide Klassen in der Form, daß über entsprechende Methoden Zugriff auf die Punkte der Figur möglich ist. Ist man als Tester außerdem daran interessiert, eine möglichst hohe Codeüberdeckung zu erreichen, sollte innerhalb der einzelnen Methoden an wichtigen Punkten (z. B. am Methodenbeginn) eine Rückmeldung erfolgen, welche Stellen erreicht wurden.

Bei Durchführung einer Quellcode-Instrumentierung muß die Struktur und die Syntax des Quellcodes bekannt sein. Ein kurzer Vergleich der Schreibweise der Klasse *Triangle* zwischen C++, Eiffel und Java soll die Unterschiede aufzeigen.

In **Java** werden die Deklarationen und Definitionen der Klasse gemeinsam aufgeführt, Superklassen werden mittels `extends` angegeben:

```
class Triangle extends Figure {
  Triangle () {
    point[0] = new Point(0,0);
    point[1] = new Point(1,0);
    point[2] = new Point(0,1);
  }

  // Klasseneigene Methode zur Ueberpruefung, ob die Punkte in Reihe
  // sind

  private boolean inLine() {
    // Hier folgt eigentlich Code zur Behandlung...
  }
  public void move(double dx, double dy) {
    for (int i=0; i < 3; i++)
    point[i].move(dx, dy);
  }
```

```
  public double getArea() {
    if (inLine())
      return -1;

    // Berechnungen...

    return value;

  }

  void movePoint(int nr, dobule dx, double dy) {
    if (nr < 3 && nr > 0)
      point[nr].move(dx, dy);
  }
}
```

In **C++** erfolgt die Deklaration und Definition normalerweise getrennt, ein Doppelpunkt kennzeichnet die Beziehung zur Superklasse:

```
class Triangle : public Figure {
  private:
    inLine();

  public:
    Triangle();
    void move(double dx, double dy);
    double getArea();
    void movePoint(int nr, double dx, double dy);
}

Triangle::Triangle() {
  point[0] = new Point(0,0);
  point[1] = new Point(1,0);
  point[2] = new Point(0,1);
}

void Triangle::move() {
  for (int i=0; i < 3; i++)
    point[i]->move(dx, dy);
}

double Triangle::getArea() {
  // siehe oben
}

bool Triangle::inLine() {
  // siehe oben
}

void Triangle::movePoint(int nr, double dx, double dy) {
  if (nr < 3 && nr > 0)
      point[nr]->move(dx, dy);
}
```

In **Eiffel**[4] wird eine Superklasse durch das Schlüsselwort `inherit` angegeben:

```
class TRIANGLE inherit FIGURE
creation
  Figure
feature
   Figure is
     do
       points.Create(0,2);
       points.enter(0, POINT.Create(0,0));
       points.enter(1, POINT.Create(0,1));
       points.enter(2, POINT.Create(1,0))
     end;

   move(dx: DOUBLE, dy: DOUBLE) is
     do
       points.entry(0).move(dx, dy);
       points.entry(1).move(dx, dy);
       points.entry(2).move(dx, dy)
   end;

   movePoint(nr: INTEGER, dx: DOUBLE, dy: DOUBLE) is
     do
       points.entry(nr).move(dx, dy)
   end;

   inLine: BOOLEAN is
     do
       -- Ueberpruefung...
   end;

   getArea: DOUBLE is
     do
       --Berechnung
   end
end;
```

Die Beispiele zeigen bereits die unterschiedliche Syntax der einzelnen Programmiersprachen. Da die jeweiligen Compiler bei der Umsetzung des Quellcodes eine ähnliche Analyse durchführen, lohnt es sich, die zugrundeliegenden Mechanismen und Ansätze zu betrachten.

In den meisten Fällen wird der Aufbau einer Sprache durch eine Regelgrammatik (eine sogenannte *Extended Backus-Naur-Form*, *EBNF*) beschrieben. Ausgehend von einem abstrakten Startsymbol geben die Regeln an, wie einzelne Symbole ersetzt werden. Durch fortwährende Transformation der Regeln erhält man aus dem Startsymbol syntaktisch korrekte Programme. Die Regeln legen damit die syntaktische Schreibweise für Klassen, Methoden usw. fest. Atomare Grundelemente jeder Sprache sind dabei die sogenannten *Terminalsymbole*, die nicht weiter umgeformt werden. Ein Beispiel fuer einen Regelsatz, der eine Deklaration einer Methode in C++ beschreibt, könnte folgendermaßen aussehen:

```
MethodDeclaration -> TypeIdent  Ident  '::' Ident '(' Arguments ')' ';'
TypeIdent -> 'int' | 'void' | 'bool' ...
Ident -> Char Ident
Char -> 'a' | 'b' | 'c' ...
Arguments -> TypeIdent Ident (',' TypeIdent Ident)*
```

[4] Zum Zeitpunkt dieser Ausarbeitung ist für SmartEiffel eine neue Version erschienen, die neue Spracheigenschaften festlegt. Diese sollen hier nicht berücksichtigt werden.

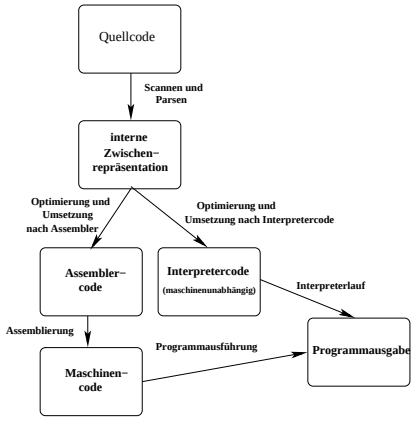

Abbildung 2.3: Phasen der Programmübersetzung.

Man beginnt also bei `MethodDeclaration` und ersetzt Symbole, die durch die linke Seite angegeben sind, durch die rechte Seite der passenden Regel, um auf eine Methodendeklaration zu kommen. Die in Anführungszeichen angegebenen Symbole repräsentieren die Terminalsymbole bzw. die Token („Wörter“) eines Compilers, die nicht weiter transformierbar sind, während senkrechte Balken Alternativen andeuten, in die das Symbol auf der linken Seite transformierbar ist.

Die gängigen modernen Compiler gehen bei der Umsetzung des Quellcodes häufig den umgekehrten Weg: Aus dem Quellcodetext werden schrittweise die einzelnen Token ermittelt, um dann abstraktere Symbole aufzubauen. Sind alle Symbole eines rechten Regelteils komplett, läßt sich die Regel jetzt reduzieren und damit das abstrakte Element identifizieren. (gängige Verfahren findet man z. B. in [ASU86]) . Für ein Finden der Instrumentierungsstellen ließe sich jetzt ein ähnliches Verfahren vorstellen: Man identifiziert anhand der Grammatik einzelne Elemente und merkt sich die passenden Punkte. Da der konkrete Code für das spätere Instrumentieren des Codes keine Rolle spielt, können größere Codeteile übergangen werden. Diese Idee wird später bei der Beschreibung der praktischen Umsetzung genauer ausgearbeitet. Im Hinblick auf die spätere Erläuterung soll hier nur kurz erwähnt werden, daß zur Bestimmung einer passenden Regel unter Umständen eine *Direktormenge* errechnet wird. Diese Menge gibt an, mit welchen Zeichen eine Regel beginnen kann und erleichtert einem Compiler beim zeichenweisen Einlesen die Zuordnung zur passenden Regel.

Abschließend soll gezeigt werden, wie eine mögliche Instrumentierung des Objektcodes funktionieren könnte. Dazu illustriert Abbildung 2.3 die einzelnen Phasen einer Umsetzung:

- Der Compiler analysiert den Quellcode und erzeugt mit Hilfe der Regeln eine interne Zwischendarstellung, die auf Korrektheit überprüft wird.

- Ist diese Repräsentation korrekt, setzt der Compiler die gehaltenen Informationen in lauffähigen Maschinencode um. Um zur Laufzeit beim Auftreten von Fehlern genauere Bezüge zum Quellcode herzustellen, können zusätzliche Informationen (als lauffähiger Code) eingebunden werden. Da der Compiler alle Informationen bezüglich des Quellcodes hält, ist dies ohne Probleme möglich.

- Unter Umständen ist der erzeugte Code kein Maschinencode, sondern ebenfalls ein Zwischenprodukt, das entweder an einen weiteren Compiler oder an einen Interpreter weitergegeben wird, der die Umsetzung in lauffähigen Code quasi zur Laufzeit vornimmt.

Soll eine Instrumentierung des Objektcodes ohne Compiler erfolgen, wären folgende Ansätze denkbar:

- Zur Instrumentierung von Methoden könnten vom Compiler eingebundene Namen verwendet werden, um die passenden Stelle zu ermitteln. Anschließend könnte eine automatisierte Umsetzung des zusätzlichen Codes erfolgen, der dann an die passende Stelle kopiert wird. [TH02] beschreibt diesen Ansatz genauer, der durch die bereits in Abschnitt 2.1 erwähnte dyninstAPI unterstützt wird.

- Eine aufwendigere Möglichkeit ist eine direkte Ermittlung wichtiger Codestellen aus der Struktur des Objektcodes. Dazu muß die genaue Umsetzung von Quellcodestrukturen in Maschinensprache bekannt sein. Dieser Ansatz ist bisher nur indirekt verfolgt worden, und zwar bei den Decompilern. Hier ist der Ansatz, den Objektcode schrittweise zu transformieren und daraus wieder abstrakte Anweisungen zu gewinnen, die in der entsprechenden Programmiersprache dargestellt werden können (z. B. bei Boomerang[5] oder JAD[6]). Die Theorie dazu ist recht weit fortgeschritten[7].

- Ist das vom Compiler erzeugt Produkt kein Maschinencode, bietet unter Umständen der ausführende Interpreter Unterstützung. In Java kann der JVMPI-Standard benutzt werden, um eine entsprechende Instrumentierung zu erreichen, der von Dyjit gegangen wird (siehe [Sun04]).

Soll die Instrumentierung jedoch sprachunabhängig sein, lassen sich die unterschiedlichen Ansätze nur schwer einheitlich behandeln: Während C++ eine direkte Umsetzung in Maschinencode vornimmt, besitzt Java eine eigene Laufzeitumgebung, die den erzeugten Code interpretiert und in Maschinensprache umsetzt. Bei Eiffel wird innerhalb des Programms zusätzlicher Code für die Laufzeitumgebung erzeugt und eingebunden. Eine Instrumentierung von Java und Eiffel erfordert genaue Kenntnisse des zu interpretierenden Codes oder eine Manipulation der Laufzeitumgebung. Beides ist nur mit erhöhtem Aufwand durchführbar, vor allem, wenn der Rückbezug zum Quellcode wieder zur Verfügung stehen soll.

Aus diesem Grund ist Quellcode-Instrumentierung das für die Schnittstelle leichter umzusetzende Instrumentierungsverfahren. Mit dieser Möglichkeit wird eine Erweiterung in Richtung anderer Sprachen offengehalten, dafür muß der Benutzer allerdings nach einer erneuten Instrumentierung den Code auch erneut kompilieren. Im Hinblick auf die leichtere Implementierung ist dieser Aufwand jedoch vertretbar.

[5] Boomerang ist unter *http://boomerang.sourceforge.net* verfügbar.
[6] JAD befindet sich unter *http://kpdus.tripod.com/jad.html*.
[7] Einen Überblick über entsprechende Mechanismen bietet [van04].

Kapitel 3

Kommunikation zwischen verschiedenen Sprachen

Für den Austausch von Informationen zwischen verschiedenen Sprachen müssen drei Voraussetzungen erfüllt sein:

- Für die Sprachen muß ein geeignetes *Austauschformat* gefunden werden, das die benötigten Informationen halten und weitergeben kann.
- Zwischen den Sprachen muß ein *Kommunikationsweg* eingerichtet werden, der die Informationen von einer Sprache an die nächste weiterleiten kann.
- Schließlich müssen beide Sprachen *Schnittstellen* für den Austausch besitzen, mit denen sie auf die übermittelten Informationen zugreifen können.

Selbst bei dem trivialen Beispiel des Austauschs über eine Textdatei sind die drei Anforderungen erfüllt: Die austauschenden Sprachen müssen für den Datenaustausch ein gemeinsames Textformat festlegen, Schnittstellen sind die Dateizugriffsfunktionen, die eigentliche Kommunikation wird durch das Betriebssystem ermöglicht (und zwar durch den tatsächlichen Zugriff auf die Datei und die korrekte Ansteuerung des Datenträgers)[1]. Im Folgenden sollen weitere Möglichkeiten beschrieben werden, die den Austausch zwischen verschiedenen Programmiersprachen ermöglichen und deren Vor- und Nachteile im Hinblick auf eine mögliche Verwendung innerhalb der Schnittstelle genannt werden.

Eine Möglichkeit zum Austausch ist die direkte Anbindung über Maschinencode. Die Idee ist dabei, verschiedene Ausführungsmöglichkeiten für den Code zur Verfügung zu stellen. Für jede beteiligte Sprache wird ein passender Einsprungspunkt zur Verfügung gestellt, von dem aus später die gemeinsame Routine aufgerufen werden kann. Ein Teil des Codes sorgt dann für die richtige Umwandlung der übergebenen Argumente und die korrekte Rückwandlung eines Ergebnisses. Alternativ können auch die beteiligten Sprachen bei Funktions-, Methoden- oder Prozeduraufrufen das Austauschformat festlegen und die Wandlung selbst vornehmen. Viele Sprachen bieten auf diese Weise zum Beispiel eine Schnittstelle an die Sprache C an. Da die unterliegenden Routinen zudem in Maschinencode geschrieben sind, geht der Austausch sehr schnell vor sich. Nachteilig ist allerdings,

[1] Wie weiter unten ersichtlich wird, ist eine textuelle Darstellung jedoch nicht immer so trivial, wie es auf den ersten Blick scheint.

daß der Code für jedes System neu umgesetzt und in das betroffene Programm eingebunden werden muß. Für eine dynamische Anpassung an verschiedene Sprachen ist dies nicht der geeignete Weg.

Alternativ kann der Austausch auch in Form von Prozeßkommunikation erfolgen. In diesem Fall wird der Nachrichtenkanal durch das Betriebssystem bereitgestellt. Im Normalfall wird für den Prozeßaustausch ein Speicherbereich als Puffer reserviert. Damit der Austausch zweier unabhängiger Prozesse funktioniert, wird außerdem durch Semaphoren, Schloßvariablen, Mutexe und Signale dafür gesorgt, daß ein Prozeß nur dann auf den Puffer zugreift, wenn der Puffer gefüllt (zum Lesen) oder leer (zum Schreiben) ist. Nachteil dieser Methode ist, daß die Kommunikationsmechanismen zum Austausch für beide Seiten detailliert festgelegt werden müssen und nicht alle Sprachen direkten Speicherzugriff auf festgelegte Adressen zulassen. Innerhalb der beteiligten Prozesse muß außerdem sichergestellt werden, daß der Speicherbereich nicht überschrieben wird.

Eine Weiterentwicklung dieser Idee ist der Austausch über standardisierte Protokolle zwischen einzelnen Prozessen. Bekannte und weit verbreitete Kommunikationsprotokolle sind z. B. TCP/IP, UDP, ICMP und IPX[2]. Hier ist die Kommunkation bereits festgelegt, so daß für den Austausch noch ein entsprechendes Text- oder Binärformat festgelegt werden muß. Heute bieten die meisten modernen Sprachen aufgrund der Entwicklung des Internets Unterstützung für diese Protokolle. Da die Prozesse dann nicht mehr notwendigerweise auf einem Rechner laufen, hat man Verfahren entwickelt, mit denen ein Prozeß entscheiden kann, ob sein Kommunikationspartner noch erreichbar ist oder nicht.

Im Hinblick auf den Test von objektorientierten Programmen spielen allerdings zwei weitere Punkte eine Rolle:

- Objektorientierte Programme sind durch Vererbung und Komposition nicht mehr linear strukturiert. Ist das Ergebnis eines Methodenaufrufs ein Objekt, muß dieses in einer angemessenen Form an den Sender zurückgegeben werden. Eine lineare Text- oder Binärrepräsentation reicht hier nicht aus.

- Es ist außerdem erforderlich, daß Objekte auf der Seite des zu testenden Programms erzeugt werden können, z. B. als Argumente eines Methodenaufrufs. In diesem Fall ist eine Umsetzung der übermittelten Daten zur Laufzeit in ein existierendes Objekt notwendig.

In vielen Sprachen gibt es die Möglichkeit, Objekte in eine binäre oder textuelle Darstellung umzuwandeln, die dann auf einem Datenträger abgespeichert werden kann. Dieser Vorgang wird *Serialisierung* genannt. Die Serialisierung für neue Klassen muß der Anwender meist selbst festlegen, für vorgegebene API-Klassen der Sprache ist die Form der Serialisierung meist schon festgelegt. Darüber hinaus ist inzwischen ein Standard festgelegt worden, der nicht nur für Objektorientierung genutzt wird, sondern auch zum Austausch allgemeiner Informationen. Die *eXtensible Markup Language* (XML) definiert einen einfachen Standard, der trotzdem Informationen geschachtelter Strukturen aufnehmen kann und als gewöhnlicher Text dargestellt wird. Die Informationen werden dabei durch sogenannte Tags definiert, die genauer beschreiben, was der Inhalt bedeutet. Die Bedeutung der Tags muß jedoch durch das einlesende Programm festgelegt werden. Beispielsweise könnte die Information für ein Objekt der Klasse *Triangle* aus dem Beispiel folgendermaßen aussehen[3]:

[2] Eine genauere Erläuterung findet man z. B. unter *http://www.networksorcery.com/enp/rfc/rfc1180.txt*.

[3] Hier soll kein Anspruch auf Vollständigkeit und Korrektheit erhoben werden, tatsächliche Serialisierungen nach XML enthalten wesentlich mehr Informationen.

```
<object name = "Triangle">
  <attribute name = "points">
  <field length = "3">
     <index = "0">
        <object class = "Point">
        <attribute name = "x"> 0 </attribut>
        <attribute name = "y"> 0 </attribut>
        </object>
     </index>
     <index = "1">
        <object class = "Point">
        <attribute name = "x"> 1 </attribut>
        <attribute name = "y"> 0 </attribut>
        </object>
     </index>
     <index = "2">
        <object class = "Point">
        <attribute name = "x"> 0 </attribut>
        <attribute name = "y"> 1 </attribut>
        </object>
     </index>
  <field>
  </attribute>
</object>
```

Wie das Beispiel zeigt, bietet XML neben der recht guten Lesbarkeit zwei weitere Vorteile: Ein Parser kann recht einfach einzelne Elemente mit Beginn und Ende identifizieren, und die Information ist unabhängig von der späteren Repräsentation. Das dargestellte Objekt könnte sowohl ein C++- wie auch Java-Objekt sein. Weitere Informationen zu diesem Standard bietet [Wor04b].

Auch für dynamische Objekterzeugung bieten viele Sprachen Unterstützung. Viele dieser Mechanismen fallen unter das Schlagwort *Reflektion* oder *Laufzeitidentifizierung*, mit der ein objektorientiertes Programm zur Laufzeit Eigenschaften seiner eigenen Objekte analysieren und auswerten kann, in einigen Fällen können sogar dynamisch Methodenaufrufe oder neue Klassen generiert werden. Auch diese Lösung ist sprachspezifisch. In Kombination mit einem weiteren Standard, *SOAP* (Simple Object Access Protocol), läßt sich allerdings dieses Feature nutzen. SOAP setzt dabei auf XML auf und definiert ein eigenes Format zur Beschreibung von Methodenaufrufen und Rückgaben von Ergebnissen dieser Aufrufe. Durch Ausnutzung der angesprochenen Laufzeitmechanismen kann der Empfänger dann die Nachricht in einen tatsächlichen Methodenaufruf umsetzen und das Ergebnis entsprechend formatiert zurückgeben. Eine genauere Erläuterung des SOAP-Standards findet man unter [Wor04a]. Da SOAP vorwiegend für die Kommunikation zwischen zwei Prozessen verwendet wird, spricht man bei der Umwandlung nicht mehr von Serialisierung, sondern vom *Marshalling* der übermittelten Objekte bzw. Aufrufe.

Abschließend kann man sagen, daß mit SOAP und XML sowie TCP/IP (und ähnlichen Protokollen) sowohl für das Austauschformat als auch den Kommunikationsweg gute Lösungen existieren. Es bleibt allerdings das Problem der Schnittstellenanbindung: Die Schnittstellen müssen für die Transformation der sprachunabhängigen Darstellung in eine Sprache entsprechende Mechanismen bereitstellen. Zwei Ansätze, die die Definition dieser Schnittstellen umfassen und bisher am weitesten entwickelt worden sind, sollen in den beiden nächsten Abschnitten behandelt werden. Der erste Ansatz ist .NET, der zweite der offene Standard CORBA.

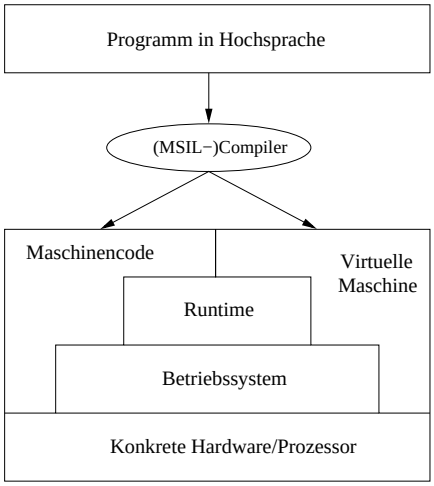

Abbildung 3.1: Beziehungen zwischen Code, Plattform und Virtueller Maschine (leicht modifiziert nach [Wes01])

3.1 .NET

.NET ist als Plattform seit 2002 für Windows verfügbar. Laut Aussagen von Microsoft ist .NET entwickelt worden, um den Austausch verschiedener Systeme vor allem über das Internet zu vereinheitlichen und zu erleichtern. In diesem Abschnitt sollen kurz die wichtigsten Aspekte von .NET erläutert werden.

Um verschiedene Sprachen über das Internet miteinander kommunizieren zu lassen, unterstützt .NET unter anderem SOAP und XML (s. Abschnitt 3). Die Entwicklung von Schnittstellen zum Austausch wird bei .NET dadurch vermieden, daß für alle Sprachen eine gemeinsame Grundlage geschaffen wird, die *MSIL* (Microsoft Intermediate Language), in die die Programmiersprache umgesetzt wird. Für jede Sprache existiert dann eine eigene Zuordnung, die die Eigenschaften der Sprache auf die MSIL abbildet. Die technische Umsetzung wird dann durch einen entsprechenden Compiler vorgenommen. Die Ausführung übernimmt – ähnlich wie bei Java – eine virtuelle Maschine. Unterstützt wird dieser durch die sogenannte *Common Language Runtime* (CLR), die benötigte Dienste und Bibliotheken bereitstellt. Um den Geschwindigkeitsverlust zu reduzieren, wird der MSIL-Code während oder vor der Ausführung kompiliert und in Maschinensprache umgesetzt. Abbildung 3.1 zeigt die Zusammenhänge grafisch.

Um die Umsetzung in die MSIL zu vereinheitlichen, hat Microsoft ein zentrales Objektmodell erstellt, daß die Merkmale und Eigenschaften von .NET beschreibt. Tabelle 3.1 zeigt die im ersten Kapitel angesprochenen Eigenschaften im Hinblick auf .NET.

Zum aktuellen Zeitpunkt unterstützt .NET verschiedene objektorientierte Sprachen, unter anderem Visual Basic, SmallTalk, Eiffel, C#, C++ und Oberon, sowie einige nicht-objektorientierte Sprachen wie PHP oder ML, die um objektorientierte Komponenten erweitert worden sind (insgesamt handelt es sich um mehr als 20 Sprachen). Da alle Sprachen in die MSIL umgesetzt werden, ist es zudem kein Problem, sprachübergreifend zu implementieren; so kann beispielsweise eine Eiffel-Klasse Eigenschaften von einer C++-Klasse erben. Gleichzeitig ist auch entfernte Kommunikation kein Problem: Durch Definition der Interfaces der entsprechenden Klassen können automatisch Beschreibungsdateien für an-

Merkmal	Ausprägung im Objektmodell
Typen	Jeder Typ in .NET ist eine Klasse. Basistypen wie z. B. int, float oder double werden aus Effizienzgründen in der Speicherverwaltung anders gehandhabt als Objekt, lassen sich jedoch durch einen automatischen Mechanismus in Objektcontainer einfügen und können dann wie Objekte gehandhabt werden.
Verweise auf Objekte	.NET kennt als Objektverweise nur Referenzen. Im Unterschied zu Java sind jedoch auch typsichere Verweise auf Funktionen, sogenannte *Delegates* möglich.
Konstruktor und Destruktor	.NET besitzt wie Java einen Garbage Collector, so daß ein Destruktor nicht gebraucht wird. Konstruktoren werden wie in Java oder C++ gehandhabt.
Abstrakte Klassen	.NET kennt keine abstrakten Klassen im Sinne von Java (`abstract`) oder C++, allerdings werden abstrakte Interfaces unterstützt. Diese Interfaces dienen außerdem zur Umsetzung der Anbindung an andere Sprachen durch den MSIL-Compiler.
Mehrfachvererbung	.NET unterstützt Mehrfachvererbung nur im Sinne von Java – eine Klasse kann nur eine Superklasse besitzen und zusätzlich beliebig viele Schnittstellen implementieren.

Tabelle 3.1: Eigenschaften des .NET-Objektmodells

gebotene Routinen erstellt werden (sog. WSDL-Dateien, Web Service Definition Language). Die CLR kümmert sich bei der Kommunikation um die Zuordnung einer Anfrage zu einem passenden Aufruf, was durch das einheitliche MSIL-Format leicht möglich ist. Der Programmierer muß dabei den Ort eines Zielobjekts nicht mehr unbedingt kennen, die Verwaltung übernimmt die CLR.

Im Hinblick auf die drei in der Umsetzung ausgewählten Sprachen wird Java aufgrund von rechtlichen Problemen von Microsoft nur bis Version 1.1.4 unterstützt, für erweiterte Features gibt es J# (eine von Microsoft an .NET angepaßte Version) oder Java .NET von RemoteSoft[4]. Für das saubere Zusammenwirken verschiedener Sprachen fordert .NET allerdings die Anpassung der Sprachen an das zentrale Objektmodell, so daß unter Umständen nicht mehr alle Eigenschaften einer Sprache unter .NET unterstützt werden. Zusätzlich müssen unter Umständen einige Anpassungen von Code vorgenommen werden, damit dieser unter .NET uneingeschränkt lauffähig ist.

.NET definiert neben dem oben genannten Objektmodell ein Sicherheitsmodell (in Bezug auf Dateizugriff, Ausführungsrechte o. ä.), dessen Einstellungen zum einen durch die Einstellungen bei Installation einer Software und zum anderen durch den ausführenden Benutzer beeinflußt wird. Der unter der MSIL (und damit unter den Sicherheitsbeschränkungen) laufende Code wird als *Managed Code* bezeichnet. Dies macht unter Umständen Schwierigkeiten bei der Übertragung:

- Bei C++ muß der Code mit einer *Managed Extension* versehen werden, um Garbage Collection und Sicherheitsbeschränkungen zu ermöglichen. Dabei muß eine C++-Klasse durch eine passende Proxy-Klasse „eingepackt“ werden, über die dann der Zugriff erfolgt.

- Auch Java definiert ein entsprechendes Sicherheitsmodell, das bei Ausführung durch die virtuelle Maschine (die Java-eigene JVM) geprüft wird. Hier können bei der

[4] Java .NET ist frei verfügbar unter *http://www.remotesoft.com/javanet/*.

Übertragung von Daten Schwierigkeiten auftreten. Bisher konnte nicht geklärt werden, in welchem Rahmen es Probleme gibt (auch bei RemoteSoft gibt es dazu keine Angaben, was die Realisierung von Java .NET angeht).

Die angedeuteten Punkte zeigen bereits einige Probleme auf. [Szy02] zieht eine allgemeinere Beurteilung, aus der sich folgende Aussagen ergeben:

- .NET bietet durch die Unterstützung vieler Sprachen eine gute Ausgangsbasis. Die Unterstützung für .NET kommt jedoch vorwiegend von Microsoft, so daß zum einen keine freie Verfügbarkeit der Unterstützung gewährleistet ist, zum anderen die Auswahl der Plattform und des Herstellers nicht besonders groß ist.

- .NET bietet auf binärer Ebene besondere Stärken (durch Festlegung der MSIL), allerdings keinen komplett festgelegten Standard auf Quellcodeebene. Hier bietet CORBA bereits genauere Festlegungen (s. auch den nächsten Abschnitt).

- .NET ist eine relativ junge Technologie, bei der sich noch zeigen muß, wie weit sie sich durchsetzt.

Die letzte Aussage ist allerdings aus heutiger Sicht zu relativieren. Auch wenn CORBA bereits der weiter entwickelte Standard ist, ist die Akzeptanz von .NET durch das Hinzufügen neuer Sprachen und einer lauffähigen Portierung auf Linux gestiegen.

Eine Instrumentierung mit Hilfe von .NET wäre auf zwei Arten denkbar:

- Man instrumentiert den Quellcode und setzt danach mit Hilfe des MSIL-Compilers die Struktur in den Code für die virtuelle Maschine um. Die Anbindung der Schnittstelle wäre ohne großen Aufwand möglich, weil .NET wie oben angedeutet z. B. Vererbung über verschiedene Sprachen hinweg unterstützt. Allerdings wird durch die beschriebenen Probleme bei der Einbindung die Instrumentierung des Quellcodes (je nach Sprache) deutlich aufwendiger.

- Alternativ kann auch der MSIL-Code instrumentiert werden, da dieser teilweise Meta-Informationen über den Quellcode enthält und einheitlich festgelegt ist. Nach wie vor bleibt aber das Problem, daß bestimmte Eigenschaften durch Anpassung an das zentrale Objektmodell verlorengehen oder bei Instrumentierung des MSIL-Codes die Programme vorher angepaßt werden müssen, um eine reibunglose Umsetzung und Lauffähigkeit zu erreichen.

Das erste Verfahren funktioniert analog zur Umsetzung mit CORBA, die im nächsten Abschnitt genauer beschrieben werden soll. Das zweite Verfahren wird beim Debugging von .NET-Programmen angewendet (Microsoft bietet eine entsprechende Anpassung seiner IDE an die Sprache an), so daß dieser Ansatz innerhalb der Arbeit nicht weiter verfolgt werden soll.

3.2 CORBA

CORBA (*Common Object Request Broker Architecture*) ist ein von OMG entwickelter Standard, der wie .NET ein passendes Objektmodell definiert. Im Fall von CORBA wird allerdings in einigen Punkten der umgekehrte Weg gegangen. Dies ist bereits daran zu erkennen, daß der Standard durch Beiträge einer Reihe von Softwarefirmen beeinflußt wurde,

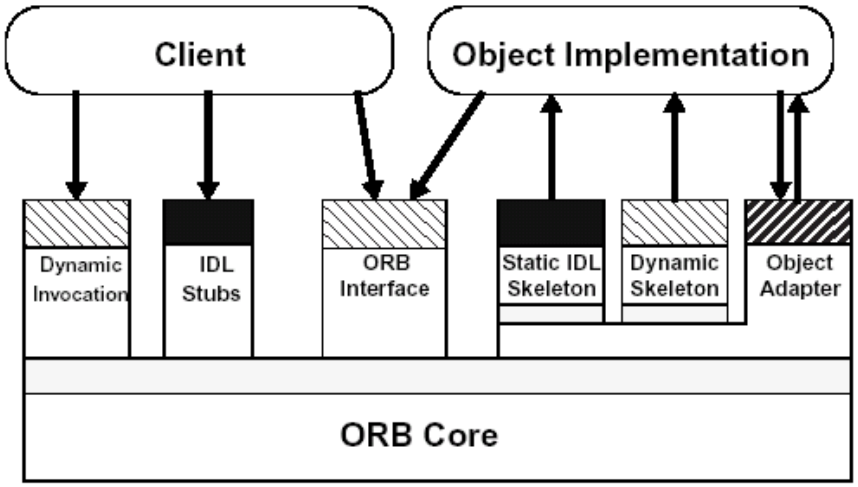

Abbildung 3.2: Struktur der CORBA-Architektur (übernommen aus [OMG02a])

die das OMG-Gremium mit in die Arbeit übernommen hat, während Microsoft das .NET-Modell zentral festgelegt hat. Der komplette Standard ist in [OMG02a] zu finden und wird in diesem Abschnitt nur in Ansätzen beschrieben.

CORBA definiert im Gegensatz zu .NET nur ein *abstraktes* Objektmodell, das keine konkrete technische Realisierung vorschreibt. Dieses Modell ist allerdings nur im Zusammenhang mit der von CORBA ebenfalls geforderten Architektur verständlich. Aus diesem Grund soll die in Abbildung 3.2 gezeigte Struktur kurz erläutert werden.

CORBA definiert die aufrufende Seite (also ein Programm, das Teile eines anderen aufruft), als Client, die empfangende Seite als Server, unabhängig davon, ob diese Rollen sich im Laufe der Kommunkation vertauschen. Dabei müssen Client und Server sich nicht notwendigerweise auf dem gleichen Rechner befinden, sondern können z. B. auch über das Internet kommunizieren. Der ORB (*Object Request Broker*) übernimmt dabei die Rolle des Vermittlers, bei entfernter Kommunkation können dies auch zwei ORBs sein, die sich gegenseitig kennen und austauschen. Analog zu .NET wird ein Methodenaufruf über den ORB umgeleitet und an die Objektimplementation der Zielsprache gegeben, wo der Aufruf wiederum in einen Methodenaufruf umgesetzt wird. Auch CORBA definiert Basistypen, die dann den Basistypen in der konkreten Sprache zugeordnet werden (wo Basistypen fehlen, wird eine passende Klasse geschaffen).

Mit diesen Eigenschaften erschöpft sich allerdings die gemeinsame Schnittmenge beider Modelle. Statt wie .NET die Aufrufe durch Umsetzung innerhalb einer gemeinsamen Zwischensprache zu realisieren, benutzt CORBA auf Seite des Clients sogenannte *Stubs*, die praktisch leere Kopien der auf Serverseite vorhandenen Klassen sind und für die Weiterleitung eines Aufrufs sorgen. Auf Serverseite existieren analoge Implementierungen für die Erzeugung einer Anfrage an ein Serverobjekt, die sogenannten *Skeletons*. Um den Code für Stubs und Skeletons zu erzeugen, benutzt man die *Interface Definition Language* (IDL), die analog zur WSDL von .NET die Signaturen der verwendeten Klassen beschreibt. Bei CORBA wird jedoch die IDL durch einen passenden IDL-Compiler in Stubs und Skeletons für die jeweils anzubindende Sprache umgesetzt. Die Anbindung erfolgt also zwischen den konkreten spracheigenen Klassen (z. B. in Form von SOAP), die aus einer gemeinsamen

Merkmal	Ausprägung im Objektmodell
Verweise auf Objekte	In CORBA werden Verweise als abstrakte Referenzen gehandhabt. Eine Referenz ist durch die Form der Architektur unabhängig von der Lebensdauer eines Objekts auf Serverseite.
Konstruktor und Destruktor	Innerhalb von CORBA existieren keine expliziten Konstruktoren oder Destruktoren. Ein Objekt wird für den Client „konstruiert“, indem er vom Server eine passende Objektreferenz erhält. Wird das Objekt auf Serverseite zerstört, erhält der Client eventuell erst eine Rückmeldung, wenn er eine neue Anfrage startet.
Abstrakte Klassen	Da in CORBA nur Interfaces definiert werden, machen abstrakte Klassen keinen Sinn. Allerdings existieren abstrakte Interfaces, für die erbende Interfaces dann passende Stubs und Skeletons zur Verfügung stellen.
Mehrfachvererbung	CORBA unterstützt Mehrfachvererbung von Interfaces; Bei Namenskonflikten muß allerdings stets festgelegt werden, welche der beiden Methoden an das aktuelle Interface vererbt wird, ein Zugriff auf beide kollidierenden Methoden ist nicht möglich.

Tabelle 3.2: Eigenschaften des CORBA-Objektmodells

(IDL-)Beschreibung entstanden sind, während .NET eine gemeinsame Sprache (MSIL) zur Anbindung benutzt, die aus speziellen Sprachen erzeugt wurde.

Tabelle 3.2 zeigt die neben dem Typsystem geltenden Eigenschaften des CORBA-Objektmodells. Um die Kommunkation weitestgehend festzulegen, definiert CORBA nun eine Reihe von Interfaces, über die beide Teile kommunizieren können. Der ORB läßt sich dabei ebenfalls durch Interfaces steuern. Zusätzlich ist es jederzeit möglich, dem ORB dynamisch Informationen über Aufrufe oder vorhandene, neu geladene Serverobjekte zu vermitteln. Die drei Schnittstellen für diese Aufgaben werden in der Abbildung mit *ORB Interface*, *Dynamic Invocation* und *Dynamic Skeleton* bezeichnet – der zuletzt genannte Teil funktioniert im Austausch mit dem *Interface Repository* des ORBs, das die bekannten Schnittstellen hält.

Im Hinblick auf konkrete Sprachanbindung ist das Objektmodell bedeutend offener. Für einzelne Sprachen existieren individuelle Mappings, die versuchen, auf Eigenheiten der Sprache einzugehen. Der Standard unterstützt ausdrücklich diese Anpassung.

Eine genauere Betrachtung der CORBA Spezifikation und einiger freier ORBs führt zu den folgenden Feststellungen:

- Aus der Zusammenfügung unterschiedlicher Vorschläge einzelner Firmen und der Anpassung des Standards an mehrere Sprachen (deren Liste bisher kürzer als bei .NET ist), ist an einigen Stellen ein Kompromiß eingegangen worden, der bei Anbindung unterschiedlicher Sprachen eine Rücksichtnahme auf individuelle Regelungen erfordert. Einzelne Probleme sollen im Praxisteil genauer erläutert werden.

- Da CORBA frei verfügbar und laut Angabe von OMG einfach zu implementieren ist, ist man als Anwender nicht auf ein einzelnes Produkt einer Firma angewiesen. Allerdings läßt der Standard dem Anbieter bei der Wahl des Binärformats und einigen Punkten der ORB-Konfiguration Freiheiten, so daß ORBs unterschiedlicher Anbieter nicht vollkommen kompatibel zueinander sind. Eine Umstellung eines ORBs auf einen anderen erfordert unter Umständen eine Anpassung der Ansteuerung des ORBs an bestimmten Stellen.

Darüber hinaus findet man in [Szy02] folgende Aussagen:

- CORBA bietet auf Quellcode-Ebene weitestgehend festgelegte Standards. Dies sorgt für eine einheitliche Schreibweise der entsprechenden Aufrufe in Bezug auf Methodennamen und Argumente. Zudem ist CORBA bereits einer Revision unterzogen worden, die auch dazu führt, daß der neue Standard nicht mehr rückwärtskompatibel zu älteren ist.

- Hinzu kommt, daß CORBA durch seinen Allgemeinheitsanspruch (Kommunikation zwischen zwei Prozessen unabhängig von Sprache, Ort und Plattform) keine zielgerichtete Anwendungsmöglichkeit anbietet. Dies kann dazu führen, daß sich in Bezug auf praktische Lösungen „Inseln“ bilden, deren Anwendung nur sehr speziell ist und keine weite Verbreitung zuläßt. Als Beispiel nennt der Autor WebSphere, bei dem der ORB in das Anwendungspaket eingebunden ist und speziell für die Anwendung innerhalb des Pakets konfiguriert wurde.

Abschließend läßt sich sagen, daß .NET und CORBA in etwa gleichwertig sind, was ihren Einsatz innerhalb der Schnittstelle betrifft:

- Mit .NET erweist sich die Instrumentierung des Quellcodes als schwieriger, damit der Code unter .NET uneingeschränkt lauffähig ist. Dafür wird keine separate Beschreibung der Anbindung benötigt.

- Mit CORBA ist zusätzlich zur Instrumentierung eine Generierung von IDL-Dateien notwendig. Dafür läßt CORBA den einzelnen Sprachen mehr Freiheit bei der Ausnutzung spracheigener Merkmale und beschränkt das Objektmodell weniger stark.

Da für .NET mit dem Debugger im Visual Studio bereits eine ähnliche Lösung entwickelt worden ist, soll hier geprüft werden, wie weit die Schnittstelle mittels CORBA realisierbar ist. Im folgenden Kapitel sollen zuerst die allgemeinen Merkmale und Anforderungen dieses Ansatzes festgelegt werden, anschließend folgt eine genauere Beschreibung der einzelnen Aufgabenbereiche und der Arbeitsweise der Schnittstelle.

Teil II

Design und Funktionsweise der Schnittstelle

Allgemeine Anforderungen

Aus den bisherigen Betrachtungen ergeben sich für die Schnittstelle folgende Anforderungen:

- Die Schnittstelle soll Quellcode-Instrumentierung der drei Sprachen Java, C++ und Eiffel unterstützen. Dabei soll die Anpassung an die gewählte Sprache dynamisch erfolgen, d. h. ohne Neustart des aufsetzenden Testwerkzeugs.
- Die Anbindung an die Schnittstelle soll über CORBA erfolgen. In dieser Hinsicht muß die Schnittstelle die passenden IDL-Dateien zur Anbindung generieren. Zur Generierung der Dateien muß aus den beteiligten Quellcode-Dateien die notwendige Information gewonnen werden.
- Im Hinblick auf die Durchführung der Testverfahren muß die Schnittstelle die Überprüfung des Zustands beteiligter Objekte und eine Überdeckungsmessung aufgerufener Methoden erlauben. Dazu sind mehrere Maßnahmen notwendig:
  - Die Schnittstelle muß die korrekten Stellen für eine potentielle Instrumentierung der Objekte identifizieren. Diese Stellen hängen von der jeweils gewählten Sprache ab.
  - Für eine Zustandsprüfung der Objekte müssen Zugriffsmethoden für die Attribute einer Klasse geschaffen werden, sofern diese nicht schon existieren.
  - Schließlich muß der eingefügte Code ebenfalls der Syntax der Sprache entsprechen. Die Schnittstelle muß also die Möglichkeit haben, kleine Codeabschnitte für die gewählte Sprache zu generieren.
- Für den Aufbau eines Testwerkzeugs müssen entsprechende Klassen geschaffen werden, die einem Benutzer möglichst einfachen Zugriff auf die Schnittstelle erlauben. Zudem soll die Schnittstelle erweiterbar sein, um die Funktionalität bei Bedarf anzupassen.

Um den Aufwand nicht unnötig zu erhöhen, ist die Funktionalität der Schnittstelle darauf ausgerichtet, daß der Quellcode korrekt ist, d. h. kompilierbar und lauffähig. Mit dieser Einschränkung verfällt der Aufwand einer syntaktischen Prüfung des Quellcodes. Die Fehlerbehandlung der Schnittstelle dient daher vorwiegend der Behandlung einer falschen Konfiguration oder fehlender Informationen. Auf konkrete Fehlerquellen wird ansatzweise eingegangen, die Beschreibung konzentriert sich allerdings mehr auf die Funktionsweise im Normalfall. Für die Fehlerbehandlung ist jedoch eine entsprechende Funktionalität vorgesehen. Die Schnittstelle geht außerdem davon aus, daß die Quellcode-Dateien innerhalb eines gemeinsamen Verzeichnisses liegen (wobei zur Gliederung weitere Unterverzeichnisse benutzt werden können – wichtig ist das gemeinsame Basisverzeichnis).

Da es vorwiegend um das Konzept der Umsetzung geht, sollen zwei weitere Anforderungen genannt werden, die allerdings eine untergeordnete Rolle spielen: Zum einen soll die Schnittstelle möglichst robust sein, zum anderen soll eine Instrumentierung des Quellcodes die Eigenschaften der Software weitestgehend unberührt lassen. Dies ist jedoch nicht in jedem Fall ohne weiteres möglich, z. B. in Bezug auf Geschwindigkeit. Eine Instrumentierung und eine zwischenzeitliche Rückmeldung an die Schnittstelle wird die Laufzeit der zu testenden Software in jedem Fall erhöhen.

Aus diesem Grund sollen die Aufgaben der Schnittstelle in zwei Richtungen weiter eingeschränkt werden:

- Die Schnittstelle unterstützt ausschließlich *funktionale Testverfahren*, wobei zusätzlich die Codeüberdeckung gemessen werden kann. Die Codeüberdeckung ist dabei auch die einzige Anwendung struktureller Testverfahren. Codeüberdeckung im Sinne von *Dataflow Coverage* (s. [Bin00]) wird dabei nicht unterstützt, weil dann lokale Variablen analysiert werden müßten. Auch nebenläufige Programme werden nicht unterstützt.

- Darüber hinaus muß für die Sprachen ein Kompromiß gefunden werden, da sie sich nicht nur in Syntax, sondern auch in der Semantik an einigen Stellen stark unterscheiden. Da CORBA mit seinem Objektmodell ohnehin einige Einschränkungen macht, werden bestimmte spezielle Testverfahren (z. B. für generische Klassen) nicht unterstützt[5].

Innerhalb der nächsten Abschnitte werden einzelne Teile der Schnittstelle genauer beschrieben. Diese Abschnitte behandeln bereits die detaillierten Regelungen des jeweiligen Vorgangs. Aus diesem Grund wird hier eine grobe Skizzierung des Gesamtablaufs gegeben:

- Der Benutzer übergibt der Schnittstelle die Namen der einzulesenden Dateien und konfiguriert das Verhalten der Schnittstelle außerdem durch Übergabe von Namen einzelner *Konfigurationsdateien*. Diese legen Details über die Sprache fest und werden in den einzelnen Abschnitten genauer beschrieben.

- Anschließend startet der Benutzer das Einlesen der Quellcode-Dateien durch einen entsprechenden Aufruf. Hier gibt die Schnittstelle bereits Rückmeldung über den Erfolg des Einlesevorgangs.

- Durch weitere Aufrufe stößt der Benutzer die Instrumentierung und die Erzeugung der IDL-Dateien an, die für die Anbindung an CORBA notwendig sind.

- Schließlich wird durch die Schnittstelle eine Kompilierung und Umsetzung aller Dateien vorgenommen. Danach kann über die beteiligten ORBs ein Testlauf vorgenommen werden.

Um den Rahmen der Arbeit nicht zu sprengen, beschränkt sich die Beschreibung auf den Vorgang der Instrumentierung und des Einlesens der gegebenen Dateien. Eine Erweiterung in Richtung Laufzeitüberwachung soll am Ende angedeutet, jedoch nicht genauer ausgearbeitet werden.

Die Beschreibung der Umsetzung basiert dabei auf den folgenden Werkzeugen und Sprachen:

- Die Umsetzung der Schnittstelle benutzt die STL und hält sich an den *C++ ANSI-Standard*. Zur Kompilierung wurde der GNU-Compiler (*g++*/*gcc*) verwendet.

- Für den Test der drei Sprachen wurden *SmartEiffel*, die *Java SDK 1.4.1* und die bereits angesprochene C++-Version verwendet.

- Als CORBA-Unterstützung wurden der *micoORB* (C++), *jacORB* (Java) und *mico/E* (Eiffel) verwendet.

[5] C++ bietet hier die Template-Klassen, Eiffel generische Klassen, die sich hervorragend als Container (z. B. Listen) einsetzen lassen, aber innerhalb der Schnittstelle nur mit bestimmten Einschränkungen instrumentierbar sind.

- Für das Einlesen der Konfigurationsdateien standen *flex* und *bison* zur Verfügung. Bison wurde als Parsergenerator verwendet, um die Struktur der Eingabedateien zu beschreiben und den Parser dafür automatisiert zu erstellen, flex lieferte als Scannergenerator den passenden Scanner.

Die Funktionsweise einzelner Werkzeuge kann bei den entsprechenden Quellen nachgeschlagen werden. Innerhalb der Erläuterungen wird nur auf diese Funktionsweise eingegangen, wenn es zum Verständnis notwendig ist oder entsprechende Besonderheiten des Werkzeugs berücksichtigt werden müssen.

Kapitel 4

Der Einlesevorgang

4.1 Allgemeiner Ablauf

Das Einlesen der Quellcode-Dateien innerhalb der Schnittstelle erfüllt zwei Aufgaben: Zum einen sollen die benötigten Informationen über existierende Klassen ermittelt werden, damit später die IDL-Dateien erzeugt werden können und zur Laufzeit entschieden werden kann, von welchem Typ ein Objekt ist. Zum anderen sollen die Stellen gekennzeichnet werden, die später für eine potentielle Instrumentierung zur Verfügung stehen. Der Benutzer soll per Methodenaufruf die Dateinamen angeben und anschließend durch einen weiteren Methodenaufruf den Einlesevorgang anstoßen.

Für den Einlesevorgang braucht die Schnittstelle Informationen über die Struktur der benutzten Sprache. Diese wird in zwei Konfigurationsdateien abgelegt: in der Regeldatei und der Optionsdatei. Die Regeldatei lehnt sich dabei an das bereits vorgestellte EBNF-Format einer Sprachgrammatik an, allerdings soll es dem Benutzer der Schnittstelle erspart bleiben, sämtliche Token einer Sprache zu definieren. Um das zu erreichen, wird ein modifizierter Ansatz gewählt, der im nächsten Abschnitt beschrieben wird. In der Optionsdatei befinden sich unter anderem Optionen, die beeinflussen, auf welche Weise der Quellcode eingelesen werden soll. Diese Optionen befinden sich in Tabelle 4.1. Im Allgemeinen läuft der Einlesevorgang folgendermaßen ab:

- Die vom Benutzer vorgegebenen Regeln der Regeldatei werden eingelesen und gespeichert. Auftretende Fehlermeldungen können nachträglich vom Benutzer ermittelt werden.
- Anschließend werden die vom Benutzer angegebenen Dateien eingelesen. Da ein kompletter Parser zu aufwendig wäre und auch nicht nötig ist, wird jede Datei mehrfach geöffnet und durchsucht. Es sollen dabei folgende Begriffe unterschieden werden:
  - **Bereich**: Als Bereich wird eine Anzahl von Zeichen innerhalb einer Quellcode-Datei aufgefaßt. Ein Bereich ist also durch eine Start- und eine Endposition erfaßbar (jeweils mit Zeile und Spalte).
  - **Kontext**: Unter Kontext soll eine Zusammenfassung aller abstrakten Begriffe verstanden werden, die einen Bestandteil einer Sprache bezeichnen z. B. Klasse, Methode, Schleife, Ausnahmebehandlungs-Abschnitt. Offensichtlich kann ein Kontext mehrere Bereiche umfassen (wie z. B. eine Klasse bei C++) oder ein Bereich mehrere Kontexte enthalten.

Eine Betrachtung des Zusammenhangs zwischen Kontexten und Bereichen ergibt zwei Regeln, die in den meisten Fällen zutreffen: Zum einen lassen sich Kontexte hierarchisch gliedern, wobei ein Kontext mehrere andere umfaßt, zum anderen nehmen in der Hierarchie niedrigere Kontexte meist auch einen kleineren Bereich innerhalb einer Datei ein (z. B. umfaßt eine Methode weniger Zeilen als ihre Klasse). Es ist also naheliegend, die Suche an der Hierarchie der Kontexte zu orientieren. Die Schnittstelle sucht dabei in Bezug auf folgende Kontexte nach Informationen:

  - Dateien
  - Module (Namespaces in C++, Packages in Java)
  - Klassen
  - innerhalb von Klassen: Methoden und Attribute
  - innerhalb von Methoden: z. B. Verzweigungen, Exception Handling, Schleifen und einfache Anweisungen

  Im ersten Durchgang wird dabei die komplette Datei betrachtet. Sofern passende Elemente gefunden werden, markiert die Schnittstelle diese Elemente und merkt sich deren Gültigkeitsbereiche für den nachfolgenden Suchvorgang. Für den gesamten Einlesevorgang werden die Dateien nacheinander eingelesen.

- Aus den eingelesenen Informationen werden interne Baumstrukturen aufgebaut, die die eingelesenen Elemente repräsentieren. Diese Elemente werden nach Abschluß des eigentlichen Einlesens zusammengefaßt und weiter strukturiert (z. B. um zu ermitteln, welche Superklassen zu einer Klasse gehören). Dies ist vor allem deshalb notwendig, damit Klassen „flachgeklopft“ werden können, denn der Zustand eines Objekts wird über alle enthaltenen Attribute festgelegt. Zudem wird an jedes Element eine eindeutige ID vergeben, die zur Identifizierung während eines Testlaufs notwendig wird.

- Zwei Sprachelemente, die die Hierarchisierung von Kontexten durchbrechen, sind Dateiimporte und der Import von Modulen (wie z. B. Namespaces in C++). Diese Importe werden als letztes vor Beginn der Umstrukturierung eingegliedert und während der Umformungen entsprechend berücksichtigt.

Aus der Baumstruktur ergibt sich für jedes eingelesene Element ein eindeutiger Pfad, der auch durch einen Bezeichner abrufbar ist. So bezeichnet

```
Example.cc:classes:classOne:methodOne:loop1
```

die erste Schleife innerhalb der Methode *methodOne* der Klasse *classOne*, die im Namespace *classes* in der Datei *Example.cc* liegt. Schleifen, Fallunterscheidungen, Alternativen, Exception Handling und atomare Anweisungen werden also durchnumeriert, um eine eindeutige Zuordnung innerhalb eines Codeblocks zu erreichen. Auf diese Weise existiert zu jeder ID genau ein Pfad.

Die für den Einlesevorgang benutzten Klassen der Schnittstelle lassen sich grob in drei Gruppen aufteilen:

- *Parsing*: Für das Parsen sind alle „Low-Level“-Klassen verantwortlich, die direkt auf den einzelnen Dateien operieren. Diese Klassen werden im nächsten Abschnitt gemeinsam mit der Regeldatei beschrieben.

Option	Bedeutung	Standardwert
nonIdentifierChars	Enthält alle Zeichen, die in der Sprache nicht Teil eines Bezeichners (Klassennamen, Typen etc.) sind. Mit Hilfe dieser Option erkennt die Schnittstelle, wann Bezeichner beginnen und enden.	`" \n\t"`
ModuleByDeclaration	Gibt an, ob ein Modul durch einfache Deklaration global für die gesamte Datei gilt. Dies ist zur Unterscheidung zwischen C++ und Java wichtig.	`"no"`
ModuleByDirectory	Gibt an, ob Module durch Verzeichnisse gegeben sind. Dies entspricht dem Mapping für Java und Eiffel.	`"no"`
ExplicitInclude	Gibt an, ob ein Datei-Import durch explizite Angabe erfolgt oder ein anderer Mechanismus dafür zuständig ist. Dies ist für die spätere Elementzuordnung wichtig.	`"no"`
ModuleSeparator	Gibt den Trennstring für ineinander geschachtelte Module an, z. B. `"."` für Java, `"::"` für C++.	`"."`
ClassSeparator	Gibt den Trennstring zwischen Klassen- und Modulnamen bzw. Klassen- und Methodennamen an.	`"."`
ModuleImportSymbol	Gibt den String für den Import von gesamten Modulen in eine Datei an. Für Java ist dies `"*"`, wie z. B. in `import java.lang.*`.	`""`
MethodHoldsClassName	Zeigt an, ob eine Methode in einer Datei separat mit Angabe der Klasse definiert wird.	`"no"`

Tabelle 4.1: Optionen, die das Einlesen beeinflussen. Einige Optionen werden nur mit `yes` oder `no` angegeben, die letzte Spalte zeigt den Wert, für den Fall, daß die Option in der Datei vollkommen fehlt.

- *Informationsassimilation*: Diese Klassen steuern die Parsing-Klassen an und speichern die Informationen innerhalb der Baumstrukturen.
- *Relationenanalyse*: Diese Klassen fassen nach den ersten beiden Phasen alle eingelesenen Informationen zusammen und stellen beispielsweise die korrekten Beziehungen zwischen Sub- und Superklassen her. Diese Informationen werden später für die Generierung der IDL-Dateien und zur Durchführung eines Testlaufs benötigt.

Zusätzlich zu diesen drei Klassengruppen gibt es außerdem Hilfsklassen zur zur Anzeige unterschiedlicher Fehlersituationen beim Einlesen. Die Abläufe und das Zusammenwirken einzelner Klassen werden im nächsten Abschnitt beschrieben.

4.2 Die Bedeutung der Regeldatei und ihre interne Darstellung

Wie in Abschnitt 4.1 angemerkt wurde, dient eine Modifizierung der EBNF-Darstellung als Beschreibung für die Syntax der Sprache. Ausgehend von der EBNF lassen sich dabei folgende Unterschiede feststellen:

- Da die Schnittstelle nur bestimmte Kontexte und Punkte innerhalb des Quellcodes erkennen muß, reicht eine Angabe der entsprechenden Stellen vollkommen aus. Betrachtet man z. B. eine Klassendeklaration in C++, stellt man fest, daß diese Deklaration durch `class` eingeleitet und durch zwei geschweifte Klammern begrenzt wird. Der Klassenname folgt direkt nach dem Schlüsselwort. Da alle inneren Elemente in einem folgenden Durchgang eingelesen werden, reicht die Angabe dieser Fragmente, um eine Klasse zu identifizieren.

- Eine gewöhnliche EBNF sagt nichts über die Bedeutung einzelner Elemente aus, sondern nur über deren Schreibweise. Für das eben angegebene Beispiel wäre es aber auch wünschenswert, wenn zusätzlich zur Identifizierung der Klasse der Name und für die Instrumentierung wichtige Stellen gespeichert werden. Im Compilerbau werden zusätzlich sogenannte *Attributregeln* zur Grammatik angegeben, die beschreiben, wie die eingelesenen Elemente verarbeitet werden sollen. Um dem Benutzer eine zusätzliche Angabe zu ersparen und die Regeldatei möglichst einfach zu halten, soll die Kennzeichnung bereits durch Angabe spezieller Elemente geschehen. Diese Elemente führen keinen Vergleich durch, sondern speichern die geforderten Informationen. Diese speziellen Elemente sollen im weiteren Text in Bezug auf eine Regel auch als *Argumente* dieser Regel bezeichnet werden.

- Eine dritte Änderung betrifft den Namen der Regeln. Dieser hat in der EBNF nur die Aufgabe, Regeln eindeutig unterscheidbar zu halten. Die Schnittstelle betreibt im Gegensatz zu einem Compiler allerdings eine aktive Suche und muß daher den Regelnamen mit einer bestimmten Bedeutung verbinden. Aus diesem Grund werden einige Namen fest für bestimmte Zwecke reserviert, ansonsten ist eine Benutzung von Hilfsregeln ohne weiteres möglich.

Tabelle 4.2 zeigt alle bisher möglichen Regelelemente die innerhalb eines Regelrumpfes verwendet werden können, und deren Bedeutung. Im Anhang befindet sich in Tabelle B die Auflistung der reservierten Regelnamen und den zugehörigen Möglichkeiten, spezielle Elemente einzusetzen.

Der Benutzer muß selbst darauf achten, daß die Regelangabe den korrekten Bereich abdeckt. Auf den ersten Blick scheint für eine Klasse in Java die Regel

```
Class -> "class" @Name "{" "}"
```

auszureichen, allerdings beginnen und enden Methoden, Schleifen und andere Konstrukte ebenfalls mit geschweiften Klammern, so daß

```
Class -> "class" @Name "{" Block* "}"
Block -> "{" Block* "}"
```

die richtige Angabe ist. Um nun noch die Position der Klasse korrekt zu kennzeichnen, reicht eine Ergänzung zu

```
Class -> "class" @Name "{" ^BEGIN Block* END^ "}"
Block -> "{" Block* "}"
```

Schreibweise	Bedeutung
`"Name"`	Sucht nach dem nächsten Vorkommen von *Name*.
`Regelname`	Verweist auf eine Regel mit dem Namen *Regelname*, die analog zum Compilerverfahren substituiert wird. Eine Verfolgung socher Verweise geht standardmäßig bis zu einer Tiefe von zehn Verweisen.
`@Name`	Liest den nächsten Bezeichner ein und speichert ihn unter *Name*. Dabei ist es ohne Probleme möglich, mehrere verschiedene Einträge unter dem gleichen Namen zu erzeugen.
`^Name`	Speichert die Dateiposition, an der das vorhergehende Element gefunden wurde, unter *Name*.
`Name^`	Speichert die Dateiposition, an der das nächste Element gefunden wird, unter *Name*.
`$Name`	Findet den nächsten Basistypen der jeweiligen Sprache und speichert ihn unter *Name*.
`Regel -> El1 El2 ...`	Sucht nacheinander nach *El1* und *El2*, *El1* muß vor *El2* gefunden werden. Bei mehreren Vorkommen von *El1* und *El2* werden immer die beiden am dichtesten zusammenliegenden Elemente gefunden. *Ausnahme*: *El1* liest einen Bezeichner ein, markiert eine Position oder speichert einen Basistypen. In diesem Fall erfolgt das Einlesen von *El1* an der nächstmöglichen Stelle, egal wo *El2* gefunden wird.
`Regel -> El1 | El2`	Sucht nach *El1* oder *El2*.
`Regel -> {El1} El2`	*El1* ist optional und muß nicht gefunden werden, *El2* muß vorkommen.
`Regel -> El1*`	*El1* kann mehrfach auftauchen, muß aber nicht gefunden werden.
`Regel -> El1+`	*El1* muß mindestens einmal, kann aber auch mehrfach vorkommen.

Tabelle 4.2: Schreibweise der Regelelemente. Die Schreibweise kann beliebig geschachtelt werden, auch Klammerung ist möglich.

Der Rest der Erläuterung bezieht sich auf Abbildung D.1 im Anhang. Die dort abgebildete Klassenstruktur soll hier genauer erklärt werden.

Die Repräsentation eines Regelelements übernimmt die Klasse *RuleItem*. Jedem der in der Tabelle angegebenen Konstrukte ist dabei eine Subklasse der Klasse *RuleItem* zugeordnet. Die Regeldatei wird mit Hilfe des Scannergenerators *flex* und des Parsergenerators *bison* eingelesen und in eine passende Baumstruktur aus Regelobjekten umgesetzt, die dann das Parsen übernehmen. Abbildung 4.1 deutet die nach dem Composite Pattern[1] funktionierende Struktur für das Beispiel `class`-Regel an. Die Speicherung der Baumstruktur wird von der Klasse *RuleRepository* übernommen, die die Regeln mit dem entsprechenden Namen verknüpft. Von dieser Klasse erhält man durch Aufruf der Methode *getRule()* eine Kopie der eingelesenen Regel, die dann mittels *search()* bzw. *searchAgain()* zur Suche verwendet werden kann.

Jedes spezielle Item implementiert die abstrakten Methoden der Klasse *RuleItem* und einige weitere speziellen Methoden, mit denen die Baumstruktur aufgebaut wird. Die Klassen *TypeItem* und *VariableItem* greifen dabei auf die Klassen *Configuration* und *TypePool* zurück. In der ersten Klasse stehen nach Einlesen der Optionsdatei alle Optionen, in

[1] Die Beschriebung des Patterns findet sich in [GHJV02].

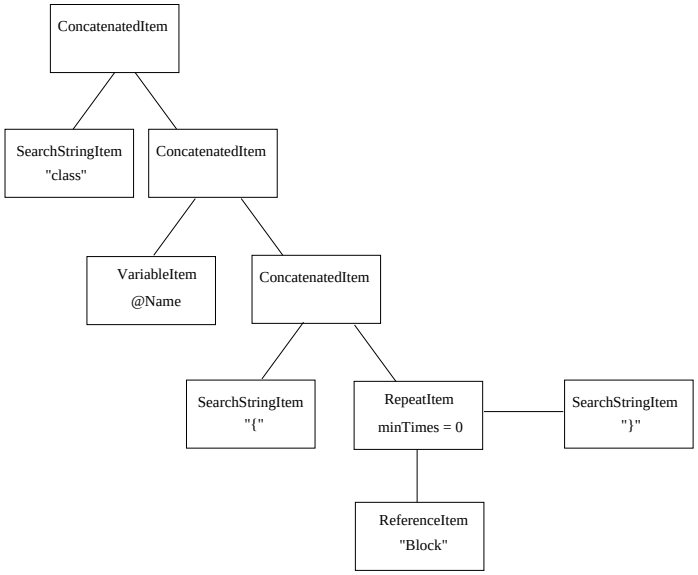

Abbildung 4.1: Struktur der `class`-Regel (mit einigen Attributen der Objekte).

der zweiten nach Einlesen einer dritten Konfigurationsdatei, der *Typendatei*, alle Basistypen der Sprache[2]. Diese Informationen werden benötigt, um Bezeichner und Basistypen korrekt einzulesen. Eingelesene Informationen einer Regel werden dabei vorläufig in der Klasse *AssignmentHolder* gespeichert. Ein Suchauftrag wird dabei über die Baumstruktur nach unten gereicht und in Teilanfragen zerlegt, gefundene Ergebnisse werden nach oben zurückgegeben. Je nach Semantik des Elements geschieht dies zu einem passenden Zeitpunkt. Auch die Rückmeldung über Erfolg oder Fehlschlag hängt von der Semantik des jeweiligen Elements ab.

Zur schnelleren Bestimmung, an welches der untergeordneten Elemente tatsächlich ein Suchauftrag weitergegeben werden soll, wird vor Durchführung der tatsächlichen Suche je eine Direktormenge für jedes Element errechnet und für beide Mengen ermittelt, welcher Textanfang zuerst in der Datei auftaucht[3]. Zu diesem Zweck existiert die Methode *getPartialString()*, die diese Berechnung durchführt. Auch hier wird die Berechnungsanfrage nach unten weitergegeben und das Ergebnis nach oben gereicht.

Die Struktur des *RuleItem*-Baumes bleibt bei der Suche bis auf eine Ausnahme statisch: Ein Regelverweis (*ReferenceItem*) expandiert den Baum und fügt die unter dem Regelverweis befindliche Regel in die verweisende Regel ein, sobald sie gebraucht wird. Zu diesem Zweck ruft die Klasse *ReferenceItem* die Methode *getLastRuleItem()* auf, die das letzte Element der Regel zurückgibt. Diesem letzten Element wird dann der Rest der Regel übergeben, um die Bestimmung der Direktormenge korrekt zu halten.

Die Sammlung der Regelelemente läßt sich ohne weiteres ergänzen. Bei einer Erweiterung müssen allerdings folgende Punkte berücksichtigt werden:

- *doSetStart()* und *doSetLimit()* müssen innerhalb einer Baumstruktur die Start- bzw. Endposition einer Suche im Baum weiter nach unten reichen. Dasselbe gilt für die Methode *getLastRuleItem()* (sofern das neue Element nicht selbst das letzte ist).

[2] Auf die Typendatei wird im nächsten Kapitel genauer eingegangen.

[3] Diese Berechnung wird beispielsweise von *ChoiceItem* verwendet, um zu entscheiden, welches der beiden untergeordneten Elemente gewählt wird – entsprechend der Formulierung `A | B`.

- Das neue Element muß selbsttätig folgende Angaben aktuell halten, damit andere Elemente darauf zurückgreifen können:
  - Aktuelle Zeile und Spalte (*line* und *col*)
  - Matchbeginn und Matchende (*currentMatchLine* und -*Col* bzw. *endMatchLine* und -*Col*
  - Informationen darüber, ob Kommentare überlesen werden (*skipping* und *delegate*, wobei *skipping* angibt, ob das Element gerade einen Kommentar überliest, während *delegate* kennzeichnet ob ein untergeordnetes Element gerade nach Kommentaren sucht)
- Im Hinblick auf den Speicherplatz sind *Buffer* und *StreamBuffer* in der Lage, nur den notwendigen Teil einer Datei zu halten, der gerade untersucht werden soll. Eine Regel steuert den Puffer nur über die Klasse *BufferIterator* an. Per *setMark()* kann an der aktuellen Position eine Marke gesetzt werden. Wird mittels *deleteMark()* eine entsprechende Marke gelöscht, prüft der Puffer selbständig, ob auf einem gehaltenen Stück keine Marke mehr existiert. Eingelesener Text wird dann bis zur nächsten Marke verworfen.

  Auch hier muß ein neues Element rechtzeitig Marken löschen, weil der Puffer in seiner maximalen Größe begrenzt ist und eine gehaltene Marke verhindert, daß der Puffer wieder geleert werden kann.

4.3 Einlesen und Repräsentation der Quellcode-Informationen

In diesem Abschnitt werden die beiden restlichen Klassengruppen beschrieben, die den Einlesevorgang unterstützen. Damit deutlich wird, welche Funktionalität die Klassen bieten müssen, ist eine genauere Betrachtung des Einlesevorgangs notwendig. Es lassen sich zwei wesentliche Punkte erkennen:

- Die Kontexthierarchie einer Sprache erlaubt die Schachtelung verschiedener Kontexte ineinander (z. B. Module und Schleifen oder Verzweigungen), so daß nur ein Durchgang für einen bestimmten Kontext nicht ausreicht. Zudem können untergeordnete Kontexte verschiedener Art in einem übergeordneten Kontext auftauchen. In Methoden finden sich z. B. einfache Anweisungen genauso wie Schleifen und Verzweigungen.

  Die Lösung dieses Problems besteht darin, abhängig vom Kontext nacheinander nach den passenden Subkontext zu suchen und bei einem Fund den ursprünglichen Suchbereich aufzuspalten. In den verbleibenden Bereichen können dann weitere Subkontexte gesucht werden (die aber nach wie vor im gleichen Kontext liegen).
- Es kann außerdem sein, daß die Identifikation der Bereiche der Subkontexte nicht eindeutig ist, weil zwei Regeln teilweise gleiche Elemente verwenden. Für die drei verwendeten Sprachen gibt es eine Stelle, an der das Phänomen auftritt: bei `while`- und `do-while`-Schleifen (ein `while` kann sowohl Anfang der einen wie auch Ende der anderen Schleife sein). Aus diesem Grund wird für C++ und Java die Einschränkung gemacht, daß `do-while`-Schleifen in `while`-Schleifen transformiert werden müssen.

Die Abbildungen D.2 und D.3 im Anhang zeigen die Klassen, die zur Speicherung der Hierarchie zuständig sind. Über das Attribut *subNodes* werden untergeordnete Knoten gehalten, während *superNode* die Navigation zum Elternknoten erlaubt. Die Methode *getIterator()* erlaubt es, einen Iterator für alle Subknoten zu erzeugen. Dieser Iterator wird gebraucht, um die Subknoten nach einem bestimmten Bezeichner zu durchsuchen. Jeder Knoten hält dabei seinen Namen im Attribut *name* und repräsentiert einen Kontext. Einige Besonderheiten einzelner Knoten sind:

- Die Klasse *ModuleNode* hält Verweise auf andere Modulteile, da sich ein Modul unter Umständen auf mehrere Dateien verteilt. Wie erwähnt, entspricht dies bei Java den Packages, bei C++ Namespaces, die wieder geöffnet werden können. Bei Eiffel werden Module direkt auf Verzeichnisnamen abgebildet, in denen einzelne Dateien liegen.

- Eine Methode wird in zwei getrennten Knoten gespeichert (*MethodDeclarationNode* und *MethodNode*). Dies erlaubt es, Sprachen zu behandeln, bei denen Deklaration und Definition einer Methode getrennt erfolgt. Darüber hinaus lassen sich durch Speicherung einer einzelnen Deklaration abstrakte Methoden behandeln. Bei Methodenargumenten werden die Argumenttypen durch eine ID beschrieben, wobei die ersten Nummern für Basistypen reserviert sind. Diese Regelung gilt auch für Klassenattribute und Rückgabewerte von Methoden. In beiden Fällen wird als Subknoten ein Objekt der Klasse *TypeNode* eingefügt, das neben dem Namen den Typ und weitere Kennzeichnungen hält. Unter anderem wird hier das Attribut *referenceMode* dazu verwendet, um zu kennzeichnen, ob ein Verweis eine Referenz, eine Variable oder ein Pointer ist.

- In der Klasse *ClassNode* wird *addSubNode()* überschrieben, weil die Klasse teilweise temporäre Informationen über Zugriffsrechte von Methoden hält. Zum Festlegen dieser Informationen dienen die Methoden *setPublicArea()*, *setPrivateArea()*, *setProtectedArea()*, *addExport()* und *addTargetFor()*. Die ersten drei Methoden dienen der korrekten Festlegung unter C++ (bei der Zugriffsrechte für die Methode bei der Klassendeklaration festgelegt werden), die letzten drei Methoden der korrekten Festlegung für Eiffel (bei der individuelle Zugriffsrechte je nach zugreifender Klasse möglich sind).

- *ImportNode* und *IncludeNode* dienen der Anzeige von Datei- und Modulimporten (z. B. per `#include` bzw. `using` in C++). Diese Knoten sind jedoch temporär und werden bei der Relationenanalyse aufgelöst (siehe unten).

- *BlockNode* ist eine Hilfsklasse, um die Baumstruktur nicht unötig kompliziert zu machen. Diese Klasse wird beim Aufstellen von `switch`-Konstrukten und geschachtelten `if`-Anweisungen verwendet.

Abbildung D.4 im Anhang zeigt die auf dieser Struktur aufsetzenden Klassen. Von zentraler Bedeutung sind dabei die Klassen *FileParser* und *InformationReader*. Beide Klassen sorgen während des Einlesevorgangs für die korrekte Umsetzung der Optionen der Optionsdatei und die Auswahl der für einen Einlesevorgang notwendigen Regeln. Die Klasse *FileParser* übernimmt dabei den kompletten Einlesevorgang und die korrekte Aufspaltung in einzelne Schritte, während eine konkrete Subklasse von *InformationReader* das Einlesen einer speziellen Struktur über den zugeordneten Bereich durchführt. Zur Verwaltung der eingelesenen Informationen dient die Klasse *Symboltable*, die jeweils den Wurzelknoten einer Baumstruktur vom Typ *FileNode* hält. Die Klasse *MappingTable* erhält dabei nach

Aufbau und Umformung des Baumes die Beziehung zwischen Pfad und ID, die später zur Instrumentierung gebraucht wird.

Die Abbildungen D.5 bis D.9 im Anhang zeigen typische Abläufe und heben einige Besonderheiten hervor:

- Abbildung D.5 zeigt die Initialisierung der Klasse *FileParser*. Die Initialisierungsroutine kümmert sich auch um das Einlesen der Konfigurationsdateien durch Aufruf passender Methoden der Klassen *TypePool*, *RuleRepository* und *Configuration*. Zusätzlich erzeugt sie die Wurzeln der Bäume für jede einzelne Datei. Darüber hinaus werden die ersten Module innerhalb der Datei-Bäume erzeugt, falls die entsprechende Option (`ModuleByDirectory`, s. Anhang) gesetzt ist. Dies wird in der Abbildung nicht dargestellt.

- Abbildung D.6 zeigt den typischen Ablauf des Einlesens innerhalb eines Kontexts. Die Klasse *FileParser* holt dazu aus einer Liste offener Kontexte (die zu Beginn durch gesamte Dateien dargestellt werden), den nächsten heraus und instantiiert einen entsprechenden Reader. Dieser durchsucht den Kontext nach passenden Informationen und erzeugt daraus neue Bereiche und Knoten, die in den aktuellen Kontextknoten eingehängt werden. Der Reader erzeugt vor allem auch Bereiche, in denen kein passendes Vorkommen gefunden wurde, indem das Ende des letzten Vorkommens und der Anfang des nächsten Vorkommens verglichen werden. Dieses Verhalten ist zur Vereinfachung aus allen Abbildungen weggelassen worden, jedoch deutet die Sortierung der Bounds durch den FileParser an, wie der Mechanismus funktioniert. Die in *localBounds* einsortierten Bereiche stellen immer den Restbereich des aktuellen Vorgangs dar.

- Abbildung D.7 illustriert ein konkretes Verhalten eines Readers am Beispiel der Klasse *ModuleReader*. Der dargestellte Ablauf gilt für den Fall, daß die Option `DeclareModuleByName` auf `yes` gesetzt ist (s. Anhang). Da in diesem Fall Modulnamen zusammengesetzt sind, werden die einzelnen Teile des Namens in eine Reihe von Modulknoten umgesetzt. Eingehängt wird der oberste Knoten, aktueller Kontext für den neuen Bereich wird der unterste. Diese Zerlegung findet bei anderen Bezeichnern vorerst nicht statt. Zudem ist der Einlesevorgang bei Modulen der einzige, bei dem keine IDs vergeben werden. Dies geschieht erst bei der Relationenanalyse (s. unten).

- Abbildung D.8 zeigt analog den Vorgang für die Klasse *ClassReader*. Hier wird die Umsetzung individueller Argumente deutlich, die in der Regeldatei angegeben werden können. Der Ablauf zeigt auch die Zuordnung zusammmengehöriger Argumente am Beispiel individueller Exporte. Durch Einordnen der gefundenen Informationen in eine nach Positionen sortierte Liste lassen sich später zusammengehörige Argumente leicht zuordnen. Eine ähnliche Verfahrensweise wird auch beim Zuordnen von Argumentnamen einer Methode zu ihren Typen vorgenommen.

- Die letzte Abbildung zeigt die Klasse *ImportReader*. Da dieser Vorgang erst zum Schluß durchgeführt wird, ist hier die Einsortierung in den Baum wichtig. Der Parser betrachtet der Reihe nach alle Dateiknoten und sucht für den aktuellen Dateiknoten und einen gefundenen Import den Knoten im zugehörigen Baum, der den kleinsten noch umfassenden Bereich besitzt. In diesen Knoten wird der Import als Subknoten eingegliedert. Dasselbe Verfahren gilt auch für Dateiimporte, die durch die Klasse *IncludeReader* behandelt werden. Da je nach Sprache Suchmechanismen statt expliziter Dateiimporte verwendet werden, kann über die Option `ExplicitImport` die

Schnittstelle so konfiguriert werden, daß automatisch Importknoten eingefügt werden (in diesem Fall importiert jede Datei alle anderen). Bei gesetzter Option wird nach erfolgreicher Auflösung einer `include`-Anweisung die Lage der Anweisung in der Klasse *IncludeTable* abgelegt, da die Informationen später zur Instrumentierung gebraucht werden (s. Abschnitt 6.2).

Nach dem Aufbau der Baumstrukturen müssen jetzt die Zugehörigkeiten korrekt aufgelöst werden, die bisher nur in Form von Zeichenketten innerhalb der Knoten gehalten werden. Dies passiert in der Relationenanalyse. Dabei werden folgende Schritte vorgenommen:

- Dateiimporte werden aufgelöst und die Importknoten durch Verweise auf die im Dateiknoten enthaltenen Subknoten ersetzt.
- Danach werden alle Module zusammengefaßt, indem gleichlautende Knoten innerhalb eines Baumes gesucht und gegenseitige Verweise in die Knoten eingetragen werden. Da nun alle Module eindeutig sind, können für die Module IDs vergeben werden. Auch hier geschieht die Zusammenfassung durch das Aufstellen von Verweisen.
- Nach Auflösung der Module können auch Modulimporte durch passende Verweise ersetzt werden. Anschließend werden innerhalb der Dateien die Klassen zu angegebenen Superklassen zugeordnet.
- Jetzt werden noch nicht zugeordnete Methodendefinitionen zu Klassen zugeordnet (wie z. B. bei C++, wo Deklaration und Definition einer Methode häufig in verschiedenen Dateien stehen). Anschließend können Verweise von Subklassen auf vererbte Methoden und Attribute der Superklassen gesetzt werden.
- Zum Schluß werden alle noch offenen Typverweise aufgelöst (Resultat- und Argumenttypen von Methoden, Attributtypen).
- In einem nachfolgenden Durchlauf werden alle Methodendeklarationen, für die keine Definition existiert, als abstrakt gekennzeichnet (und zugehörige Klassen ebenfalls). Existieren noch unaufgelöste Verweise, werden diese gesammelt und innerhalb einer Fehlerliste zurückgegeben. Ein Testlauf ist aber dennoch möglich. Unaufgelöste Verweise treten beispielsweise dann auf, wenn innerhalb des Quelltextes Bibliotheksklassen verwendet werden, deren Quelltext nicht zur Verfügung steht.

Abbildung 4.2 zeigt das Beispiel eines solchen Baumes schematisch vor (oben) und nach (unten) Auflösen der Verweise.

Das Auflösen der Verweise wird durch die in Abbildung D.10 im Anhang abgebildeten Klassen vorgenommen. Dabei erfolgt die gesamte Steuerung durch die Klasse *Transformer*. Die verschiedenen genannten Schritte werden jeweils durch eine Subklasse von *TreeOperation* vorgenommen. Jede dieser Subklassen erhält außerdem ein Iterator vom Typ *TreeIterator*, mit dem über den Baum iteriert werden kann, konkrete Subklassen der Iteratorklasse legen dabei die Traversierungsrichtung fest. Zusätzlich kann durch die Operation ein Filter vom Typ *NodeComparator* an den Iterator übergeben werden. Auf diese Weise liefert *getNext()* nur den nächsten Knoten, der bestimmte Kriterien erfüllt. Der Rückgabewert von *execute* bestimmt dabei, ob eine Änderung am konkreten Baum stattgefunden hat oder nicht.

Die Verweise werden, wie bereits angedeutet, entweder in Form einer Referenz oder in Form einer ID gesetzt. Im ersten Fall wird durch Aufruf von *getProxy()* am gefundenen

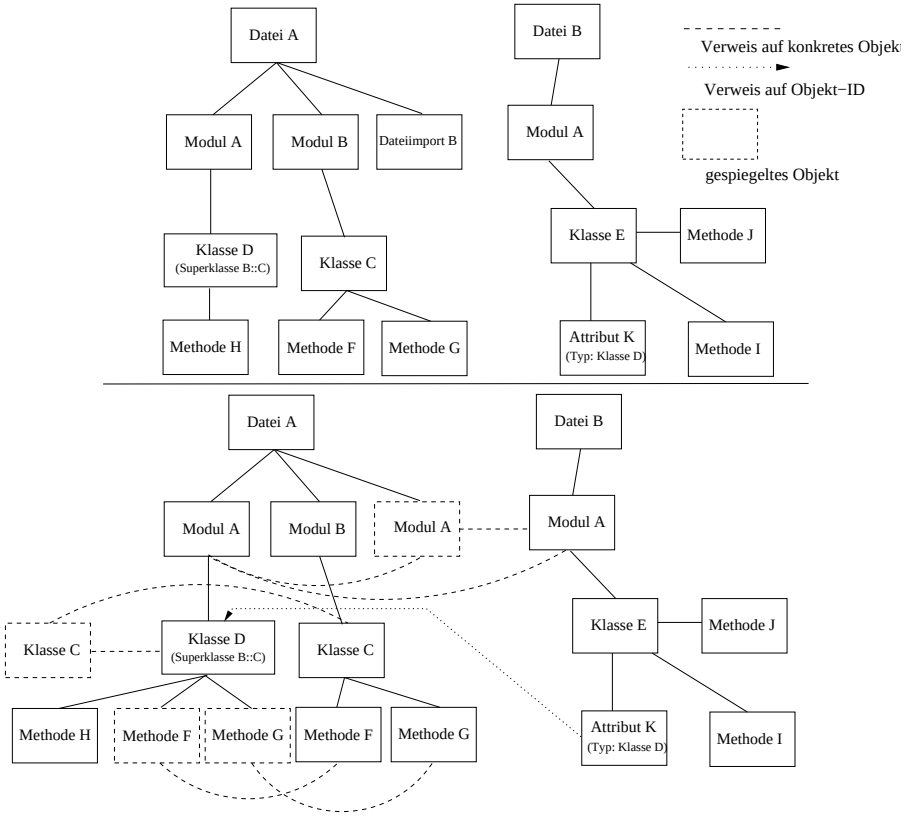

Abbildung 4.2: Auflösung von Verweisen in der Baumstruktur.

Knoten ein Proxy erzeugt, der eine Subklasse des jeweiligen Knotentyps ist und nur die Aufgabe hat, den Zugriff auf den Originalknoten weiterzuleiten. (Eine Ausnahme bildet die Klasse *MethodAliasNode* – siehe unten). Diese Proxyklasse existiert für jeden Kontextknoten, der ein Methodenknoten ist oder diesen umfaßt. Da die Proxies dasselbe Interface wie der jeweilige Knoten besitzen, wird auf eine Abbildung verzichtet. Für viele Operationen ist der Ablauf folgender:

- Die Operation sucht mit Hilfe des Iterators im Baum den nächsten passenden Knoten, der noch nicht aufgelöst wurde.
- An diesem Knoten wird jetzt die Auflösung vorgenommen. Bei Auflösung der Klassenbeziehung und der späteren Typauflösung für Attribute, Methodenargumente und Rückgabetypen wird durch Aufruf von *resolve()* bzw. *resolveClasses()* die Verantwortlichkeit an den jeweiligen Knoten weitergegeben. Der Knoten führt jetzt selbst eine Suche nach fehlenden Verweisen über den Baum durch, wobei er selbst der Startknoten ist. Dies hat den Vorteil, daß dadurch der Bezeichner und Zugriffsrechte für die Suche mit einbezogen werden können. Analog funktioniert die Ausflösung der Zugehörigkeit von Methoden zu Klassen mittels *resolveClass()*.
- Falls die Suche erfolgreich war, gibt die *resolve*-Methode Methode *true* zurück. In diesem Fall benachrichtigt der Knoten außerdem den Superknoten im Baum und setzt das Attribut *open* auf *false*. Der Superknoten kann dann selbst wieder bestim-

men, ob er noch offene Elemente als Subknoten enthält oder selbst offene Verweise besitzt, und kann seinerseits *open* entsprechend setzen (anfangs steht *open* im Normalfall auf *true*).

Die Suche wird dabei durch die Methode *search()* gesteuert. Das erste Argument, der zu findende Name, steuert dabei die Suchrichtung. Ausgehend vom Startknoten wird zuerst im eigenen Kontext gesucht, bei Mißerfolg jeweils im darüberliegenden Kontext. Durch die Proxies können auch importierte Kontexte berücksichtigt werden, wobei dann das letzte Argument der Methode als *true* weitergegeben wird, um anzuzeigen, daß innerhalb importierter Kontexte die Suche nur noch nach unten durchgeführt werden darf. Ist der zu suchende Bezeichner ein zusammengesetzter, wird zuerst der erste Teil verglichen und bei Erfolg im passenden Knoten der Restbezeichner wiederum abwärts gesucht.

Die beiden anderen Argumente dienen der Berücksichtigung der Zugriffsrechte. Beide speichern den bisherigen Suchpfad als String, wobei *origin* nur die Rollen der Elemente speichert, während *originName* die Namen der bisher durchsuchten Elemente hält. Auf diese Weise kann z. B. eine Methode, die als `protected` eingetragen wurde, entscheiden, ob der Suchaufruf von einer Klasse weitergegeben wurde, die eine Subklasse ihrer Klasse ist. Ist der Suchpfad über eine Superklassenbeziehung gegangen, kann die Methode den Zugriff zulassen, ansonsten sofort *NULL* zurückgeben und damit die Suche in dieser Richtung abbrechen. Die Bezeichner aus *originName* werden vor allem für Eiffel benötigt, wo der Export von Methoden an explizite Klassen erfolgt. Abbildung 4.3 zeigt das Beispiel eines Suchverlaufs. Der Baum ist bereits teilweise expandiert, und für die Methode *g* (das am weitesten rechts liegende Blatt) soll die Klasse *E* gesucht werden. Die Pfeile zeigen die Aufruf-Richtung an, die Nummern die einzelnen Schritte:

1. Innerhalb des Knotens, der die Methode repräsentiert, wird die Suche mit dem Aufruf `search("E", "method", "g", false)` angestoßen. Aus dem Aufruf läßt sich eindeutig erkennen, daß er von der Methode *g* gekommen ist.

2. Der Knoten für die Klasse *G* führt die Suche weiter nach oben fort, da offensichtlich keine Methode *E* existiert. Der Aufruf wird erweitert und als `search("E", "class/method", "G/g", false)` an den Dateiknoten weitergegeben.

3. Vom Dateiknoten kann die Suche nur nach unten gegeben werden. Der Knoten für die Klasse *H* gibt *NULL* zurück, weil der Bezeichner nicht übereinstimmt. Da der Aufruf von *G* kommt, kann durch Überprüfung des bisherigen Pfades vermieden werden, daß ein Zyklus bei der Suche entsteht. Der Aufruf wird hier in der Form `search("E", "file/class/method", "B/G/g", false)` an den Proxy für Modul *C* weitergegeben. Dieser fügt dem Suchpfad einen String zur Kennzeichnung hinzu und setzt das letzte Argument auf *true*, so daß der Aufruf jetzt mit Anhang als `search("E", "import/file/class/method", "B/G/g", false)` weitergereicht wird.

4. Der Modulknoten reicht den Aufruf an die Knoten für die Klassen *D* und *E* weiter, und zwar durch `search("E", "module/import/file/class/method", "C/B/G/g", false)`. Die Suche über *D* ist erfolglos, während *E* der geeignete Kandidat ist. Zwei Fälle sollen demonstrieren, wie der Knoten entscheiden kann, ob der Zugriff erlaubt ist:

   - Hat der Knoten das Zugriffsrecht *public*, wird der Zugriff ohne Einschränkung zugelassen, und die Suche ist erfolgreich. Der Knoten gibt seine Referenz an den Methodenknoten *g* zurück.

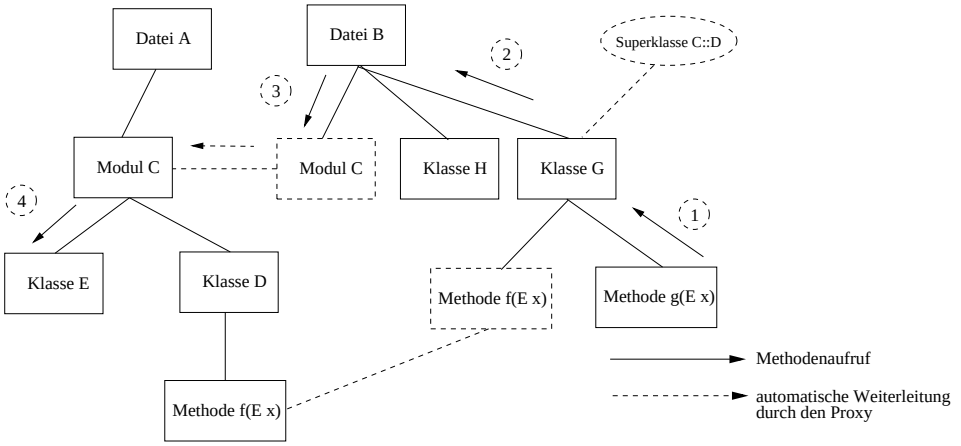

Abbildung 4.3: Suchverlauf beim Auflösen von Verweisen.

- Ist der Zugriff dagegegen nur dem Modul zugehörigen Klassen erlaubt, kann *E* durch zwei Überprüfungen festgestellen, ob der Zugriff erlaubt ist oder nicht: Zum einen muß für eine Berechtigung der Name des übergeordneten Moduls im Pfad enthalten sein, gleichzeitig darf das Modul aber nicht importiert worden sein, was hier am enthaltenen Teilstring *module/import* erkennbar ist. In diesem Fall gibt *E* ebenfalls *NULL* zurück und die Suche scheitert.

Analog kann die Suche für das Argument *E* des Methodenknotens *f* durchgeführt werden (der Algorithmus gibt aufgelöste Verweise nicht an andere Knoten mit dem gleichen Verweis weiter). Da bei der Suche nicht der Typ des Knotens mitgegeben wird, muß bei einem passenden Kandidaten falschen Typs die unterbrochene Suche fortgesetzt werden. Zu diesem Zweck werden das Attribut *marked* und die Methoden *getMarkColor()* bzw. *unmark()* in der Knotenklasse gebraucht. Mittels einer entsprechenden Codierung werden durchsuchte Knoten als schwarz (= Unterbäume und Superknoten vollständig durchsucht) oder grau (= einige verbundene Knoten sind noch offen) gekennzeichnet. Auf diese Weise kann entschieden werden, welche Knoten noch durchsucht werden müssen. Mittels *unmark()* können alle Knoten nach durchgeführter Suche wieder auf weiß (= nicht durchsucht) gesetzt werden.

Die Abbildungen D.11 bis D.13 im Anhang zeigen beispielhaft den Ablauf für einige Operationen. In Abbildung D.11 wird das Auflösen der Module dargestellt, wobei auch die nachträgliche Vergabe der IDs miteinbezogen wird. Abbildung D.12 demonstriert die Auflösung von Modulimporten und zeigt, wie ein Import durch einen passenden Proxy ersetzt wird. Ein ähnliches Verfahren kommt auch beim Auflösen von Dateiimporten zur Anwendung, allerdings wird hier Rücksicht auf die Option `IncludeTransitive` genommen: Ist diese Option auf `yes` gesetzt, wird vor dem Kopieren der inkludierten Knoten geprüft, ob diese schon vorhanden sind (die genaue Beschreibung kann in der Tabelle in A nachgesehen werden). Schließlich zeigt Abbildung D.13 das Erzeugen von Fehlermeldungen aus unaufgelösten Symbolen.

Leider bringt die Auflösung auch technischen Aufwand für den Benutzer mit sich, und zwar im Hinblick auf Mehrfachvererbung. Beim Einfügen von Methodenknoten an einem Subklassen-Knoten wird überprüft, ob eine entsprechende Methode bereits existiert (in Bezug auf den Namen und deren Argumente). In diesem Fall geht die Schnittstelle davon aus, daß die Methode in der Subklasse überschrieben wird und erzeugt für die entsprechen-

de Methode keine Kopie der Superklassenmethode. Dieser Ansatz ist kompatibel zu der Interface-Implementierung von Java (Interfaces werden dann als abstrakte Superklassen gesehen), die Mehrfachvererbung ersetzt. Bei Eiffel wird bei Namenskollision eine Umbenennung vorgenommen. Zu diesem Zweck existiert die Klasse *MethodAliasNode*, die diese Umbenennung speichert. Später wird ein Objekt dieser Klasse durch einen Methoden-Proxy ersetzt, der bei Zugriff auf den Namen allerdings den neuen Bezeichner zurückgibt. Für die Mehrfachvererbung, die C++ definiert, gibt es in dieser Hinsicht keine passende Lösung. Zudem bietet CORBA auch keine Möglichkeit an, bei Namenskollision in C++ gezielt die Variante der Methode anzusprechen, die in einer bestimmten Superklasse liegt. Aus diesem Grund muß der Umbenennungsmechanismus von Eiffel in C++ auf folgende Weise nachgebildet werden:

```
class A {
public:
  A();
  int attrib;
  void method();
};

class B {
public:
  B();
  int attrib;
  virtual void method();
};

//Hilfsklasse

class C : public B {
public:
  C();
  int& attrib_b;
  void method_b();
};

// Erbende Klasse (in der normalerweise Namenskollision auftritt)

class D : public C, public A {
public:
  D();
  int& attrib_a;
  virtual void print();
  int getX();
  int getX_b();
};

...

void A::method() {...}

void B::print() {...}

C::C() : x_b(B::x) {}

void C::method_b() {
  B::method();
}

D::D() : C(), x(A::x) {
}
```

```
void D::method_a() {
  A::method();
}
```

Zusätzlich muß der Benutzer bei der Instrumentierung bestimmte Angaben machen, damit die Überdeckung aller Methoden gemessen werden kann. Diese Problematik wird daher im Kapitel zur Instrumentierung noch einmal angesprochen.

Kapitel 5

Generierung der IDL-Dateien

5.1 Anforderungen der IDL und generierte Hilfsklassen

Die von der Schnittstelle generierten IDL-Dateien bilden die Grundlage für eine Anbindung an CORBA. Mittels Aufruf des IDL-Compilers für die passende Sprache werden die Klassen erzeugt, die später den Austausch zwischen der Schnittstelle und der zu testenden Software handhaben[1]. Die hier beschriebenen Regelungen beziehen sich auf die von OMG beschriebenen Mappings ([OMG02b], [OMG03]). Einzige Ausnahme ist das Mapping für Eiffel: Da OMG bisher noch keinen verbindlichen Standard definiert hat, wird an dieser Stelle das vom mico/E-Team definierte Mapping als Ausgangsbasis verwendet ([Swi02]). Dieses Mapping hat jedoch keine Allgemeingültigkeit und kann daher bei anderen ORBs als dem mico/E-ORB zu Anpassungsschwierigkeiten führen.

Damit die Notwendigkeit einiger IDL-Deklarierungen deutlich wird, ist eine Betrachtung der Architektur notwendig, die zur Laufzeit den Austausch zwischen Schnittstelle und der zu testenden Software ermöglicht. Diese Architektur ist in Abbildung 5.1 dargestellt.

Um eine Durchführung von Testfällen zu ermöglichen, wird auf Seite der zu testenden Software (in der Abbildung der mit *instrumentierte Klassen* gekennzeichnete Bereich) mit einer zusätzlichen Klasse, dem Objekt-Manager, ausgestattet. Die instrumentierten Klassen übernehmen dabei größtenteils die Rolle des Clients. Der Objekt-Manager unterstützt die Erzeugung von Objekten auf Anfrage oder gibt bestimmte Ereignisse über am Testlauf beteiligte Objekte bekannt. Gleichzeitig wird zu jeder instrumentierten Klasse eine Wrapper-Klasse geschaffen, die die Kommunikation über die ORBs ermöglicht. Jedes Wrapperobjekt hält dabei eine Referenz auf das eigentliche Objekt, das getestet werden soll. Auf Anforderung erzeugt der Objekt-Manager zu jedem existierenden Objekt ein passendes Wrapper-Objekt, das die Methodenaufrufe an die instrumentierte Klasse weiterleitet. Auf Seite der Schnittstelle hat der Benutzer die Möglichkeit, den Ablauf zu steuern (eine Beschreibung, wie dies möglich ist, erfolgt in Abschnitt 6.3). Die Schnittstelle bietet zur Überwachung des Ablaufs außerdem eine Klasse mit der Bezeichnung *EventSink* an, die von den instrumentierten Objekten während eines Testlaufs Meldungen über bestimmte Ereignisse (z. B. das Erreichen bestimmter Punkte im Quellcode) entgegennimmt. Diese Meldungen werden über den Objekt-Manager nach außen gegeben, der auch eigene Ereignisse an die Schnittstelle meldet, wie z. B. die Zerstörung eines Objekts. Die Steuerung des Benutzers hat während des Testlaufs die Möglichkeit, auf diese Ereignisse zu reagieren und entsprechende Maßnahmen zu ergreifen.

[1] Tatsächlich werden nur die Skeleton-Klassen benötigt – der Grund wird in Abschnitt 6.3 erläutert.

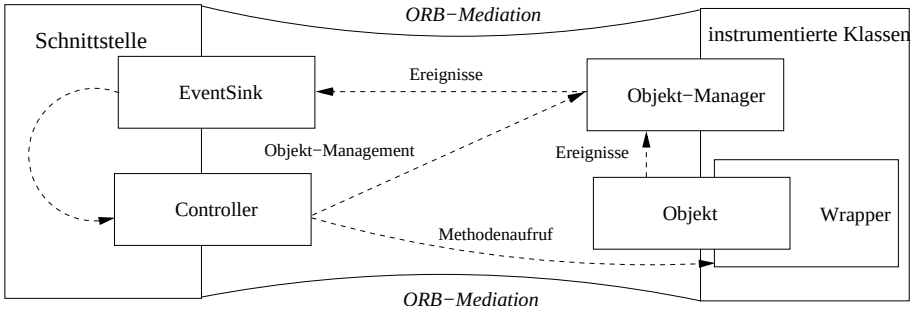

Abbildung 5.1: Darstellung der Laufzeitarchitektur.

In Abbildung D.16 im Anhang ist die konkrete Klassenstruktur zu sehen. Alle kursiv geschriebenen Bezeichner stehen dabei nicht für abstrakte Elemente, sondern drücken eine Parametrisierung aus. Innerhalb dieser kursiven Ausdrücke steht *T* bzw. *t* für die jeweils instrumentierte Klasse. Wird also die Klasse *Figure* instrumentiert, steht *tList* für *figureList* und *TServant* für *FigureServant*. Aktueller Wert für diesen Parameter ist jeder Klassenbezeichner, der beim Einlesen ermittelt wurde.

Für jede instrumentierte Klasse *T* wird eine Wrapperklasse mit der Bezeichnung *TServant* geschaffen. Dieser Wrapper fungiert als Proxy und bietet nach außen dieselben Methoden an wie die instrumentierte Klasse. Er sorgt außerdem für die Wandlung einzelner Methodenargumente vom CORBA-eigenen Format des Mappings in das sprachspezifische und die umgekehrte Wandlung bei Rückgabewerten. Darüber hinaus erhält jede instrumentierte Klasse ein zusätzliches Attribut, das eine eindeutige Laufzeit-ID hält. Über diese ID sind später einzelne Objekte identifizierbar. Der Objekt-Manager (*GlobalObjectAdmin*) verwaltet dabei für jede instrumentierte Klasse eine eine Liste von aktiven Wrapper-Objekten. Damit diese Verwaltung nicht von sprachspezifischen Bibliotheksklassen abhängig ist, werden zusätzlich zwei Hilfsklassen *TNode* und *TList* geschaffen, die eine einfach verkettete Liste repräsentieren. Innerhalb dieser Liste werden die einzelnen Servants gehalten.

Nach außen bietet der Objekt-Manager vier Methoden an, mit denen Objekte erzeugt sowie Objektreferenzen geholt oder freigegeben werden können. Die Identifizierung der gewünschten Klasse bzw. des Objekts erfolgt dabei über die beim Einlesen der Dateien vergebenen IDs bzw. die Laufzeit-ID eines Objekts. Die Freigabe mittels *releaseObject()* ist notwendig, damit bei Java und Eiffel der Garbage Collector nicht mehr benötigte Objekte aufsammeln kann, die ansonsten auf Seite der zu testenden Software gehalten werden würden. Eine Besonderheit stellen die Methoden *createNilRef()* und *castObject()* dar. Die erste der beiden Methoden erzeugt `null`-Referenzen zu einem jeweiligen Klassentyp und gibt sie zurück, die zweite ist zum Duchführen eines Casts eines Objekts während des Test zuständig. Übergeben wird dabei die Laufzeit-ID des Objekts und die ID der gewünschten direkten Superklasse. Als Ergebnis erhält der Benutzer die neue Objektreferenz und eine neue ID. Die neue Referenz muß ebenfalls mit *releaseId()* freigegeben werden, damit das Objekt später wieder gelöscht werden kann. Um auf eine Superklasse einer Superklasse zu kommen, ist ein zweifacher Cast erfoderlich, ein Downcast ist nicht möglich. Die Methode dient der Unterstützung von abstrakten Klassen (die nicht direkt erzeugt werden können) und der Durchführung des *Polymorphic Server Test* bzw. des *Polymorphic Message Test*. Schließlich kann die Methode *shutdown()* zum Beenden des ORBs nach Abschluß des Testlaufs verwendet werden. Alle internen Methoden dienen entweder dazu, Ereignisse an

die Klasse *EventSink* zu melden, die innerhalb der Schnittstelle dieselben Methoden besitzt (*reachedPoint()*, *objectDeleted()*, *objectChanged()* und *objectCreated()*) oder sie sind für die interne Verwaltung der Objekte zur Laufzeit gedacht. Die Klasse *AdminAccessor* dient dem Zugriff auf den Objekt-Manager und ist aus Gründen der Einheitlichkeit notwendig, weil der Objekt-Manager als Singleton implementiert wird und in Eiffel durch eine Hilfsklasse der Zugriff vorgesehen werden muß.

Die Klasse *EventSink*, die auf der Seite der Schnittstelle arbeitet, hat entsprechende Methoden zur Behandlung der auftretenden Ereignisse:

- *reachedPoint()*, *objectCreated()*, *objectChanged()* und *objectDeleted()* entsprechen den Methoden des Objekt-Managers.

- Zusätzlich besitzt die Klasse eine Methode *initDone()*, mit deren Aufruf gemeldet wird, daß der ORB auf der Seite der zu testenden Software jetzt aktiv ist und sich im Naming Service angemeldet hat. Analog gibt es eine Methode *shutdown()*, mit der beide ORBs wieder heruntergefahren werden können.

- Für die Benutzung während des Testlaufs besitzt die Klasse zwei weitere Methoden, *attachObserver()* und *detachObserver()*, mit denen Objekte einer Klasse *Observer* eingetragen werden können. Diese Objekt übernehmen die eigentliche Ereignisbehandlung und müssen daher für alle Ereignisse Methoden mit gleichem Namen wie dei Klasse *EventSink* besitzen. Die Klasse *EventSink* dient also nur der Weiterleitung der Ereignismeldungen.

Um die Kommunikation zwischen den einzelnen Objekten anzudeuten, zeigt Abbildung D.17 im Anhang beispielhaft die Kommunikation für die Methoden *createObject()* und *castObject()* des Objekt-Managers. Die Objekttypen beziehen sich dabei auf das bislang verwendete Figurenbeispiel, wobei die Suche nach der übergebenen ID innerhalb aller Listen durch *findServant()* weggelassen wurde. Durch Aufruf dieser Methode auf allen Listen läßt sich die korrekte Liste feststellen, die bearbeitet werden muß.

Zusätzlich zum bisher beschriebenen Verhalten (das im weiteren als *aktiver Modus* bezeichnet werden soll) bietet die Schnittstelle eine Möglichkeit, Applikationstests durchzuführen (in diesem Fall befindet sie sich im *passiven Modus*). Die in Abbildung D.16 im Anhang gezeigten Klassen bleiben dabei größtenteils bestehen, es ändern sich lediglich die Rollen von Originalobjekt und Servant. Der Servant dient jetzt als Memento, das den aktuellen Zustand des Objekts hält, wobei jetzt das Objekt eine Referenz auf das Memento besitzt. Der Benutzer der Schnittstelle kann jetzt bei Benachrichtigung durch den Objekt-Manager den Zustand einzelner Attribute abfragen. Abbildung 5.2 zeigt für diesen Modus die Laufzeitarchitektur.

Beide der bisher erwähnten Modi erfordern vom Benutzer der Schnittstelle eine korrekte Handhabung, damit ein Testlauf gültige Ergebnisse erzielt. Aus diesem Grund sollen einige Aspekte genauer betrachtet werden.

- **Zustandsuntersuchung der zu testenden Software:** Im aktiven Modus bezeichnet der Begriff Zustand den Zustand aller Objekte, die über die Methode *createObject()* des Objekt-Managers erzeugt worden sind, sowie alle Objekte, auf die von diesen Objekten durch Attribute verwiesen wird. Der Benutzer hat hier nur die Möglichkeit, als Ausgangspunkt einer Zustandsuntersuchung die über *createObject()* erzeugten Objekte anzusprechen, und deren Attributreferenzen zu verfolgen, um den Zustand eines bestimmten Objekts zu ermitteln; die Methode *getObject()* sollte nur in Ausnahmefällen benutzt werden. Bei Benutzung von *getObject()* muß zum Ende der

Zustandsuntersuchung in jedem Fall *releaseObject()* aufgerufen werden, damit keine Speicherlecks entstehen. Dies entspricht allerdings eher einer Debugging-Strategie und sollte in der Praxis daher nur selten nötig sein.

Im passiven Modus ist der Zustand nur nach Verlassen einer Methode definiert, Zwischenzustände können nicht abgefragt werden, da die Aktualisierung eines Objekts nur beim Austritt aus der Methode erfolgt. Darüber hinaus sollte eine Änderung von Attributen eines Objekts nur innerhalb der objekteigenen Methoden erfolgen, weil die Schnittstelle die Änderung bei direktem Zugriff von außen nicht registriert. Da von vorneherein nicht durch die Schnittstelle entschieden werden kann, welche Objekte den Zustand der Software definieren, kann der Benutzer hier mit *getObject()* auf beliebige Objekte (bzw. deren Mementos) zugreifen.

- **öffentlicher Zugriff:** Die Schnittstelle sieht für die Instrumentierung alle Methoden als öffentlich an, die nicht privat sind (oder im Falle von Eiffel nicht mit `{NONE}` deklariert worden sind). Um den Zugriff auf die Methoden zu gewährleisten, werden für den aktiven Modus die Zugriffsrechte der betroffenen Methoden passend geändert.

- **Beteiligte Klassen:** Da für die an einem Testlauf beteiligten Klassen der Quellcode zur Verfügung stehen kann oder nicht, sollen im weiteren zwei Kategorien von Klassen unterschieden werden:

  – *reguläre Klassen* sind Klassen, für die der Quellcode zur Verfügung gestellt worden ist. Für diese Klassen können zur Laufzeit alle öffentlich zugreifbaren Methoden (im Sinne des oben genannten öffentlichen Zugriffs) aufgerufen und eine Instrumentierung zur Überdeckungsmessung durchgeführt werden. Für diese Klassen werden die bereits beschriebenen Hilfsklassen erzeugt, unabhängig davon, ob der Benutzer eine Instrumentierung zur Messung einer Codeüberdeckung anfordert.

  – *externe Klassen* sind alle Klassen, deren Quellcode nicht zur Verfügung steht, die aber trotzdem an einem Test beteiligt sind (z. B. Bibliotheksklassen). Für diese Klassen kann der Benutzer einen eigenen Wrapper erzeugen (das Verfahren wird im nächsten Kapitel beschrieben). Der Wrapper und die passenden Listenklassen sind in Abbildung D.16 im Anhang mit *XServant*, *XServantList* und *XServantNode* bezeichnet worden. Dieser Wrapper funktioniert genauso wie alle anderen Wrapper auch, allerdings mit einigen Einschränkungen, die ebenfalls im nächsten Kapitel beschrieben werden.

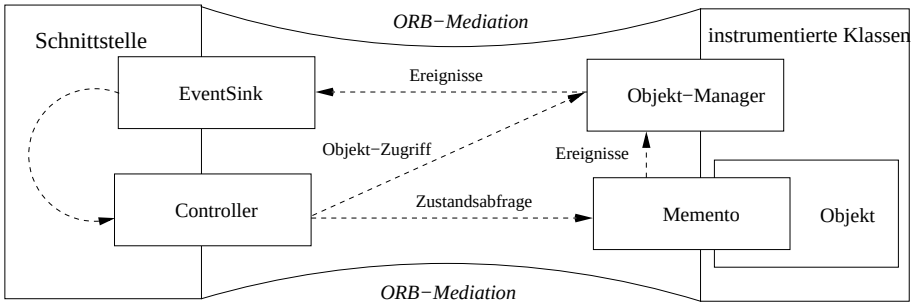

Abbildung 5.2: Laufzeitarchitektur des passiven Modus.

Da alle an einem Testlauf beteiligten Klassen innerhalb einer IDL-Datei deklariert werden müssen, ist es Aufgabe der Schnittstelle, für jede reguläre Klasse eine Wrapper-Deklaration in IDL sowie die Deklarationen für die Klassen *EventSink* als auch für den Objekt-Manager bereitzustellen. Um ein konkretes Beispiel einer IDL-Deklaration zu betrachten, wird Bezug auf das bisher verwendete Figuren-Beispiel genommen. Ein Umsetzung der Klassen *Figure* und *Triangle* in IDL hätte folgendes Aussehen:

```
interface FigureServant {
  void move(in double x, in double y);
  double getArea();
  void movePoint(in long nr, in double dx, in double dy);
};

interface TriangleSevant : FigureServant {};
```

An diesem Beispiel wird deutlich, daß abstrakten Klassen ebenfalls Interfaces zugeordnet werden. Private Methoden (wie z. B. *inLine()*) werden nicht im Interface aufgeführt. Diese Festlegung entspricht der oben genannten Definition für öffentlichen Zugriff. Die Schnittstelle bietet jedoch die Möglichkeit, diese Methoden zur Messung der Codeüberdeckung zu instrumentieren und dann mittels *castObject()* für die Testauswertung mit einzubeziehen.

Innerhalb der IDL-Notation werden muß auch beachtet werden, daß Argumente von Operationen als Ein-/Ausgabeparameter (`inout`), Ausgabeparameter (`out`) oder Eingabeparameter (`in`) festgelegt werden können. Für die Schnittstelle wird eine einfache Regelung getroffen: Basistypen werden als Eingabeparameter deklariert, sofern es sich nicht um ein Array oder eine C++-Referenz bzw. einen C++-Pointer auf diesen Typen handelt, Objekttypen werden immer als Ein-/Ausgabeparameter deklariert. Dies erlaubt es dem Benutzer, in Testaufrufen übergebene Objekte auf Veränderungen nach der Rückkehr zu überprüfen. CORBA unterstützt diesen Ansatz für die drei Sprachen, indem die spätere Klasse für Ein-/Ausgabetypen Zugriff auf den alten und den neuen Wert nach Beenden des Aufrufs ermöglicht. Die Regelung ist bewußt einfach gehalten und verzichtet darauf, die Klassenrepräsentation der Eiffel-Basistypen zu berücksichtigen.

Zur Umsetzung der sprachspezifischen Basistypen innerhalb einer IDL-Datei muß die Schnittstelle das spezifische Mapping zu einer Sprache kennen. Um dieses Problem zu lösen, wird die bereits angesprochene Typendatei verwendet. Innerhalb dieser Datei gibt der Benutzer in Tabellenform die Beziehung von IDL- und Sprachtypen an. Die Datei besteht dabei aus drei Spalten, in denen der Benutzer folgende Angaben macht:

- **erste Spalte**: Hier steht der sprachspezifische Typ, wie er innerhalb der Sprache verwendet wird.
- **zweite Spalte**: An dieser Stelle wird die Schreibweise des entsprechenden IDL-Typen angegeben. Über diese Spalte kann die Schnittstelle erkennen, welche ID während des Einlesens für den Typen vergeben wird.
- **dritte Spalte**: In dieser Spalte steht der sprachspezifische Ausdruck für den in der zweiten Spalte angegebenen IDL-Typen. Diese Spalte benutzt die Schnittstelle zur Instrumentierung.

Diese Informationen werden vor Beginn des Einlesevorgangs in die Klasse *TypePool* eingelesen und stehen damit der Klasse *TypeItem* zur Verfügung. Die folgenden drei Beispiele zeigen das Mapping des `double`-Typen für Java, Eiffel und C++.

- **Java**: `"double" "double" "double"`
- **Eiffel**: `"DOUBLE" "double" "DOUBLE"`
- **C++**: `"double" "double" "CORBA::Double"`

Speziell für C++ existieren in der Optionsdatei die Optionen `PointerPostfix` und `ReferencePostfix`, die es erlauben, Pointer- und Referenztypen zu identifizieren. Diese Informationen werden später für die Instrumentierung benötigt, um die Typen korrekt anzusprechen und z. B. in Zuweisungen einzusetzen. Generische Typen werden nicht unterstützt und müssen zum Test in einer Hilfsklasse gekapselt werden, die passende Zugriffsmethoden bereitstellt. Arrays werden nur in eindimensionaler Form unterstützt, um den Aufwand für die Schnittstelle in Grenzen zu halten. Da in Eiffel ein Array eine generische Klasse ist und in Java und C++ eckige Klammern ein Array kennzeichnen, unterstützt die Schnittstelle die Identifikation von Arrays auf zwei Arten: Zum einen durch die Option `ArrayPostfix`, mittels derer Arrays in C++ und Java erkannt werden, zum anderen kann der Benutzer in der Typendatei einen Eintrag `"ARRAY[type]"` in der ersten Spalte anlegen. Das Schlüsselwort `type` wird dabei durch die Klasse *TypeItem* berücksichtigt und führt zu einem Matchversuch innerhalb der Klammern. Existiert kein passender Basistyp, der innerhalb der Klammern gematcht werden kann, wird davon ausgegangen, daß es sich bei dem Bezeichner um eine Klasse handelt. Die Klasse *TypeAssignment* unterstützt den Eintrag eines Arrays durch geeignete Attribute (die Einordnung beider Klassen ist in Abbildung D.1 im Anhang zu sehen). In IDL werden Arraytypen durch eine Sequenz ausgedrückt, um dynamische Arrays ebenfalls zu erfassen. Arrays sind zudem der einzige Datentyp, der nicht auf der Seite der zu testenden Software, sondern auf Seite der Schnittstelle erzeugt werden muß (s. Beispiel in der Zusammenfassung auf S. 97). Da in C++ Arrays keine automatische Längenüberprüfung besitzen, muß das Array in einer Klasse gekapselt werden, die die Längenüberprüfung unterstützt. Dieser Aspekt wird im nächsten Kapitel genauer beschrieben.

Eine Reihe spezieller Regelungen der Interface Definition Language sind aus dem verwendeten Beispiel allerdings nicht ersichtlich. CORBA unterstützt Groß- und Kleinschreibung mittels einer speziellen Regelung: Bezeichner mit gleichen Buchstaben, aber verschiedener Groß- und Kleinschreibung werden als gleich angesehen. Darüber hinaus muß ein Bezeichner über die komplette IDL-Datei in der zuerst verwendeten Schreibweise aufgeführt werden und darf innerhalb eines Gültigkeitsbreichs nur einmal zur Deklaration benutzt werden. Kollisionen mit IDL-Schlüsselwörtern (z. B. `interface`) können durch Vorstellen eines Unterstriches aufgehoben werden. Diese Festlegung hat folgende Konsequenzen:

- Bei Schachtelung von Modulen müssen die Bezeichner über alle Module bis zum innersten unterschiedlich sein. In der Praxis tritt eine gleichartige Benennung geschachtelter Module allerdings eher selten auf.
- CORBA unterstützt das Überladen von Methoden nicht. Aus diesem Grund nimmt die Schnittstelle bei *einer* der betroffenen Methode eine Umbenennung vor und merkt sich den Originalnamen dieser Methode in einer Tabelle. Die aus dieser Umbenennung entstandene Methode im Servant wird dann wieder an die ursprüngliche Methode der instrumentierten Klasse gebunden. Um die Namen eindeutig zu halten, wird dem ursprünglichen Methoden-/Operationsnamen die ID angehängt, die beim Einlesen vergeben wurde.

Zusätzlich ist unter CORBA eine Schachtelung von Interfaces nicht erlaubt. Dies bedeutet, daß innere Klassen, wie sie aus Java oder C++ bekannt sind, nicht durch die Schnittstelle unterstützt werden.

Werden Bezeichner von Interfaces oder Modulen als Typangabe für ein Methodenargument oder ein Attribut verwendet, müssen sie vorher deklariert worden sein. Bei zyklischen Bezügen ist daher analog zu C++ eine Forward Declaration notwendig. Die Generierung der IDL sollte eine geschickte Deklarationsreihenfolge festlegen, um möglichst viele Forward Declarations zu vermeiden.

Im Hinblick auf die Mehrfachvererbung (s. Abschnitt 3.2) läßt sich die im letzten Abschnitt getroffene Regelung sehr einfach auf die IDL abbilden: Umbenannte Methoden werden im erbenden Interface einfach neu deklariert, die andere, nicht umbenannte Variante kann daher eindeutig einem vererbenden Interface zugeordnet werden. Ist zum Beispiel *C* eine von *A* und *B* erbende Klasse und *f()* eine in *A* und *B* deklarierte Methode, die bei der Vererbung von *A* in *g()* umbenannt wurde, sieht die IDL-Deklaration folgendermaßen aus:

```
interface A {
  void f();
}

interface B {
  void f();
}

interface C : A, B {
  void g();
  B::f();
}
```

Für Attribute bietet CORBA zwar Unterstützung und generiert sogar automatisch Zugriffsmethoden, jedoch ist deren Schreibweise für verschiedene Mappings unterschiedlich. Aus diesem Grund wird für die Schnittstelle eine pragmatischere Festlegung getroffen: Eine lesende Operation für ein Attribut *attr* wird *getAttr()* genannt, eine schreibende *setAttr()*. Existiert bereits eine der beiden zugehörigen Methoden in der ursprünglichen Klasse, geht die Schnittstelle davon aus, daß diese Methode den Zugriff entsprechend unterstützt und generiert keine eigene Variante.

Abschließend sollen einige technische Aspekte erwähnt werden, die die Kompilierung der IDL-Dateien mit sich bringt. Die zu einer IDL-Datei erzeugten Quellcode-Dateien sind je nach Mapping in verschiedenen Verzeichnissen zu finden oder unterscheiden sich in der Anzahl für die verschiedenen Sprachen. Trotzdem muß für die Kompilierung des gesamten instrumentierten Codes bekannt sein, wo diese Dateien liegen und welche Dateien neu hinzugekommen sind. Die drei Sprachen treffen hier für die Festlegung einer zu einem Softwarepaket gehörenden Dateiensammlung unterschiedliche Maßnahmen:

- C++ verwendet `include`-Anweisungen für den Präprozessor, um die Abhängigkeiten zwischen einzelnen Dateien festzulegen. Damit die Dateien gefunden werden, kann dem Compiler über eine Kommandozeilenoption mitgeteilt werden, innerhalb welcher Verzeichnisse nach einer inkludierten Datei gesucht werden soll. Eine Erweiterung um die betreffenden Dateien muß also zusätzliche `include`-Anweisungen vorsehen und unter Umständen den Suchpfad beim Aufruf des Compilers erweitern.

- Java besitzt ebenfalls eine Kommandozeilenoption zur Festlegung des Suchpfades. Zusätzlich bewirkt ein Import eines Packages ein Durchsuchen eines Unterverzeichnisses aus dem aktuellen Verzeichnis der Datei. Analog zu C++ muß also auch hier der Suchpfad erweitert werden. Alle Dateien, die im gleichen Verzeichnis wie die aktuell kompilierte Datei liegen, werden automatisch berücksichtigt.

- Bei Eiffel existiert eine dritte Variante: Der SmartEiffel-Compiler erhält zur Festlegung des Suchpfades entweder eine Datei mit der Endung `.ace`, die die Dateien innerhalb sogenannter Cluster organisiert. Dabei können Dateien innerhalb eines Clusters auch aus verschiedenen Verzeichnissen stammen. Alternativ wird eine spezielle Datei `loadpath.se` innerhalb eines Verzeichnisses abgelegt, die auf weitere Verzeichnisse verweist. Im zweiten Fall kann innerhalb der Vezeichnisse, auf die verwiesen wird, erneut eine `loadpath.se` abgelegt werden, so daß der Mechanismus auch transitiv funktioniert. Bei Eiffel ist also entweder ein Eintrag zusätzlicher Dateien innerhalb eines Clusters oder eine Erweiterung der eingetragenen Pfade notwendig.

Um die Änderungen in passender Weise durchzuführen, ist eine Unterstützung der Schnittstelle durch den Benutzer notwendig. Durch Implementieren einer abstrakten Klasse kann der Benutzer steuern, auf welche Weise Dateien in ein Softwarepaket integriert werden. Zum anderen kann diese abstrakte Klasse bestimmen, welche Dateien aus einer IDL-Datei erzeugt worden sind, und diese an die Schnittstelle zurückgeben. Da die Bindung an die Schnittstelle zur Laufzeit geändert werden kann, ist eine Implementierung der abstrakten Klasse durch den Benutzer für jede der verwendeten Sprachen notwendig, um die benötigte Flexibilität zu erreichen.

Zusätzlich sind bei Java die Zugriffsrechte für Klassen innerhalb einer Datei oder eines Packages wichtig: Soll eine Klasse auch außerhalb einer Datei oder eines Packages verfügbar sein, muß sie innerhalb einer Datei mit gleichem Namen aufgeführt werden. Würde man also eine Klasse innerhalb eines Packages öffentlich machen, müßte man diese Klasse in eine eigene Datei kopieren. Um diesen Aufwand zu vermeiden, wird die Veröffentlichung auf andere Weise erreicht: Der Servant wird innerhalb des Packages bzw. Moduls deklariert, in dem die zugehörige Klasse steht, und erhält öffentliche Zugriffsrechte. Auf diese Weise ist nur für den Servant eine eigene Datei notwendig.

Im folgenden Abschnitt soll gezeigt werden, wie die getroffenen Regelungen durch eine entsprechende Klassenstruktur unterstützt werden.

5.2 Schnittstellenklassen zur Unterstützung der Generierung

Nach Einlesen der Quellcode-Dateien hat der Benutzer der Schnittstelle die Möglichkeit, Fehlermeldungen über fehlende Informationen auszulesen und externe Klassen zu ergänzen, die für den Test benötigt werden. Eine Ergänzung fehlender Quellcode-Dateien ist nicht ohne weiteres möglich, in diesem Fall muß der Einlesevorgang erneut durchgeführt werden.

Nach Festlegung dieser Kritierien startet der Benutzer durch Methodenaufruf die eigentliche IDL-Generierung. Diese Generierung wird in drei Schritten durchgeführt:

- Zuerst werden die bisher eingelesenen Informationen aufbereitet und für die IDL-Generierung teilweise in Tabellen eingetragen. Dies betrifft das Vergeben neuer Na-

men für überladene Methoden und die Behandlung von Mehrfachvererbung mit Namenskollision und das Einrichten von *get*- und *set*-Methoden.

- Anschließend werden mit einer Hilfsstruktur die bisher eingelesene Information sortiert, um zu erreichen, daß innerhalb der IDL-Datei möglichst wenige Vorwärtsdeklarationen benötigt werden.

- Nun wird aus den gewonnenen Informationen die IDL-Datei generiert und als Vorbereitung für die Instrumentierung werden die zugehörigen Quellcode-Dateien bestimmt. Dies geschieht über die vom Benutzer bereitgestellten Klassen, die auch dafür sorgen, daß die neuen Dateien später bei der Kompilierung integriert werden.

Der Benutzer kann jetzt die Optionen für den IDL-Compiler, den spracheigenen Compiler und Linker sowie Laufzeitargumente für den Start der instrumentierten Software festlegen. Für Compiler und Linker ist das noch nicht notwendig, allerdings müssen diese Optionen spätestens vor Aufruf der Instrumentierung festgelegt werden.

Abbildung D.14 im Anhang illustriert die Klassen, die zur Vorbereitung der IDL-Generierung benötigt werden. Der erste Schritt der Informationsgewinnung besteht darin, mittels der Klassen *OverloadingSearcher* und *NameCollisionSearcher*, die aufgebauten Bäume auf überladene Methoden (d. h. gleiche Methodenknoten an einem Klassenknoten) und Namenskollision bei Mehrfachvererbung zu untersuchen. Die Klasse *NameCollisionSearcher* vergleicht dabei die Superklassen einer Klasse auf gleiche Methodennamen (über das Attribut *superClasses* des aktuellen Knotens), um entsprechende Kollisionen zu finden. Eine bei Überladung umbenannte Methode wird als Alias in die Klasse *IdentifierTable* eingetragen. In derselben Klasse wird auch die Superklasse als Präfix eingetragen, deren Methode bei der Namenskollision nicht umbenannt worden ist. Beide Informationen werden später beim Generieren von Methodennamen benötigt. Eine weitere Operationsklasse, *OpenAttributeSearcher*, sucht innerhalb der Bäume nach Attributen, für die keine entsprechende *get*- oder *set*-Methoden vorgesehen sind. Diese Attribute werden in die Klasse *AttributeTable* eingetragen, damit später entschieden werden kann, ob innerhalb der IDL-Dateien *get*- oder *set*-Methoden generiert werden müssen. Jede dieser drei Operationsklassen greift dabei auf passende Objekte der Klassen *TreeIterator* und *NodeComparator* zurück (diese Klassen wurden schon beim Einlesen erläutert, s. Abbildung D.10).

Nach Ermitteln dieser Informationen kann nun der zweite Schritt durchgeführt werden. Die restlichen in der Abbildung gezeigten Klassen übernehmen dabei die Aufgabe, die an zu Knoten gehörenden Unterknoten zu sortieren, die innerhalb der Knotenklasse als Liste repräsentiert werden. Die Hauptaufgabe übernimmt dabei die Klasse *ReferencePool*, die einen Verweis auf einen Datei-, Modul-, Klassen- oder Methodenknoten hält. Aus mehrereren Objekten dieser Klasse wird ein isomorpher Baum zu den zu instrumentierenden Objekten aufgebaut, der dann durch rekursive Aufrufe der Methode *sort()* von unten nach oben sortiert wird. Das Verfahren gründet dabei auf der Tatsache, daß jeder Knoten im Normalfall Verweise auf andere Knoten enthält, die unter Umständen Forward Declarations erfordern. Wird dagegen in der IDL-Datei die Deklaration vor dem Verweis vorgenommen, ist diese überflüssig. Bei entsprechender Sortierung des Baumes brauchen die einzelnen Knoten nur noch von oben nach unten und von links nach rechts in die entsprechenden IDL-Konstrukte gewandelt werden, und die Anzahl der Vorwärtsdeklarationen wird minimal gehalten.

Abbildung 5.3 stellt einen Schritt des Sortierverfahrens schematisch dar. Jedes Objekt vom Typ *ReferencePool* hält die IDs, auf die vom zugehörigen Objekt der Klasse *Node* verwiesen wird, hat aber darüber hinaus auch Zugriff auf alle Verweise *unterliegender*

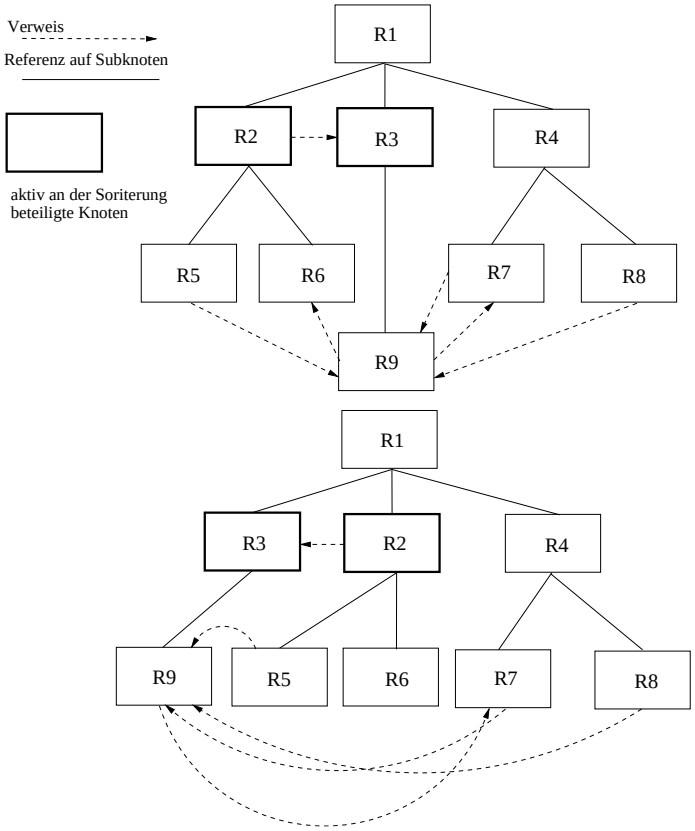

Abbildung 5.3: Sortierung der *ReferencePool*-Objekte.

Knoten des aktuellen Knotens. Nun werden alle auf der gleichen Ebene liegenden Knoten mittels des Bubble Sort-Verfahrens sortiert. Nebeneinanderliegende Knoten werden dabei auf folgende Weise verglichen: Ist die Anzahl der Verweise des weiter links liegenden Knotens auf den weiter rechts liegenden Knoten größer als die Anzahl umgekehrter Verweise, werden beide Knoten getauscht. Unterliegende Knoten werden bei dieser Zählung miteinbezogen. Nach Sortierung der Ebene wird auf die nächsthöhere Ebene zurückgekehrt und die Sortierung dort vorgenommen. Auf diese Weise wird die Anzahl der Vorwärtsdeklarationen minimal gehalten. Im nachhinein können dann alle noch verbleibenden Vorwärtsdeklarationen ermittelt werden, indem für jeden Knoten des gesamten Baums erfragt wird, welche Verweise *nicht* durch davor- oder darüberliegende Knoten erfüllt werden. Für das in der Abbildung gezeigte Beispiel heißt dies: Die Objekte *R2* und *R3* werden getauscht, weil *R3* nur einen Verweis auf Objekte von *R2* besitzt (den von *R9* auf *R6*) während *R2* zwei Verweise (auf *R3* und *R9*) hält. Nach dem Tausch ist nur noch der Verweis von *R9* auf *R7* nicht erfüllt und muß später durch eine Vorwärtsdeklaration aufgelöst werden.

Vor der Sortierung wird der Aufbau der Struktur durch die Klasse *PoolFactory* vorgenommen. Auch diese Klasse traversiert mittels eines Iterators den kompletten Baum und greift auf die in den Tabellen stehende Information zu. Die Klasse *InterfaceOpFilter* prüft dabei folgende Bedingungen für jedes Objekt vom Type *Node*:

- Es handelt sich um einen Knoten, der ein Modul, eine Klasse, ein Attribut oder eine Methode darstellt.

- Handelt es sich um einen Attributknoten, muß zumindest eine *get*- oder *set*-Methode fehlen. Dies wird durch Aufruf der dafür vorgesehenen Methoden der Klasse *AttributeTable* geprüft.

Zu jedem ausgewählten Knoten wird ein entsprechendes Objekt vom Typ *ReferencePool* erzeugt. Anschließend kann durch Aufruf der Methode *getRefPoolRoot()* der Wurzelknoten der Struktur geholt werden. Alle aus den einzelnen Bäumen erzeugten Strukturen werden wiederum durch ein Objekt von Type *ReferencePool* zusammengefaßt, das keinen Verweis auf einen Knoten enthält. Dies ist notwendig, weil alle Deklarationen innerhalb einer Datei aufgeführt werden sollen.

Danach wird durch Aufruf von *sort()* auf dem globalen Knoten das oben beschriebene Sortierverfahren angestoßen. Dabei übernehmen die einzelnen Methoden folgende Aufgaben:

- Mittels *hasId()* kann ein Objekt vom Typ *ReferencePool* ein anderes fragen, ob es einen Knoten mit einer bestimmten ID darstellt oder als Subknoten enthält. Unter anderem ruft diese Methode die Methode *getReferenceIds()* der Klasse *Node* auf, die alle aufgelösten Verweise des eigentlichen Knotens zurückgibt.

- *hasId()* wird dann von *getForwardCount()* und *getBackwardCount()* verwendet. Mittels dieser Methoden können jetzt die gegenseitigen Verweise zweier benachbarter Knoten vom Typ *ReferencePool* festgestellt werden.

Über das Attribut *subPools* kann ein Knoten mittels Aufruf dieser Routinen seine eigenen Subknoten sortieren.

Nach Beenden der Sortierung sind die eigentlichen Forward Declarations zu ermitteln. Dies geschieht durch Aufruf der Methode *addForwardDecl()*, die entsprechende Objekte vom Typ *ForwardDeclaration* erzeugt und in die Klasse *ForwardDeclTable* einträgt. Dabei wird die Methode *searchId()* verwendet. Diese Methode sucht innerhalb vorhergehender und übergeordneter Knoten nach der geforderten ID. Wird diese nicht gefunden, erzeugt das betroffene *ReferencePool*-Objekt einen Eintrag für eine Vorwärtsdeklaration. Dabei sind zwei wesentliche Punkte zu berücksichtigen:

- Benötigen zwei Entitäten dieselbe Vorwärtsdeklaration (z. B. zwei verschiedene Interfaces einen Verweis auf ein anderes), so ist eine doppelte Vorwärtsdeklaration zu vermeiden. Aus diesem Grund trägt ein *ReferencePool*-Objekt nach Erzeugen einer Vorwärtsdeklaration die ID des aufgelösten Verweises in dem Attribut *closedRefs*. Dieses Attribut wird bei der Suche berücksichtigt, so daß untergeordnete und nachfolgende Objekte erkennen können, daß schon ein Eintrag erzeugt wurde.

- Unter Umständen reicht eine einfache Vorwärtsdeklaration nicht aus. Abbildung 5.3 (unterer Teil) macht das Problem deutlich: Repräsentieren *R3* und *R4* Module und alle untergeordneten Knoten Interfaces, so reicht eine Vorwärtsdeklaration von *R7* für *R9* nicht aus. Zusätzlich muß auch das Modul *R4* deklariert werden, um die Namensbereiche korrekt zu erfassen. Das hat außerdem zur Folge, daß die Vorwärtsdeklaration nicht mehr direkt vor *R9* stehen darf, sondern bereits vor *R3* erfolgen muß. Um diese Verschiebung zu berücksichtigen, übergibt jedes *ReferencePool*-Objekt bei Erzeugung eines Eintrags seine eigene Referenz an die Tabelle. Diese kann jetzt für die übergeordneten Knoten dieses Objekts die Vorwärtsdeklarationen holen und die zugehörigen *ReferencePool*-Objekte ermitteln. An jedem dieser gefundenen Objekte wird jetzt *isSubPoolOf()* aufgerufen, um eine entsprechende Beziehung festzustellen.

Ist dies der Fall, wird bei dem entsprechenden *ForwardEntry*-Objekt über *addSubEntry()* der entsprechende Untereintrag erzeugt.

Nach Eintrag aller Vorwärtsdeklarationen wird nun die Erzeugung durchgeführt. Die beteiligten Klassen zeigt Abbildung D.15 im Anhang.

Das eigentliche Schreiben der Datei geschieht über die Methode *write()* der Klasse *ReferencePool* und die Klasse *FileWriter*. Bei der Erzeugung des IDL-Codes bezieht jeder Knoten folgende Informationen mit ein:

- Bei Deklaration einer Entität wird in der Klasse *IdentifierTable* nachgesehen, ob für den Bezeichner ein Alias existiert oder ein Präfix vorangestellt werden muß. (Tatsächlich wird auch beim Eintragen der Forward Declaration der Alias eingetragen, falls er existiert.)

- Für korrekte Verweise ist die Bestimmung der Lage des Typs wichtig, auf den verwiesen wird. Dieser wird mit Hilfe der Klassen *NameCreator* und *CreatorFactory* ermittelt. Dabei schafft die Klasse *CreatorFactory* bei Aufruf der Methode *getNameCreatorFor()* einen Verweis auf einen Knoten (vom Typ *Node*) und betimmt dessen vollständigen Namenspfad bis zum Wurzelknoten (also dem Dateiknoten) als String. Setzt jetzt ein Objekt vom Typ *ReferencePool* seine eigene ID in der Methode *setImporter()* ein, kann das *NameCreator*-Objekt dessen Pfad bis zur Wurzel ebenfalls ermitteln und damit einen gemeinsamen Präfix bestimmen, der dann abgeschnitten wird. Über *getName()* kann dann der Rest zurückgegeben werden, der den Bezeichner ausmacht. Damit die aufwendige Ermittlung nicht jedesmal komplett erneut durchgeführt werden muß, hält die Klasse *CreatorFactory* nach dem in [GHJV02] beschriebenen Flyweight Pattern das verwendete *NameCreator*-Objekt weiter, um es bei Bedarf erneut zu verwenden.

- Vor Deklaration eines Typs wird in der Klasse *ForwardDeclTable* nachgesehen, welche Forward Declarations eingefügt werden müssen. Diese werden dann mittels Aufruf von *toString()* an den entsprechenden Objekten vom Typ *ForwardEntry* in die Datei geschrieben.

Die Erzeugung des konkreten IDL-Codes wird dabei davon abhängig gemacht, von welchem speziellen Typ ein *Node*-Objekt ist, auf das ein *ReferencePool*-Objekt verweist. Darüber hinaus erzeugt jedes dieser Objekte, das auf einen Knoten vom Typ *ClassNode* verweist, nach der eigenen Deklaration eine Deklaration für eine entsprechende Sequenz, die später für Arrays benötigt wird. Diese Deklaration entfällt, falls vorher bereits eine Forward Declaration erfolgt ist. Dies kann durch Aufruf von *hasForwardEntry()* festgestellt werden. Falls ein Eintrag in der Tabelle existiert, ist das Array schon vorher deklariert worden.

Die letzte noch zu erläuternde Klasse, *InterfaceTransformer*, ist abstrakt, die Funktionalität muß vom Benutzer durch eine konkrete Subklasse bereitgestellt werden. Die Subklasse erfüllt zwei Aufgaben: Zum einen gibt sie für die konkrete Programmiersprache zurück, welche Quellcode-Dateien der IDL-Deklaration entsprechen und welchen Bezeichner die entsprechende Klasse trägt, zum anderen sorgt die Klasse für die Aufnahme der neuen Datei in ein bisheriges Softwarepaket. Damit kann der Benutzer beispielweise unter Java die entsprechenden Erweiterungen des Suchpfades vornehmen. Um mehrere Sprachen dynamisch zu unterstützen, muß der Benutzer an dieser Stelle auch für jede Sprache eine Implementation bereitstellen. Zum Erzeugen der entsprechenden Einträge greifen alle Objekte vom Typ *ReferencePool*, die auf einen Klassenknoten verweisen, auf

die bereits erwähnte Klasse *MappingTable* aus Abbildung D.4 zu, um ihre eigene ID in den korrekten Bezeichner zu wandeln. Die gewonnene Information wird dann später zur Instrumentierung benutzt.

Mit dieser Klassenstruktur ist die Erzeugung der regulären Klassen abgedeckt. Die Erzeugung der Deklarationen für Servants der externen Klassen übernimmt die Klasse *Instrumentator* direkt. Für diese Klassen werden vor Beginn aller anderen Deklarationen Forward Declarations erzeugt und am Ende der IDL-Datei die entsprechenden Deklarationen eingefügt. Dabei wird auf die vom Benutzer über *addExternalClass()* und *addExternalMethod()* eingefügten Informationen zurückgegriffen. Mittels dieser Methoden kann der Benutzer zuerst externe Klassen angeben, für die ebenfalls eine eigene ID vergeben wird. Über diese ID und die Klasse *MethodStruct* kann der Benutzer jetzt angeben, welche Methoden der Klasse hinzugefügt werden. Dazu ruft er die Methode *addExternalMethod()* auf. Die Klasse *ReferencePool* greift auf diese Informationen mittels *hasExternalId()* zurück, um überflüssige Vorwärtsdeklarationen zu vermeiden, nachdem diese Informationen mittels *integrateExternalClasses()* integriert worden sind. Die letztgenannte Methode ist Teil des Gesamtablaufs, daher findet man eine genauere Erläuterung im Abschnitt 6.2.

Kapitel 6

Die Instrumentierung der Software

6.1 Allgemeiner Ablauf

Nach Erzeugung des Codes für die Skeleton- und Stub-Klassen muß nun die eigentliche Instrumentierung durchgeführt werden. Dieser Vorgang integriert auch die in Abschnitt 5.1 beschriebenen Hilfsklassen. Dabei wird der ursprüngliche Quellcode nur dahingehend verändert, daß Rückmeldungen über Ereignisse möglich sind:

- Jede der beteiligten Klassen muß ein zusätzliches Attribut erhalten, das die Laufzeit-ID darstellt. Diese ID wird innerhalb des Konstruktors vergeben. Zusätzlich meldet jede Klasse über den Aufruf der Methode *objectCreated()* des Managers, daß sie erstellt worden ist. Innerhalb des Destruktors wird (soweit vorhanden) *objectDeleted()* aufgerufen.

- Im passiven Modus wird außerdem vor Verlassen einer Methode *objectChanged()* für das zugehörige Objekt aufgerufen. Mittels dieser Ereignisse kann der Benutzer individuell auf Änderungen des Objektzustands reagieren.

- Die restlichen Einfügungen der Schnittstelle in den Originalcode steuert der Benutzer mittels Wahl der zu instrumentierenden Punkte. Die Schnittstelle unterstützt dabei folgende Instrumentierungsarten[1]:

  – **Individuelle Einfügung eines Codestücks an einer mit einer ID bezeichneten Stelle:** Diese Instrumentierungsart erlaubt flexible Instrumentierung und könnte z. B. auch für Dataflow-Coverage genutzt werden (s. [Bin00]). Dabei soll beispielsweise bei einer ID eines Blockknotens nur die Stelle bezeichnet werden, bei der ein Block beginnt. Für eine `if`-Anweisung gilt wird die Instrumentierung für Bedingung, und `if`- sowie `else`-Teil durchgeführt, bei einer Schleife für den Bedingungsteil und den Beginn des Schleifenrumpfes. Bei einer `switch`-Anweisung wird der Beginn eines jeden Blocks instrumentiert, bei einer Methode der Beginn des Methodenrumpfes und jeder Austrittspunkt. Für eine Exception ist nur eine Einfügung am Eintrittspunkt vorgesehen.

[1] Die letzten drei Instrumentierungsarten orientieren sich an den in [Bin00] beschriebenen Überdeckungskategorien.

- **Statement Coverage:** Diese Instrumentierungsart unterstützt Statement Coverage, instrumentiert also jede einzelne Anweisung. Das Zählen von Schleifendurchläufen muß der Benutzer übernehmen. Im Ausblick wird eine mögliche Lösung zur Durchführung der Zählung angedeutet. Dabei werden abhängig von der übergebenen ID alle Klassen einer Datei oder eines Moduls in allen Methoden instrumentiert, oder nur eine Methode einer ausgewählten Klasse.
- **Branch Coverage:** In diesem Modus wird nur der Beginn einer jeden Verzweigung eines Methodenrumpfes instrumentiert. Der instrumentierte Bereich kann analog zur Statement Coverage mittels Id angegeben werden.
- **Multiple Condition Coverage:** Auch hier werden alle Verzweigungen einer Methode instrumentiert.

Jede dieser Instrumentierungsarten fügt an den entsprechenden Stellen zwei Aufrufe ein: Das Holen der Objekt-Manager-Referenz und den Aufruf von *reachedPoint()*. Da innerhalb von Methoden außerdem auch Methoden der Superklasse aufgerufen werden können (z. B. in Java durch `super`), bietet die Schnittstelle an, auch die überschriebenen Methoden der Superklassen zu instrumentieren. Dies löst auch das am Ende des vorhergehenden Kapitels angesprochene Problem der Mehrfachvererbung unter C++ (s. Seite 65).

Bei Instrumentierung von Bedingungen hat der Benutzer außerdem die Wahl, kombinierte Prädikate als eine Einheit oder als Verkettung mehrerer atomarer Prädikate zu instrumentieren. Um die Überdeckung korrekt zu messen, wird eine Methoden für jede instrumentierte Klasse eingeführt, die beim Aufruf jeweils die ID der Bedingung übergeben bekommt – *trueEvent()*. Diese Methode liefert außerdem immer *true* als Ergebnis. Innerhalb dieser Methode wird wiederum *reachedPoint()* mit dem übergebenen Argument aufgerufen. Bei atomarer Betrachtung der Bedingung genügt es, mittels einer Und-Verknüpfung einen Aufruf von *trueEvent()* vor den Code zu setzen. So wird aus

```
if (a == b) {....}
```

nach Instrumentierung beispielsweise

```
if ((a == b) && (trueEvent(125)) {....}
```

Bei feingranularer Betrachtung wird die Instrumentierung komplizierter. So wird beispielsweise aus

```
if ((a < b) && ((b < c) || (c < d))) {....}
```

das Codestück

```
if (((a < b)  && (trueEvent(125))) &&
    (((b < c) && (trueEvent(126))) ||
    ((c < d) &&  (trueEvent(127))))
   ) {....}
```

Auf diese Weise kann leicht festgestellt werden, welche Prädikate noch erreicht werden.

Für die zusätzlich erzeugten Klassen vom Typ *TServant* und muß der Code neu generiert werden. Teilweise können dazu die Regeln des Rule Repository wiederverwendet werden. Mit Hilfe der entsprechenden Regeln und den Informationen aus der konkreten Subklasse von *InterfaceTransformer* können alle Methoden der generierten Skeleton-Klassen eingelesen werden. Anschließend können die ermittelten Informationen dazu verwendet werden, dieselben Methoden innerhalb der Subklasse zu deklarieren. Danach müssen die

Methodenrümpfe gefüllt werden. Dies ist auch für den Objekt-Manager notwendig, da von vornherein nicht bekannt ist, welche Klassen später am Test beteiligt sind. Für alle Klassen vom Typ *TList* und *TNode* ist dagegen eine komplette Neugenerierung notwendig.

Innerhalb bestimmter Grenzen sind die Routinen der im letzten Absatz genannten Klassen aber fest. So ist beispielsweise die Methode *createObject()* eine Aneinanderreihung mehrerer `if`-Abfragen, bei der sich nur die ID und die zugehörige Klasse bzw. die zugehörige Liste ändert, an die der Aufruf weitergegeben wird. Genauso unterscheiden sich die Routinen einer Verwaltungsliste nur in den Bezeichnern für die Elementklasse und der verwalteten Klasse. Ansonsten setzen sich die Methoden aus sprachspezifischen Bausteinen zusammen, die ebenfalls fest sind.

Dies führt zu folgendem Ansatz: Die Schnittstelle liest bei der Initialisierung zwei weitere Dateien ein, die die notwendigen Informationen für die Instrumentierung liefern. Innerhalb einer Datei, der *Syntaxdatei*, sind die Syntax-Bausteine enthalten, die später zum Aufbau des konkreten Codes benutzt werden. In der anderen Datei, die hier mit *Routinendatei* bezeichnet werden soll, stehen die zum größten Teil festgelegten Routinen für die unterstützenden Klassen in Form einer einfachen Sprache. Diese Routinen enthalten an bestimmten Stellen Parameter, deren Werte dann aus den eingelesenen Informationen geholt und bei Erzeugung des Codes eingesetzt werden. Diese Herangehensweise bietet zwei Vorteile:

- Die Erzeugung des Codes wird nicht innerhalb der Schnittstellenklassen varankert. Das hat zur Folge, daß die Schnittstellenhierarchie für diesen Teil kleiner gehalten wird, weil sie sich auf den Umsetzungvorgang konzentriert. Darüber hinaus kann der Code für die vorgegebenen Klassen (z. B. den Objekt-Manager) auch an eine Sprache angepaßt werden, falls dies notwendig ist.

- Bei geschickter Aufteilung der Funktionalität kann die Codeerzeugung nicht nur durch Einlesen des Codes aus einer Datei vorgenommen werden. Alternativ können die Objekte, die diesen Code repräsentieren, zur Laufzeit erzeugt und an die Klassen übergeben werden, die für die Codeerzeugung zuständig sind. Diese Regelung erlaubt es, einige Teile dynamisch zu erzeugen, deren Inhalt nicht durch parametrisierte Codeteile ausgedrückt werden kann (z. B. die Übergabe der Initialisierungsargumente für den ORB).

Tabelle C.1 im Anhang zeigt die Bestandteile, die innerhalb der Syntaxdatei aufgelistet werden. Dabei besitzen einige Syntaxkonstrukte Parameter, in die später konkrete Werte eingesetzt werden (z. B. Objektname und Methodenname beim Methodenaufruf). Diese Parameter sind nicht mit den Parametern für die einzelnen Routinen zu verwechseln. Die spätere Beschreibung der Klassen trennt beide Parameterarten und zeigt, daß die Codeerzeugung in zwei Schritten vorgenommen wird: Zuerst werden die Parameter der Routinen ersetzt, um nur noch konkrete Bezeichner innerhalb der Coderepräsentation zu benutzen. Danach werden die im Speicher gehaltenen Codekonstrukte in Code umgesetzt, wobei die Syntaxbestandteile herangezogen und deren Parameter durch die konkreten Bezeichner ersetzt werden.

Tabelle C.2, die ebenfalls im Anhang zu finden ist, zeigt die Bestandteile des Codes, der später in eine konkrete Sprache umgesetzt wird. Die Routinen werden dabei durch Bezeichner in eckigen Klammern abgegrenzt, die ähnlich einer XML-Beschreibung eine Anfangs- und eine Endversion besitzen. Dabei unterscheidet die Schnittstelle zwei Varianten: Die erste Variante bezeichnet komplette Routinen, z. B.

```
[createObject]
```

```
[active]
...Code...
[/active]

[passive]
...Code...
[/passive]

[/createObject]
```

Dabei kennzeichnen die Schlüsselworte `active` und `passive` die Varianten für den aktiven und passiven Modus. Die zweite Variante bezeichnet Codestücke für die Wandlung von Basistypen, beispielsweise

```
[string]

[from]
...Code...
[/from]

[to]
...Code...
[/to]

[/string]
```

Hier bezeichnen die Schlüsselwörter `from` und `to` die Codestücke für die Wandlung vom und zum CORBA-Typen. Diese Routinen werden bei der Weiterleitung von Methodenaufrufen von den Servants an die eigentlichen Objekte gebraucht. Bei Arrays ergibt sich hier die Notwendigkeit, jedes Element innerhalb einer Schleife zu wandeln, so daß eine Überprüfung der Länge möglich ist. In C++ muß der Benutzer hier eine eigene Array-Klasse zur Verfügung stellen, die ein normales Array kapselt. Diese Klasse muß folgende Merkmale aufweisen:

- Es muß eine Methode zur Längenabfrage bereitgestellt werden.
- Die Operatoren `[]` und `new` bzw. `delete` müssen überladen werden, damit der Umgang mit dieser Klasse ansonsten wie bei einem normalen Array funktioniert.
- Die Klasse muß generisch sein, damit alle Typen unterstützt werden können.
- Es muß eine Version für Basistypen und eine Version für Objekttypen definiert werden.

Für alle erzeugten Codestücke gibt es bestimmte Anforderungen, die sich aus den Möglichkeiten der einzelnene Sprachen ergeben:

- Da Eiffel als Rückgabewert die Variable *Result* benutzt, muß der Code, um nicht ständig an die Sprache angepaßt zu werden, so geschrieben sein, daß am Ende der Routine das Resultat das korrekte ist, d. h. die Semantik des Ausgabewertes `$%` entspricht nicht immer der einer `return`-Anweisung.

- Bei Schleifen wird in der Regel eine Init-Anweisung gefordert. Auch diese Regelung wird durch Eiffel motiviert. Bei einer Java- oder C++-Schleife wird dieser Init an den Anfang der Schleife gesetzt. Da die Schleife in Eiffel außerdem eine Abbruchsbedingung statt einer Eintrittsbedingung verwendet, werden hier alle in der Schleife angegebenen Bedingungen negiert.

- Im Hinblick auf C++ wird durch die Option *StandardRefType* angegeben, von welcher Art Verweise auf neue Attribute oder lokale Variablen sind. Die Schnittstelle unterscheidet hier Referenz und Pointer. Dies ist unter anderem auch bei der Erzeugung einer Zuweisung oder eines Methodenaufrufs wichtig, bei der unter Umständen der Adreß-Operator oder Dereferenzierung benutzt werden muß. Casts sind nicht möglich, weil diese nicht direkt von Eiffel unterstützt werden und jede Sprache ihren eigenen Fehlerbehandlungsmechanismus beim Fehlschlag von Casts bietet. Damit Zuweisungen und Methodenaufrufe korrekt umgesetzt werden können, sind darüber hinaus verkettete Methodenaufrufe verboten.

- Zusätzlich werden bei der Erzeugung des Codes einige angegebene Tags der Syntaxdatei benutzt, die einige Regelungen in Bezug auf das CORBA-Mapping angeben:

  – Die Verweisart von CORBA-Referenzen (Pointer, Referenz)
  – Die Verweisart des `out`-Typs und dessen Benennung
  – Der Zugriff auf den `out`-Typ (schreibend und lesend)
  – Die korrekte Schreibweise für Narrowing und die `this`- bzw. `nil`-Referenz in CORBA

Da der Benutzer für die integrierten Hilfsklassen ebenfalls Code erzeugt, muß er sich ebenfalls an die oben genannten Bedingungen halten.

Im aktiven Modus ist die Erzeugung des Hauptprogramms ebenfalls Teil der Instrumentierung. Die Schnittstelle unterstützt dabei das Einfügen durch die Option *MainLocation* in der Optionsdatei und das Tag `<main-program>` innerhalb der Syntaxdatei. Die an den ORB weitergegebenen Initialisierungsargumente werden bei Aufruf der Methode *init()* in entsprechenden Attributen gehalten. Das Hauptprogramm selbst besteht nur aus einer Initialisierung des Objekt-Managers durch Holen der Singleton-Referenz. Der Code für die Hauptroutine ist damit fest und muß vollständig innerhalb des Tags eingetragen werden.

Neben der eigentlichen Codeerzeugung muß die Schnittstelle Funktionalität bereitstellen, um die korrekte Stelle für das Einfügen des Codes oder die Änderung des Originalcodes zu finden. Der Vorgang ist dabei recht einfach gehalten: Die Aufträge für eine Datei werden zusammengefaßt und die Datei schrittweise durchlaufen. An den passenden Stellen wird eine Instrumentierung vorgenommen und der neue Inhalt in eine andere Datei geschrieben. Anschließend werden die instrumentierten Dateien über die Originaldaten kopiert. Dies ist notwendig, weil Java fordert, daß öffentliche Klassen in einer Datei mit gleichem Namen liegen. Umbenennungen von Klassen werden durch die Schnittstelle nicht vorgenommen.

Die folgende Beschreibung bezieht sich auf Abbildung D.15 und stellt dar, welche Funktion einzelne Methoden der Klasse *Instrumentator* übernehmen und in welcher Reihenfolge deren Aufruf sinnvoll ist.

- Durch *setInstrumentationMode()* wird der Modus gewählt(aktiv oder passiv). Mittels *addInstrumentationOrder()* werden dann Instrumentierungsaufträge in der oben angesprochenen Form an die Klasse übergeben.

- Nun kann mit *addExternalClass()* und *addExternalMethod()* das Hinzufügen von externen Klassen vorgenommen werden. Ein Aufruf von *integrateExternalClasses()* integriert diese Klassen.
- Anschließend kann durch *generateTasks()* die Auftragsliste in eine äquivalente Liste mit Codeerzeugungsaufträgen umgewandelt werden, die zur Durchführung der Instrumentierung notwendig sind (eine genauere Beschreibung folgt im nächsten Abschnitt).
- Dann sorgt der Aufruf von *generateIDL()* für die Generierung der IDL-Dateien und abschließend (nach Kompilierung der IDL-Dateien) der Aufruf von *instrumentate()* für die eigentliche Instrumentierung der Dateien.

Der folgende Abschnitt beschreibt die technischen Maßnahmen zur Instrumentierung genauer.

6.2 Schnittstellenklassen zur Instrumentierung

Die Klassen zur Unterstützung der Instrumentierung machen den größten Teil der Schnittstelle aus. Die im vorhergehenden Abschnitt beschriebenen Funktionalitäten werden dabei in mehreren Abstraktionsschichten realisiert, auf die nachfolgend im einzelnen eingegangen wird. Die Schichten werden dabei von unten nach oben betrachtet, das heißt, zuerst werden die Klassen zur Codeerzeugung beschrieben, zum Schluß die Klassen, die die Interfaces für den Benutzer zur Verfügung stellen.

Basis zur Erzeugung des einzufügenden Codes und zur Generierung der Hilfsklassen sind die in den Abbildungen D.18 und D.19 dargestellten Klassen. Die in Abbildung D.18 dargestellten Klassen dienen dabei zum Aufbau einer Tabelle, die die in der Syntaxdatei abgelegten Konstrukte hält. Die Tabelle wird durch die Klasse *SyntaxBasis* repräsentiert, während alle Subklassen der Klasse *SyntaxElement* einzelne Teile der Konstrukte aufnehmen. Der Aufbau beim Einlesen geschieht durch die Klasse *ElementBuilder*, wobei zum Einlesen der Datei wiederum *flex* und *bison* verwendet werden. Durch Aufbau per *setNext()* kann ein Baum von Elementen zu einem kleinen Stück Code zusammengesetzt und per Aufruf von *write()* in den sprachspezifischen Code umgewandelt und in eine Datei geschrieben werden. Dabei übernimmt jede Subklasse die Repräsentation eines bestimmten Elements:

- *StringElement* repräsentiert eine einfache Zeichenkette.
- *ParamElement* stellt einen Parameter dar, z. B. *Name*. Mittels *insertParam()* kann für einen Parameter (der mit *setParamName()* benannt worden ist) ein Wert eingesetzt werden. Um die richtige Stelle zu finden, wird dabei der Aufruf rekursiv weitergegeben, bis er das *ParamElement*-Objekt mit dem korrekten Parameternamen erreicht. Bei der Umwandlung mit *write()* wird später der Parameterwert eingesetzt und in eine Datei geschrieben. Dabei kann der Parameterwert wiederum eine komplexe Struktur sein.
- *MarkerElement* dient nur dazu, beim Instrumentieren eine bestimmte Position zu markieren. Diese Markierung wird benötigt, weil die Instrumentierung die neuen Klassen in zwei Durchgängen erzeugt: Zuerst werden alle Klassen und Methoden generiert, danach werden die Methodenrümpfe gefüllt. Die Klasse ist auch die einzige Subklasse von *SyntaxElement*, die die Methode *addObserver()* ausnutzt. Auf

diese Weise kann das Element später bei der Umwandlung einer Observer-Klasse mitteilen, daß in der Datei eine Position erreicht wird (die genaue Beschreibung des Observer Patterns findet man in [GHJV02]).

Die in der anderen Abbildung gezeigten Klassen nutzen nun diese Elemente für ihre eigene Funktionalität. Die zentrale Klasse für die Repräsentation eines Codestücks ist *CodeSnippet*, die Subklassen stellen jeweils konkrete Konstrukte dar, die in der Routinendatei benutzt werden. Dabei existiert zu jedem in Tabelle C.1 im Anhang angegebenen Konstrukt eine Subklasse. Die in der Abbildung gezeigten Klassen stellen nur einen Ausschnitt der Gesamtmenge dieser Klassen dar.

Wichtig für den Aufbau sind die Klassen *IdentifierSnippet*, *ParamSnippet* und *ClassparamSnippet*, die die Basiselemente darstellen. *IdentifierSnippet* repräsentiert einen Bezeichner innerhalb der Routinendatei, der z. B. einen Variablennamen oder einen Typen bezeichnet. Die Klasse *ParamSnippet* steht für einen allgemeinen Parameter, der an Stelle eines Bezeichners stehen kann. Die Werte für die Parameter holt sich diese Klasse aus einem Objekt der Klasse *ParamTable*, die vor Beginn der Umwandlung in Strings übergeben werden muß. Auf diese Weise werden nur die Parameterwerte ausgelesen, die innerhalb des Konstrukts gebraucht werden, übergeben werden müssen immer *alle* in der Tabelle C.1 (im Anhang) angegebenen Argumente. Die Handhabung funktioniert dabei folgendermaßen:

- Mit der Klasse *CodeSnippetBuilder* werden die entsprechenden Codestücke entweder aus der Routinendatei eingelesen und und in der Klasse *CodePieceTable* gespeichert, oder sie werden durch direkte Aufrufe aufgebaut . Aus der Klasse *CodePieceTable* können für die spätere Instrumentierung die einzelnen Routinen ausgelesen werden. Der Aufbau wird dabei über einen internen Stack gehandhabt. Beispielweise nimmt die Methode *buildIf()* die obersten drei Elemente vom Stack, bildet daraus eine Fallunterscheidung mit Bedingung, if- und else-Zweig und legt das resultierende *CodeSnippet*-Objekt wieder auf dem Stack ab. Mittels *getCode()* kann später der komplette Baum eines Routinenstücks zurückgegeben werden.

- Mit *setParamTable()* werden die Parameter übergeben, die dieser Baum erhalten soll. Dabei kann ist über den in der Klasse *CodePieceTable* zugeordneten Namen festgelegt, welche Parameter das Codestück erwartet. Das Einsetzen der Parameter geschieht durch *resolveParams()*. Dieser Aufruf wird rekursiv durch den Baum weitergegeben. Der Ausgabewert `$%` wird dabei als besonderer Parameter behandelt (für die Zuweisung zu diesem Element wird eine Objekt der Klasse *ReturnSnippet* erzeugt, die ebenfalls Subklasse von *CodeSnippet* ist). Wird dieser Parameter auf `"Return"` gesetzt, wird die entsprechende Zuweisung in eine `return`-Anweisung umgesetzt, ansonsten in eine Zuweisung an die benannte Variable. Analog wird ein entsprechender Name für die Eingabevariable `$$` gesetzt und später in den Code integriert.

- Schließlich kann mit *write()* die Wandlung in einen String und das Schreiben in eine Datei durchgeführt werden. Dabei greifen jetzt die einzelnen Subklassen von *CodeSnippet* auf die Konstrukte zurück, die in der Klasse *Syntaxbasis* gespeichert sind, und erzeugen ein passendes Codestück. Unter Umständen benutzt ein *CodeSnippet*-Objekt ebenfalls die die Klasse *ElementBuilder* (z. B. um Argumente in eine Methodendeklaration einzusetzen). Die Klasse *SnippetList* dient dabei zur Kapselung eines solchen Routinenstücks und gibt die Aufrufe weiter.

Für die Klasse *ClassParamSnippet* muß beim Aufbau angegeben werden, welche Ausprägung des Klassenparameters tatsächlich eingesetzt werden soll (Klassenname, Servant-

name usw.). Dies geschieht über den Aufruf der Routine *setParType()*. Die gehaltene Information ist ein einfacher Enumerationstyp.

Die Abbildung illustriert zwei weitere Besonderheiten, die von Klassen mit höherer Funktionalität gebraucht werden:

- Mit *getLocals()* können vor der Umwandlung alle lokalen Variablendeklarationen ausgelesen werden. Diese werden für die korrekten Umwandlungen von Zuweisungen und Methodenaufrufen benötigt (s. unten). Darüber hinaus werden diese Deklarationen gesammelt und vor dem eigentlichen Codestück angeordnet (dies ist für Eiffel notwendig, für C++ und Java ohne Bedeutung). Die Objekte der Klasse *LocalDefSnippet* verbleiben dabei an ihrem Platz in der Baumstruktur, der Aufruf von *write()* erzeugt aber keinen Code. Dies geschieht an anderer Stelle. Die Methode *getType()* greift später teilweise auf diese Einträge zurück. Darüber hinaus wird für die Umwandlung der Deklarationen auf die Klasse *TypePool* zurückgegriffen (dies ist in der Abbildung nicht dargestellt).

- Die Klasse *ClassDeclarationSnippet* steht für alle Klassen, deren Element mindestens einen Positionsmarker enthält und repräsentiert damit einen Observer. Die aufgerufene Methode ist dabei *markerReached()*, der der Name des erreichten Markers übergeben wird. Wie leicht an der Methode *addObserver()* zu erkennen ist, wird jedoch der Aufruf weiter nach oben gereicht. Diese Funktionalität wird nur von wenigen Klassen benutzt (unter anderem für die Erzeugung von Klassenrümpfen, Modulen und lokalen Deklarationen).

Aufbauend auf dieser Funktionalität arbeiten jetzt die in im Anhang in Abbildung D.20 abgebildeten Klassen. Die Schnittstelle repräsentiert dabei einzelne Instrumentierungsvorgänge durch Subklassen der Klasse *CodeTask* und hält diese in einer entsprechenden Liste (der Klasse *TaskList*). Dabei hält ein Task-Objekt den Ort, an dem die Instrumentierung durchgeführt werden soll, sowie eine interne ID, die der Schnittstelle die Zuordnung eines Tasks zu einer erreichten Stelle ermöglicht, die durch die schon erwähnte Methode *markerReached()* zurückgemeldet wird. Zur Ausführung wird die Methode *execute()* aufgerufen, bei deren Durchführung die beiden Klassen *FileReader* und *FileWriter* verwendet werden, um den Inhalt aus einer Datei einzulesen und den veränderten Inhalt in eine andere Datei zu schreiben. Es werden drei verschiedene Tasks unterschieden:

- Objekte der Klasse *CodeCreationTask* stellen Tasks dar, die *Codeeinfügungen* vornehmen und einen Großteil der Instrumentierung ausmachen. Dieser Task hält dabei mehrere Codestücke, die der Reihe nach in der Datei an der bezeichneten Stelle eingefügt werden. Die Datei wird dabei bis zu der bezeichneten Stelle eingelesen und in die andere Datei geschrieben, danach der zusätzliche Code eingefügt. Der Zeitpunkt, zu dem ein neues Objekt aus der Attributlist *codePiece* entnommen wird, wird durch die Rückmeldung per *markerReached()* bestimmt. Auf diese Weise können geschachtelte Module mit enthaltenen Klassen erzeugt oder Klassen mit enthaltenen Methoden erzeugt werden. Bei der letzten Rückmeldung, bei der die Liste dann leer ist, wird die Meldung an den per *addObserver()* eingetragenen Generator gegeben, der die Position speichert. Diese wird dann für die Generierung der Methodenrümpfe verwendet.

- Objekte der Klasse *CodeReplacementTask* dienen zur *Codeersetzung*. Diese Objekte nutzt die Schnittstelle zur Änderung von Zugriffsrechten bei Methoden. Angegeben werden muß dabei die Stelle, bis zu der überschrieben werden soll. Der Task liest bis

zur gewünschten Stelle aus der alten Datei, trägt den neuen Inhalt ein und überliest alle alten Inhalte bis zum gewünschten Ende.

- Objekte der Klasse *CodeDeletionTask* löschen Codestücke. Diese Klasse ist nur für C++ notwendig, weil hier Veränderungen an den `include`-Anweisungen vorgenommen werden müssen. In diesem Fall wird bis zur Startstelle eingelesen und der alte Inhalt bis zur Endstelle einfach verworfen.

Zusätzlich zu den bisher beschriebenen Informationen hält jeder Task ein Objekt der Klasse *Environment*, das für eine Methode alle lokal deklarierten Variablen, alle zur entsprechenden Klasse gehörenden Attribute und alle Methodenargumente mit Typangaben enthält (für andere Codestücke, z. B. Klassendeklarationen, bleiben alle enthaltenen Listen leer). Diese Informationen werden von den *CodeSnippet*-Objekten, die eine Zuweisung, einen Vergleich oder einen Methodenaufruf darstellen, benötigt. Für jeden enthaltenen Ausdruck (der durch weiter *CodeSnippet*-Objekte dargestellt wird) wird dabei mit *getType()* geprüft, von welchem Typ er ist. Handelt es sich um einen Referenztypen, wird jetzt z. B. bei einer Zuweisung verglichen, ob der Referenztyp auf beiden Seiten derselbe ist. Dies ist notwendig, um bei C++ die korrekten Operatoren zu erzeugen, damit die Anweisung komplierbar ist. Analog gilt dies für Vergleiche und Methodenaufrufe mit ihren Argumenten. Die *CodeSnippet*-Objekte greifen dabei auf ihr eingetragenes *Environment*-Objekt zurück, um ihren eigenen Typ festzustellen. Wird dagegegen das Ergebnis eines Methodenaufrufs zugewiesen, ist darüber hinaus auch der Typ einer Methode festzustellen, der nicht im Environment gespeichert ist. Um diese Information bereitzustellen, greift die *Enviroment*-Klasse auf die Klasse *NodeFinder* zurück, um innerhalb der eingelesenen Bäume den passenden Knoten zu einem Typ zu suchen und aus den eingetragenen Informationen den korrekten Typ zu ermitteln. Da innerhalb des generierten Codes jeweils die vollen Bezeichner (mit Modulen) für Klassen verwendet werden müssen, liefert die Suche immer ein eindeutiges Ergebnis. Darüber hinaus hält die Klasse *NodeFinder* einmal gefundene Knoten, um den Suchaufwand bei weiteren Anfragen zum gleichen Knoten zu verringern. Der Instrumentierungsablauf stellt außerdem sicher, daß zum Zeitpunkt des Vergleichs keine Parameter mehr im Code enthalten sind (s. u.). Der hier beschriebene Vergleich für Methodenargumente und -resultate funktioniert auch für Hilfsklassen, weil die Schnittstellen der Hilfsklassen fest sind (ausgenommen der Bezeichner der Klasse, zu der sie gehören). Daher kann diese Festlegung in der Methode *getTypeOf()* festgeschrieben werden. Analog wird durch den Algorithmus auch der korrekte Zugriff auf CORBA-Klassen (z. B. Repräsentationen eines `inout`-Typs) geregelt. Der Algorithmus ersetzt jedoch keinen kompletten Typcheck. Auf passende Typen muß der Benutzer bei Vorgabe des Codes achten, nur die Art des Verweises (Pointer, Referenz, lokale Variable) darf verschieden sein.

Zusätzlich bietet die *Environment*-Klasse den Nutzen, die Zusammensetzung eines kompletten Methodenrumpfes zu unterstützen. Die Zusammensetzung wird mit Hilfe der Klasse *CodeAssembler* vorgenommen. Diese setzt stückweise einzelne Objekte vom Typ *SnippetList* zusammen. Um zu vermeiden, daß zwei lokale Variablen gleichen Namens deklariert werden, werden in einem mittels *addSnippet()* neu angefügten Codestück Variablennamen umbenannt, deren Benennung mit bereits im vorhergehenden Code benutzten Variablennamen kollidiert. Findet eine Umbenennung statt, liefert *addSnippet()* außerdem *true* als Rückgabewert. Gleichzeitig werden die umbenannten Einträge auch im zugehörigen *Environment*-Objekt aktualisiert. Mittels *getChangedVariables()* kann direkt nach der Einfügung außerdem ermittelt werden, welche Variablen umbenannt wurden. Da diese Variablen später unter Umständen bei der dynamischen Generierung von Codeteilen eine Rolle spielen, muß diese Information nach außen gegeben werden. Mittels *getEnvironment()*

kann die aufrufende Klasse nach Zusammenfügen des kompletten Codes das endgültige Environment und mit *getCodePiece()* den zusammengefügten Codeteil ermitteln.

Mit diesen Klassen steht jetzt die Funktionalität zum Erzeugen kompletter Codeteile zur Verfügung. Der Ablauf sieht dabei folgendermaßen aus:

- Die Schnittstelle holt sich aus der Klasse *CodePieceTable* parametrisierte Codeteile oder generiert zusätzlich benötigte Teile dynamisch. Dynamisch generierte Codeteile dürfen keine Parameter mehr enthalten, bei parametrisierten Teilen werden die entsprechenden Parameter vorgegeben und eingesetzt. Die notwendigen Informationen erhält die Schnittstelle aus den enthaltenen Dateien (siehe unten).
- Dann werden die einzelnen Teile mit Hilfe der Klasse *CodeAssembler* zusammengesetzt und anschließend in ein Objekt der Klasse *CodeCreationTask* eingetragen. Die Tasks werden in ein Objekt der Klasse *TaskList* eingetragen, wo sie durch Aufruf von *sort()* nach Dateinamen und Position sortiert werden. Dies vermeidet mehrmalige Durchläufe derselben Datei.
- Zum Schluß wird mit *execute()* die Ausführung der Instrumentierung angestoßen. Dabei hält das *TaskList*-Objekt ein *FileReader*- sowie ein *FileWriter*-Objekt und initialisiert diese entsprechend des aktuellen Tasks, der dann die beiden Objekte nur noch zum Lesen und Schreiben anspricht. Tasks mit gleicher Position werden dabei in gleicher Reihenfolge gelassen und nacheinander in die Datei eingefügt.

Für die Positionsbestimmung der einzufügenden Codeteile und der zu übergebenden Parameter sind jetzt weitere ansteuernde Klassen notwendig. Die korrekten Stellen für die generierten Hilfsklassen lassen sich hier am besten aus einer Betrachtung von C++ herleiten, weil C++ engere Anfoderungen stellt als die anderen beiden Sprachen: Deklaration und Definition von Methoden sind getrennt, wobei die Deklaration immer vor der Definition oder innerhalb einer vorher inkludierten Datei liegen soll. Eine Aufteilung der einzelnen Hilfsklassen auf neue oder vorhandene Dateien zeigt Abbildung 6.1 am Ende des Abschnitts.

Ablagestelle für die Hilfsklassen der regulären Klassen (*Servant*-, *ServantList*- und *ServantNode*-Klasse) ist die jeweilige Header-Datei, in der die Klasse definiert wird (hier durch *Class.h* gekennzeichnet). Dabei hängt der Ort der Deklaration der *Servant*-Klasse davon ab, welcher Modus gesetzt ist: Bei aktivem Modus muß die *Servant*-Klasse *nach*, bei passivem Modus *vor* der instrumentierten Klasse deklariert werden. Für die Definitionen der einzelnen Klassen werden separate Dateien vorgesehen (*(X)ServantNode.cc*, *(X)ServantList.cc*, *(X)Servant.cc* und *(X)Class.cc*). Zusammengebracht werden die Deklarationen und Definitionen über die Header-Datei des Objekt-Managers (*OM.h*), dessen Funktionalität auch den anderen Klassen zur Verfügung stehen muß. Zusätzlich werden in dieser Header-Datei alle Servants der externen Klassen deklariert, bevor der Objekt-Manager deklariert wird. Auf diese Weise können im Objekt-Manager alle Listenattribute ohne Probleme deklariert werden. Die einzelnen Implementation der Klassen können dann auf die anderen Definition zurückgreifen. An dieser Stelle wird davon ausgegangen, daß die vom Benutzer angegebenen externen Klassen tatsächlich in einer anderen Klasse verwendet und dort auch korrekt inkludiert werden. Das gesamte Klassensystem muß also abgeschlossen sein. Für Java und Eiffel sind die hier beschriebenen Maßnahmen nicht notwendig, da Deklaration und Definition an gleicher Stelle erfolgen und die Reihenfolge der Klassen egal ist.

Aus der Beschreibung des letzten Absatzes wird deutlich, daß die Positionen der einzelnen Deklarationen für die Hilfsklassen leicht aus dem Anfang bzw. dem Ende der eigentlichen Klassendeklaration ermittelt werden kann. Weiter oben wurde bereits beschrieben,

daß nach Einfügen einer Klasse ebenfalls ermittelt wird, wo später der Methodenrumpf einzufügen ist. Wie bereits im vorhergehenden Abschnitt beschrieben wurde, wird für die *Servant*-Klassen, die von den erzeugten Skeleton-Klassen abgeleitet werden müssen, auf die Funktionalität der Klasse *RuleRepository* zurückgegriffen und damit die zu erzeugenden Klassen und Funktionen ermittelt. Der Ablageort dieser Klassen wird wiederum durch die vom Benutzer implementierte Subklasse der Klasse *InterfaceTransformer* angegeben. Dieser Ablage wird dann in eine entsprechende `include`-Anweisung umgesetzt. Die einzelnen Information für die tatsächliche Instrumentierung kann schließlich aus den intern aufgebauten Bäumen ermittelt werden.

Abbildung D.21 im Anhang zeigt die zur Informationsermittlung benötigten Klassen. Die zentralen Klassen sind dabei die Subklassen der Klasse *Generator*. Jeder Generator erzeugt beim Aufruf von *prepare()* die passenden Klassen und Methoden mit leeren Rümpfen, wobei zu den Methoden die lokal deklarierten Variablen bereits hinzugehören. Diese Variablen werden durch Füllen eines Environments ermittelt, d. h. nach Erzeugen der Rümpfe stehen auch die entsprechenden Inhalte zur Verfügung. Da die Klassen und Methoden erst geschrieben werden können, nachdem die lokalen Variablen zur Verfügung stehen, müssen die Task-IDs intern gehalten werden, um später zuzuordnen, welche Positionsmarkierung zu welchem Methodenrumpf gehört. Der Aufbau der Environments und Tasks geschieht dabei auf folgende Weise:

- Die *Instrumentator*-Klasse trägt jeden Generator als eigenen Observer in die Klasse *ClassSearcher* ein. Diese Klasse iteriert jetzt über die eingetragenen Bäume und benachrichtigt bei Erreichen jedes Klassenknotens die eingetragenen Generatoren.

- Die Generatoren können nach Benachrichtigung durch entsprechende Navigation im Baum die benötigten Informationen herausholen. In allen Fällen werden dabei die Klassennamen, die zugehörigen Methodensignaturen und die Attribute einer Klasse generiert und ins Environment eingetragen. Die Informationen werden dabei im Hinblick auf die im Anhang angegebene Tabelle für die Routinendatei (Seite 123) und die bereits weiter oben vorgestellte Struktur der Hilfsklassen ermittelt und teilweise durch Anpassung des Klassennamens generiert.

- Die restlichen Tasks und Information gewinnen die Generatoren aus den anderen, bereits beschriebenen Klassen:

  - aus *InterfaceTransformer* die Lage der Skeletonklassen,
  - aus *IncludeTable* alle `include`-Anweisungen, die gestrichen werden müssen und
  - über *RuleRepository* und *RuleItem* die Namen und Methoden der Skeleton-Klassen.

Eine Ausnahme von diesem Verfahren ist die Klasse *XServantGenerator*. Bei dieser Klasse löst der Aufruf von *prepare()* eine direkte Anfrage beim Instrumentator aus. Über den Aufruf von *getExternalClasses()* der *Instrumentator*-Klasse kann eine Liste aller Klassen ermittelt werden, über den wiederholten Aufruf von *getExternalClassMethods()* alle Methoden zu den jeweiligen Klassen. Für diese Informationen werden jetzt wiederum die entsprechenden Klassen- und Methodengerüste mit den passenden Inhalten generiert.

Die Reihenfolge der Aufrufe von *prepare()* und *execute()* richtet sich dabei nach dem Auftauchen der betroffenen Inhalte innerhalb der Dateien. Daher wird die Klasse *XServantGenerator* vor der Klasse *ManagerGenerator* aktiviert. Ebenso müssen die Aufträge

von *ClassPreparator* und *ServantGenerator* bzw. *ListGenrator* teilweise in dasselbe *TaskList*-Objekt geschrieben werden, weil die Deklarationen innerhalb der gleichen Datei auftauchen. Die Klasse *ClassPreparator* übernimmt hier die Erzeugung der Aufträge für die änderung der Zugriffsrechte und die Einfügung zusätzlicher *get*- oder *set*-Routinen.

Schließlich zeigt Abbildung D.22 die Klassen, die der Benutzer anspricht, um seine eigenen Aufträge zu vergeben. Wie oben angegeben, kann der Benutzer mit der Klasse *InstrumentationOrder* die gewünschte Instrumentierung angeben. Diese müssen dann in die Klasse *OrderList* eingetragen werden und sorgen für die Generierung weiterer Tasks. Diese werden vor Beginn der Aufrufe der *execute*-Methoden mit *generateTasks()* in die entsprechende Taskliste eingefügt. Die Umwandlung der *OrderList*-Objekt in entsprechende Tasks geht dabei schrittweise vor sich:

- Wird für eine Klasse auch eine Instrumentierung der Superklassen gewünscht, wird über die Klasse *NodeFinder* der Klassenknoten gefunden und von dort alle Superklassen ermittelt. Für diese werden dann weitere Instrumentierungsaufträge eingetragen, sofern sie für die entsprechenden IDs nicht vorhanden sind.

- Sobald dies geschehen ist, wird für einen Auftrag der passende Knoten gefunden und in Aufträge für entsprechende Unterknoten umgewandelt. Beispielsweise wird ein Instrumentierungsauftrag für eine Klasse in entsprechende Aufträge für die enthaltenen Methoden umgewandelt, eine Auftrag für eine Datei wird auf jede enthaltene Klasse und jedes enthaltene Modul transformiert. Dies entspricht einer einfachen Abwärtsbewegung zu den jeweiligen Subknoten innerhalb des Baumes.

- Ab einem Instrumentierungsauftrag zu einer Methode wird außerdem die Klasse *InstrumentationFilter* zu Hilfe genommen. Diese filtert mittels *getNext()* (der gewöhnlichen Iterator-Methode) alle Knoten heraus, die nun individuell instrumentiert werden müssen.

Dieses Verfahren wird solange durchgeführt, bis nur noch individuelle Aufträge verbleiben. Diese können nun mit wenig Aufwand in einen entsprechenden Task umgesetzt werden (die Ermittlung des Dateinamens ist noch notwendig). Dabei wird das Attribut *helplist* verwendet, um alle Elemente zu halten, die noch umgeformt werden müssen. Gleichzeitig kann durch Analyse festgestellt werden, ob Aufträge kollidieren (sich teilweise überdecken), und ein entsprechender Kompromiß als Auftrag gespeichert werden.

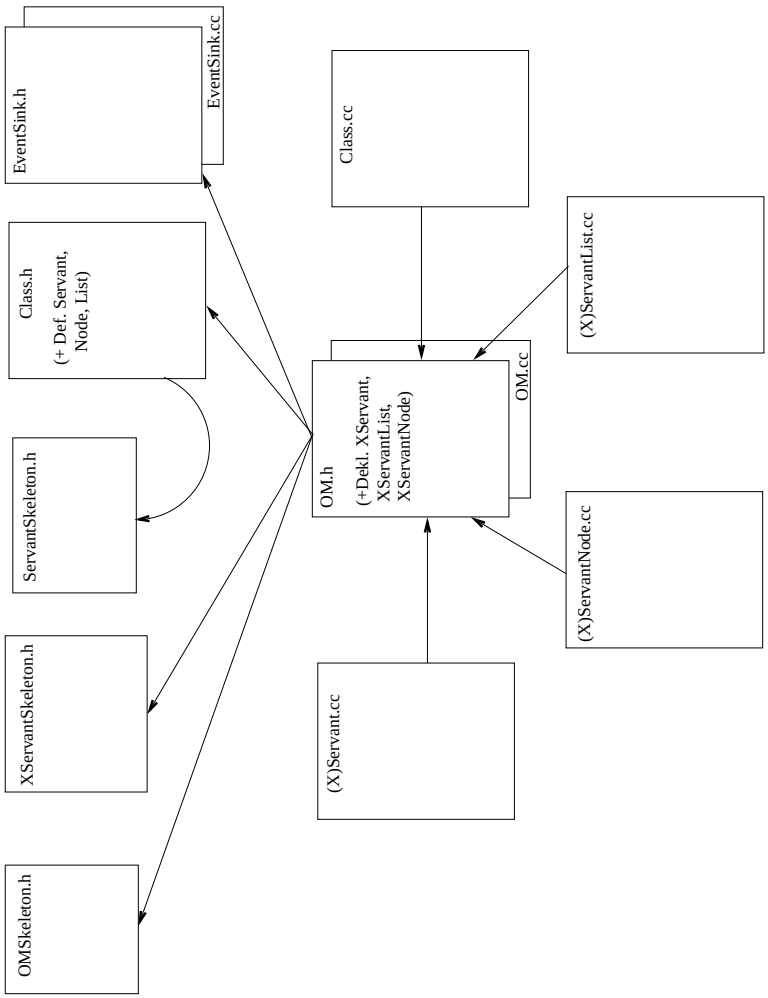

Abbildung 6.1: Verteilung einzelner Klassen auf die Dateien.

6.3 Die Benutzung der Schnittstelle zur Instrumentierung

Nachdem die Schnittstelle nun weitestgehend in ihrer Funktionalität beschrieben worden ist, sollen noch einmal die Schritte zusammengefaßt werden, die der Benutzer bei Verwendung der Schnittstelle durchführen muß. Anschließend folgt eine kurze Skizzierung, wie aus den vorhandenen Klassen ein Testwerkzeug aufgebaut werden kann.

- Der erste Schritt, der vorgenommen werden muß, ist eine Erstellung der benötigten Konfigurationsdateien: Eine Regeldatei zum Einlesen der beteiligten Klassen, eine Optionsdatei zur Festlegung des Verhaltens, eine Typendatei für alle Basistypen und die beiden Instrumentierungsdateien. Bei Bedarf müssen außerdem Anpassungen der bereits vorgegebenen Standard-Instrumentierungsdateien vorgenommen werden.

- Danach kann die Testvorbereitung vorgenommen werden. Dies umfaßt die Einrichtung der ORBs und eines Naming Services sowie das Kopieren aller Quellcode-Dateien in einen eigenen Bereich, damit die Originale nicht überschrieben werden. Der Naming Service muß bereits vor Beginn eines Testlaufs gestartet werden, damit die Adresse der Schnittstelle später als Argument übergeben werden kann.

- Jetzt kann der Benutzer durch Erzeugen eines Objekts vom Typ *FileParser* und dem Aufruf der Methoden *addFileName()*, *init()* und *parse()* die Dateinamen übergeben und deren Inhalte einlesen lassen. Fehlermeldungen kann er aus der Klasse *ErrorPool* auslesen.

- Anschließend wird die Klasse *Instrumentator* benutzt, um die übrigen Vorgänge durchzuführen: Durch *generate()* können die IDL-Dateien generiert und umgesetzt werden, durch *instrumentate()* wird die Instrumentierung durchgeführt. Auch hier finden sich entsprechende Fehlermeldungen in der Klasse *ErrorPool*. Durch *compile()* wird die Kompilierung vorgenommen. Vor Beginn dieser Vorgänge muß der Benutzer durch *chooseTestModus()* festlegen, ob die Schnittstelle die Instrumentierung für den aktiven oder den passiven Modus durchführen soll.

Nach Abschluß der Instrumentierung kann der Benutzer der Schnittstelle nun in seinem eigenen Code den ORB initialisieren und ein Objekt vom Typ *EventSink* erzeugen. Dieses muß er beim Naming Service unter der Bezeichnung `EventSink` veröffentlichen, damit der instrumentierte Code die Rückmeldung an die korrekte Stelle geben kann. Anschließend kann die instrumentierte Software als eigener Prozeß gestartet werden. Sobald die Initialisierung des zweiten ORBs und aller Schnittstellen vollständig ist, wird an dem *EventSink*-Objekt die Methode *initDone()* aufgerufen. Jetzt kann der Benutzer je nach Modus Testobjekte erzeugen und Tests durchführen oder auf die Rückmeldungen eines Applikationstests warten. Auf Ereignisse kann er reagieren, indem er eigene Subklassen vom Typ *Observer* beim *EventSink*-Objekt anmeldet.

Um ein konkretes Beispiel für den Code eines Tests zu zeigen, wird hier auf das Figurenbeispiel zurückgegriffen. Der Ausschnitt zeigt einen Test, der zum Beispiel Teil des beschriebenen *Modal Class Test* sein könnte. Der Code erzeugt ein Dreiecks-Objekt und läßt dann den Flächeninhalt berechnen, wobei das Ergebnis gespeichert wird. Die Implementation greift dabei auf das Dynamic Invocation Interface von CORBA zu, weil der Schnittstelle vor Kompilation die verwendeten Klassen nicht bekannt sind (aus diesem Grund werden die generierten Stubs nicht benötigt):

```
// statische Hilfskonstante

long TestCase::INVALID = -1

TestCase::initDone() {
  active = true;
}

TestCase::isActive() {
  return active;
}

// Der eigentliche Testfall

TestCase::run() {

    // Objekt-Manager und DynAnyFactory holen

    CORBA::Object_var adminobj =
      ns->resolve_str("GlobalObjectAdmin");
    admin = Admin::_narrow(adminobj);

    CORBA::Object_var dynfacobj =
      orb->resolve_initial_references("DynAnyFactory");

    DynamicAny::DynAnyFactory_ptr dynFac =
      DynamicAny::DynAnyFactory::_narrow(dynfacobj);

    CORBA::Long_out rid = INVALID;

    MappingTable* table = MappingTable::getTable();
    long id = table->getIdFor("Triangle");
    CORBA::Object_ptr triangle = admin->createObject(id, rid);

    // Punkte initialisieren

    CORBA::TypeCode* tc1 =
       CORBA::TypeCode::create_sequence_tc(0, CORBA::_tc_Object)

    DynamicAny::DynAny_ptr points =
    dynFac->create_dyn_any_from_type_code(tc1);
    DynamicAny::DynSequence_ptr pointseq =
      DynamicAny::DynSequence::_narrow(pointseq);

    // Punkt erstellen

    CORBA::Long_out rid2 = INVALID;
    id = table->getIdFor("Point");
    CORBA::Object_ptr p1 = admin->createObject(id, rid2);

    CORBA::Request_var r1 = p1->_request("setx");
    r1->add_in_arg("x") = 1;
    r1->invoke();
```

```
    CORBA::Request_var r2 = p1->_request("sety");
    r1->add_in_arg("y") = 1;
    r1->invoke();

    // analog für die Punkte p2 und p3 ...

    ...

    // und das Dreieck initialisieren

    DynamicAny::DynAnySeq s1;
    s1.length(3);
    DynamicAny::DynAny_ptr pointobj1 =
      dynFac->create_dyn_any_from_type_code(CORBA::_tc_Object);
    pointobj1->insert_reference(p1);
    s1[0] = pointobj1;
    DynamicAny::DynAny_ptr pointobj2 =
      dynFac->create_dyn_any_from_type_code(CORBA::_tc_Object);
    pointobj2->insert_reference(p2);
    s1[1] = pointobj1;
    DynamicAny::DynAny_ptr pointobj3 =
      dynFac->create_dyn_any_from_type_code(CORBA::_tc_Object);
    pointobj2->insert_reference(p2);
    s1[2] = pointobj3;

    pointseq->length(3);
    pointseq->set_elements_as_dyn_any(s1);

    CORBA::Request_var call = triangle->_request("setPoints");
    call->add_in_arg("points") = *(pointseq->to_any());
    call->invoke();

    // eigentlicher Testaufruf

    CORBA::Request_var call2 = triangle->_request("getArea");
    *call2->result()->value()->set_type(CORBA::_tc_double);
    call2->invoke();

    CORBA::Long res <<= *(call2->result->value());
}

// Hauptprogramm

int main(int argc char* argv[]) {
  // ORB-Initialisierung ....

  ...

  // EventSink-Objekt erstellen

  EventSink* eventobj = new EventSink();
```

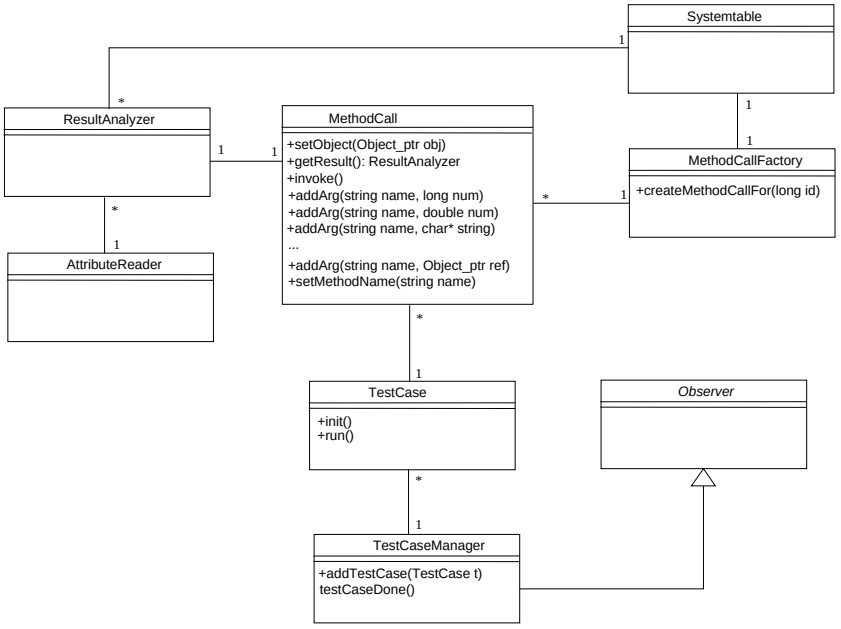

Abbildung 6.2: Klassen zur Kapselung von Testabläufen.

```
  // weitere Initialisierungen, Veroeffentlichung des
  // EventSink-Objekts...

  ...

  // Testcase ist Subklasse von Observer

  Testcase* test1 = new Testcase();

  // eintragen

  eventobj->attach(test1);

  // Software starten

  ...

  // und auf Rückmeldung warten (Polling)

  while (!(test1->isActive())) {
    if (orb->work_pending())
    orb->perform_work();
  }
  test1->run();
}
```

Auf diese Weise sind bereits handgeschriebene Tests wie z. B.bei JUnit möglich. Um für umfangreichere Tests die komplizierten Aufrufe zu vermeiden, sollten die einzelnen Aspekte gekapselt werden. Abbildung 6.2 zeigt eine mögliche Lösung: Da im Dynamic In-

vocation Interface ohnehin Strings für Methoden und Argumentnamen übergeben werden, ist es von Vorteil, diese für einen Methodenaufruf in einer Klasse *MethodCall* zu kapseln. Dabei muß die Referenz auf das CORBA-Objekt übergeben werden. Die Erzeugung kann durch eine passende Factory (*MethodCallFactory*) übernommen werden, die sich an den Einträgen in der Klasse *Systemtable* orientiert. Das Resultat kann wiederum eine Referenz auf ein CORBA-Objekt sein, das nun mit Hilfe der in der Klasse *Systemtable* enthaltenen Informationen analysiert und beispielsweise in einen String umgewandelt werden kann. Basistypen können direkt gewandelt werden. Die Klasse *ResultAnalyser* erzeugt aus den Attributknoten, die unter einem eine Klassenknoten eingeordnet sind (nach Expandierung sind dies *alle* Attribute, soweit instrumentierbar), eine Liste von Objekten der Klasse *AttributeReader*. Diese erzeugen wiederum weitere Objekte vom Typ *MethodCall*, um die *get*-Methoden des CORBA-Objekts anzusprechen. Mit Hilfe dieses Mechanismus können Verweise jetzt beliebig tief verfolgt werden. Wird darüber hinaus eine Tabelle gehalten, die die Laufzeit-ID einem Typen zuordnet, kann auch eine Referenz mit einem konkreten Typen identifiziert werden (was bei Aufruf von *getObject()* nützlich ist). Dabei ist darauf zu achten, daß nicht mehr benötigte Objekte auch wieder mit *release()* freigegeben werden.

Ein weiterer Vorteil ergibt sich durch Kapselung der Tests (die ebenfalls in Abbildung 6.2 zu sehen ist). Während ein Objekt vom Typ *TestCase* für die Durchführung eines einzelnen Tests und der Erzeugung von Objekten, Methodenaufrufen und dem Verwalten der Ergebnisse zuständig ist, hält ein Objekt der Klasse *TestManager* die Referenz auf den Objekt-Manager der zu testenden Software. Aufrufe an diese Klasse werden an den Objekt-Manager nach außen weitergegeben, während Ereignismeldungen an den korrekten Testfall zurückgegeben werden. Der Testmanager übernimmt hier die Rolle eines Mediators und kann als Subklasse von *Observer* auch die Initialisierung des ORBs vornehmen. Die Testfälle werden in einer Liste gehalten und können durch *testCaseDone()* melden, daß der nächste Testfall aktiviert werden soll.

Auch für die Analyse der Codeüberdeckung ergeben sich verschiedene Möglichkeiten. Die Messung ist bei höherer Instrumentierungsdichte genauer. Die einfachste Unterstützung eines Testwerkzeugs für Codeüberdeckung besteht in einer Kennzeichnung der erreichten Codezeilen. Eine bessere, weitaus kompliziertere Handhabung ist die Unterteilung in zwei Klassen von Punkten: Schleifeneintrittspunkte und gewöhnliche Punkte bei Verzweigung oder Anweisungsüberdeckung. Für Schleifenpunkte könnte mitgezählt werden, wie oft die Schleife durchlaufen wurde. Damit ein abruptes Verlassen einer Schleife und ein Wiedereintritt nicht zu falschen Ergebnissen führt, muß das Testwerkzeug eine History über besuchte Punkte führen. Wird zwischen dem mehrfachen Erreichen eines Schleifenpunktes ein anderer Instrumentierungspunkt erreicht, kann der Zähler für den Schleifenpunkt auf null zurückgesetzt und eine neue Zählung begonnen werden. Eine prozentuale Angabe kann grob aus der Anzahl der Gesamtzeilen für eine Methode und der Anzahl der überdeckten Zeilen errechnet werden. Diese läßt sich wiederum aus den erreichten Punkten ableiten.

Kapitel 7

Zusammenfassung und Ausblick

Das in dieser Arbeit entworfene Konzept steht innerhalb eines Spannungsfeldes verschiedener Anforderungen. Neben den in der Einleitung genannten Anforderungen kamen technische Anforderungen der einzelnen Sprachen und die Anforderungen von CORBA hinzu. Innerhalb des Konzepts sind dabei folgende Punkte der anfangs genannten Zielsetzung umgesetzt worden:

- Die Anbindung der Schnittstelle an eine konkrete Software in Java, Eiffel oder C++ ist innerhalb gewisser Grenzen möglich. Bestimmte Eigenschaften der einzelnen Sprachen (z. B. generische Klassen oder benutzderdefinierte Datentypen) wurden dabei vernachlässigt, um nicht zu viele Optionen zur Anpassung einzuführen und die Generizität zu erhalten.

- Die Architektur zur Laufzeit läßt die instrumentierten Klassen in ihren bisherigen Eigenschaften weitestgehend unberührt. Lediglich die Einführung der ID und der Methode *trueEvent* kann in Ausnahmefällen zu Problemen führen.

- Der Benutzer kann durch Zugriff auf einige wenige Klassen und die Konfigurationsdateien die Schnittstelle steuern (nach dem in [GHJV02] genannten *Facade Pattern*). Bei Bedarf ist es möglich, einzelne unterliegende Klassen direkt anzusprechen oder zu erweitern. Einige Möglichkeiten werden weiter unten angedeutet.

- Im Hinblick auf unterstützte Testverfahren sind funktionale Tests mit Messung der Codeüberdeckung möglich. Zusätzlich läßt sich durch den passiven Modus ein Applikationstest durchführen. Dies deckt einen großen Teil der in [Bin00] genannten Testverfahren ab.

Die individuellen Mappings von CORBA unterstützen teilweise den vorgestellten Ansatz, Servants mit Referenz auf ein Objekt zu verwenden (z. B. durch sinnvolle Mappings für Strings). Dies erleichterte den Entwurf einheitlicher Methodenrümpfe für die Hilfsklassen. Auch die enge Vorgabe von Interfaces für ORB-Methoden oder Mappings für IDL-Deklarationen lassen erwarten, daß zumindest für Java und C++ verschiedene ORBs zum Austausch verwendet werden können.

Es gibt aber auch einige Punkte, die sich bei diesem Konzept ungünstig auswirken:

- Die Tabellen im Anhang zeigen, daß zur Berücksichtigung einzelner Sprachmerkmale teilweise umfangreiche Regelungen durch Optionen eingeführt werden mußten,

um eine einheitliche Behandlung zu ermöglichen. Dieser Aufwand ist aber gerechtfertigt, da man sich vor Augen halten muß, daß die Generizität dynamisch ist, d. h. zur Laufzeit erreicht wird.

- Auch für kleinere Unterschiede in den CORBA-Mappings müssen einige Ausnahmeregelungen getroffen werden (z. B. für die Methode `this()`, siehe auch in der Tabelle zur Syntaxdatei im Anhang).

- Die Unterschiede zwischen den drei betrachteten Sprachen lassen außerdem darauf schließen, daß das Konzept nicht ohne weitere Anpassungen auf andere Sprachen erweitert werden kann, da jede Sprache ihre eigenen Besonderheiten hat. Das betrifft sowohl syntaktische wie auch semantische oder technische Unterschiede.

Ein Ansatz, der zumindest dem Benutzer die Handhabung erleichtert, wäre ein Werkzeug, das eine Veränderung der Optionen über eine strukturierte Menüführung erlaubt. Dabei könnte diese Menüführung auch direkt in das Testwerkzeug integriert werden. Alternativ müßten die Klassen, die die Optionen speichern, in ihrer Funktionalität erweitert werden, so daß über den Aufruf entsprechender Methoden die Optionen geschrieben oder gelesen werden können.

Im Folgenden sollen einige zusätzliche Erweiterungsmöglichkeiten der Schnittstelle genannt werden.

Eine mögliche Erweiterung betrifft die Hierarchie der Subklassen von *RuleItem*. Durch Einführung zusätzlicher Semantiken und Erweiterung des durch *bison* generierten Parsers können Einlesevorgänge abgekürzt werden. So wäre beispielsweise ein Element sinnvoll, das eine Regel fehlschlagen läßt, sobald es erkannt wird: Beispielsweise könnte

```
Attribute -> @Type @Name !"(" ";"
```

ausdrücken, daß vor dem Semikolon keine Klammer gefunden werden darf, ansonsten wird ein Mißerfolg gemeldet. Dies ermöglicht es der Schnittstelle, in C++ schneller zwischen Attribut und Funktion zu unterscheiden, so daß die Schnittstelle nicht gezwungen ist, Funktionen vor Attributen zu parsen (im ursprünglichen Entwurf ist dies der Fall, weil davon ausgegangen wird, daß Funktionsdeklarationen im allgemeinen länger sind als Deklarationen für Attribute).

Ein weiterer Aspekt ist der iterative Aufbau von Elementen. Der Einlesevorgang muß momentan bei Behebung von Fehlermeldungen der Schnittstelle durch Hinzufügen neuer Dateien komplett wiederholt werden. Bisher fehlt eine Unterstützung, die die entsprechenden Subklassen von *TreeOperation* flexibler ausnutzt und bei Angabe einer zusätzlichen Datei die neuen Informationen einfach integriert. Diese Unterstützung sollte ohne großen Aufwand möglich sein, weil für einen neuen (Datei-)Baum die Operationsreihenfolge zur Expandierung dieselbe bleibt. Darüber hinaus könnten zusätzliche Operationen bzw. Klassen vorgesehen werden, die das Laden und Speichern eingelesener Informationen ermöglichen. Dies erlaubt dann auch die Testunterbrechung und eine schnellere Fortsetzung der Tests mit einem geringen Aufwand der Testvorbereitung. Ob eine Testsuite selbst speicherbar ist, hängt von der Implementation des Benutzers ab.

Für die Durchführung des Testlaufs können ebenfalls zwei Varianten implementiert werden:

- Eine Variante betrifft überladene Konstruktoren. Um diese Konstruktoren zu berücksichtigen, muß der Aufruf von *createObject()* im aktiven Modus auf mehrere Aufrufe mit einem festen Ablaufprotokoll erweitert werden. Nach diesem Aufruf hält

der Servant noch kein konkretes Objekt. Dieses wird jetzt durch einen zusätzlichen Aufruf am Servant erzeugt, der jetzt auch die Bindung für die überladenen Konstruktoren anbietet. Der Objekt-Manager muß zu diesem Zweck eine eigene Methode anbieten, die den Servant zurückgibt, ohne den Verweis auf das eigentliche Objekt zu initialisieren. Gleichzeitig müssen alle Konstruktoren in passender Weise instrumentiert werden (d. h. durch eine entsprechende Wandlung der übergebenen Argumente). Diese Erweiterung macht die Schnittstelle zur Laufzeit weniger sicher, weil es jetzt in der Verantwortlichkeit des Benutzers liegt, ob ein Servant tatsächlich ein konkretes Objekt hält, das für einen Test verwendet werden kann.

- Die zweite Variante betrifft die Verteilung von Schnittstelle und zu testender Software. Da CORBA die Objekte unabhängig von dem tatsächlichen Ort transparent hält, ist es zur Laufzeit egal, ob die zu testende Software auf dem gleichen Rechner liegt wie die Schnittstelle oder das darauf aufbauende Werkzeug. Mit Hilfe von CORBA könnte *Remote Testing* betrieben werden: dazu trennt man die Klassen des Testwerkzeugs und die Klassen der Schnittstelle und fügt eine zusätzliche Schicht zur Kommunikation ein (die mittels mittels CORBA realisiert ist). Die Schnittstelle übernimmt in diesem Fall die Rolle eines Servers. Dabei muß nur dafür gesorgt werden, daß die Schnittstelle öffentlich erreichbar ist.

Insgesamt ist die Funktionsfähigkeit der Umsetzung an kleineren Beispielen erfolgreich getestet worden; für Regelungen, deren Aufwand nur durch umfangreiche Tests nachprüfbar gewesen wäre, wurde eine Prüfung gegen die entsprechenden technischen Anleitungen der einzelnen Tools vorgenommen. Trotz des detaillierten Entwurfs bleiben weiterhin Details innerhalb des gezeigten Konzepts zu klären, deren Prüfung nur im Rahmen einer größeren Umsetzung möglich ist, z. B. die Optimierung von Algorithmen oder Besonderheiten einzelner ORB-Implementationen, die genutzt werden. Auch im Hinblick auf eine Verallgemeinerung des Konzepts besteht noch Klärungsbedarf. Möglicherweise stellen sich einige getroffene Ausnahmen als günstig für weitere Sprachen heraus, während andere aufgelöst und in allgemeinere Regelungen überführt werden könnten. Das hier vorgestellte Konzept bietet einen guten ersten Ansatz, der durch weitere Forschung verfeinert werden kann.

Literaturverzeichnis

[ASU86] AHO, ALFRED V., RAVI SETHI und JEFFREY D. ULLMAN: *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bei90] BEIZER, BORIS: *Software Testing Techniques*. Van Nostrand Reinhold, zweite Auflage, 1990.

[Bin00] BINDER, ROBERT V.: *Testing Object-Oriented Systems*. Addison-Wesley, 2000.

[CK03] CAMPBELL-KELLY, MARTIN: *From Airline Reservations to Sonic the Hedgehog*. The MIT Press, 2003.

[Cor03] CORNWALL, JAY: *Aspect-Oriented Instrumentation & Automated Search for CPU/Storage Bottlenecks*. Technischer Bericht, Imperial College, London, 2003. http://www.doc.ic.ac.uk/∼jlc01/dyjit-ng/reports/aopi-ascsb.ppt (Juli 2004).

[DC01] DETERS, MORGAN und RON K. CYTRON: *Introduction of Program Instrumentation using AspectJ*. Technischer Bericht, OOPSLA Workshop 2001, 2001. http://www.cs.ubc.ca/∼kdvolder/workshops/OOPSLA2001/submissions/15-deters.pdf.

[Dou99] DOUGLASS, BRUCE POWELL: *Doing Hard Time*. Addison-Wesley, 1999.

[Fla02] FLANAGAN, DAVID: *Java in a Nutshell*. O'Reilly Verlag, 2002.

[GHJV02] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2002.

[HNSS00] HELKE, STEFFEN, ANDRÉ NORDWIG, THOMAS SANTEN und DEHLA SOKENOU: *Scaling-Up von V&V-Techniken durch Integration und Abstraktion*, September 2000. http://swt.cs.tu-berlin.de/Staff/SteffenHelke.html.

[JH02] JÄHNICHEN, STEFAN und STEPHAN HERRMAN: *Was, bitte, bedeutet Objektorientierung?* Informatik Spektrum/Springer Verlag, August 2002.

[Lig90] LIGGESMEYER, PETER: *Modultest und Modulverifikation*. Helmut Balzert/ Bibliographisches Institut and F.A. Brockhaus AG, Mannheim, 1990.

[Lou03] LOUIS, DIRK: *C/C++ - Die praktische Referenz*. Markt+Technik, 2003.

[Mey97] MEYER, BERTRAND: *Object-oriented Software Construction*. Prentice Hall, zweite Auflage, 1997.

[MH02] MONK, SIMON und STEVEN HALL: *Virtual Mock Objects using AspectJ with JUNIT*. XP Magazine 2002, Oktober 2002. Verfügbar unter http://www.xprogramming.com/xpmag/virtual (Juli 2004).

[OMG02a] OMG: *Common Object Request Broker Architecture: Common Specification (Version 3.0)*, Dezember 2002. http://www.omg.org/cgi-bin/doc?fomal/04-03-01 (Juli 2004).

[OMG02b] OMG: *IDL to Java Language Specification*, August 2002. http://www.omg.org/cgi-bin/doc?formal/02-08-05.

[OMG03] OMG: *C++ Language Mapping Specification*, Juni 2003. http://www.omg.org/cgi-bin/doc?formal/03-06-03.

[OMG04] OMG: *UML 1.5 Specification*, Juli 2004. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[Som02] SOMMERVILLE, IAN: *Software Engineering*. Pearson Studium, sechste Auflage, 2002.

[Sun04] SUN MICROSYSTEMS: *Java Virtual Machine Profiler Specification*. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html, Juli 2004.

[Swi02] SWITZER, ROBERT: *The Manual of mico/E*, Juli 2002. http://www.math.uni-goettingen.de/micoe.

[Szy02] SZYPERSKI, CLEMENS: *Component Software – Beyond Object-Oriented Programming*, Kapitel 17. Addison-Wesley, zweite Auflage, 2002.

[TH02] TIKIR, MUSTAFA M. und JEFFREY K. HOLLINGSWOOD: *Efficient Instrumentation for Code Coverage Testing*, 2002. http://www.cs.umd.edu/∼tikir/papers/issta02.pdf (August 2004).

[van04] VAN EMMERICK, MIKE: *Is Decompilation possible?* Technischer Bericht, Program-Transformation.Org, http://www.program-transformation.org/Transform/DecompilationPossible, Juli 2004.

[Wes01] WESTPHAL, RALF: *.NET kompakt*. Spektrum Verlag, 2001.

[Whe04] WHEELER, DAVID: *Counting Source of Lines Code*. http://www.dwheeler.com/sloc/, Juli 2004.

[Wor04a] WORLD WIDE WEB CONSORTIUM: *SOAP Tutorial*, Juli 2004. http://www.w3.org/TR/2003/REC-soap12-part0-20030624.

[Wor04b] WORLD WIDE WEB CONSORTIUM: *XML Tutorial*, Juli 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

Teil III

Anhang

Anhang A

Optionen der Optionsdatei

Die folgende Tabelle listet alle Optionen der Optionsdatei auf, die die Schnittstelle in ihrem Verhalten beeinflussen.

Option	Bedeutung	Standardwert
nonIdentifierChars	enthält alle Zeichen, die in der Sprache nicht Teil eines Bezeichners (Klassennamen, Typen etc.) sind. Mit Hilfe dieser Option erkennt die Schnittstelle, wann Bezeichner beginnen und enden.	`" \n\t"`
ModuleByDeclaration	Gibt an, ob ein Modul durch einfache Deklaration global für die gesamte Datei gilt. Dies ist zur Unterscheidung zwischen C++ und Java wichtig.	`"no"`
ModuleByDirectory	Gibt an, ob Module durch Verzeichnisse gegeben sind. Dies entspricht dem Mapping für Java und Eiffel.	`"no"`
ExplicitInclude	Gibt an, ob ein Datei-Import durch explizite Angabe erfolgt oder ein anderer Mechanismus dafür zuständig ist. Dies ist für die spätere Elementzuordnung wichtig.	`"no"`
IncludeTransitiv	Gibt an, ob ein Datei-Import auch transitiv fortgesetzt wird, wenn die importierte Datei wiederum Dateien importiert. Auch diese Option ist für die Elementzuordnung wichtig.	`"no"`
ModuleSeparator	Gibt den Trennstring für geschachtelte Module an, z. B. `"."` für Java, `"::"` für C++.	`"."`
ClassSeparator	Gibt den Trennstring zwischen Klassen- und Modulnamen bzw. Klassen- und Methodennamen an.	`"."`
ModuleImportSymbol	Gibt den String für den Import von gesamten Modulen an. Für Java ist dies die Angabe `"*"`, wie z. B. in `import java.lang.*`.	`""`
StandardModuleAccess	Gibt an, welches Zugriffsrecht für Klassen innerhalb von Modulen gelten soll, wenn keine explizite Deklaration erfolgt. Mögliche Werte sind `"public"`, `"private"` und `"module"` (also nur Zugriff innerhalb des gleichen Moduls oder der gleichen Datei).	`"public"`
StandardClassAcess	Gibt an, welches Zugriffsrecht für Methoden und Attribute einer Klasse gilt, wenn keine weiteren Angaben zum Zugriff gemacht werden. Zusätzlich zu den unter *StandardModuleAccess* angegebenen Möglichkeiten kommt noch `"protected"` hinzu	`"private"`

Option	Bedeutung	Standardwert
MethodHoldsClassName	Zeigt an, ob eine Methode in einer Datei separat mit Angabe der Klasse definiert wird. Die Deklaration wird von der Schnittstelle innerhalb der Klasse gesucht.	`"no"`
ClassNameForConstructor	Gibt an, daß der Konstruktor den Namen der Klasse trägt.	`"yes"`
ConstructorName	Gibt einen einheitlichen Namen für einen Klassenkonstruktor an.	`""`
DestructorPrefix	Diese Option gibt im Zusammenhang mit *ClassNameForConstructor* den Präfix für den Destruktor an (an den der Klassenname gehängt wird).	`""`
PointerPostfix	Diese Option gibt den Postfix für Pointer an (für C++ also `"*"`), der an Typen gehängt wird.	`""`
ReferencePostfix	Diese Option gibt den Postfix für Referenzen an (für C++ also `"&"`), der an Typen gehängt wird.	`""`
BaseDirectory	Diese Option gibt das gemeinsame Basisverzeichnis an, unter dem alle Quellcode-Dateien abgelegt worden sind. Diese Option darf nicht weggelassen werden.	keine
DeclarationFileEnding	Gibt die Endung für die Datei an, die die Deklarationen enthält. Ist diese Endung gleich der Endung von *DefinitionFileEnding*, geht die Schnittstelle davon aus, daß Deklaration und Definition in einer Datei erfolgen.	`""`
DefinitionFileEnding	Gibt die Endung für die Dateien an, in der die Definitionen einer Klasse stehen.	`""`
CompilerName	Gibt den Namen des Compilers an, der zur Umsetzung der Quellcode-Dateien notwendig ist.	`""`
CompilerOptions	Gibt (z. B. für CORBA) notwendige Optionen an, mit denen der Compiler aufgerufen werden muß.	`""`
LinkerName	Gibt den Namen des Linkers an, der zur Umsetzung der Quellcode-Dateien notwendig ist.	`""`
LinkerOptions	Gibt (z. B. für CORBA) notwendige Optionen an, mit denen der Linker aufgerufen werden muß.	`""`
IDLCompilerName	Gibt den Namen des IDL-Compilers an.	`""`
IDLCompilerName	Gibt Optionen für den IDL-Compiler an (z. B. Benutzung des POA).	`""`
TargetName	Gibt den Namen des auszuführenden Programms an, in das der Quellcode umgesetzt werden soll.	`"TestMain"`
RunTimeOptions	Gibt die Optionen an, die an das laufende Programm übergeben werden müssen.	`""`
MainLocation	Gibt die Stelle an, an der das Hauptprogramm steht. Erlaubt sind nur die Werte `"function"` (globale Funktion innerhalb der Datei des Objekt-Managers) oder `"method"` (Methode innerhalb des Objekt-Managers)	`"method"`
MainIsConstructor	Gibt an, ob das Hauptprogramm ebenfalls Konstruktor des Objekt-Managers ist.	`"no"`
InitArgCount	Gibt an, ob für die an den ORB übergebenen Initialisierungselemente auch die Anzahl separat übergeben werden muß	`"no"`

Anhang B

Feste Regelnamen innerhalb der Regeldatei

Dieser Abschnitt beschreibt die Regelnamen, deren Bedeutung fest vergeben sind. Auf den Namen einer Regel folgen dann die Elemente, die innerhalb der Regel verwendet werden können, wobei die letzte Spalte die jeweilige Bedeutung der Regel oder des Elements erläutert. Einige Elemente oder Regeln haben im Zusammenhang mit einigen Optionseinstellungen unterschiedliche Auswirkungen. Die Optionen können im vorhergehenden Abschnitt nachgeschlagen werden.

Regelname	Element	Bedeutung
FileImport		Bezeichnet den Import einer Datei (also eine `#include`-Anweisung in C++). Diese Regel kommt nur dann zum Tragen, wenn die Option `ExplicitImport` auf `"yes"` gesetzt wird. Ansonsten geht die Schnittstellen davon aus, daß sich alle angegebenen Dateien gegenseitig importieren.
	@FileName	Bezeichnet den importierten Dateinamen.
Module		Diese Regel identifiziert ein Modul (d. h. einen Namespace in C++ oder ein Package in Java). Existieren in der Sprache keine Module, kann die Regel weggelassen werden. In diesem Fall wird der entsprechende Einlesevorgang durch die Schnittstelle übersprungen.
	@ModuleName	Bezeichnet die Stelle, an der der Name des Moduls steht. Durch dieses Argument werden auch komplexere Bezeichner erfaßt, die dann durch die Option `ModuleSeparator` in der Konfigurationsdatei zerlegt wird.
	^BEGIN	Kennzeichnet den Beginn eines Moduls. Diese Angabe ist unwirksam, wenn die Option `DeclareModuleByName` auf `"yes"` gesetzt ist.
	^END	Kennzeichnet das Ende eines Moduls. Diese Angabe ist unwirksam, wenn die Option `DeclareModuleByName` auf `"yes"` gesetzt ist.
ModuleImport		Gibt den Import eines Moduls oder Modulelements an (`import` in Java, `using` in C++). Analog zur Modulregel kann auch diese Regel weggelassen werden.
	@ModuleName	Gibt die Stelle an, an der der Bezeichner des importierten Moduls steht. Analog zur Modulregel können hier ebenfalls komplexere Bezeichner erfaßt werden.
Class		Diese Regel bezeichnet eine Klasse (die auch teilweise abstrakt sein kann). Ein Fehlen dieser Regel führt zu einem Fehler.

Regelname	Element	Bedeutung
	@ClassName	Gibt den Namen der Klasse an.
	@SuperClass	Bezeichnet die Stelle, an der ein Bezeichner ein Superklasse auftauchen kann. Durch entsprechende Formulierung können mit Hilfe diese Elements auch mehrere Superklassen eingelesen werden.
	^BEGIN	Dieses Element kennzeichnet den Beginn des Klassenbereichs (also in C++ die Stelle der öffnenden geschweiften Klammer).
	^END	Dieses Element kennzeichnet das Ende des Klassenbereichs.
	@ExportName	Dieses Element dient dazu, eine Klasse zu halten, an die ein Attribut oder eine Methode einer anderen Klasse individuell exportiert wird. Dieses Argument dient dazu, die Schreibweise von Eiffel abzudecken.
	@TargetStart, @TargetEnd	Diese Elemente geben den Bereich an, der mittels mehrerer durch *ExportName* angegebenen Exporte abgedeckten Bereich an. Auch diese Elemente sind nur für Eiffel notwendig.
	@RenamedMethod	Dieses Element gibt den Namen einer umbenannten Methode an, die bei Mehrfachvererbung notwendig ist. Diese Angabe wird für Eiffel gebraucht. Der Name wird dabei auf die im Programmtext vor der Umbenennung auftauchende Methode bezogen.
	@NewName	Dieses Element gibt im Zusammenhang mit *RenamedMethod* den neuen Namen einer umbenannten Funktion an.
	^PUBLIC_METHOD_DECL, ^PUBLIC_METHOD_DECL_END, ^PUBLIC_CLASS_DECL	Diese Elemente dienen nur der Markierung, daß eine Methode oder Klasse als `public` deklariert wurde. Die Position dieses Elements (Klasse oder Methode) wird im entsprechenden Knoten gespeichert, um später einen schnellen Zugriff auf die Deklaration der Zugriffsrechte zu bieten.

Regelname	Element	Bedeutung
	^PROTECTED_METHOD_DECL, ^PROTECTED_METHOD_DECL_END, ^PROTECTED_CLASS_DECL	Diese Elemente dienen nur der Markierung, daß eine Methode oder Klasse als `protected` deklariert wurde. Analog zu den beiden vorhergehenden Elementen wird auch hier die Position im entsprechenden Knoten vermerkt.
	^PRIVATE_METHOD_DECL, ^PRIVATE_METHOD_DECL_END, ^PRIVATE_CLASS_DECL	Analog zu den vorhergehenden beiden Zeilen dienen diesen beiden Elemente der Deklaration eines privaten Zugriffs.
AttributeDeclaration		Diese Regel gibt das Aussehen eines Klassenattributs an.
	@Type, $Type	Dieses Element gibt den Typen eines Attributs an. Dabei bezeichnet die erste Variante einen Klassentypen, die zweite einen Basistypen (s. auch *MethodDeclaration*).
	@Name	Dieses Element gibt den Namen des Attributs an.
MethodDeclaration		Diese Regel bezeichnet eine Methodendeklaration oder eine abstrakte Methode . Innerhalb eines Klassenbereichs wird nach einer Methodendeklaration gesucht; wenn `MethodHoldsClassName` gesetzt ist, innerhalb der Datei nach der passenden Definition. Ist diese Option nicht gesetzt, wird ebenfalls innerhalb der Klasse nach Methodendefinitionen gesucht.
	@Result / $Result	Jedes der beiden Elemente kennzeichnet die die Stelle des Rückgabetyps. Im Fall von @Result erwartet die Schnittstelle einen Klassenbezeichner, im anderen Fall einen Basistypen. Im Normalfall wird man beide Elemente in der Form ($Result | @Result) verwenden, um zuerst einen Basistypen zu matchen, bei Fehlschlag eine Klasse.
	@MethodName	Bezeichnet die Stelle des Methodennamens.

Regelname	Element	Bedeutung
	@ArgType / $ArgType	Analog zum Resultat einer Methode wird mit diesen beiden Elementen der Typ eines Arguments für eine Methode festgelegt. Durch Zuordnung der Positionen kann bei mehreren Argumenten auch die Reihenfolge der Argumente durch die Schnittstelle erkannt werden.
	@ArgName	Bezeichnet den Namen eines Arguments. Die Zuordnung zum Typen erfolgt dabei ebenfalls nach Position, d. h. der erste gefundene Argumentname wird dem ersten gefundenen Argumenttypen zugeordnet.
	^PUBLIC_DECL	Dieses Element wird wie in der Klassenregel zur Angabe der Zugriffsrechte für die Methode verwendet.
	^PROTECT_DECL	Dieses Element wird wie in der Klassenregel zur Angabe der Zugriffsrechte für die Methode verwendet.
	^PRIVATE_DECL	Dieses Element wird wie in der Klassenregel zur Angabe der Zugriffsrechte für die Methode verwendet.
Method		Diese Regel dient der Angabe einer kompletten Methode mit Rumpf. Alle aus *MethodDeclaration* verwendeten Elemente besitzen hier ebenfalls Gültigkeit, zusätzlich kommen die Elemente ^BEGIN und ^END hinzu, die den Beginn und das Ende des Methodenrumpfes angibt. Auch das Fehlen dieser Regel führt zu einem Fehler.
Conditional		Diese Regel gibt die Schreibweise einer Verzweigung an (kein `switch`!).
	^BEGIN-COND	Gibt den Beginn eines Bedingungsausdrucks an, über den die Verzweigung entscheidet. Bei mehreren geschachtelten Verzweigungen (wie bei Eiffel mittels `elsif`) können auch mehrere Bedingungen angegeben werden.
	^END-COND	Gibt das Ende des jeweiligen Bedingungsausdrucks der Verzweigung an.

Regelname	Element	Bedeutung
	^BEGIN	Gibt den Beginn der jeweiligen Alternative an (auch hier können mehrere Alternativen auftreten).
	^END	Gibt das Ende der jeweiligen Alternative an.
Switch		Diese Regel gibt die Schreibweise einer `switch`-Anweisung aus Java oder C++ wieder.
	^BEGIN	Gibt den Beginn eines case-Konstruktes an.
	^END	Gibt das Ende eines case-Konstruktes an.
Loop		Mit dieser Regel wird die Schreibweise für Schleifen angegeben. Auch hier führt das Fehlen der Regel zu einer Fehlermeldung.
	^BEGIN	Gibt den Beginn des Schleifenrumpfs an.
	^END	Gibt das Ende des Schleifenrumpfs an.
	^BEGIN-PRED	Mit diesem Element wird die Stelle angegeben, an der das Prädikat beginnt, das für einen erneuten Schleifendurchlauf geprüft wird.
	^END-PRED	Mit diesem Element wird die Stelle angegeben, an der das Prädikat endet, das für einen erneuten Schleifendurchlauf geprüft wird.
ExceptionBlock		Gibt einen Block an, der zum Exception Handling vorgesehen ist. Die Schnittstelle kann dies benutzen, um eine interne Exception innerhalb der zu testenden Software zu erkennen.
	^BEGIN-TRY	Dieses Element wird nur gebraucht, um den Einlesevorgang der Schnittstelle korrekt zu steuern, und gibt den Beginn eines `try`-Blocks an.
	^END-TRY	Analog zum vorhergehenden Element wird durch dieses Element das Ende des `try`-Blocks gekennzeichnet.

Regelname	Element	Bedeutung
	^BEGIN	Dieses Element bezeichnet den Beginn eines Blocks, der das tatsächliche Exception Handling übernimmt (z. B. `catch` in Java und C++ und `rescue` in Eiffel).
	^END	Dieses Element bezeichnet das Ende des Blocks für Exception Handling.
Statement		Diese Regel bezeichnet eine einzelne atomare Anweisung (z. B. eine Zuweisung, einen Methodenaufruf oder eine `return`-Abweisung). Für diese Regeln gibt es keine verpflichtenden Argumente, allerdings darf auch diese Regel innerhalb des Regelsatzes nicht fehlen.
ReturnStatement		Diese Regel gibt die Rücksprungsanweisung der entsprechenden Sprache an und ist optional. Fehlt die Regel, geht die Schnittstelle davon aus, daß Rückgabewerte durch Zuweisung an eine spezielle Variable geschehen. Zusätzlich Argumente sind nicht vorgesehen, daher wird die Regel nur verwendet, um die Stelle eines Rücksprungs für die spätere Instrumentierung zu speichern.
AndPredicate		Diese Regel bezeichnet eine Und-Verknüpfung und ist für eine feinere Bestimmung der Codeüberdeckung wichtig. Durch ^PRED und ^END-PRED können die Bereiche für untergeordnete Prädikate angegeben werden, die dann im nächsten Durchgang auf ihre Zusammensetzung durchsucht werden können.
OrPredicate		Analog zur Und-Verknüpfung bezeichnet diese Regel die Oder-Verknüpfung. Die Elemente sind die gleichen und haben dieselbe Bedeutung.
NotPredicate		Analog zur Und-Verknüpfung bezeichnet diese Regel eine nicht-atomare Negierung. Die Elemente sind die gleichen und haben dieselbe Bedeutung.

Anhang C

Angaben innerhalb der Instrumentierungsdateien

Die folgenden Tabellen geben die Angaben innerhalb der Instrumentierungsdateien wieder. Die erste Tabelle zeigt alle erlaubten Syntaxkonstrukte innerhalb der Routinendatei. Dabei stehen die Bezeichner in den Klammern nur als Platzhalter für tatsächliche Werte. Ausnahmen sind die Klassenparameter, bei denen die Bezeichner `Class`, `Servant`, `ServantList` und `ServantNode` feststehend sind.

Die nachfolgende Tabelle zeigt die Schlüsselwörter für die Routinendatei. Zu jedem Schlüsselwort wird angegeben, welcher Routine es entspricht, welche Aufgabe diese Routine im aktiven oder passiven Modus besitzt und welche Parameter erwartet werden. Zusätzlich enthält die Datei alle Wandlungsroutinen für Basistypen, wie es bereits im Abschnitt 6.1 beschrieben worden ist. Diese Teile werden für alle Servant-Methoden verwendet, die die eigentlichen Objektmethoden kapseln. Im aktiven Modus wird dabei innerhalb der Methode eine Wandlung der übergebenen Argumente und eine Rückwandlung des Resultats vorgenommen. Im passiven Modus werden diese Teile dazu benutzt, bei der Methode *refresh()* eines Servants die Attribute korrekt zu wandeln und zu speichern.

Die letzte Tabelle zeigt alle Angaben innerhalb der Syntaxdatei mit den zugehörigen Argumenten. Gezeigt wird dabei nur das einleitende Tag, das abschließende Tag wird mit einem einleitenden Schrägstrich begonnen. Zwischen den Tags werden Argumente mit @, Argumentkollektionen mit $ angegeben und normale Zeichenketten in Anführungszeichen eingeschlossen. Die Schnittstelle geht außerdem davon aus, daß der Additionsoperator durch + repräsentiert wird.

Ausdruck	Bedeutung
$$	Eingabewert. Dieser Ausdruck bezeichnet den Eingabewert einer Transformation bei Wandlung eines Basistypen. Hier setzt die Schnittstelle je nach Kontext den Namen eines Methodenarguments, eines Attributs oder einer lokalen Variablen ein.
$%	Ausgabewert. Dieser Ausdruck wird normalerweise in einer Zuweisung benutzt (`assign`) und vom Interpreter je nach Kontext in eine `return`-Anweisung oder eine Zuweisung zu einer Variablen umgesetzt.
`<paramName:Servant>`	Servant-Klasse zu einem Klassenparameter
`<paramName:ServantNode>`	Node-Klasse zu einem Klassenparameter
`<paramName:ServantList>`	List-Klasse zum Klassenparameter
`<paramName:ListAttribute>`	Listen-Attributname zu einem Klassenparameter innerhalb des Objekt-Managers
`<Class: paramName>`	Name eines Klassenparameters
`<paramName>`	allgemeiner Parameter einer Routine
`VOID`	`void`, `NULL` oder `null`
`condition(Bedingung)(If)(Else)`	Verzweigung
`loop(Init)(Bed)(Rumpf)`	Schleife
`decl(Typ)(Name)`	lokale Deklaration
`corbadecl(Typ)(Name)`	lokale Deklaration eines CORBA-Typs
`arraydecl(Typ)`	lokale Deklaration eines Arrays
`assign(Name)(Ausdruck)`	Zuweisung
`access(Name)(Index)`	Array-Zugriff
`arrayconstruct(Name)(Lower)(Upper)(Length)`	Konstruktoraufruf eines Arrays
`arraydelete(Name)`	Löschen eines Arrays
`equals(Ausdruck)(Ausdruck)`	Vergleich auf Gleichheit
`differs(Ausdruck)(Ausdruck)`	Vergleich auf Ungleichheit
`call(Objekt)(Name)((Argument)(Argument) ...)`	Methodenaufruf

Ausdruck	Bedeutung
`callstatic(Klasse)(Name)((Argument)(Argument) ...)`	statischer Methodenaufruf
`delete(Name)`	Löschen einer Variablen. Dies entspricht einem `delete` in C++, d. h. in Java und Eiffel einer einfachen Zuweisung von `null` bzw. `void`.
`and(Bed1)(Bed2)`	Und-Verknüpfung
`or(Bed1)(Bed2)`	Oder-Verknüpfung
`not(Bed)`	Negierung
`construct(Name)(Typ)(Args)`	Konstruktoraufruf
`narrow(Quelle)(Ziel)`	Narrowing-Operation mit CORBA-Typen

Tabelle C.1: Inhalte der Syntaxdatei

Schlüsselwort	Parameter	Bedeutung
`ReachedPoint`	keine	Diese Routine meldet sowohl im aktiven wie auch im passiven Modus das Erreichen eines instrumentierten Punktes an die Klasse *EventSink*.
`Init`	keine	Diese Routine gibt alle Aufrufe der *init*-Methode des Objekt-Managers wieder, die dessen feste Referenzen initialisieren (EventSink, ORB etc.) und die Initialisierung des ORBs vornehmen (inklusive der Rückmeldung an die Schnittstelle). Im aktiven Modus ist diese Routine Hauptprogramm (bei Eiffel) oder wird durch als statische Methode bei Programmbeginn aufgerufen, im passiven Modus wird der Init beim ersten Zugriff auf den ORB ausgeführt. Im voraus fügt die Schnittstelle einen Konstruktoraufruf für jede Liste (vom Typ *TList*) ein.
`ObjectDeleted`	keine	Diese Routine entspricht der gleichlautenden Methode des Objekt-Managers. Die Methode meldet sowohl im aktiven als auch im passiven Modus das Löschen eines Objektes an die Klasse *EventSink*.

Schlüsselwort	Parameter	Bedeutung
`ObjectCreated`	keine	Diese Routine entspricht der gleichlautenden Methode des Objekt-Managers. Die Methode meldet sowohl im aktiven als auch im passiven Modus das Löschen eines Objektes an die Klasse *EventSink*.
`Activate`	keine	Diese Routine entspricht der Methode gleichen Namens im Objekt-Manager, die sowohl im aktiven als auch im passiven Modus eine Servant im POA aktiviert.
`Deactivate`	keine	Diese Routine ist das Gegenstück zur vorhergehenden (der Servant wird wieder aus der Map des POA ausgetragen).
`GetServantList`	`Class` (Klassenparameter)	Diese Routine entspricht der Methode *getTServantList()* im Objekt-Manager. Sowohl im aktiven wie auch im passiven Modus dient diese Methode dazu, die entsprechende List zurückzugeben, damit auf Anfrage ein passender Servant zu einem Objekt innerhalb dieser List aktiviert werden kann. Der Parameter gibt an, zu welcher Klasse diese Routine gehört.
`ObjectChanged`	keine	Diese Routine entspricht der *objectChanged*-Methode des Objekt-Managers und meldet im passiven Modus zurück, daß ein Objekt unter Umständen verändert worden ist.
`CreateObject`	`Id`, `Class` (Klassenparameter)	Diese Routine entspricht einem Fall der Fallunterscheidung innerhalb der Methode *createObject()* des Objekt-Managers. Als Paramter muß die Klassen-Id und die zugehörige Klasse übergeben werden. Im passiven Modus spielt diese Routine keine Rolle und wirft eine Exception.
`CreateNilRef`	`Id`, `Class` (Klassenparameter)	Diese Routine entspricht analog zur vorhergehenden einem Teil der Methode *createNilRef()* des Objekt-Managers und gibt eine CORBA-Nullreferenz zurück. Analog zur vorhergehenden Methode wird auch hier im passiven Modus eine Exception geworfen.

Schlüsselwort	Parameter	Bedeutung
`ReleaseObject`	`Class` (Klassenparameter)	Diese Routine ist ein Teil der Objekt-Manager-Methode *releaseObject*. Dieser Teil fragt eine Liste nach der übergebenen ID und ruft bei Erfolg *deleteServant()* auf der entsprechenden Liste auf. Sowohl im aktiven wie auch im passiven Modus wird damit eine CORBA-Referenz aufgelöst.
`GetObject`	`Class` (Klassenparameter)	Analog zur vorhergehenden Routine stellt diese Routine einen Teil der Methode *getObject()* des Objekt-Managers dar. Dieser Teil ruft *findServant* auf der Liste des übergebenen Klassenparamters auf und gibt bei Erfolg den entsprechenden Servant zurück. Die Schnittstelle fügt für den Mißerfolgsfall eine Exception am Ende der Methode hinzu.
`castObject`	`Id`, `Class` (Klassenparameter)	Diese Routine gibt einen Teil der Methode *castObject()* des Objekt-Managers an. Dieser Teil sorgt für die korrekte Zuordnung einer bestimmten ID zu einer Liste und ruft an dieser *createCastedServant()* auf.
`Shutdown`	keine	Diese Routine sorgt für das Herunterfahren des ORBs, wenn der Test abgeschlossen ist und entspricht der Methode *shutdown()* im Objekt-Manager.
`GetAdminIntern`	keine	Diese Routine entspricht der *getAdmin*-Methode des Objekt-Managers und liefert den Administrator zurück. Diese Methode ist entweder statisch oder eine once-Methode (in Eiffel).
`GetAdminExtern`	keine	Diese Routine entspricht der *getAdmin*-Methode der Klasse *AdminAccessor*. Diese Routine ruft die *getAdmin*-Methode des Objekt-Managers auf.
`GetIDAdmin`	keine	Diese Routine entspricht der *getId*-Methode des Objekt-Managers und liefert eine neue ID für ein Objekt zurück.
`GetIdObject`	keine	Diese Methode entspricht der *getId()*-Methode einer instrumentierten Klasse und gibt die enthaltene Laufzeit-ID zurück.

Schlüsselwort	Parameter	Bedeutung
`ConstructorObjectBegin`	`Class` (Klassenargument)	Diese Routine stellt den Teil dar, der in den Konstruktor einer instrumentierten Klasse am Anfang eingefügt wird. Im aktiven Modus holt dieser Konstruktor eine passende Laufzeit-ID, im passiven Modus wird zusätzlich der Servant konstruiert und gehalten.
`ConstructorObjectEnd`	keine	Diese Routine stellt den Teil dar, der ans Ende eines Konstruktors innerhalb einer instrumentierten Klasse eingefügt wird. Im aktiven Modus wird nur *objectCreated()* zurückgemeldet, im passiven Modus wird zusätzlich vorher der Servant mittels *refresh* aktualisiert.
`DestructorObject`	`Class` (Klassenparameter)	Diese Routine wird am Ende des Destruktors eingefügt (soweit vorhanden). Diese Routine holt in beiden Modi vom Objekt-Administrator die passende Liste und ruft dort *deleteServant()* auf. Darüber hinaus wird *objectDeleted()* aufgerufen.
`ConstructorServant`	keine	Diese Ruotine gibt den Konstruktor des Servants an.
`GetIdServant`	keine	Diese Routine gibt die *getId*-Methode des Servants an. Die Laufzeit-ID entspricht der des gehaltenen Objekts.
`SetImpl`	keine	Diese Routine entspricht der *setImpl*-Methode im Servant und setzt die Referenz des Servants auf das eigentliche Objekt, das am Test beteiligt ist.
`GetImpl`	keine	Diese Routine entspricht der *getImpl*-Methode des Servants und liefert die Referenz auf das Objekt zurück. Diese Routine ist nur im aktiven Modus von Bedeutung, im passiven Modus wird sie nicht benutzt.
`ConstructorServantNode`	keine	Diese Routine entspricht dem Konstruktor der Servant-Knotenklasse. Im Normalfall werden die Attribute *value* und *next* auf VOID gesetzt.
`SetValue`	keine	Diese Routine entspricht der *setValue*-Methode der Servant-Knotenklasse. Mit ihr wird der entsprechende Verweis auf den Servant des Knotens gesetzt.

Schlüsselwort	Parameter	Bedeutung
`GetValue`	keine	Diese Routine ist das Gegenstück zur vorhergehenden und liefert den Servant des Elements zurück.
`SetNext`	keine	Diese Routine entspricht der *setNext*-Methode der Servant-Knotenklasse. Mit Hilfe dieser Methode wird der Verweis auf den nächsten Knoten gesetzt.
`GetNext`	keine	Diese Routine ist das Gegenstück zur vorhergehenden und gibt die Referenz auf das nächste Objekt zurück.
`ConstructorServantList`	keine	Diese Routine entspricht dem Konstruktor der Servant-Listenklasse und initialisiert den Parameter *start*.
`CreateServant`	`Class` (Klassenparameter)	Diese Routine entspricht der *createServant*-Methode der Servant-Listenklasse. Diese Routine wird im aktiven Modus dazu verwendet, ein Objekt mit passendem Servant zu erzeugen und innerhalb der Liste abzulegen.
`CreateServantFor`	`Class` (Klassenparameter)	Diese Routine entspricht der *cerateServantFor*-Methode der Servant-Listenklasse und wird sowohl im aktiven als auch im passiven Modus dazu verwendet, einen Servant für ein bereits existierendes Objekt zu erzeugen (beispielsweise bei Abfrage eines Attributs eines Objektes).
`FindServant`	`Class` (Klassenparameter)	Diese Routine steht für die *findServantFor*-Methode der Servant-Listenklasse. Diese Routine wird gebraucht, wenn von der Schnittstelle eine Objektreferenz als Argument übergeben wird, um das zugehörige Objekt zu finden. Im passiven Modus wird diese Methode nicht benutzt.
`DeleteServant`	`Class` (Klassenparameter)	Diese Routine entspricht der *deleteServant*-Methode der Servant-Listenklasse und wird dazu benutzt, einen eingetragenen Servant wieder zu entfernen.

Tabelle C.2: Inhalt der Routinendatei

Tag	Variablen	Bedeutung
<equality-operator>	*keine*	Der Gleichheits-Operator bezieht sich sowohl auf den Vergleich zwischen Objekten als auch Basistypen
<diff-operator>	*keine*	Analog zum Gleichheits-Operator definiert dieses Tag den Ungleichheits-Operator.
<and-operator>	*keine*	Dieses Tag definiert die logische Und-Verknüpfung.
<not-operator>	*keine*	Dieses Tag legt die logische Negierung fest.
<assignment-operator>	*keine*	Dieses Tag definiert den Zuweisungs-Operator für Objekte und Basistypen.
<arg-decl-separator>	*keine*	Dieses Tag legt das Trennzeichen für Argumente einer Methode bei der Deklaration fest.
<arg-call-separator>	*keine*	Dieses Tag definiert das Gegenstück zum vorhergehenden. Mit Hilfe dieses Tags wird angegeben, wie Argumente beim Methodenaufruf getrennt werden.
<dereference-operator>	*Name*	Dieses Tag beschreibt den Operator zur Dereferenzierung eines Pointers. Dabei wird der Operator als Präfix-Operator aufgefaßt.
<adress-operator>	*Name*	Dieses Tag repräsentiert den Operator zur Umwandlung eines Objektes oder einer Referenz in einen Pointer. Auch dieser Operator wird als Präfix-Operator aufgefaßt.

Tag	Variablen	Bedeutung
`<return-statement>`	*Value*	Dieses Tag stellt das Äquivalent der Anweisung dar, mit der die Sprache ein Resultat zurückgibt. In Eiffel muß hier eine Zuweisung an die spezielle Variable *Result* stehen.
`<type-Declaration>`	*Type, Name*	Dieses Tag steht für eine lokale Variablendeklaration.
`<attribute-Declaration>`	*Access, Type, Name*	Dieses Tag steht für eine Attributdeklaration einer Klasse.
`<if-statement>`	*Condition, If*	Dieses Tag stellt die Verzweigung der Programmiersprache (ohne `else`-Zweig) dar.
`<else-statement>`	*Else*	Dieses Tag ergänzt das vorherige durch den Else-Teil einer Verzweigung. Es wird davon ausgegangen, daß der Else-Teil nach dem If-Teil folgt.
`<method-call>`	*Name, Operator, Method, Args*	Dieses Tag repräsentiert einen Methodenaufruf. An Stelle von *Args* werden später die Argumente, getrennt durch den vorgegebenen Separator, eingesetzt.
`<loop>`	*Init, Condition, Body*	Dieses Tag repräsentiert eine Schleife der jeweiligen Sprache, wobei in C++ und Java damit die `while`-Schleife angegeben werden muß. Über die Option *LoopConditionNegated* wird in Eiffel ausgedrückt, daß die angegebene Bedingung negiert werden muß, um den korrekten Ablauf der Schleife zu gewährleisten.

Tag	Variablen	Bedeutung
`<class-declaration>`	*Class, Superclass, Body*	Dieses Tag stellt eine Klassendeklaration dar. *Body* ist dabei eine Positionsargument, das zurückgibt, an welcher Stelle die Methoden und Attribute eingefügt werden sollen.
`<method-declaration>`	*Access, Name, Args*	Dieses Tag repräsentiert eine Methodendeklaration.
`<singleton-method-def>`	*Access, Name, Args*	Dieses Tag stellt eine Methodendeklaration der *get*-Methode einer Singleton-Klasse dar. Dieses Tag wird für C++ benötigt, um eine statische Methode zu deklarieren, die das Singleton-Pattern erfüllt.
`<method-definition>`	*Access, Class, Name, Args, Body, Locals*	Dieses Tag stellt eine Methodendefinition dar. *Body* stellt eine Positionsmarkierung dar, die zurückgibt, an welcher Stelle später der Rumpf der Method eingefügt wird. analog gilt dies für *Locals* und lokale Variablendeklarationen bzw. textitArgs und die Methodenargumente. *Class* wird nur für C++ gebraucht und repräsentiert den zugehörigen Klassennamen.
`<singleton-method-def>`	*Access, Class, Name, Args, Body, Locals*	Dieses Tag stellt eine Methodendefinition einer *get*-Methode für eine Singleton-Klasse dar (analog zu `<singleton-method-decl>`). Diese Angabe wird für Java und Eiffel benötigt.

Tag	Variablen	Bedeutung
`<array-declaration>`	*Name, Type*	Dieses Tag repräsentiert eine Array-Deklaration.
`<array-init>`	*Lower, Upper, Name, Length*	Dieses Tag beschreibt die Initialisierung eines Arrays. Dabei ist es von der verwendeten Sprache abhängig, welche der verfügbaren Argumente tatsächlich verwendet werden.
`<constructor-operator>`	*Name, Type*	Dieses Tag beschreibt den eindeutigen Aufruf zum Erzeugen eines Objekts. In C++ könnte dieser Aufruf in der Form `@Name "= new" @Type "()"` festgelegt werden, in Eiffel dagegen mit `"!!" @Name ".make"`.
`<call-operator>`	*keine*	Dieses Tag definiert Operator zum Aufruf einer Methode (normalerweise „“).
`<static-call-operator>`	*keine*	Dieses Tag definiert den Operator zum Aufruf einer statischen Methode (z. B. „::“ in C++).
`<array-access-operator>`	*Name, Index*	Dieses Tag definiert den Operator zum Zugriff auf einen Array-Index.
`<module-declaration>`	*Name*	Dieses Tag definiert eine Moduldeklaration und wird eingesetzt, wenn die Option *ModuleByDeclaration* auf `"yes"` gesetzt ist.
`<module-definition>`	*Name, Body*	Dieses Tag definiert eine Moduldefinition und wird eingesetzt, wenn die Option *ModuleByDeclaration* auf `"no"` gesetzt ist. *Body* ist der übliche Positionsparameter.

Tag	Variablen	Bedeutung
<main-program>	*Body*	Dieses Tag definiert das Gerüst für das Hauptprogramm. *Body* gibt dabei die Stelle zurück, an der das Hauptprogramm eingefügt werden muß.
<corba-this>	*Name*	Dieses Tag definiert die Schriebweise der this-Methode von CORBA. Diese Angabe wird zur impliziten Aktivierung eines Objekts und zur Umwandlung eines Servants in eine Referenz benötigt, der Name gibt dabei das Objekt an.
<corba-nil>	*Name*	Dieses Tag definiert die Schreibweise eine nil-Referenz in CORBA. Auch hier gibt der Name das Objekt an.
<narrow-operation>	*Classname, Source, Target*	Dieses Tag definiert die Operation, die für das Narrowing unter CORBA verwendet wird. Eine Angabe für C++ wäre z. B. @Target "=" @Name "::_narrow(" @Target ")".
<corba-out-read>	*Classname, Source*	Dieses Target gibt den lesenden Zugriff auf einen out-Typen von CORBA in der jeweiligen Sprache an, z. B. @Classname ".value" in Java, @Source ".item" in Eiffel und @Source in C++. Die Information wird z. B. bei der Generierung einer Zuweisung benutzt. Analog existiert ein Tag <corba-out-write>.

Anhang D

Diagramme zur Darstellung der Schnittstelle

Die folgenden Diagramme illustireren die Funktionsweise der Schnittstelle. Diese Diagramme zeigen die wichtigsten Aspekte der Schnittstelle und sind daher nicht als Referenz für einzelne Klassen gedacht. Einzelne Diagramme geben nur in Verbindung mit Erläuterungen in den entsprechenden Abschnitten Sinn, so daß zur Erläuterung dort nachgelesen werden muß. An anderen Stellen sind Konzepte vernachlässigt worden, deren Umsetzung bekannten Pattern enspricht (z. B. Iteratoren oder die Modellierung einer Baumstruktur durch das Composite Pattern) oder deren Zweck aus der Namensgebung beteiligter Klassen und Objekte offensichtlich wird.

Die Zuordnung der einzelnen Abbildungen zu den entsprechenden Abschnitten ist aus folgender Auflistung ersichtlich:

- Die Abbildungen D.1, D.2, D.3, D.4 zeigen die statischen Klassenstrukturen für den Einlesevorgang und werden in den Abschnitten 4.2 und 4.3 erläutert.
- Die Abbildungen D.5, D.6, D.7, D.8 und D.9 illustrieren einige der zugehörigen Abläufe. Die genauen Beschreibungen findet man in Abschnitt 4.3.
- Die in den Abbildungen D.10, D.12, D.11 und D.13 gezeigten Darstellungen dienen der Illustration der Transformationsvorgänge beim Einlesen. Auch diese Abbildungen werden in Abschnitt 4.3 beschrieben.
- Die Abbildungen D.14 und D.15 zeigen die statischen Klassendiagramme zur IDL-Generierung und gehören zu der Beschreibung in Abschnitt 5.2.
- Die Abbildungen D.16 und D.17 demonstrieren die Struktur und Arbeitsweise der generierten Hilfsklassen. Die Beschreibung befindet sich in 5.1.
- Alle folgenden Abbildungen sind Bestandteil der Beschreibung zur eigentlichen Instrumentierung und gehören zu Abschnitt 6.2.

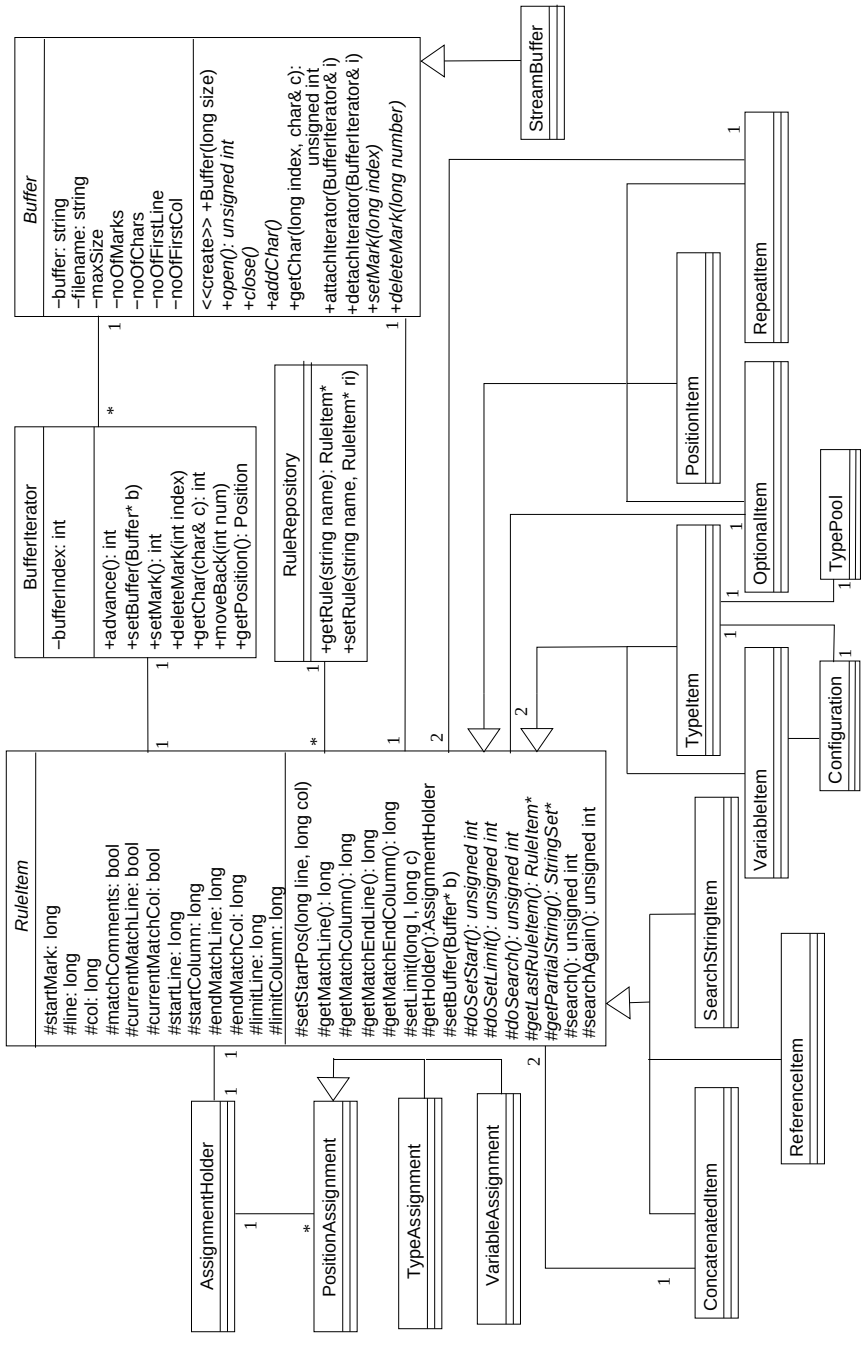

Abbildung D.1: Items und unterstützende Klassen.

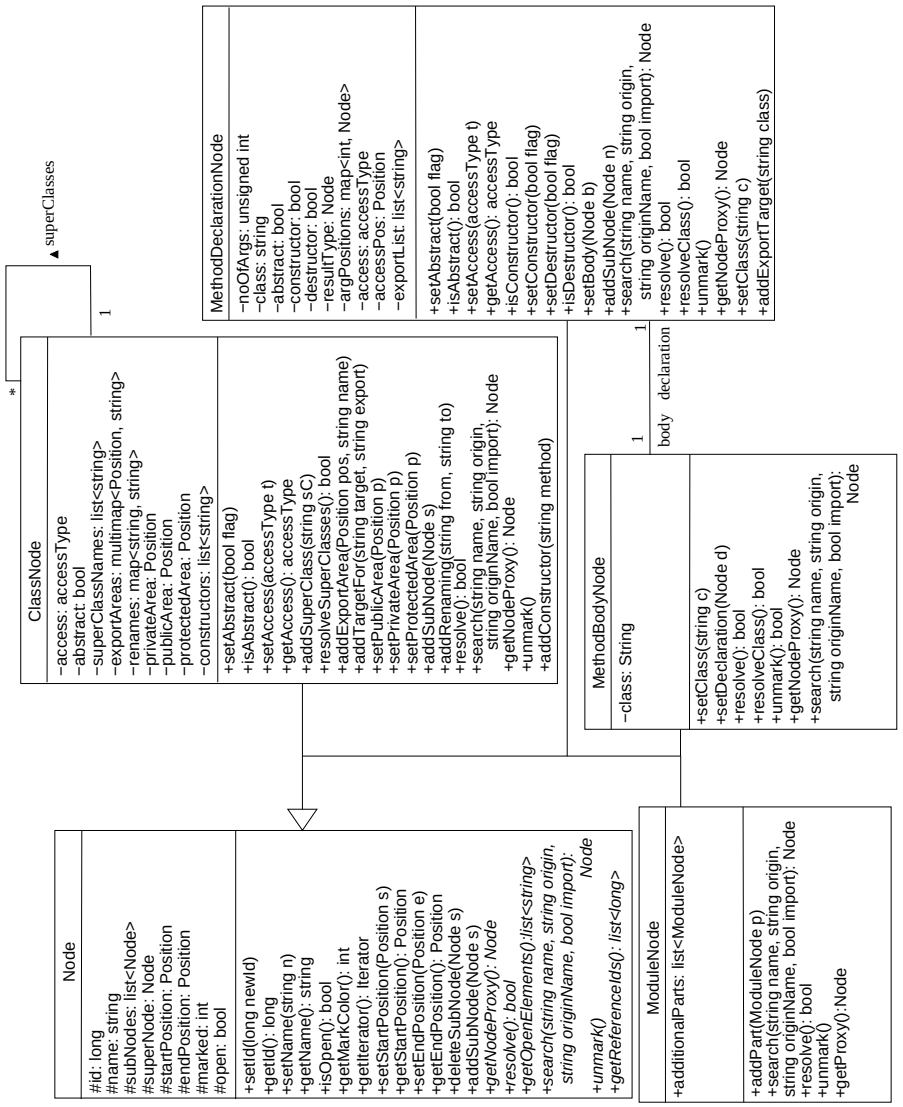

Abbildung D.2: *Node*-Klassen zur Speicherung der Daten.

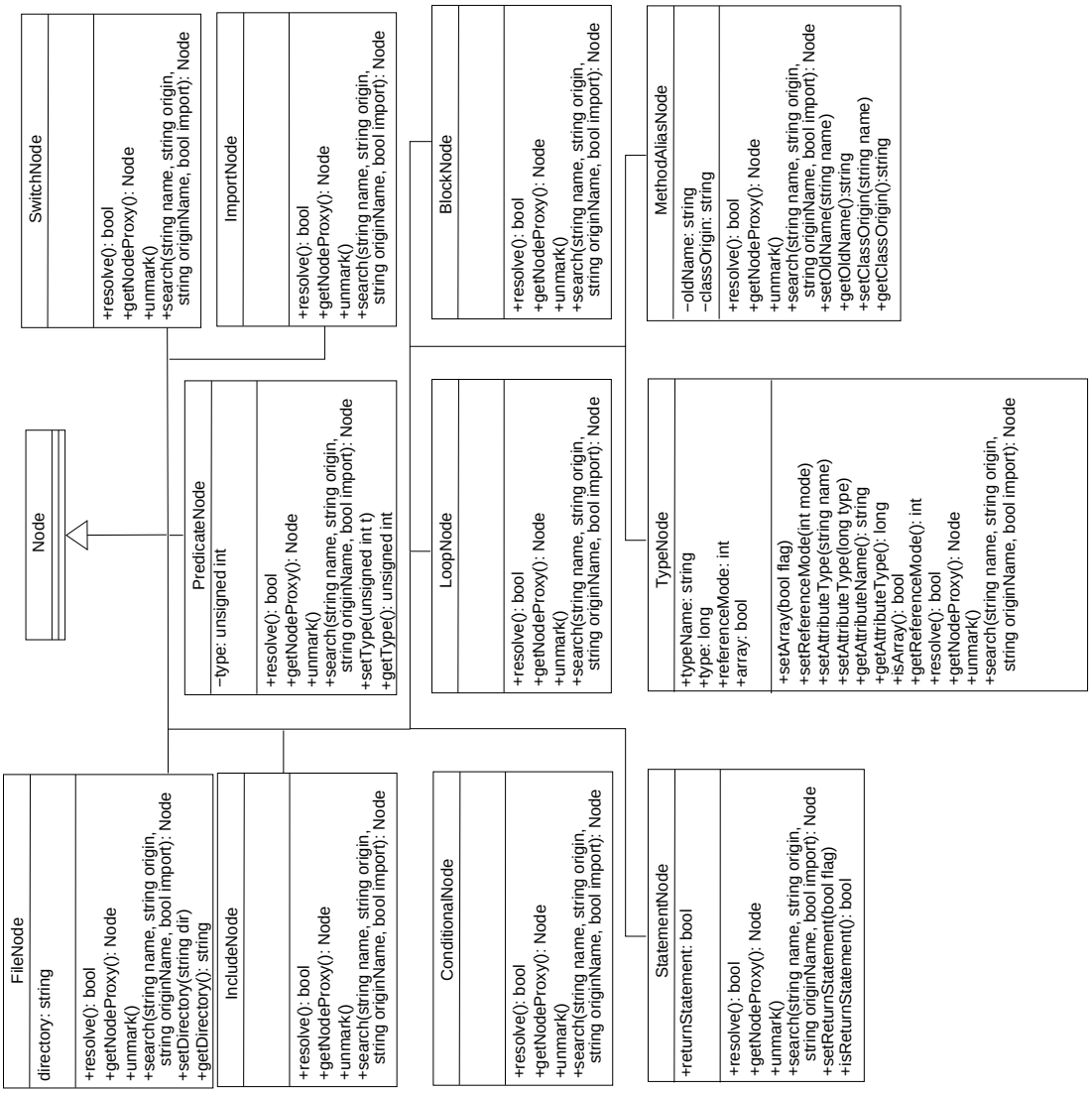

Abbildung D.3: Weitere *Node*-Klassen zur Speicherung der Daten.

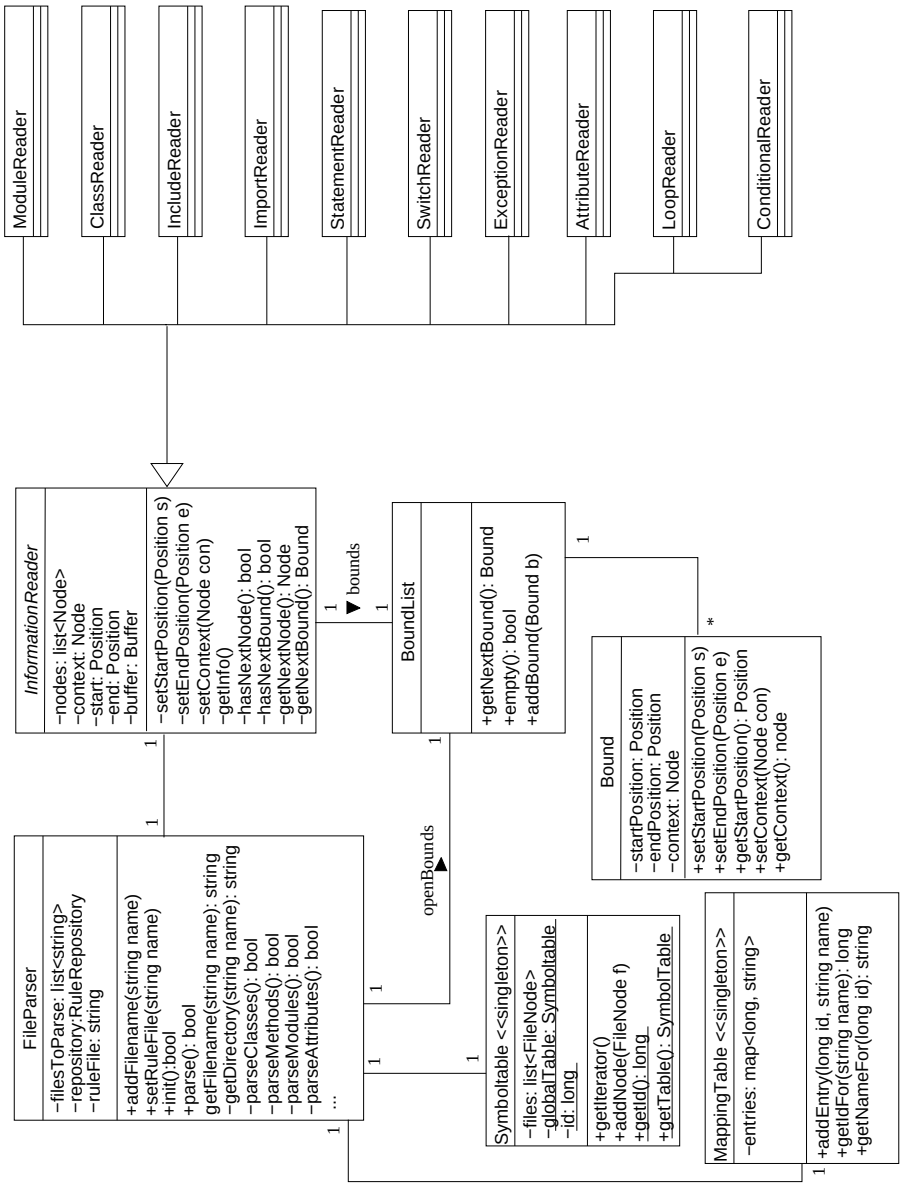

Abbildung D.4: Verantwortliche Klassen für den Einlesevorgang.

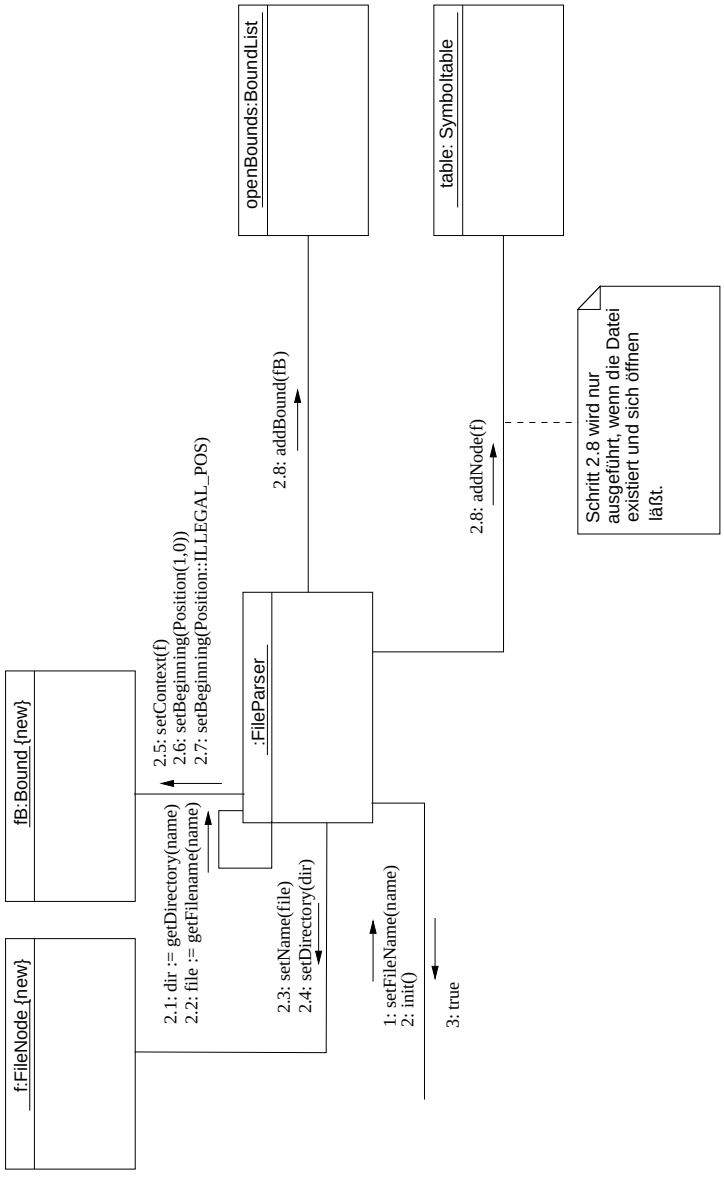

Abbildung D.5: Init der FileParser-Klasse.

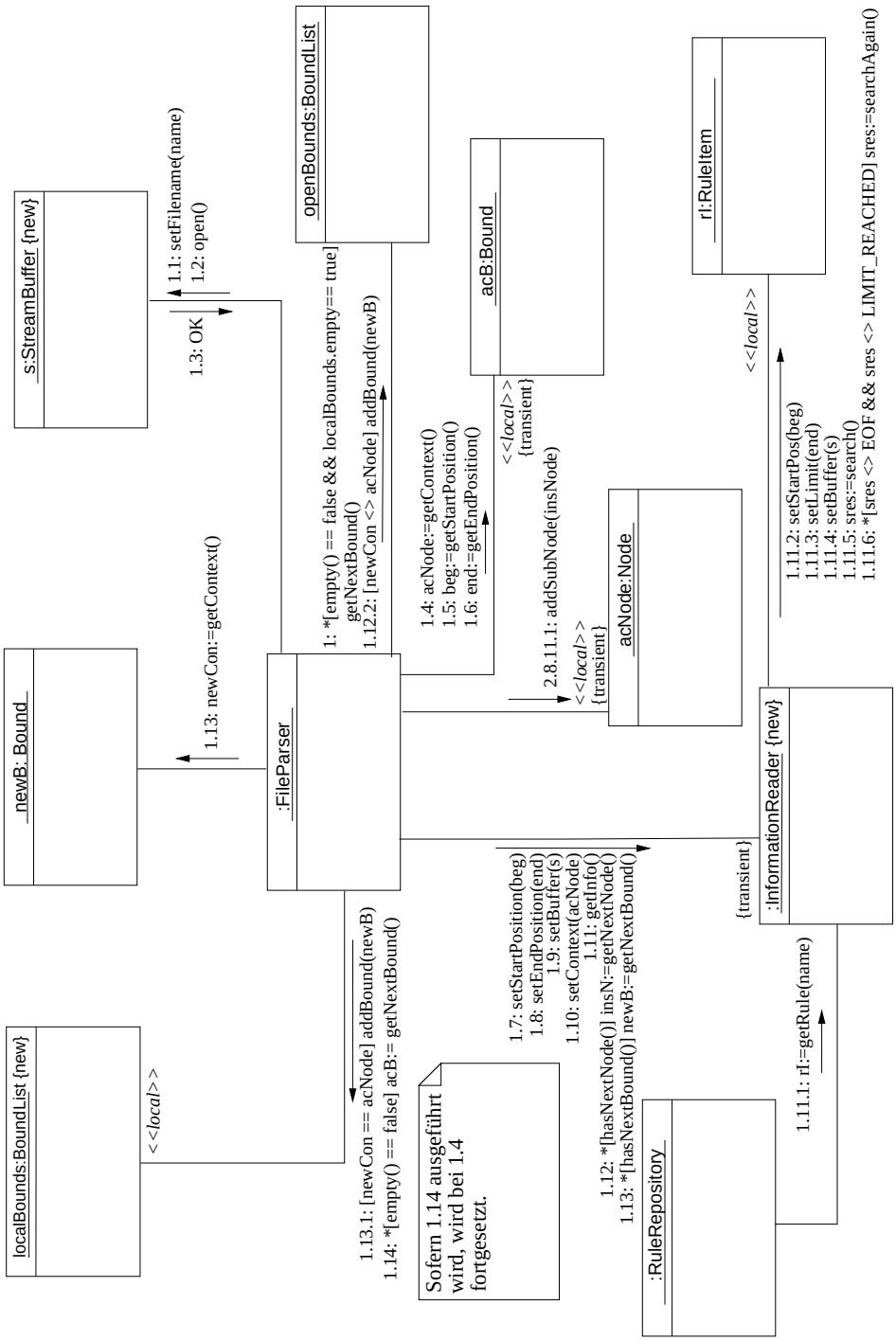

Abbildung D.6: Allgemeiner Ablauf des Einlesens innerhalb eines Kontexts.

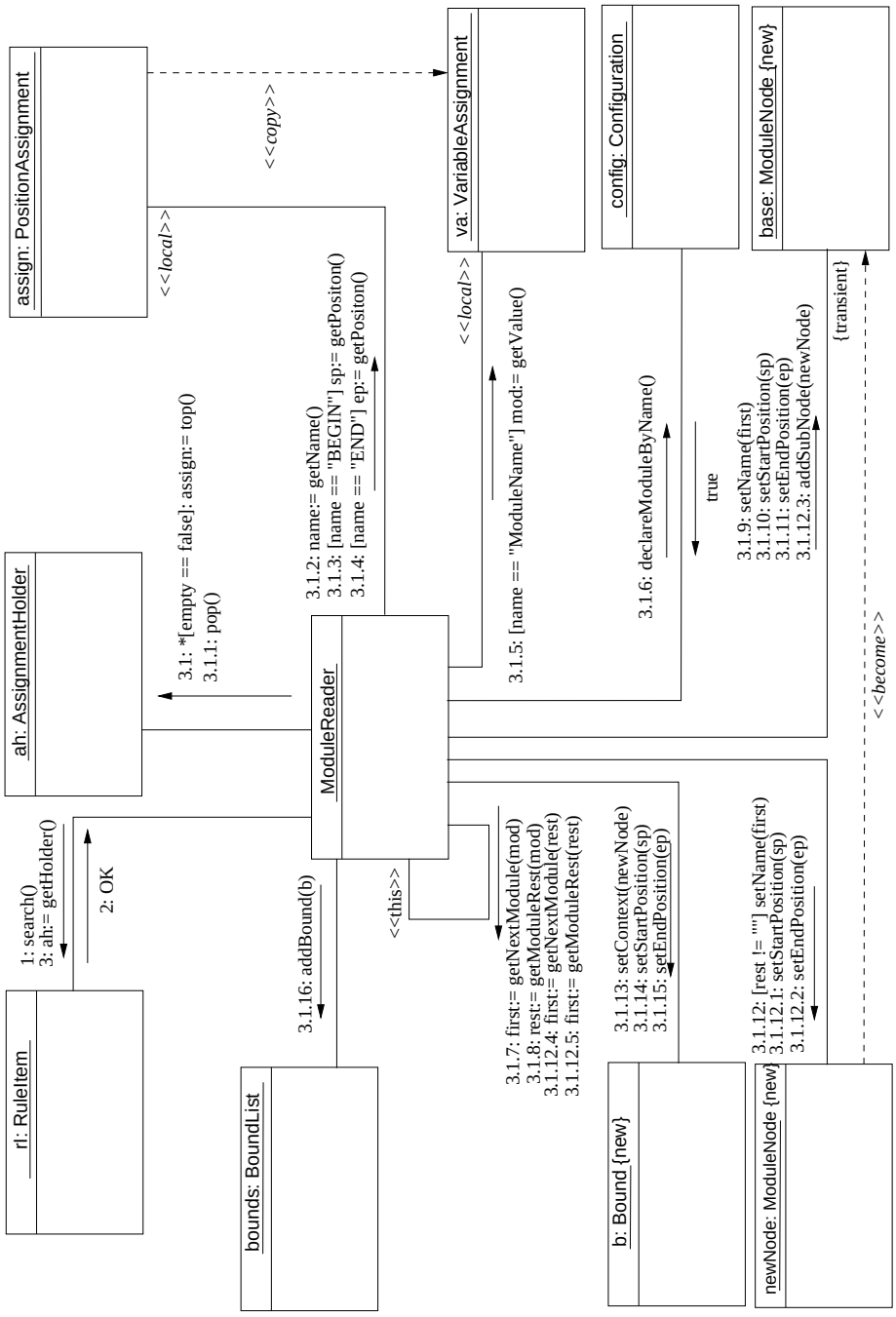

Abbildung D.7: Arbeitsweise der Klasse *ModuleReader*.

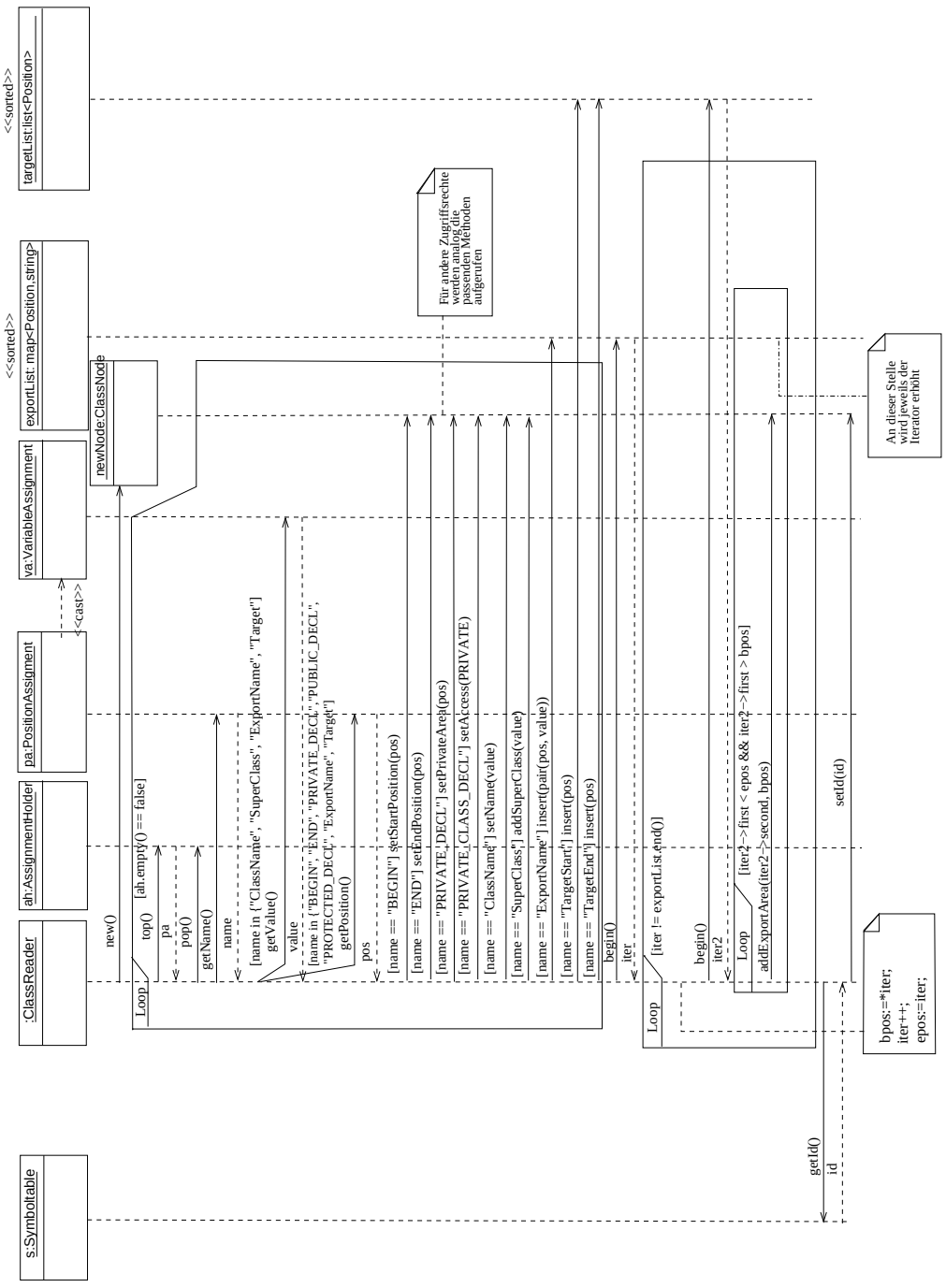

Abbildung D.8: Arbeitsweise der Klasse *ClassReader*.

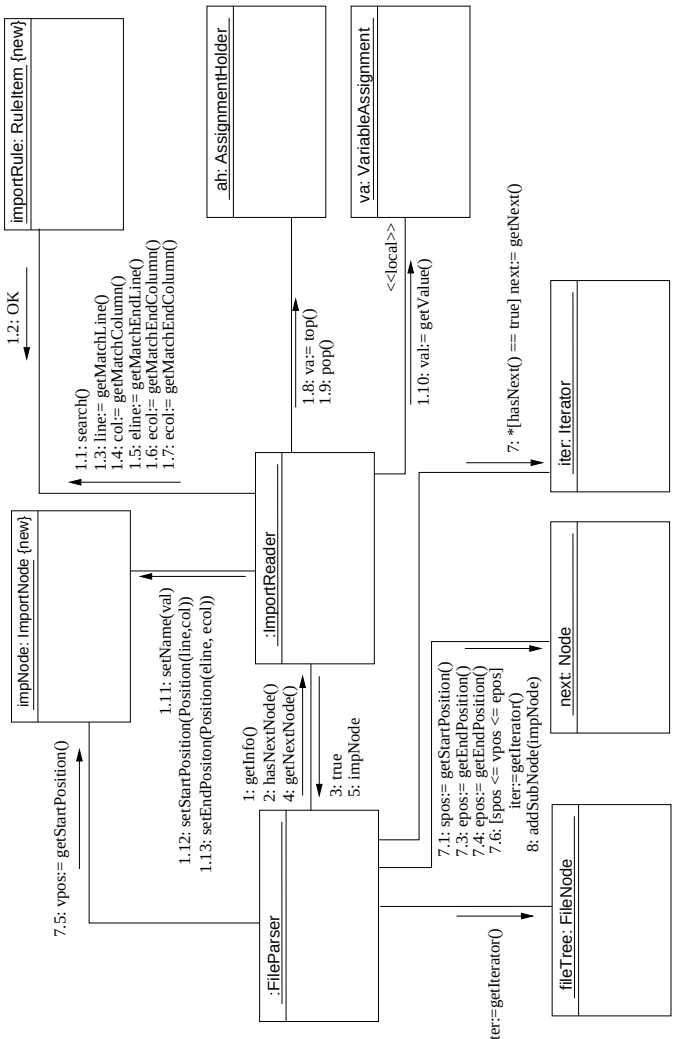

Abbildung D.9: Vorgang beim Einlesen eines Imports.

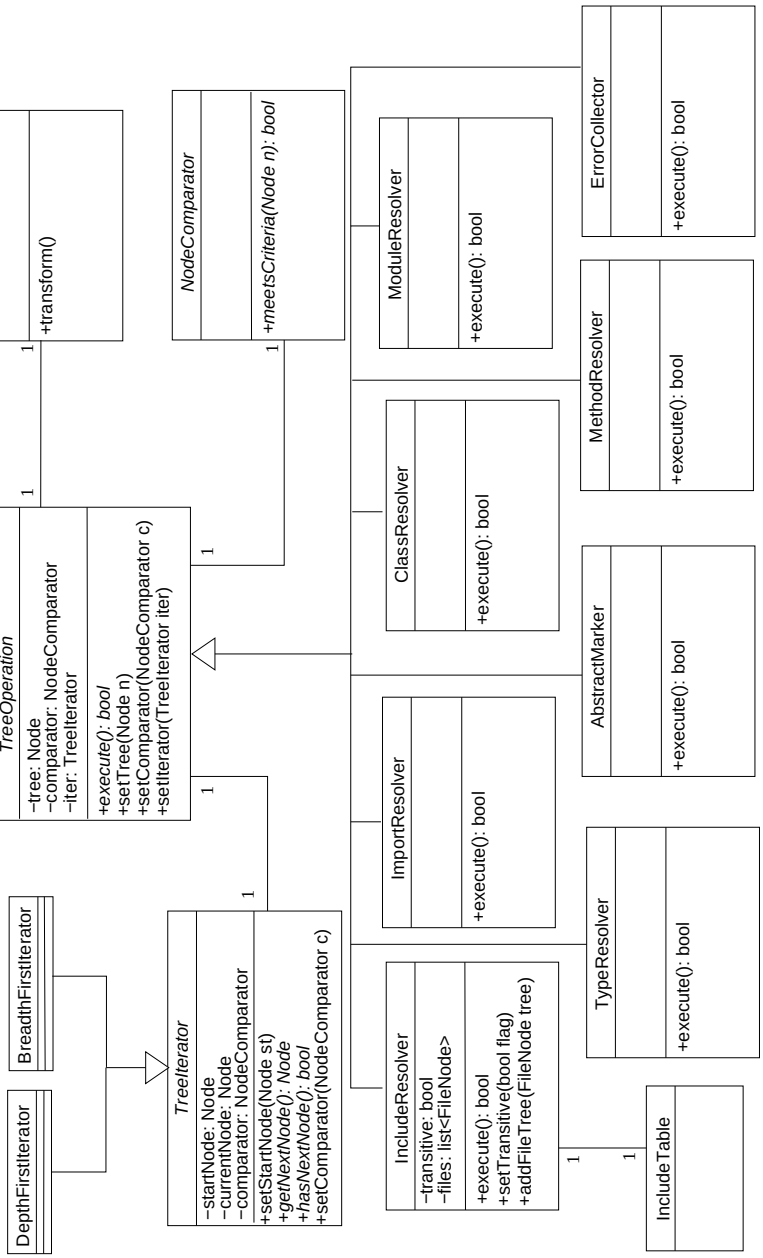

Abbildung D.10: Klassen zur Relationenanalyse.

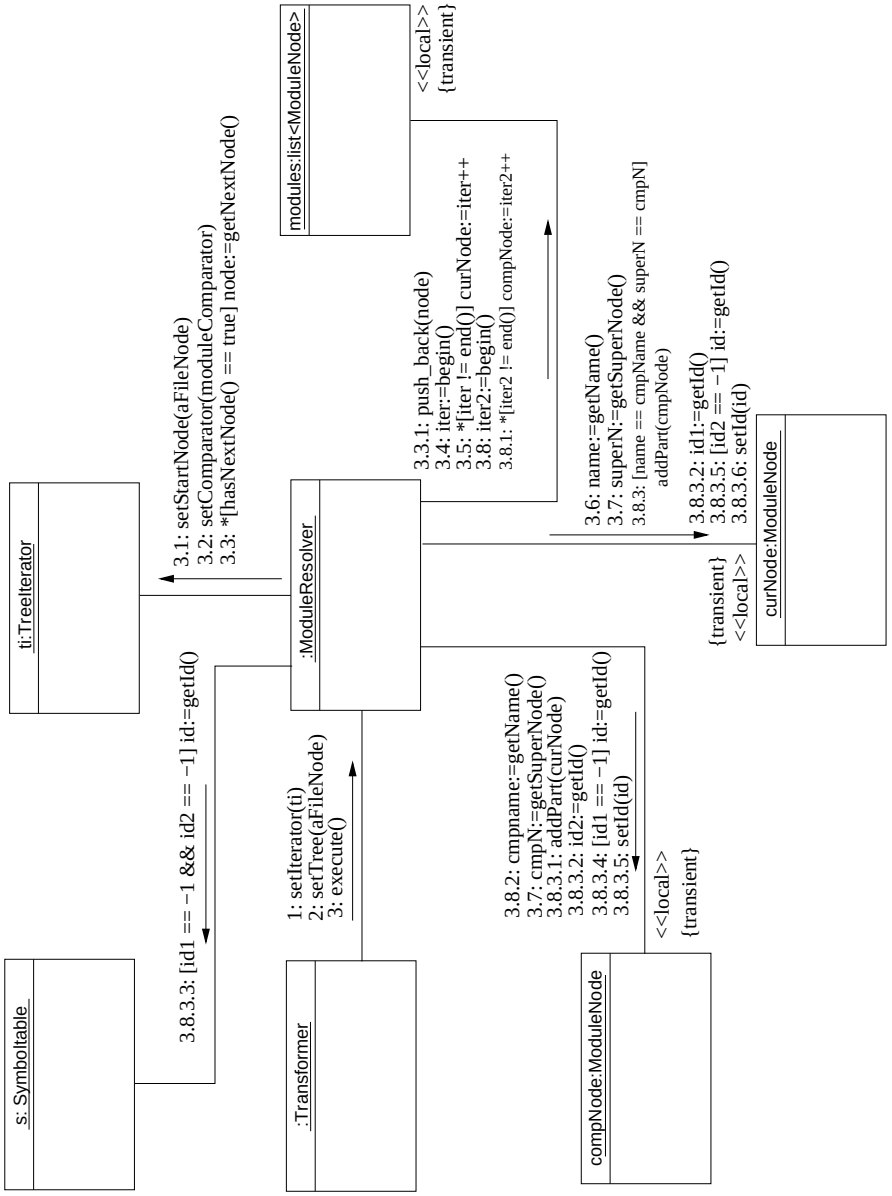

Abbildung D.11: Arbeitsweise der Klasse *ModuleResolver*.

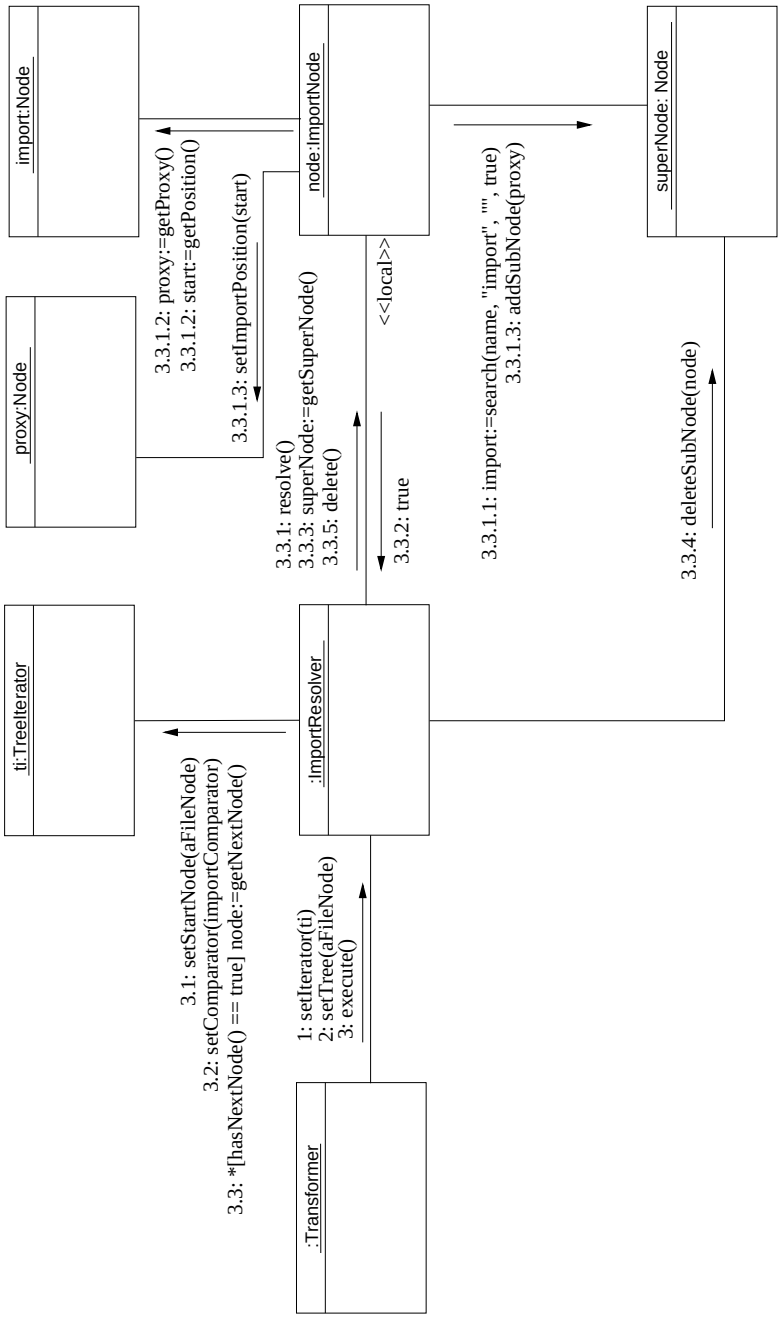

Abbildung D.12: Arbeitsweise der Klasse *ImportResolver*.

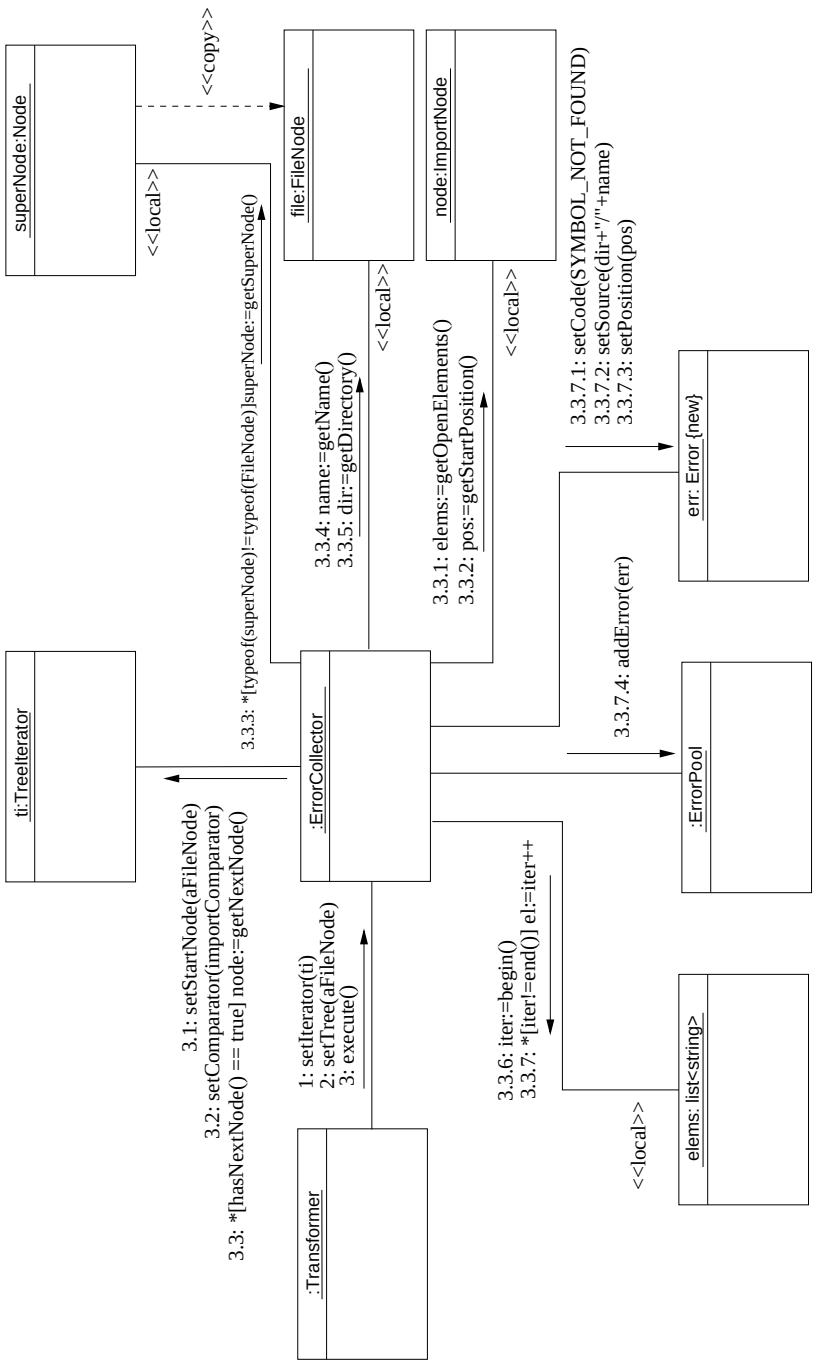

Abbildung D.13: Arbeitsweise der Klasse *ErrorCollector*.

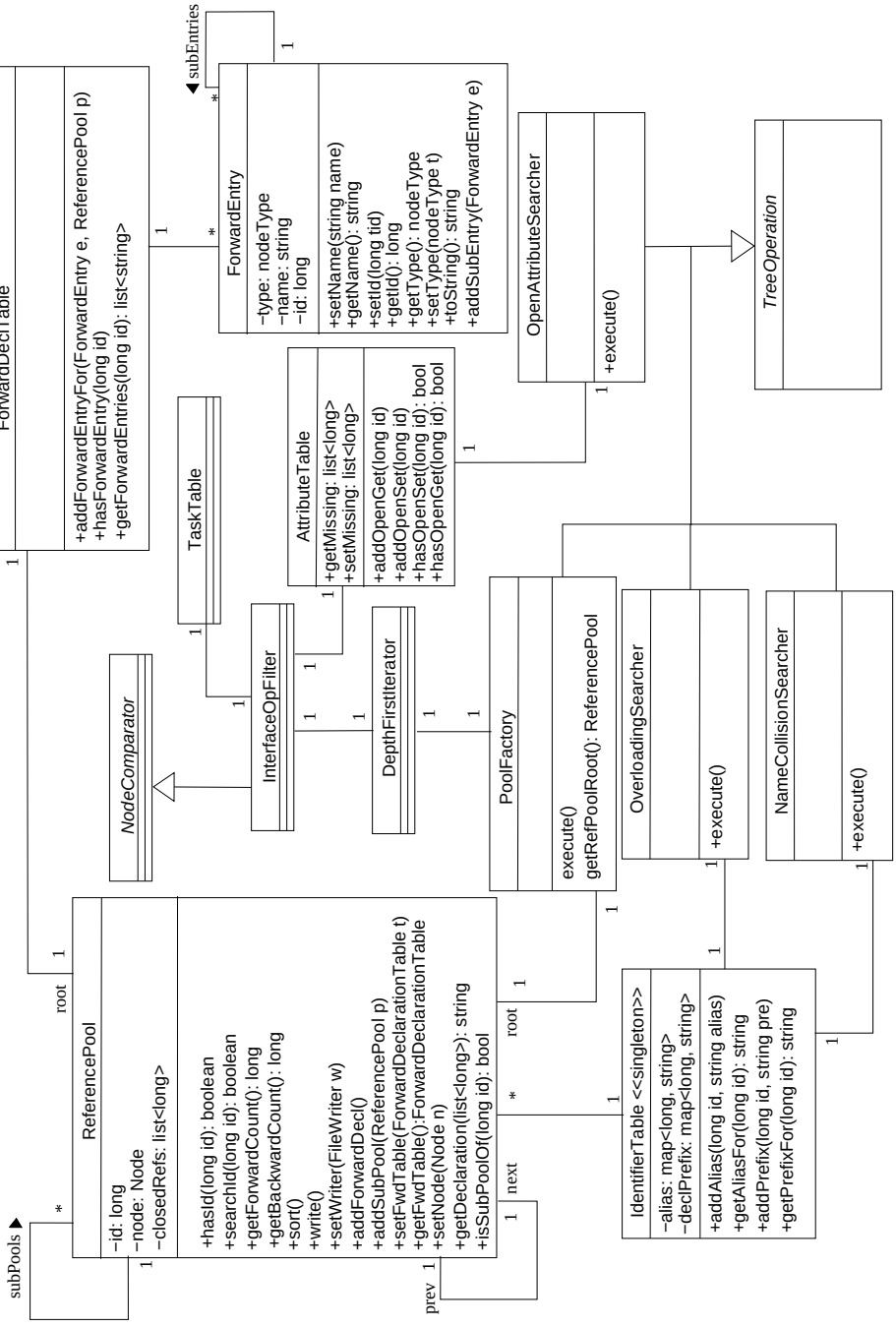

Abbildung D.14: Klassen zur Vorbereitung der IDL-Generierung.

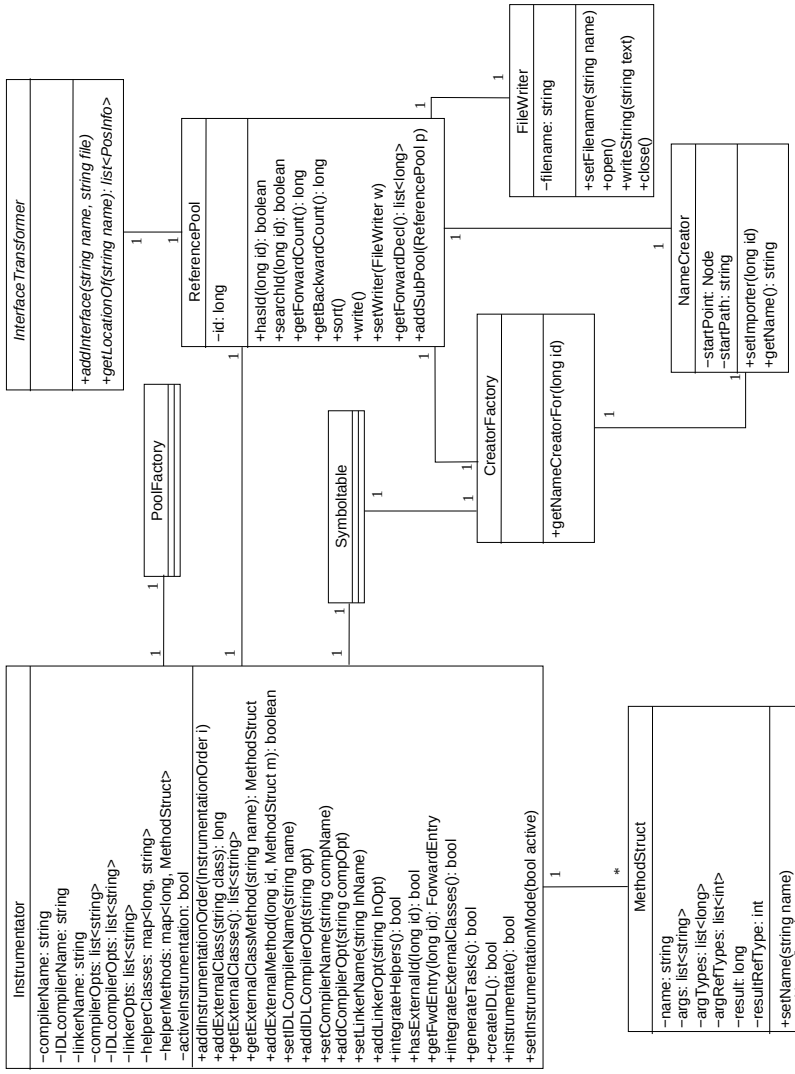

Abbildung D.15: Klassen zur Durchführung der IDL-Generierung.

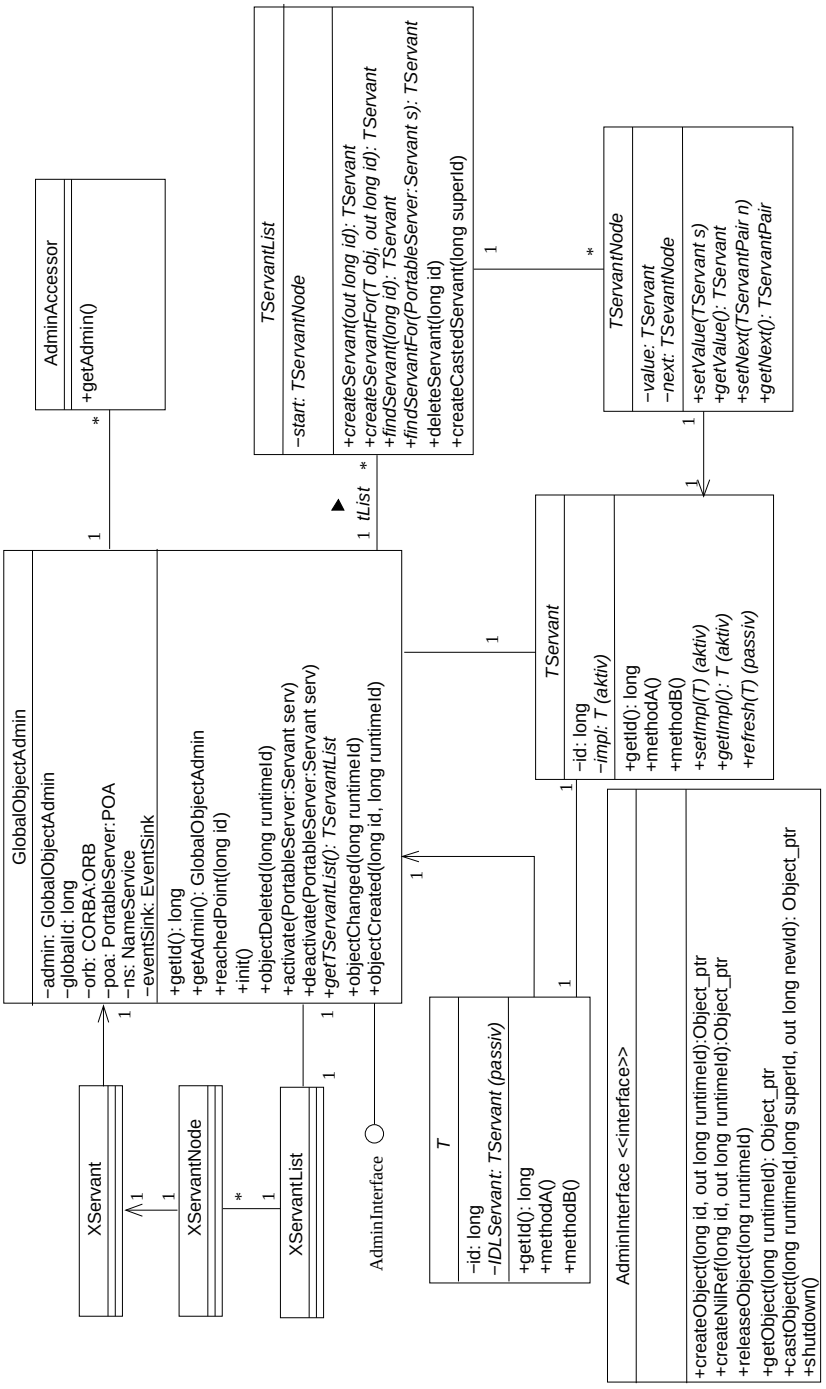

Abbildung D.16: (erzeugte) Klassen zur Unterstützung des Laufzeitverhaltens – *T* ist hier Parameter in der Namensgebung.

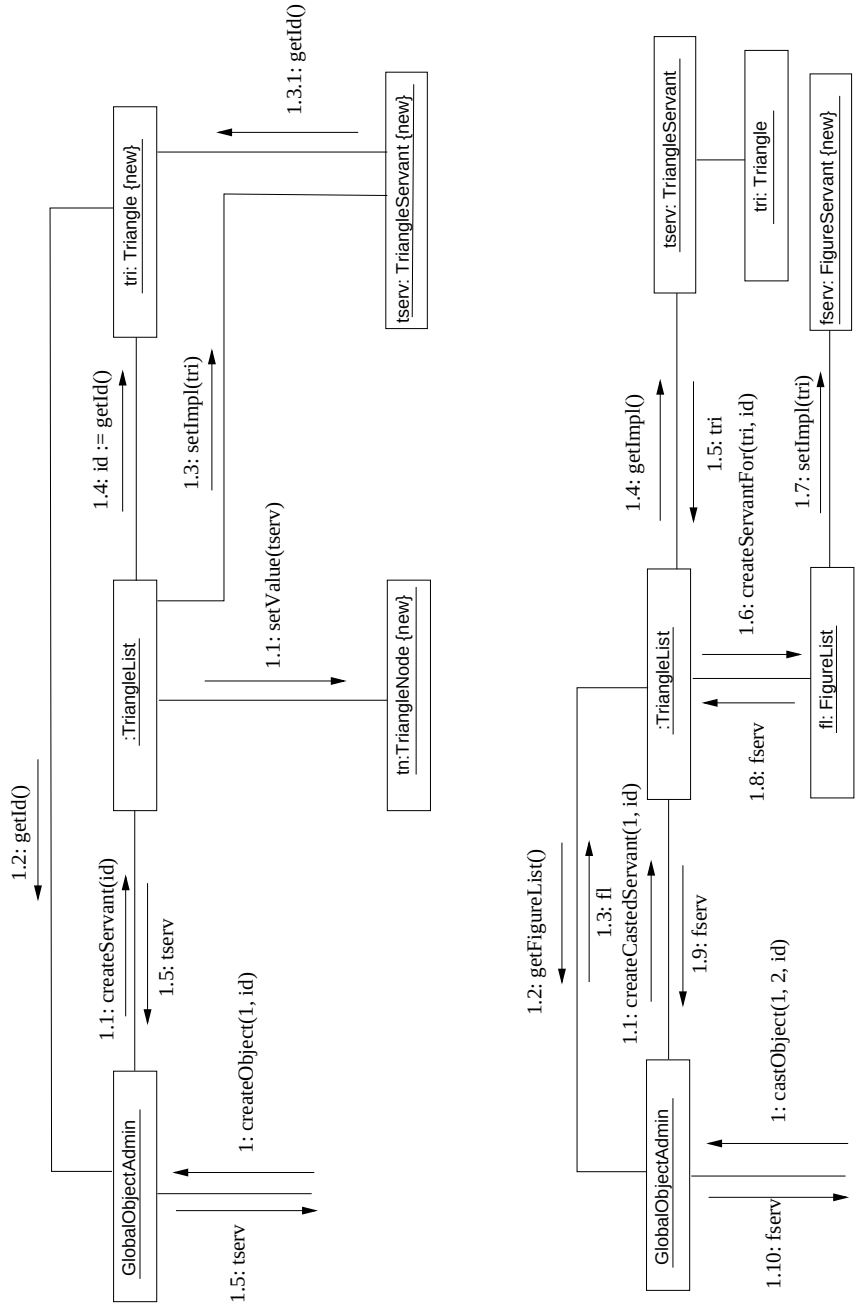

Abbildung D.17: Kommunikation für die Aufrufe *createObject()* und *castObject()*.

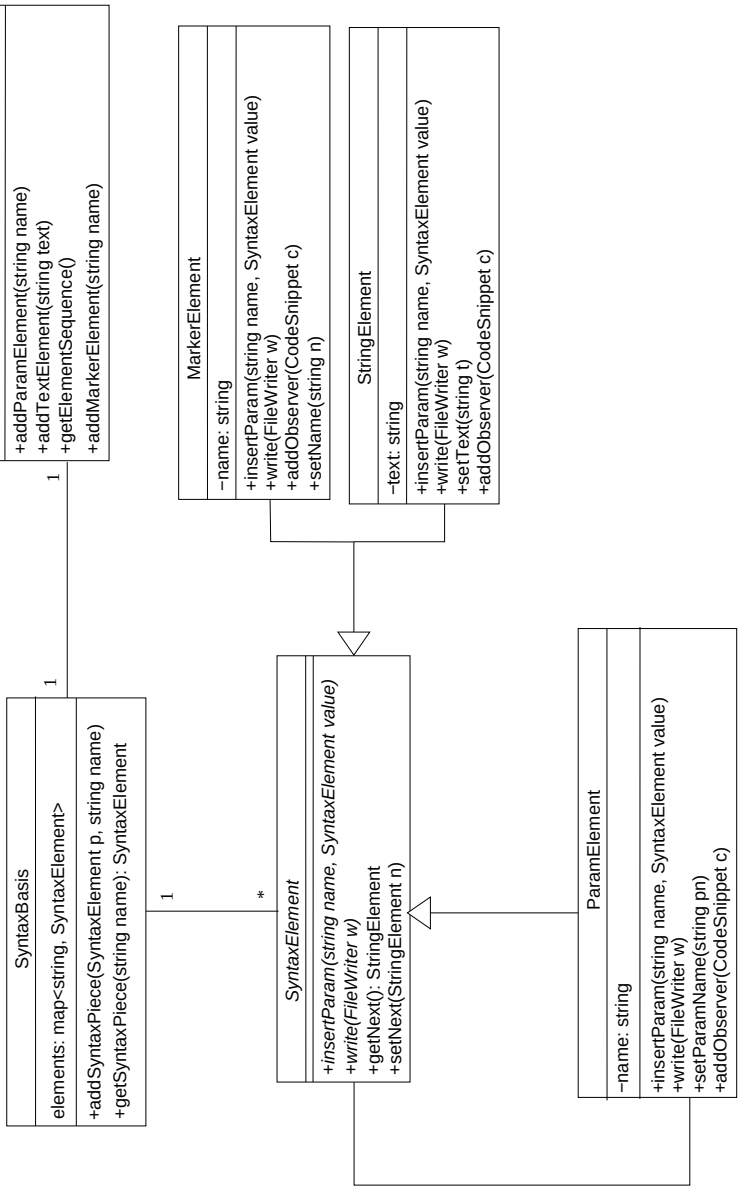

Abbildung D.18: Basisklassen zum Aufbau des einzufügenden Codes.

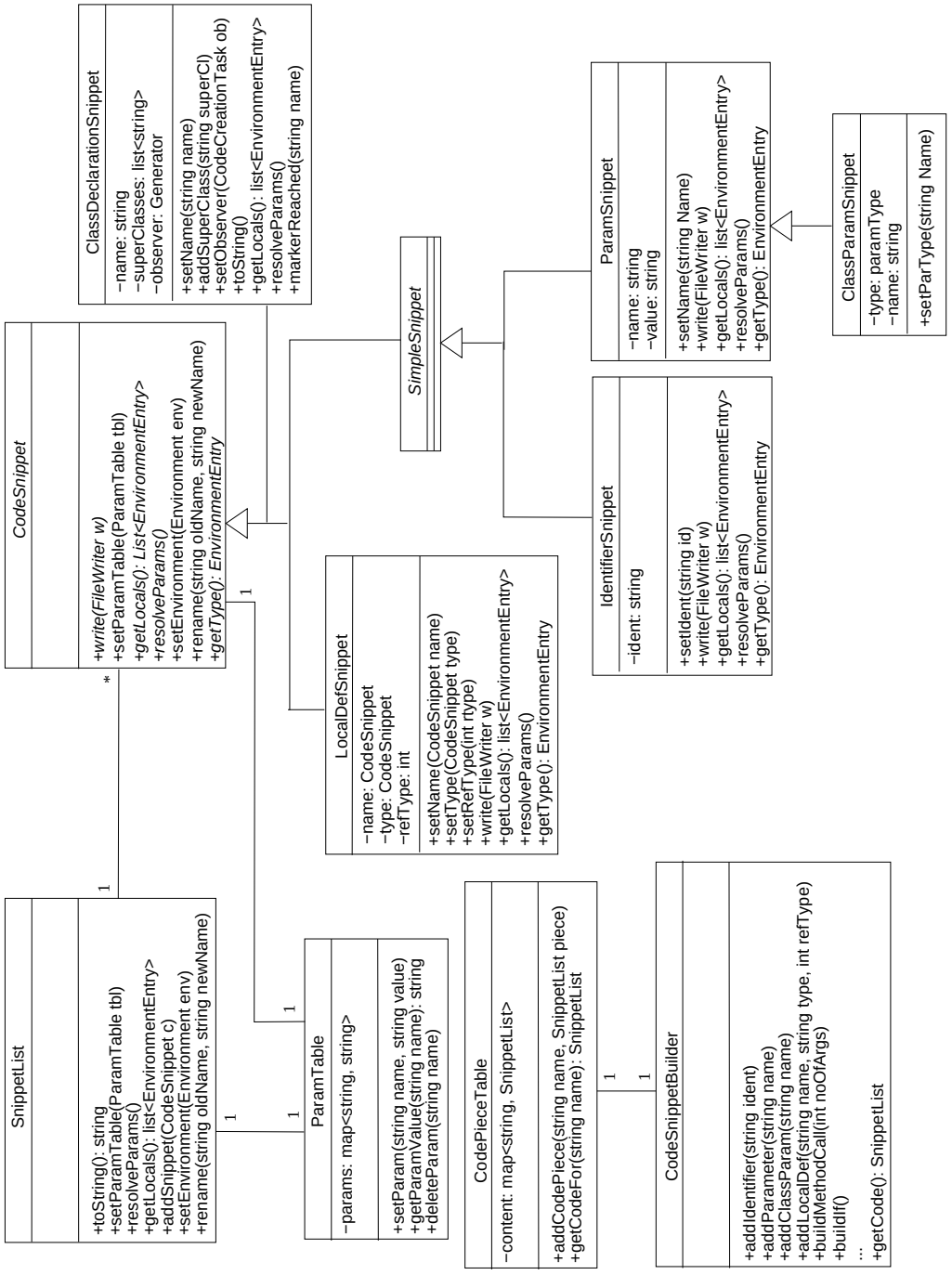

Abbildung D.19: Low-Level-Funktionalität zur Instrumentierung.

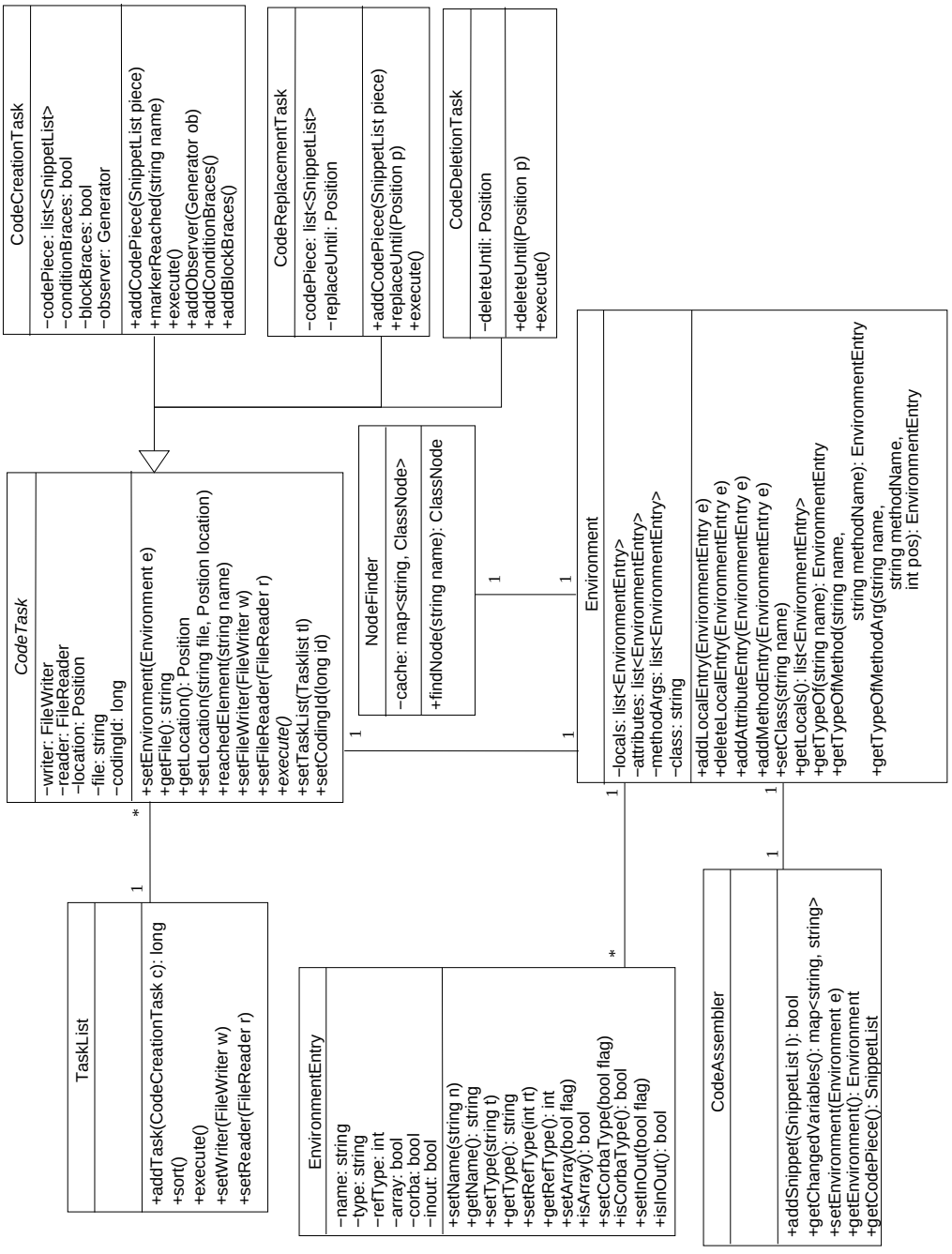

Abbildung D.20: Mid-Level-Funktionalität zur Instrumentierung.

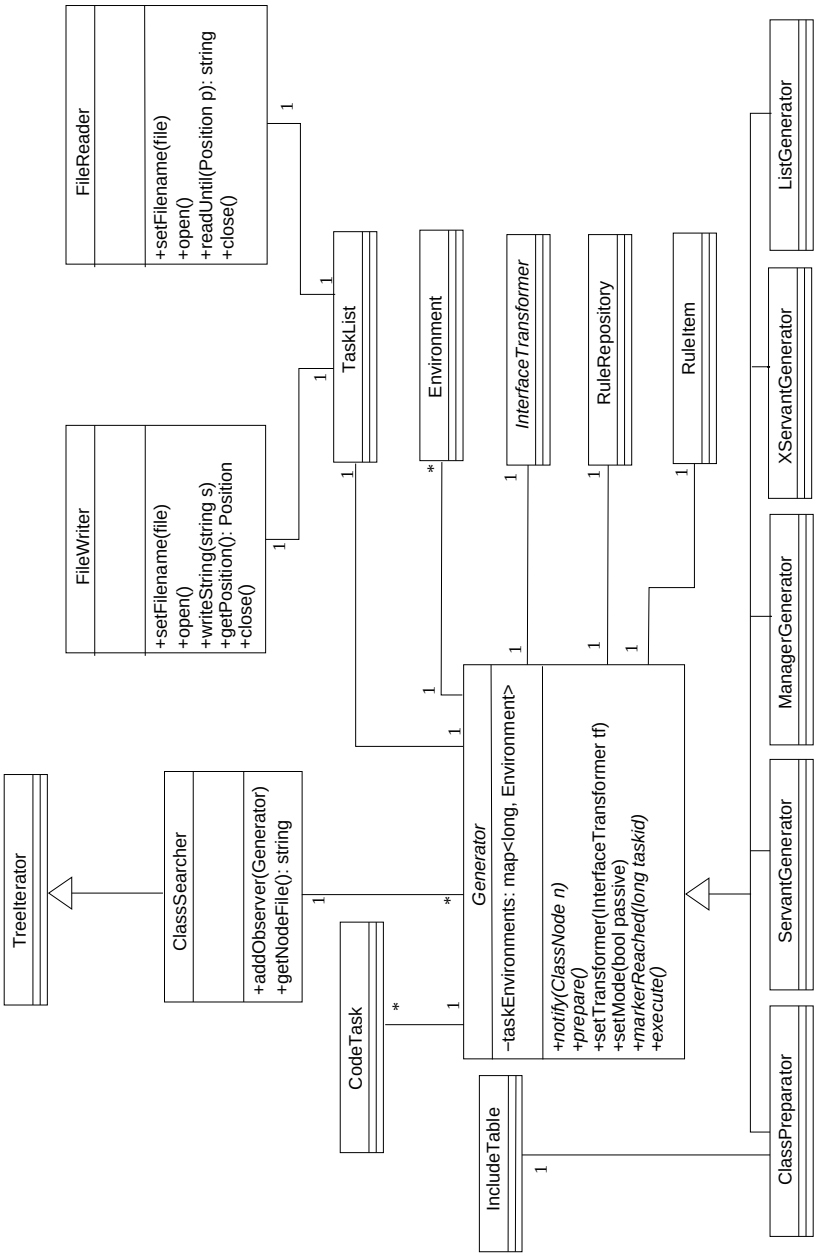

Abbildung D.21: High-Level Funktionalität zur Instrumentierung.

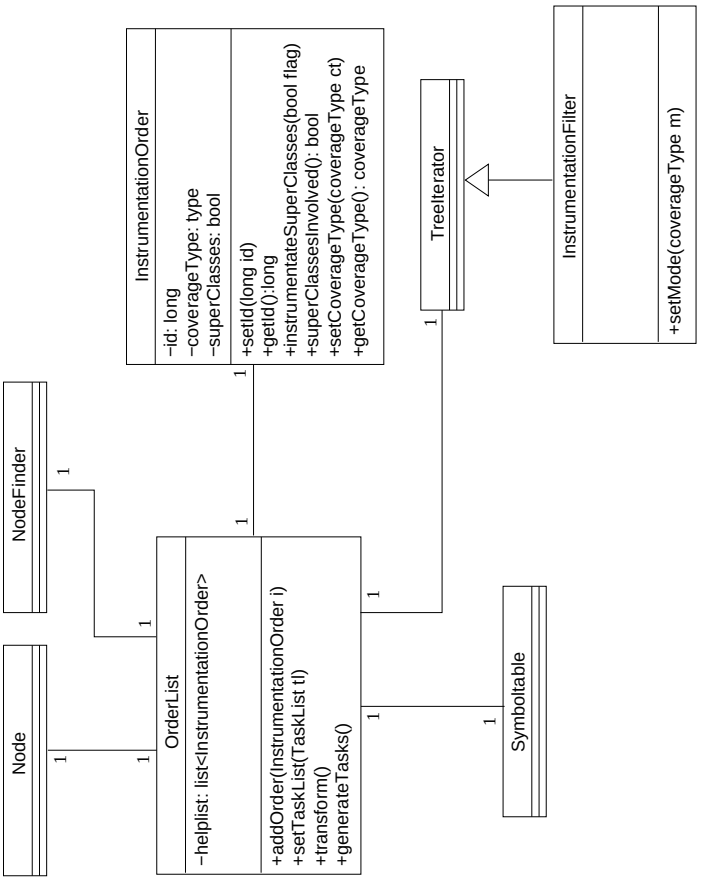

Abbildung D.22: Benutzerklassen zur Instrumentierung und Hilfsklassen zur Vorbereitung.