Design by Contract und Java - Konzeption und Implementierung eines Präprozessors

Diplomarbeit
bei Prof. Dr. Manfred Kaul

Zweitprüfer
Karl-Heinz Sylla

vorgelegt von Jan Hermanns
am Fachbereich Angewandte Informatik der
Fachhochschule Bonn-Rhein-Sieg

Bonn, 13. Februar 2001

Inhaltsverzeichnis

Danksagung

Ich möchte mich hiermit bei der Firma *Coodex Consulting GmbH* für die Unterstützung während meiner Diplomarbeit bedanken. Mein besonderer Dank gilt dabei meinem Betreuer *Arno Haase* für seine wertvollen Tips und Anmerkungen.

Des weiteren möchte ich mich sehr herzlich bei *Manfred Kaul* und *Karl-Heinz Sylla* für die sehr gute Betreuung von Seiten der Fachhochschule Bonn-Rhein-Sieg bedanken.

Kapitel 1

Einleitung

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist eine intensive Auseinandersetzung mit dem Thema Design by Contract und die Implementierung eines Werkzeugs, um diese Methodik für die Programmiersprache Java verfügbar zu machen.

1.2 Überblick über die Kapitel

Im folgenden wird ein kurzer Überblick über die verschiedenen Kapitel der vorliegenden Arbeit gegeben.

1.2.1 Kapitel 2 - Korrektheit

In diesem Kapitel wird der Begriff der Korrektheit definiert und es wird ein Beweissystem vorgestellt, mit dessen Hilfe die partielle Korrektheit von Programmen nachgewiesen werden kann. Anschließend wird anhand eines Beispiels die Anwendung des Beweissystems veranschaulicht.

1.2.2 Kapitel 3 - Design by Contract

Die Methodik Design by Contract wird vorgestellt und es wird gezeigt, wie mit Hilfe der Vertragsmethapher die Korrektheit von Klassen bzw. Operationen zur Laufzeit sichergestellt werden kann. Daran anknüpfend werden verschiedene Seiteneffekte diskutiert, die bei der Verwendung der Methodik auftreten können. Abschließend werden die Grenzen dessen, was Design by Contract leisten kann, aufgezeigt.

1.2.3 Kapitel 4 - Unit-Tests

Das Kapitel beschreibt das Konzept der Unit-Tests und erläutert die Entwicklungsmethode eXtreme-Programming, die Unit-Tests als wesentliches Grundkonzept beinhaltet. Anschließend wird das Java-basierte Testframework JUnit genauer untersucht. Am Ende des Kapitels wird anhand eines Beispiels gezeigt, wie sich Design by ontract und das Konzept der Unit-Tests ergänzen können.

1.2.4 Kapitel 5 - Vergleich vorhandener Tools

In diesem Kapitel werden vier verschiedene Werkzeuge untersucht, die Design by ontract für die Programmiersprache Java bereits unterstützen. Dabei werden insbesondere die Stärken und die Schwächen der einzelnen Werkzeuge untersucht.

1.2.5 Kapitel 6 - Der *sushi*-Präprozessor

Zu Anfang des Kapitels werden die Ziele und die Anforderungen beschrieben, die für die Entwicklung des Präprozessors entscheidend waren. Danach werden die Architektur und die Implementierung des Präprozessors genauer betrachtet. Das Ende des Kapitels gibt einen Ausblick darüber, wie der Präprozessor verbessert werden kann.

Kapitel 2

Korrektheit

2.1 Einleitung

Ein wesentliches Qualitätsmerkmal von Programmen ist die Korrektheit. Dabei bezeichnet der Begriff Korrektheit ganz allgemein die Eigenschaft eines Programms, sich so zu verhalten, wie es seine Spezifikation definiert.

In diesem Kapitel wird der Begriff Korrektheit genauer erläutert, und es wird ein Überblick gegeben, wie sich Korrektheit beweisen läßt.

2.2 Korrektheit

Beim Programmieren kommt es immer wieder vor, daß ein Programm falsche Ergebnisse liefert, sich in einer Endlosschleife "aufhängt" oder unerklärliche Fehlermeldungen erscheinen. Darum hat man sich in der Informatik schon sehr früh darum bemüht, Methoden zu finden, um diese Probleme zu vermeiden oder wenigstens zu beheben.

Die Funktionalität von Programmen steckt in den Prozeduren und Funktionen der Module bzw. in den Operationen der Klassen (bei der objektorientierten Programmierung). Prozeduren, Funktionen und Operationen implementieren Algorithmen, die auf Variablen bzw. Attribute zugreifen und diese entsprechend verändern. Dabei wird unter einem Algorithmus eine eindeutige, endliche Beschreibung eines allgemeinen, endlichen Verfahrens zur schrittweisen Ermittlung gesuchter Größen aus gegebenen Größen verstanden [Balzert99].

Die Herausforderung beim Programmieren liegt nun darin, Algorithmen zu implementieren, die einen gewünschten Zweck erfüllen. Die wohl am häufigsten verwendete Methode, um sich davon zu überzeugen, daß ein Algorithmus funktioniert, ist das Testen. Durch Ausprobieren verschiedener Ein-

gabewerte bekommt man ein Gefühl dafür, ob der Algorithmus arbeitet wie erwartet. Es liegt jedoch klar auf der Hand, daß man durch Testen keinesfalls sicherstellen kann, daß ein Algorithmus korrekt funktioniert, da der Aufwand in der Regel viel zu groß ist. Denn es müßten alle möglichen Kombinationen von Eingabewerten getestet werden.

Daher hat man Methoden entwickelt, um durch Analysen die Konsistenz zwischen Spezifikationen und Programmen zu beweisen[1]. In diesem Kontext spricht man auch von Verifikation. Dabei sei angemerkt, daß das Finden eines Beweises dafür, ob ein Programm eine Spezifikation erfüllt, eine intellektuelle Leistung darstellt, für die folgerichtiges Denken eine Voraussetzung ist und die somit nicht algorithmisch gelöst werden kann.

2.3 Verifikation

Durch Verifikation der Korrektheit beweist man, daß sich ein Algorithmus konsistent zu seiner Spezifikation verhält. Um die Spezifikation eines Algorithmus anzugeben, benutzt man *Zusicherungen (Assertions)*, welche die *Vorbedingungen (Preconditions)* und die *Nachbedingungen (Postconditions)* eines Algorithmus definieren. Vorbedingungen bezeichnen dabei die zulässigen Werte und Beziehungen der Variablen vor Begin des Algorithmus. Nachbedingungen bezeichenen die Werte und Beziehungen, die nach Ablauf des Algorithmus gültig sein müssen. Zusicherungen definieren also nicht nur gültige Wertebereiche (z.B. $x > 0$ **and** $x < 10$), sondern auch auch das Verhältnis der Variablen zueinander (z.B. $x = 2y + z$).

Um einen Algorithmus mit seiner Spezifikation darzustellen, benutzt man eine Notation der Form: $\{pre\}\ A\ \{post\}$. Dabei ist $\{pre\}$ die Vorbedingung, A der Algorithmus und $\{post\}$ die Nachbedingung. Notationen dieser Art werden oft auch als *Hoare-Tripel* oder *Korrektheitsformeln* bezeichnet.

Bei der Verifikation von Algorithmen unterscheidet man zwei Arten von Korrektheit:

partielle Korrektheit: Unter der Voraussetzung, daß der Algorithmus A terminiert, garantiert die Vorbedingung $\{pre\}$, daß nach Beendigung die Nachbedingung $\{post\}$ gilt.

totale Korrektheit: Die Vorbedingung $\{pre\}$ garantiert, daß der Algorithmus A terminiert und danach die Nachbedingung $\{post\}$ gilt.

[1] Die Korrektheit eines Programmes steht also immer in Relation zu seiner Spezifikation.

Ein Weg, um *totale Korrektheit* zu beweisen ist daher, zuerst die *partielle Korrektheit* zu beweisen und danach zu beweisen, daß der Algorithmus terminiert.

2.4 Verifikationsregeln

Im folgenden wird ein Beweissystem vorgestellt, mit dessen Hilfe die *partielle Korrektheit* von sequentiellen Programmen bewiesen werden kann. Die Idee dieser Methode ist, daß man ein komplexes Programm in einfache Teilstrukturen zerlegt und dann für jede Teilstruktur beweist, daß sie korrekt ist. Um den Beweis zu führen, stehen Verifikationsregeln und Axiome zur Verfügung, wobei Verifikationsregeln in Form einer Schlußregel angegeben werden:

$$\frac{Prämissen}{Konklusion}$$

Dabei sind Prämissen die Voraussetzungen, die aus Korrektheitsformeln oder auch Zusicherungen bestehen, wohingegen die Konklusion die Schlußfolgerung bezeichnet, die aus der Gültigkeit der Prämissen folgt. Die Konklusion besteht daher immer aus genau einer Korrektheitsformel.

Da Axiome von Natur aus keine Prämissen haben, läßt man bei der Notation von Axiomen auch den Strich weg und schreibt nur die Korrektheitsformel nieder.

Folgende Verifikationsregeln und Axiome stehen zur Verfügung, um die partielle Korrektheit von Programmen zu beweisen:

Axiom 1: Zuweisungsaxiom

$$\{r[x := A]\}x := A\{r\}$$

Der Ausdruck $x := A$ bezeichnet eine Zuweisung, die den Wert der Variablen x verändert[2].

Das Axiom besagt nun, daß durch die Ersetzung aller x durch A in der Nachbedingung $\{r\}$ eine passende Vorbedingung $\{r[x := A]\}$ ermittelt werden kann.

[2]Die in diesem Kapitel enthaltenen Beispiele verwenden jedoch das Gleichheitszeichen (=) als Zuweisungsoperator, da es sich um Java-Quelltexte handelt.

Beispiel:

$\{pre?\}\ x = x + 25\ \{x = 2y\}$

Ersetzt man in der Nachbedingung alle x durch $x + 25$,
führt das zu der Vorbedingung $\{x + 25 = 2y\}$.
Daraus ergibt sich dann:

$\{x + 25 = 2y\}\ x = x + 25\ \{x = 2y\}$

Regel 2: Sequentielle Komposition

$$\frac{\{p\}\, A_1\, \{q\},\ \{q\}\, A_2\, \{r\}}{\{p\}\, A_1,\ A_2\, \{r\}}$$

Diese Regel besagt, daß zwei Programmteile A_1 und A_2 zusammengesetzt werden können, wenn die Vorbedingung von A_2 gleich der Nachbedingung von A_1 ist.

Regel 3: Bedingte Anweisung

$$\frac{\{q\ \textbf{and}\ B\}\, A_1\, \{r\},\ \{q\ \textbf{and not}\ B\}\, A_2\, \{r\}}{\{q\}\textbf{if}\ B\ \textbf{then}\ A_1\ \textbf{else}\ A_2\{r\}}$$

Die Programmteile A_1 und A_2 können zu einer **if**-Anweisung zusammengesetzt werden, sofern sie die gleiche Nachbedingung $\{r\}$ haben und für die Vorbedingungen gilt, daß $\{q\ \textbf{and}\ B\}$ eine Vorbedingung für A_1 ist und $\{q\ \textbf{and not}\ B\}$ eine Vorbedingung für A_2 ist.

Regel 4: Schleife

$$\frac{\{p\ \textbf{and}\ B\}\, A\, \{p\}}{\{p\}\ \textbf{while}\ B\ \textbf{do}\ A\, \{p\ \textbf{and not}\ B\}}$$

Die Schleifenregel besagt, daß wenn die Zusicherung p nach jeder Ausführung des Schleifenrumpfes A gültig ist, dann p auch nach der Terminierung der Schleife **while** B **do** A gültig ist.

Bei p spricht man von der *Schleifeninvariante*, da p jedesmal gültig sein muß, wenn die Schleifenbedingung B geprüft wird. Diese Schleifeninvariante beschreibt also den Teil der Schleife, der immer gleich bleibt. Das Herausfinden einer sinnvollen Schleifeninvariante ist eine der Hauptschwierigkeiten beim Beweisen von partieller Korrektheit.

Regel 5: Konsequenzregel

$$\frac{p \Rightarrow p_1,\ \{p_1\}\, A\, \{q_1\},\ q_1 \Rightarrow q}{\{p\}\, A\, \{q\}}$$

Die Konsequenzregel besagt, daß jederzeit eine Vorbedingung p_1 durch eine "schärfere" Vorbedingung p und eine Nachbedingung q_1 durch eine "schwächere" Nachbedingung q ersetzt werden kann.

2.5 Beispielhafter Beweis

Um die Anwendung des Beweissystems zu veranschaulichen, wird die *partielle Korrektheit* bezüglich der Vorbedingung $\{x \geq 0$ **and** $y \geq 0\}$ und der Nachbedingung $\{quo \cdot y + rem = x$ **and** $0 \leq rem < y\}$ für die folgende Operation bewiesen:

```
01:  void div ( int x, int y )
02:  {
03:      int quo = 0;
04:      int rem = x;
05:
06:      while ( rem >= y ) {
07:          rem -= y;
08:          quo++;
09:      }
10:  }
```

Abbildung 2.1: Berechnung des Quotienten und des Rests für x/y

Um den Beweis durchzuführen, muß das Programm in Teilstücke zerlegt werden, auf die sich dann die Regeln bzw. das Axiom anwenden lassen. Betrachtet man das Programm, so fällt auf, daß es sich grob in zwei Teile zerlegen läßt. Den ersten Teil bilden die Zuweisungen in Zeile 3 und Zeile 4, den zweiten Teil bildet die Schleife von Zeile 6 bis Zeile 9. Diese grobe Aufteilung ist in der Abbildung 2.2 im ersten Schritt veranschaulicht (Anwendung der Sequenzregel und der Konsequenzregel). Der zweite Schritt in Abbildung 2.2 verdeutlicht die feinere Aufteilung des Zuweisungsteils (hellgraue Kästchen) und der Schleife (dunkelgraue Kästchen).

Die Pfeile zwischen den Zusicherungen zeigen immer von unten nach oben. Dies soll die Rückwärtsausrichtung des Zuweisungsaxioms veranschaulichen,

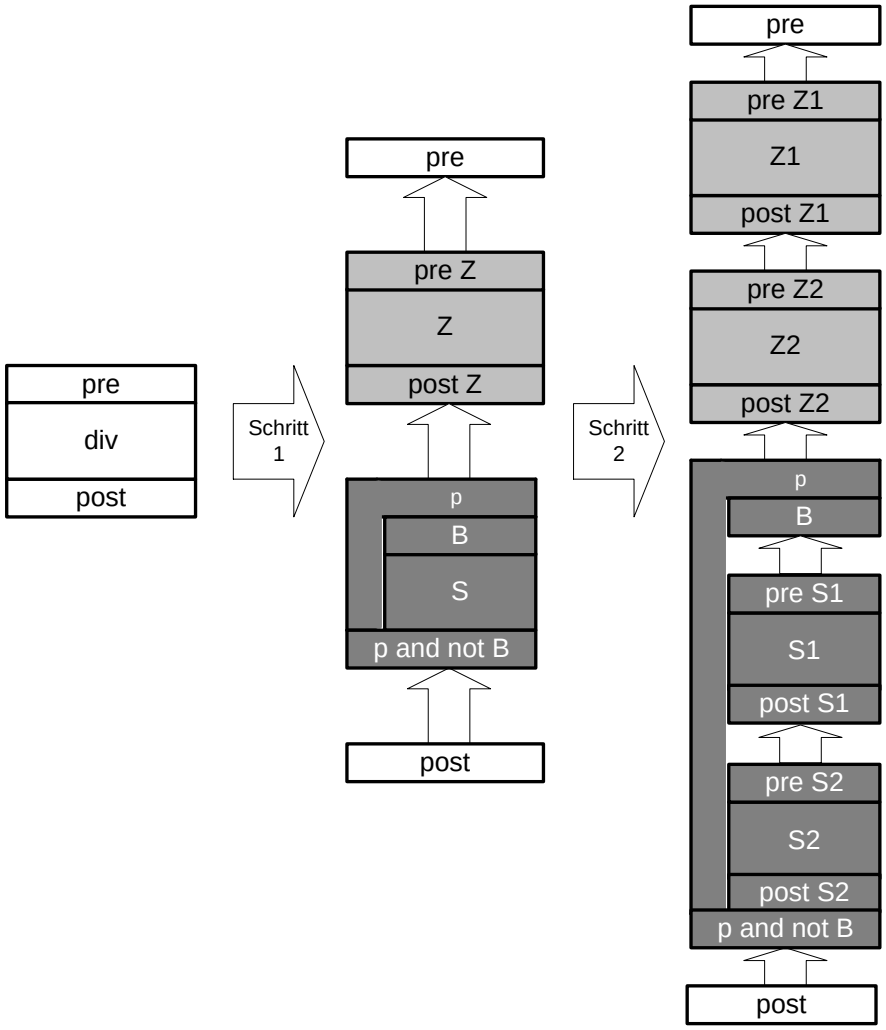

Abbildung 2.2: Veranschaulichung des Beweises

die maßgeblich für die Vorgehensweise des Beweises ist. Unter Anwendung der Regeln und des Axioms wird nun gezeigt, daß sich aus der gegebenen Nachbedingung $\{post\}$ die gegebene Vorbedingung $\{pre\}$ herleiten läßt (d.h. im Schaubild verläuft der Beweis von unten nach oben).

Die Schleife

Da die Schleife die letzte Anweisung des Programms ist, muß zuerst gezeigt werden, daß die Nachbedingung der Schleife äquivalent zu der gegebenen Nachbedingung $\{post\}$ ist.

Gesucht wird zunächst die Nachbedingung der Schleife. Die Konklusion der Schleifenregel besagt, daß sich die Nachbedingung der Schleife aus der Schleifeninvariante $\{p\}$ und der Negation der Abbruchbedingung $\{\textbf{not } B\}$ zusammensetzt. Die Abbruchbedingung $\{B\}$ der Schleife ist offensichtlich:

$$\{B\} \equiv \{rem \geq y\} \Rightarrow \{\textbf{not } B\} \equiv \{y > rem\}$$

Damit man die Nachbedingung $\{p \textbf{ and not } B\}$ der Schleife bilden kann, muß allerdings noch die Schleifeninvariante $\{p\}$ ermittelt werden. Nach einigem Ausprobieren und Überlegen wählt man als Schleifeninvariante $\{p\}$:

$$\{p\} \equiv \{quo \cdot y + rem = x \textbf{ and } rem \geq 0\}$$

Setzt man nun $\{p\}$ und $\{\textbf{not } B\}$ zusammen, so erhält man die Nachbedingung der Schleife. Formt man diese entsprechend um, so ist die Äquivalenz zwischen $\{p \textbf{ and not } B\}$ und $\{post\}$ zu erkennen:

$$\begin{aligned} \{p \textbf{ and not } B\} &\equiv \{quo \cdot y + rem = x \textbf{ and } rem \geq 0 \textbf{ and } y > rem\} \\ &\Leftrightarrow \{quo \cdot y + rem = x \textbf{ and } 0 \leq rem < y\} \\ &\Leftrightarrow \{post\} \end{aligned}$$

Mit Hilfe der Konklusion der Schleifenregel wurde gezeigt, daß $\{p \textbf{ and not } B\}$ äquivalent zu *post* ist. Die Konklusion folgt jedoch erst aus der Gültigkeit der Prämisse $\{p \textbf{ and } B\}A\{p\}$. Daher muß die Gültigkeit der Prämisse nachgewiesen werden. A steht stellvertretend für den Schleifenrumpf, der sich in diesem Beispiel aus den Zuweisungen $S1$ und $S2$ zusammensetzt. Zu zeigen ist also, daß sich aus $\{p\}$ eine Vorbedingung $\{p \textbf{ and } B\}$ entwickeln läßt. Da sich der Schleifenrumpf aus zwei Zuweisungen zusammensetzt, muß zuerst von $\{p\}$ aus eine Vorbedingung für $S2$ ermittelt werden:

$$\begin{array}{lcl} \{pre\ S2\} & S2 & \{p\} \\ \{pre\ S2\} & quo{+}{+} & \{p\} \\ \{pre\ S2\} & quo = quo + 1 & \{p\} \\ \{pre\ S2\} & quo = quo + 1 & \{quo \cdot y + rem = x \textbf{ and } rem \geq 0\} \end{array}$$

$$\Rightarrow \{pre\ S2\} \equiv \{(quo + 1) \cdot y + rem = x \textbf{ and } rem \geq 0\}$$

Die Vorbedingung $\{pre\ S2\}$ wurde mit Hilfe des Zuweisungsaxioms berechnet. Sie wird nun für die Nachbedingung $\{post\ S1\}$ eingesetzt (Sequenz-Regel), um die Vorbedingung $\{pre\ S1\}$ zu berechnen:

$$
\begin{array}{lll}
\{pre\ S1\} & S1 & \{post\ S1\} \\
\{pre\ S1\} & S1 & \{pre\ S2\} \\
\{pre\ S1\} & rem{-}{=}\, y & \{pre\ S2\} \\
\{pre\ S1\} & rem = rem - y & \{pre\ S2\} \\
\{pre\ S1\} & rem = rem - y & \{(quo+1)\cdot y + rem = x \textbf{ and } rem \geq 0\}
\end{array}
$$

$$\Rightarrow \{pre\ S1\} \equiv \{(quo+1)\cdot y + (rem - y) = x \textbf{ and } (rem - y) \geq 0\}$$

Die Vorbedingung $\{pre\ S1\}$ wurde wieder mit Hilfe des Zuweisungsaxioms berechnet. Jetzt muß $\{pre\ S1\}$ lediglich noch umgeformt werden, um die Äquivalenz zu $\{p \textbf{ and } B\}$ zu zeigen:

$$
\begin{array}{rcl}
\{pre\ S1\} & \equiv & \{(quo+1)\cdot y + (rem - y) = x \textbf{ and } (rem - y) \geq 0\} \\
& \Leftrightarrow & \{quo\cdot y + y + (rem - y) = x \textbf{ and } (rem - y) \geq 0\} \\
& \Leftrightarrow & \{quo\cdot y + y + rem - y = x \textbf{ and } (rem - y) \geq 0\} \\
& \Leftrightarrow & \{quo\cdot y + rem = x \textbf{ and } (rem - y) \geq 0\} \\
& \Leftrightarrow & \{quo\cdot y + rem = x \textbf{ and } rem \geq 0 \textbf{ and } rem \geq y\} \\
& \Leftrightarrow & \{p \textbf{ and } B\}
\end{array}
$$

Damit ist die Gültigkeit der Prämisse bewiesen.

Der Zuweisungsteil

Der Zuweisungsteil läßt sich in zwei Zuweisungen $Z1$ und $Z2$ zerlegen. Ausgehend von der Vorbedingung $\{p\}$ der Schleife, die als Nachbedingung $\{post\ Z2\}$ benutzt wird (Sequenzregel), soll nun mit Hilfe des Zuweisungsaxiom eine Vorbedingung $\{pre\ Z2\}$ ermittelt werden:

$$
\begin{array}{lll}
\{pre\ Z2\} & Z2 & \{post\ Z2\} \\
\{pre\ Z2\} & Z2 & \{p\} \\
\{pre\ Z2\} & rem = x & \{p\} \\
\{pre\ Z2\} & rem = x & \{quo\cdot y + rem = x \textbf{ and } rem \geq 0\}
\end{array}
$$

$$\Rightarrow \{pre\ Z2\} \equiv \{quo\cdot y + x = x \textbf{ and } x \geq 0\}$$

Die ermittelte Vorbedingung $\{pre\ Z2\}$ wird als Nachbedingung für $Z1$ eingesetzt (Sequenzregel), um mit Hilfe des Zuweisungsaxioms eine Vorbedingung $\{pre\ Z1\}$ zu finden:

$$\begin{array}{lll} \{pre\ Z1\} & Z1 & \{post\ Z1\} \\ \{pre\ Z1\} & Z1 & \{pre\ Z2\} \\ \{pre\ Z1\} & quo = 0 & \{pre\ Z2\} \\ \{pre\ Z1\} & quo = 0 & \{quo \cdot y + x = x \textbf{ and } x \geq 0\} \end{array}$$

$$\Rightarrow \{pre\ Z1\} \equiv \{x = x \textbf{ and } x \geq 0\}$$

Zuletzt wird noch gezeigt, daß die ermittelte Vorbedingung $\{pre\ Z1\}$ durch die gegebene Vorbedingung $\{pre\}$ impliziert wird:

$$\{pre\} \Rightarrow \{pre\ Z1\}$$

$$\{x \geq 0 \textbf{ and } y \geq 0\} \Rightarrow \{x = x \textbf{ and } x \geq 0\}$$

Damit ist die partielle Korrektheit der Operation *div* bezüglich der Vorbedingung $\{x \geq 0 \textbf{ and } y \geq 0\}$ und der Nachbedingung $\{quo \cdot y + rem = x \textbf{ and } 0 \leq rem < y\}$ bewiesen.

Die Operation ist tatsächlich nur partiell korrekt, da sie für den gültigen y-Wert 0 nicht terminiert.

2.6 Fazit

Mit Hilfe von theoretischen Analysen, wie dem vorgestellten Beweissystem, läßt sich die Korrektheit von Programmen beweisen. Damit ist es möglich, dafür zu garantieren, daß sich ein Programm entsprechend seiner Spezifikation verhält.

Das Beispiel hat gezeigt, daß es sehr aufwendig ist, den Korrektheitsbeweis "von Hand" durchzuführen. Daher gibt es Programme, die den Entwickler bei der Verifikation seiner Module unterstützen. Diese Programme können jedoch meistens nicht automatisch die Korrektheit eines Programms beweisen (also die Konsistenz zwischen Spezifikation und Implementierung), sondern lediglich die lästigen (automatisierbaren) Routineaufgaben erledigen, die während der Verifikation anfallen.

Verifikation ist ein überwiegend statischer Ansatz, da sich Programme erst verifizieren lassen, wenn sie fertig implementiert sind. Es gibt zwar Ansätze, Beweise zu modularisieren[3], doch im allgemeinen können Beweise nur schlecht wiederverwendet werden. Hier ist auch ein Problem bei der

[3]Unter[url prover] findet man eine sehr umfangreiche Auflistung von verschiedensten Verifikationsprogrammen.

praktischen Anwendung der Verifikation zu sehen, denn starre Methodiken lassen sich eher schlecht in einen (üblicherweise verwendeten) iterativen Softwareentwicklungsprozeß integrieren.

Kapitel 3

Design by Contract

3.1 Einleitung

Im vorangegangenen Kapitel wurde gezeigt, wie sich die Korrektheit eines Programms durch Verifikation beweisen läßt. Diese Methodik ist jedoch sehr starr und - trotz Toolunterstützung - mit einem gewissen Aufwand verbunden, was ihre Praxistauglichkeit zumindest in Frage stellt.

In diesem Kapitel wird *Design by Contract*[1] als praxisorientierte Methode vorgestellt, die dabei hilft, die Korrektheit von Programmen sicherzustellen.

3.2 Design by Contract

Design by Contract wurde von *Bertrand Meyer* entwickelt [Meyer97] und ist ein wesentlicher Bestandteil der Programmiersprache *Eiffel*. Die Ideen und Konzepte von Design by Contract sind jedoch nicht auf Eiffel limitiert und lassen sich zu großen Teilen in vielen Programmiersprachen (insbesondere in objektorientierten Programmiersprachen) umsetzen.

Die Kernidee von Design by Contract ist, ganz ähnlich wie in dem vorgestellten Beweissystem, die Verwendung von Zusicherungen, um das Verhalten von Operationen und Klassen zu beschreiben. Zusicherungen bei Design by Contract unterscheiden sich jedoch von den Zusicherungen des Beweissystems dadurch, daß sie keine mathematischen Notationen sind, sondern Konstrukte, die sich durch die Programmiersprache ausdrücken lassen. Genauer ausgedrückt: Zusicherungen sind Ausdrücke, die einen boolschen Wert zurückliefern. Die Zusicherungen werden während der Laufzeit des Programms über-

[1] *Design by Contract* ist ein eingetragenes Markenzeichen der Firma Interactive Software Engineering (ISE)

prüft, d.h. sie werden nicht benutzt, um die Korrektheit eines Programms zu beweisen, sondern um das korrekte Verhalten eines Programms zur Laufzeit zu überprüfen.

Durch die Tatsache, daß Zusicherungen in der Programmiersprache ausgedrückt werden können, ist man nun in der Lage, im Quelltext selbst das Verhalten eines Programms zu spezifizieren. Das ist ein sehr wichtiger Aspekt, denn Programmiersprachen sind eigentlich nur geeignet, das *Wie* (etwas getan werden soll) zu spezifizieren. Das *Was* (wird gemacht) kann höchstens in Kommentaren beschrieben werden, die jedoch nur einen informellen harakter haben.

Die Zusicherungen, die bei Design by ontract zur Verfügung stehen, sind: Vorbedingungen, Nachbedingungen und Invarianten. Dabei können Vorbedingungen und Nachbedingungen, analog zu den Vor- und Nachbedingungen des Beweissystems, als Beschreibung dessen verstanden werden, was vor bzw. nach dem Ausführen einer Operation gelten muß. Invarianten dürfen in diesem Kontext jedoch nicht mit den *Schleifeninvarinaten* des Beweissystems verwechselt werden. Invarianten beschreiben bei Design by ontract vielmehr das Verhalten einer Klasse.

3.3 V■rträg■

Im Alltag dienen Verträge dazu, zwei Parteien an ihre Rechte und Pflichten zu binden. Im Regelfall profitieren beide Parteien aus diesem Vertragsverhältnis. Als Beispiel sei hier der Mietvertrag genannt, der den Mieter verpflichtet dem Vermieter einen gewissen Geldbetrag zu überweisen und den Vermieter verpflichtet, dem Mieter Wohnraum zur Verfügung zu stellen. Beide profitieren aus diesem Vertrag, denn der Mieter kann die Wohnung nutzen und der Vermieter erhält Geld.

Das Zusammenspiel von Objekten mit Zusicherungen kann ebenfalls als Vertragsverhältnis verstanden werden. Objekte interagieren miteinander, indem sie Nachrichten untereinander verschicken, d.h. Objekte rufen Operationen von anderen Objekten auf. Dabei nimmt das aufrufende Objekte die Rolle des Kunden (lient) ein und das aufgerufene Objekte die Rolle des Anbieters (Supplier). Der Kunde muß nun dafür Sorge tragen, daß die Vorbedingungen erfüllt sind, wohingegen die Einhaltung der Nachbedingungen und der Invariante in der Verantwortung des Anbieters liegen.

3.4 Defensives Programmieren

Defensives Programmieren wird oft als Schlüssel zur Wiederverwendbarkeit gesehen. Bei defensiver Programmierung wird das Programm derart gestaltet, daß es mit jeder Art von Eingabe zurecht kommt. Das führt jedoch bei großen Programmen dazu, daß die einzelnen Klassen durch zahlreiche *if-then-else*-Statements "aufblähen". Oft kommt es auch vor, daß auf bestimmte Eingaben "unnatürlich" reagiert wird, da eine Klasse eigentlich gar nicht in der Lage ist, auf spezielle Eingabe angemessen zu reagieren. Oder was noch schlimmer ist, die Klasse reagiert auf bestimmte Eingaben entsprechend dem Kontext, in den sie eingebettet ist. Wenn das geschieht ist die Klasse in anderen Programmen nicht mehr wiederverwendbar, da der Kontext möglicherweise ein anderer ist.

Da bei Design by Contract die Verträge ganz explizit Teil der Schnittstelle einer Klasse sind, ist somit genau definiert, in welchem Kontext und unter welchen Voraussetzungen die Klasse bzw. eine bestimmte Operation benutzt werden kann.

Dem lieg die Idee zugrunde, daß eine Operation ihre Arbeit nur dann beginnt, wenn auch die Vorbedingungen erfüllt sind. Sind die Vorbedingungen nicht erfüllt, so liegt es nicht in der Verantwortung der Operation, angemessen auf diesen Umstand zu reagieren. In [Meyer97] wird dies als *Non-Redundancy principle* zur Regel gemacht, die besagt, daß unter keinen Umständen eine Vorbedingung im Quelltext nocheinmal geprüft werden darf, denn die Verletzung einer Vorbedingung (bzw. einer Zusicherung ganz allgemein) ist kein Sonderfall, sondern die Manifestierung eines Fehlers im Programm. Dabei deutet eine nicht eingehaltene Vorbedingung auf einen Fehler beim Kunden hin. Eine Verletzung der Nachbedingung oder der Invariante zeigt einen Fehler beim Anbieter an.

Ein wesentlicher Aspekt bei der Verwendung von Design by Contract ist, daß Verträge ausschließlich für die Kommunikation von Objekten untereinander gedacht sind. D.h. sobald ein Objekt mit der "Außenwelt" in Interaktion tritt (z.B. Benutzer-Eingaben entgegennimmt oder Eingaben von einer Netzwerkverbindung erwartet), muß auf herkömmliche Weise (*if-then-else*) geprüft werden, ob die Eingaben adäquat sind oder nicht.

Damit ist auch die Brücke zum defensiven Programmieren geschlagen. Design by Contract sollte anstelle eines defensiven Programmierstils bei der Objekt-zu-Objekt Interaktion eingesetzt werden, um die Wiederverwendbarkeit zu steigern. Bei der Objekt-zu-Außenwelt Interaktion muß jedoch nach wie vor defensiv programmiert werden, um das Programm robust zu gestalten.

3.5 Abgrenzung zu Assert

Einige Programmiersprachen (z.B. , ++) stellen eine *assert*-Funktion bzw. ein Makro bereit, mit dem getestet werden kann, ob eine Bedingung an einer bestimmten Stelle des Programms erfüllt ist oder nicht. Diese Bedingung ist, genau wie bei den Zusicherungen eines Vertrags, ein boolscher Ausdruck.

Ein Assert stellt jedoch keinen Vertrag dar, denn Verträge sind explizit Teil der Schnittstelle einer Klasse, um die Semantik zu beschreiben. Dadurch daß Asserts nur an bestimmten Stellen des Quelltextes "von Hand" eingefügt werden, besitzen sie nicht die Explizitheit von Verträgen und können daher nur schwer die Semantik einer Klasse bzw. Operation beschreiben.

Ein weiterer Nachteil von Asserts gegenüber Design by ontract ist, daß bei einem Fehlschlagen eines Asserts nicht offensichtlich ist, ob der Kunde oder der Anbieter für das Fehlschlagen des Asserts verantwortlich ist. Bei Design by ontract sind die Verantwortlichkeiten klar definiert, denn der Kunde muß für die Einhaltung der Vorbedingungen sorgen und der Anbieter muß die Einhaltung der Invarianten und Nachbedingungen sicherstellen.

3.6 Ein konkretes Beispiel

Im folgenden wird ein Beispiel gezeigt, um die Anwendung von Verträgen zu verdeutlichen. Das Beispiel ist entnommen aus [Hunt/David00] wurde jedoch an die Syntax des *sushi*-Tools[2] angepaßt.

Gegeben ist die Schnittstelle für einen Küchenmixer:

```
public interface Blender
{
    public int     getSpeed();
    public void    setSpeed( int s );
    public boolean isFull();
    public void    fill();
    public void    empty();
}
```

Der Mixer kann mit 10 verschiedenen Geschwindigkeiten betrieben werden. Beim Wechseln der Geschwindigkeit ist darauf zu achten, daß die Geschwindigkeit immer nur um eine Einheit erhöht bzw. verringert werden kann. Der

[2] *sushi* ist ein Präprozessor für die Programmiersprache Java, der Verträge aus speziellen Javadoc-Kommentaren entnimmt. Er wurde im Rahmen der vorliegenden Arbeit entwickelt.

Mixer darf nur gefüllt werden, wenn er leer ist und kann natürlich nur geleert werden, wenn er voll ist.

```
/**
 * @invariant: Range check : getSpeed()>=0 && getSpeed()<10;
 */
public interface Blender
{
  public int     getSpeed();
  public boolean isFull();

  /**
   * @require: Only change by one : Math.abs(getSpeed()-s)<=1;
   * @require: Range check        : s >= 0 && s < 10;
   * @ensure:  s assigned         : getSpeed() == s;
   */
  public void    setSpeed( final int s );


  /**
   * @require: Don't fill it twice : !isFull();
   * @ensure:  Ensure it was done  : isFull();
   */
  public void    fill();


  /**
   * @require: Don't empty it twice : isFull();
   * @ensure:  Ensure it was done   : !isFull();
   */
  public void    empty();
}
```

Anhand des Beispiels lassen sich einige Merkmale von Design by ontract sehr gut veranschaulichen. Die Syntax der Zusicherungen ist sehr einfach gehalten und lehnt sich an die Syntax von Eiffel an. Zuerst kommt ein Schlüsselwort, das den Typ der Zusicherung bezeichnet. Dabei zeigt das Schlüsselwort *require* an, daß es sich um eine Vorbedingung handelt, das Schlüsselwort *ensure* kennzeichnet eine Nachbedingung und *invariant* verdeutlicht, daß es sich um eine Invariante handelt. Diesem Schlüsselwort folgt ein optionales Label, das bei der Verletzung eines Vertrages zur Laufzeit ausgegeben wird (zusam-

men mit einem Stacktrace und der Art der Vertragsverletzung). Als letztes wird die Bedingung angegeben, die geprüft werden soll.

3.7 Die Prüfbedingung

Die Prüfbedingung einer Zusicherung ist ein boolscher Ausdruck, der zur Laufzeit geprüft wird. Liefert der Ausdruck bei der Überprüfung `false`, so ist der Vertrag verletzt worden.

Bei der Formulierung des Vertrags bzw. der Bedingung sind einige Dinge zu beachten.

Alle Operationen, die in einer Vorbedingung benutzt werden, müssen dem aufrufenden Objekt "zugänglich" sein. Der Aufrufer muß die hance haben, überprüfen zu können, ob er eine Vorbedingung überhaupt erfüllen kann. Wäre beispielsweise in der obigen Schnittstelle des Mixers die Operation `isFull()` als `private`[3] dekaliert worden, so könnte man vor einem Aufruf der Operationen `fill()` und `empty()` niemals überprüfen, ob die Vorbedingung erfüllt ist. Alle Operationen, die in der Prüfbedingung einer Vorbedingung benutzt werden, müssen also mindestens die gleiche Sichtbarkeit haben wie die Operation, für die die Vorbedingung gelten soll. Dies entspricht der *Precondition Availability rule* [Meyer97] und wird in Eiffel sogar automatisch geprüft.

Es ist "guter Stil" innerhalb der Prüfbedingungen, nicht direkt auf Attribute der Klasse zuzugreifen. Statt dessen sollte immer über Operationen auf Attribute Bezug genommen werden. Wird nämlich direkt auf ein Attribut zugegriffen, so ist der Vertrag sehr stark an eine konkrete Implementierung gebunden.

3.8 Vererbung von Verträgen

Vererbung ist ein wichtiger Aspekt der objektorientierten Programmierung, da Vererbung (neben Delegation) eine Methode ist, um die bestehende Funktionalität von vorhandenen Klassen wiederverwenden zu können.

In der Programmiersprache Java kann Vererbung in zwei unterschiedlichen Ausprägungen verwendet werden. Die beiden Ausprägungsformen sind *Schnittstellenvererbung* und *Klassenvererbung*[4]. Beide Vererbungsformen

[3] abgesehen von der Tatsache, daß Java lediglich in Klassen, nicht aber in Schnittstellen, die Deklaration von `private` Methoden erlaubt

[4] Zur Unterscheidung dienen die Schlüsselwörter `implements` (Schnittstellenvererbung) und `extends` (Klassenvererbung).

bewirken *Subtyping*[5], definieren also eine Typ-Subtyp Beziehung zwischen Schnittstelle bzw. Oberklasse und Klasse.

Schnittstellenvererbung ist im Gegensatz zur Klassenvererbung jedoch reines Subtyping, d.h. es wird lediglich die Menge aller durch die Operationen definierten Signaturen vererbt.

Klassenvererbung geht über Subtyping hinaus, da zusätzlich zur Schnittstelle auch die Implementierungen der Operationen vererbt werden.

Da die Methodik Design by ontract Verträge (Vorbedingungen, Nachbedingungen und Invarianten) explizit als Teil der Schnittstelle definiert, ist es notwendig, daß Verträge ebenfalls vererbt werden.

Im folgenden wird daher gezeigt, wie sich Vererbung auf die Zusicherungen der Verträge auswirkt und wie abgeleitete Klassen ihrerseits Verträge definieren bzw. redefinieren können. Insbesondere soll Design by ontract die durch Vererbung definierten Typ-Subtyp Beziehungen nicht verletzen.

3.8.1 Vererbung von Vor- und Nachbedingungen

Erbt eine Klasse K_{sub} von einer anderen Klasse (oder Schnittstelle) K, so sollten sowohl die Vor- als auch die Nachbedingungen der Operationen von K auch für die entsprechenden Operationen von K_{sub} gelten, denn ein Aufrufer der Klasse K_{sub}, kennt diese Klasse selbst möglicherweise gar nicht, da er sie nur über ihre Basisklasse (oder Schnittstelle) K anspricht[6]. Da der Aufrufer jedoch die Klasse K kennt und sich auf den Vertrag der Klasse K verläßt, muß der Vertrag auch in Unterklassen Gültigkeit haben. Anderenfalls wäre Polymorphie in Verwendung mit Verträgen nicht sinnvoll.

Das gilt natürlich nicht nur für Operationen, deren Funktionalität in einer abgeleiteten Klasse einfach nur geerbt wurde, sondern auch für Operationen, die in einer Unterklasse überschrieben wurden und somit dynamisch gebunden werden. Bei überschriebenen Operationen wird die Sache jedoch interessanter, da bei der Redefinition einer Operation auch der Vertrag der Operation geändert werden kann.

Das Liskovsche Substitutionsprinzip

Die obigen Forderungen manifestieren sich auch im *Liskovschen Substitutionsprinzip (LSP)*[Liskov/Wing94], denn es besagt, daß eine Basisklasse nur

[5]Ein Typ ist ein Name, der eine bestimmte Schnittstelle bezeichnet. Von einem Subtyp spricht man, wenn ein Typ die Schnittstelle eines Supertyps enthält.

[6]Dieses Szenario ergibt sich sehr häufig, wenn man mit Bibliotheken oder Frameworks arbeitet, die Implementierungen von Schnittstellen nur über eine Factory anbieten.

dann durch eine abgeleitete Klasse substituiert werden kann, wenn die abgeleitete Klasse alle Eigenschaften der Basisklasse erfüllt. Die Einhaltung des LSP ist sehr wichtig, um die sinnvolle und korrekte Anwendung von Polymorphie zu gewährleisten.

Vorbedingungen

Vorbedingungen definieren, unter welchen Umständen eine Operation aufgerufen werden kann. Sie beschreiben also die "Arbeit", die vom Aufrufer vor Benutzung einer Operation erledigt werden muß. Definiert eine überschriebene Operation nun eine neue Vorbedingung, so darf die neue Vorbedingung dem Aufrufer nicht mehr Arbeit abverlangen, als die Vorbedingung der überschriebenen Operation. Das liegt wieder in der Tatsache begründet, daß nur auf diese Weise garantiert werden kann, daß Polymorphie in allen Fällen funktioniert, weil das LSP gilt.

Eine redefinierte Operation darf in ihrer Vorbedingung jedoch weniger verlangen als die Vorbedingung der überschriebenen Operation. Auch in diesem Fall ist die Gültigkeit des LSP gewährleistet.

Die "Abschwächung" einer Vorbedingung wird dadurch sichergestellt, daß zur Laufzeit geprüft wird, ob die "veroderten" Vorbedingungen gültig sind. Es muß also gelten:

$$\{pre\}\mathbf{or}\{pre_{sub}\}Operation\{post\}$$

Dabei bezeichnet $\{pre\}$ die Vorbedingung der Operation aus der Basisklasse und $\{pre_{sub}\}$ die redefinierte Vorbedingung der Operation aus der abgeleiteten Klasse.

Nachbedingungen

Nachbedingungen geben Auskunft darüber, was von einer Operation geleistet wird. Der Aufrufer kann sich also darauf verlassen, daß nach Beendigung der Operation sämtliche Nachbedingungen erfüllt sind. Um die Einhaltung des LSP (und damit Polymorphie) zu gewährleisten, muß eine redefinierte Operation dafür Sorge tragen, daß alle Nachbedingungen der überschriebenen Operation gelten, da Aufrufer evtl. nur den Vertrag der überschriebenen Operation kennen und sich darauf verlassen, daß die dort versprochenen Nachbedingungen auch eingehalten werden.

Die Implementierung der redefinierten Operation kann jedoch leistungsfähiger sein als die der überschriebene Operation. D.h. es können Nachbedingungen angegeben werden, die zusätzlich zu den Nachbedingungen der überschriebenen Operation gelten.

Um sicherzustellen, daß Nachbedingungen nur "verstärkt" werden können, werden die Nachbedingungen der Operation aus der Basisklasse mit den Nachbedingungen der Operation aus der abgeleiteten Klasse "verundet". Daher wird zur Laufzeit folgendes geprüft:

$$\{pre\}Operation\{post\}\textbf{and}\{post_{sub}\}$$

$\{post\}$ ist die Nachbedingung der Operation aus der Basisklasse. $\{post_{sub}\}$ ist die redefinierte Nachbedingung der Operation aus der abgeleiteten Klasse. $\{pre\}$ bezeichnet sämtliche Vorbedingungen, die vor Aufruf der Operation erfüllt sein müssen.

Überspezifizierung

Bei der Definition von Verträgen sollte man immer die Auswirkungen, die die Vererbung auf Vor- und Nachbedingungen hat, bedenken. Es kann nämlich leicht passieren, daß Verträge so strikt definiert sind, daß praktisch keine Vererbung mehr möglich ist. Das nachfolgende Beispiel stellt eine solche Situation dar:

```
public interface PosReciprocCalculator
{
    /**
     * @require: Only positive numbers     : d > 0.0;
     * @ensure:  Result is always positive : $return > 0.0;
     */
    public double calcReciproc ( double d );
}
```

In der Schnittstelle `PosReciprocCalculator` wird die Operation `calcReciproc` definiert, die den Kehrwert der Zahl d zurückliefert. Die Vorbedingung der Operation besagt, daß der Kehrwert nur für Zahlen größer 0 berechnet wird. Die Nachbedingung der Operation garantiert, daß der berechnete Kehrwert immer größer 0 ist[7].

Nun wird eine zweite Schnittstelle `ReciprocCalculator` definiert, welche die vorhandene Schnittstelle `PosReciprocCalculator` erweitert:

[7] mittels `$return` wird dem *sushi* Präprozessor angezeigt, daß man sich auf den Rückgabewert einer Operation bezieht

```
public interface ReciprocCalculator extends PosReciprocCalculator
{
    /**
     * @require: Everything but zero  : d != 0.0;
     * @ensure:  Result is never zero : $return != 0.0;
     */
    public double calcReciproc ( double d );
}
```

Die Operation `calcReciproc` soll erweitert werden. Aus der Vorbedingung ist ersichtlich, daß die Operation jetzt auch den Kehrwert für negative Zahlen berechnen kann. In der Nachbedingung wird angegeben, daß der Kehrwert niemals 0 ist.

Bei genauerer Betrachtung fällt auf, daß die Nachbedingung aus der erweiterten Schnittstelle ungültig ist, da sie nicht "stärker" ist als die Nachbedingung aus der Basisschnittstelle (d.h. `$return != 0.0` impliziert nicht `$return > 0.0`).

Man könnte nun vermuten, daß sich das Problem durch Weglassen der ungültigen Nachbedingung löst. Das ist jedoch ein Trugschluß, da die Nachbedingung, die in der Basisschnittstelle definiert wurde, in allen abgeleiteten Klassen ihre Gültigkeit behalten muß, um Polymorphie zu gewährleisten.

Offensichtlich ist die Nachbedingung, zur Berechnung des Kehrwerts für positive Zahlen, so ungünstig, daß praktisch keine Vererbung möglich ist (zumindestens nicht, um die Operation so zu erweitern, daß auch für negative Zahlen der Kehrwert berechnet werden kann).

Die Schwierigkeit beim Definieren von Verträgen ist, zu erkennen, wann eine Zusicherung "ungünstig" ist. Hier wird bewußt nicht von "falschen" Zusicherungen gesprochen, denn die obige Nachbedingung, die in der Schnittstelle `PosReciprocCalculator` definiert wurde, ist durchaus richtig, da der Kehrwert jeder positiven Zahl immer positiv ist. Sie ist lediglich "ungünstig", da diese Nachbedingung eine sinnvolle Vererbung (im Sinne einer Erweiterung auf negative Zahlen) a priori ausschließt.

3.8.2 Vererbung von Invarianten

Invarianten beschreiben die globalen Eigenschaften, die für jedes Exemplar einer Klasse gelten. Damit unterscheiden sie sich maßgeblich von Vor- und Nachbedingungen, da diese die Eigenschaften einer Operation beschreiben. Mit Hilfe von Invarianten können Zusammenhänge zwischen verschiedenen Operationen beschrieben werden. Das folgende Beispiel verdeutlicht das:

```
/**
 * @invariant: Speed 0 means off : isOff() == (getSpeed()==0);
 * ...
 */
public interface Blender
{
    public boolean isOff();
    public int     getSpeed();
    // ...
}
```

Die Schnittstelle des Küchenmixers wurde um eine Operation `isOff()`[8] ergänzt, die anzeigt, ob der Mixer ausgeschaltet ist. Betrachtet man nun die Invariante, so erkennt man genau den Zusammenhang der beiden Operationen `isOff()` und `getSpeed()`. Nur wenn die Operation `getSpeed()` den Wert 0 liefert, gibt die Operation `isOff()` den Wert `true` zurück.

Die globalen Eigenschaften, die durch Invarianten beschrieben werden, können auch als *gültiger Zustand* eines Exemplars verstanden werden. Sie müssen daher direkt nach der Erzeugung eines Exemplars sowie vor und nach jedem Operationsaufruf gelten. Das Zusammenspiel zwischen Vorbedingungen, Nachbedingungen und Invarianten läßt sich daher durch folgende Regel beschreiben:

$$\{inv\}\textbf{and}\{pre\}Operation\{inv\}\textbf{and}\{post\}$$

Da Invarianten den gültigen Zustand eines Exemplars beschreiben, müssen sie auch in abgeleiteten Klassen ihre Gültigkeit bewahren, um Polymorphie zu gewährleisten. Der Anbieter darf daher lediglich Invarianten ergänzen, die dann zusätzlich zu den geerbeten Invarianten geprüft werden. Daher werden die geerbten Invarianten mit den Invarianten der redefinierten Klasse "verundet". Es gilt also:

$$\{inv\}\textbf{and}\{inv_{sub}\}\textbf{and}\{pre\}Operation\{inv\}\textbf{and}\{inv_{sub}\}\textbf{and}\{post\}$$

Dabei bezeichnet $\{inv\}$ die Invarianten der Basisklasse, $\{inv_{sub}\}$ die Invarianten der abgeleitenten Klasse, $\{pre\}$ sämtliche Vorbedingungen, die vor Aufruf der Operation gelten müssen und $\{post\}$ alle Nachbedingunen, die nach Beendigung der Operation erfüllt sein müssen.

[8] Die Operation könnte sinnvoll in den Vorbedingungen der Operationen `empty()` und `fill()` verwendet werden, um anzuzeigen, daß der Mixer nur geleert bzw. gefüllt werden darf, wenn das Gerät ausgeschaltet ist.

3.8.3 Diskussion über die Vererbung von Zusicherungen

Die dargestellten Regeln zur Vererbung von Zusicherungen wurden in Anlehnung an [Meyer97] beschrieben. Allerdings verbergen sich in den Konzepten einige Tücken, die im Folgenden diskutiert werden.

Betrachtet werden die folgenden beiden Schnittstellen:

```
public interface Person
{
    /**
     * @require: Positive age : age >= 0;
     */
    public void setAge ( int age );
    public int  getAge ();
}

public interface Adult extends Person
{
    /**
     * @require: Age greater or equal 18 : age >= 18;
     */
    public void setAge ( int age );
}
```

Interessant ist die Vorbedingung der Operation `setAge()` aus der Schnittstelle `Adult`. Die Intention dieser Vorbedingung ist klar, denn die Operation `setAge()` sollte auf einem Exemplar vom Typ `Adult` nur mit Werten aufgerufen werden, die 18 oder größer sind. Würde die Operation jedoch mit Werten aufgerufen, die im Bereich {0...17} liegen, so würde keine Verletzung der Vorbedingung angezeigt, da die Vorbedingung aus der Schnittstelle `Adult` mit der Vorbedingung aus der Schnittstelle `Person` "verodert" wird.

Es ist zwar durchaus als positiv zu bewerten, daß im dargestellten Fall keine Vertragsverletzung auftritt (denn sie würde einen Fehler beim Kunden anzeigen, obwohl er sich an den Vertrag aus der Basisklasse gehalten hat), aber auf der anderen Seite ist es ärgerlich, daß die Verletzung des Liskovschen Substitutionsprinzips nicht auffällt.

Die Schnittstelle `Adult` wird nun um eine Invariante erweitert:

```
/**
 * @invariant: Age always greater or equal 18 : getAge() >= 18;
 */
public interface Adult extends Person
{
    /**
     * @require: Age greater or equal 18 : age >= 18;
     */
    public void setAge ( int age );
}
```

Wird nun die `setAge()` Operation mit einem Wert im Bereich {0...17} aufgerufen, so führt das zu einer Verletzung der Invariante. Bei der Verwendung der Invariante fällt also im Gegensatz zur Vorbedingung die Verletzung des LSP auf.

Ein Fehlschlagen der Invariante kann daher nicht a priori als Fehler in der Implementierung des Anbieters gesehen werden, sondern es muß auch die Möglichkeit in Betracht gezogen werden, daß das Fehlschlagen einer Invariante möglicherweise durch eine Verletzung des LSP hervorgerufen wurde (etwa durch eine kovariante Redefinition wie im obigen Beispiel).

3.9 Laufzeitüberprüfung

Die Verträge werden zur Laufzeit überprüft. Vorbedingungen werden direkt nach dem Eintritt in eine Operation geprüft, d.h. bevor die Operation mit ihrer eigentlichen Arbeit beginnt. Nachbedingungen werden geprüft, bevor die Operation terminiert (d.h. unmittelbar vor einer `return` Anweisung bzw. nach der letzten Anweisung im Falle einer `void` Operation). Invarianten werden sowohl mit den Vor- als auch mit den Nachbedingungen geprüft.

Tritt eine Vertragsverletzung auf, so beendet sich das gesamte Programm und zeigt an, wo die Verletzung des Vertrags aufgetreten ist. Würde das Programm nicht beendet, so wäre das fatal, da eine Vertragsverletzung einen Fehler im Programm anzeigt, der behoben werden muß und nicht ignoriert werden darf.

Klassen können in unterschiedlicher Intensität mit Verträgen bestückt werden. Dabei können entweder nur die Vorbedingungen oder die Vorbedingungen und die Nachbedingungen oder die Vorbedingungen, die Nachbedingungen und die Invarianten aktiviert werden. Es ist z.B. nicht möglich, nur die Nachbedingungen zu aktivieren. Würde man die Nachbedingungen ohne die Vorbedingungen aktivieren, so würde zur Laufzeit nicht überprüft, ob der

Kunde alle Anforderungen erfüllt hat, um die Operation in Anspruch nehmen zu können. Wären die Anforderungen nicht erfüllt, so würde die Operation aber trotzdem mit ihrer Arbeit beginnen, da die nicht erfüllten Anforderungen, wegen der abgeschalteten Vorbedingungen nicht zu einer Vertragsverletzung führen würden. Das könnte dann in der Konsequenz dazu führen, daß die Operation die versprochene Nachbedingung nicht erfüllen kann. Der Fehler liegt in diesem Fall zwar klar beim Anbieter, da er die Vorbedingung nicht erfüllt, doch dieser Fehler würde nur schwer gefunden werden, da die nicht erfüllte Nachbedingung einen Fehler beim Anbieter vermuten ließe.

3.10 Seiteneffekte

Die unbedachte Verwendung von Design by Contract kann unter Umständen zu Seiteneffekten führen. Solche Seiteneffekte sind für den Programmierer äußerst unangenehm, da sie schwer zu finden sind. Im folgenden werden drei verschiedene Arten von Seiteneffekten beschrieben und wie man sie verhindern kann.

3.10.1 Endlose Rekursionen

Endlose Rekursionen treten immer dann auf, wenn ein Programmstück sich selber aufruft (bzw. verschiedene Programmstücke sich gegenseitig aufrufen), aber keine Bedingung definiert wurde, die irgendwann zum Abbruch der Aufrufe führt. Betrachtet wird das folgende Beispiel:

```
public interface List
{
    /**
     * @ensure: Length can't be negative : length() >= 0;
     */
    public int length();
}
```

Die Schnittstelle definiert eine Liste. Mittels der Operation `length()` kann die Liste über die Anzahl ihrer Elemente befragt werden. Die Nachbedingung der Liste besagt, daß die Anzahl der Elemente der Liste immer größer oder gleich 0 ist.

Das Problem liegt nun darin, daß in der Nachbedingung der Operation auf selbige Bezug genommen wird[9]. Da die Nachbedingung zur Laufzeit direkt

[9]Offensichtlich hätte man die Zusicherung der Nachbedingung besser wie folgt definiert (das eigentliche Problem ist jedoch von grundsätzlicher Natur): `$return >= 0`

vor dem Verlassen der Operation geprüft wird, führt das in diesem Beispiel dazu, daß sich die Operation vor dem Verlassen selber aufruft[10]. Dieser Aufruf wird sich dann wieder selber aufrufen und der folgende auch usw.

Als Faustregel gilt daher, daß weder die Vorbedingung noch die Nachbedingung auf die Operation Bezug nehmen darf, die sie spezifiziert.

Diese Faustregel ist jedoch kein Allheilmittel, denn wäre die Nachbedingung der Operation als Invariante formuliert worden, so wäre die Operation ebenfalls in eine endlose Rekursion gelaufen. Interessanterweise hätte die endlose Rekursion durch die Invariante schon früher begonnen, da die Invariante auch schon beim Eintritt in die Operation geprüft wird.

3.10.2 Heisenbug

Der Begriff Heisenbug[url Heisenbug][11] bezeichnet einen Fehler, der verschwindet bzw. das Verhalten des Systems verändert, wenn man versucht, diesen Fehler zu finden. Das folgende Beispiel veranschaulicht dieses Phänomen:

```
public interface Stack
{
    /**
     * @ensure: Element was pushed : element.equals ( pop() );
     */
    public void   push ( Object element );
    public Object pop  ();
}
```

Gegeben ist die Schnittstelle eines Stapelspeichers. Die Nachbedingung der Operation `push()` benutzt die Operation `pop()`, um festzustellen, ob das Element auch tatsächlich auf den Stapel gelegt wurde. Die Verwendung der Operation `pop()` ist allerdings problematisch, da sie den Zustand des Stapels verändert. In diesem Fall würde die Aktivierung der Nachbedingungen dazu führen, daß die Operation `push()` das Objekt auf den Stapel legt und die Nachbedingung der Operation dieses Objekt wieder entfernen würde (d.h. die Operation würde "von außen" gesehen nichts tun). Sind die Nachbedingungen nicht aktiviert, so würde die Operation wie erwartet funktionieren.

[10] Der *sushi* Präprozessor besitzt daher einen Mechanismus, um solche endlosen Rekursionen zu vermeiden. Wird Design by Contract jedoch ohne solche Unterstützung benutzt, so liegt es in der Verantwortung des Programmierers, endlose Rekursionen zu vermeiden.

[11] Der Begriff ist eine Anspielung auf die Heisenbergsche Unschärferelation.

Es sollte daher darauf geachtet werden, daß in Zusicherungen jeglicher Art keine Operationen verwendet werden, die zu einer Zustandsänderung des Systems führen.

3.10.3 Callbacks

Eine weitere Gefahrenquelle für mögliche Seiteneffekte stellen allbacks dar. Der allback-Mechanismus wird in der Praxis verwendet, um Zustandsänderungen eines Objekts an andere Objekte zu propagieren. In der objektorientierten Programmierung kann dieser Mechanismus mit Hilfe des *Observer*-Entwurfsmusters realisiert werden[Gamma et al.94]. Die Problematik wird anhand des folgenden Beispiels diskutiert:

```
public interface ObservableSet
{
    /**
     * @require: Not in List yet : !exists ( element );
     * @ensure:  Element added   :  exists ( element );
     */
    public void    add      ( Object   element  );


    /**
     * @require: Element exists  :  exists ( element );
     * @ensure:  Element removed : !exists ( element );
     */
    public void    remove   ( Object   element  );
    public boolean exists   ( Object   element  );
    public void    register ( Observer observer );
}


public interface Observer
{
    public void notifyElementAdded   ( Object element );
    public void notifyElementRemoved ( Object element );
}
```

Die Schnittstelle `ObservableSet` definiert eine Menge zum Verwalten von Elementen. Mittels der Operationen `add()` und `remove()` können der Menge Elemente hinzugefügt bzw. entfernt werden. Die Operation `exists()` prüft, ob sich das übergebene Element in der Menge befindet. Die Menge bietet

zusätzlich die Möglichkeit, `Observer`-Objekte zu registrieren, die benachrichtigt werden, falls sich der Zustand der Menge ändert. Dabei wird auf dem `Observer`-Objekt die Operation `notifyElementAdded()` aufgerufen, wenn der Menge ein Element hinzugefügt wurde und die Operation `notifyElementRemoved()`, wenn ein Element aus der Menge entfernt wurde. Den beiden Operationen wird jeweils das Objekt übergeben, daß hinzugefügt bzw. entfernt wurde. Das Problem liegt in der Benachrichtigung der `Observer`

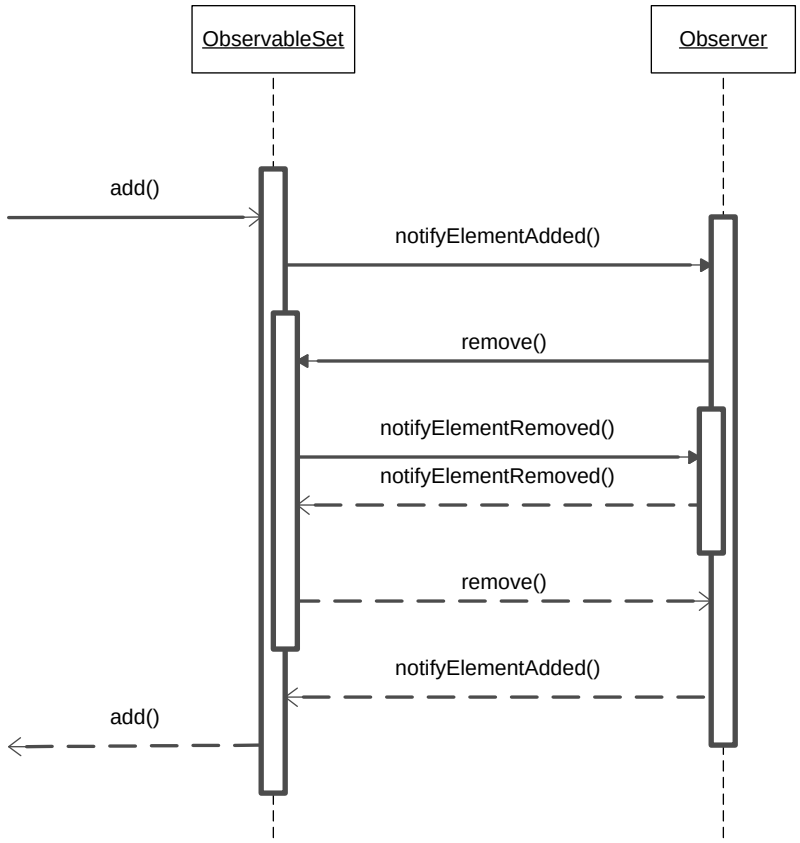

Abbildung 3.1: Ablaufsequenz für den Aufruf der Operation `add`

Objekte, da mit dem Aufrufen der Benachrichtigungsoperation die Kontrolle über den Programmfluß an das jeweilige `Observer`-Objekt abgegeben wird. Erst nach Beendigung der Benachrichtigungsoperationen gelangt die Kontrolle zurück an das benachrichtigende Objekt. Man stelle sich vor, daß ein `Observer`-Exemplar des folgenden Typs bei der Menge registriert wird:

```
public class BadApple implements Observer
{
    private ObservableSet _set;

    public void notifyElementAdded   ( Object element )
    {
        _set.remove ( element );
    }

    public void notifyElementRemoved ( Object element )
    { /* Empty implementation */ }
}
```

Der Einfachheit halber wird davon ausgegangen, daß das Exemplar das einzige `Observer`-Objekt ist, das bei der Menge angemeldet wurde.

Wird nun auf der Menge die Operation `add()` aufgerufen, so stellt sich die Situation wie folgt dar: Die Operation `add()` beginnt ihre Arbeit und fügt das Element der Menge hinzu. Danach benachrichtigt sie den `Observer` (ein Exemplar des Typs `BadApple`), der dann das gerade eingefügte Element wieder aus der Menge löscht. Der Kontrollfluß gelangt zurück an die Operation `add()`. Vor dem Terminieren der Operation wird die Nachbedingung (`exists(element);`) geprüft, die dann fehlschlägt, da das eingefügte Element in der Zwischzeit vom Observer gelöscht wurde. Das Sequenzdiagramm in Abbildung 3.1 veranschaulicht die Situation.

Durch das Fehlschlagen der Nachbedingung wird der Eindruck erweckt, daß der Fehler beim Anbieter liegt. Diese Schuldzuweisung ist allerdings unfair, da die Arbeit der Operation `add()` durch den `Observer` vom Typ `BadApple` kompromittiert wurde.

Zur Lösung des Problems schlägt *Clemens Szyperski* [Szyperski97] vor, daß Operationen, die den Zustand des Objektes ändern, während der Benachrichtigungsphase nicht aufgerufen werden dürfen. Diese Idee kann durch eine einfache Erweiterung der Schnittstelle umgesetzt werden:

```
public interface ObservableSet
{
    /**
     * @return true, if the notification is in progress.
     */
    public boolean inNotification ();
```

```
    /**
     * @require: Not in notification : !inNotification();
     * ...
     */
    public void    add            ( Object   element  );


    /**
     * @require: Not in notfication  : !inNotification();
     * ...
     */
    public void    remove         ( Object   element  );
    // ...
}
```

Die Operation `inNotification()` kann sehr einfach mit Hilfe eines Flags implementiert werden. Bevor die Opertionen `add()` bzw. `remove()` mit der Benachrichtigung der `Observer`-Objekte beginnen, setzen sie das Flag. Nachdem der letzte `Observer` benachrichtigt wurde, wird das Flag dann wieder zurückgesetzt. Auf diese Weise würde dann im obigen Beispiel der `Observer` vom Typ `BadApple` gegen die Vorbedingung der `remove()`-Operation verstoßen und könnte somit nicht mehr die Verletzung der Nachbedingung der `add()`-Operation provozieren.

3.11 Grenzen von Design by Contract

Es wurde gezeigt, wie die Verwendung von Zusicherungen dabei hilft, das Verhalten von Klassen und Operationen zu beschreiben und wie die Prüfung der Zusicherungen zur Laufzeit sicherstellt, daß sich Klassen und Operationen korrekt, bezüglich der Vorbedingungen, Nachbedingungen und Invarianten verhalten.

Zusicherungen sind jedoch auf Ausdrücke der Programmiersprache begrenzt, die einen boolschen Wert zurückliefern. Dadurch lassen sich viele Sachverhalte, welche die Semantik von Klassen und Operationen beschreiben, nicht ausdrücken.

Betrachtet wird die Schnittstelle eines Stapelspeichers:

```
/**
 * @invariant: Empty : isEmpty() == (count()==0);
 * @invariant: Full  : isFull() == (count()==capacity());
 * @invariant: Count non negative : count() >= 0;
 * @invariant: Count bounded      : count() <= capacity();
 */
public interface Stack
{
   public boolean isEmpty  ();
   public boolean isFull   ();
   /**
    * @return the maximum number of items, that can be pushed.
    */
   public int     capacity ();

   /**
    * @return the number of items on the stack.
    */
   public int     count    ();

   /**
    * @require: Not empty : !isEmpty();
    */
   public Object  top      ();

   /**
    * @require: Not empty  : !isEmpty();
    * @ensure:  Not full   : !isFull();
    * @ensure:  One fewer  : ($old count()) - 1 == count();
    * @ensure:  Return top : $return.equals ($old top());
    */
   public Object  pop      ();

   /**
    * @require: Not full      : !isFull ();
    * @ensure:  Not empty     : !isEmpty();
    * @ensure:  Item on top   : item.equals ( top() );
    * @ensure:  One more item : ($old count()) + 1 == count();
    */
   public void    push     ( Object item );
}
```

Die Invarianten der Schnittstelle beschreiben das Zusammenspiel der Operationen `isEmpty()`, `isFull()`, `capacity()` und `count()`. Diese einfachen Zugriffsoperationen sind jedoch eher uninteressant, da die eigentliche Funktionalität eines Stapelspeichers in den Operationen `push()` und `pop()` liegt.

Zunächst werden die Zusicherungen der Operation `push()` untersucht:

Not full: Die Operation darf nur aufgerufen werden, wenn der Stapel noch nicht voll ist.

Not empty: Die Operation garantiert, daß nach ihrem Aufruf der Stapel nicht mehr leer ist.

Item on top: Nach dem Aufruf der Operation liefert die Operation `top()` das gleiche Element, das der Operation `push()` zuvor übergeben wurde.

One more item: Es wird garantiert, daß sich die Anzahl der Elemente des Stapels um eins erhöht hat[12].

In den Nachbedingungen der Operation `push()` werden zwei Axiome beschrieben, die wesentlich für die Funtionsweise eines Stapels sind. Zum einen ist das die Nachbedingung $\{Not\ empty\}$, zum anderen die Nachbedingung $\{Item\ on\ top\}$. Dabei definiert die Nachbedingung $\{Item\ on\ top\}$ den Zusammenhang der Operationen `push()` und `top()`.

Nun werden die Zusicherungen der Operation `pop()` betrachtet:

Not empty: Die Operation darf nicht aufgerufen werden, wenn sich keine Elemente auf dem Stapel befinden.

Not full: Die Operation garantiert, daß nach Beendigung der Operation, der Stapel nicht voll ist.

One fewer: Nachdem die Operation aufgerufen wurde, befindet sich ein Element weniger auf dem Stapel als vor dem Aufruf der Operation.

Return top: Das oberste Element des Stapels wird zurückgeliefert.

---

[12] Wird in einer Nachbedingung einem Ausdruck das Schlüsselwort `$old` vorangestellt, so wird der Wert referenziert, den der Ausdruck bei Eintritt in die Operation hatte. Der *sushi*-Präprozessor unterstützt diesen Mechanismus allerdings in der aktuellen Version noch nicht.

Auffällig ist, daß das Zusammenspiel der Operationen `push()` und `pop()` in keiner Zusicherung beschrieben ist. Die Nachbedingung {*Return top*} deutet zwar das Verhältnis der Operationen zueinander an, aber eine genaue Spezifikation der Beziehung wird nicht gegeben[13]. Eine Implementierung der `push()` Operation, die das übergebene Element auf den Stapel legt und zusätzlich die schon auf dem Stapel befindlichen Elemente durch beliebige andere Elemente ersetzt, würde alle Zusicherungen der Schnittstelle erfüllen (d.h. die Implementierung wäre korrekt bezüglich der Spezifikation, die durch die Verträge gegeben ist). Natürlich würde es sich bei dieser Implementierung nicht mehr um einen Stapelspeicher im klassischen Sinne halten, da die Implementierung das FIFO-Prinzip verletzen würde, aber nur mit Hilfe der boolschen Zusicherungen, wie sie *Design by Contract* bietet, läßt sich die vollständige Funktionsweise des FIFO-Prinzips nicht spezifizieren.

Ein naiver Ansatz, um das Problem zu lösen, ist die Benutzung einer "Schattenimplementierung", doch hier stellt sich natürlich wieder die Frage, wie die Korrektheit der Schattenimplementierung gewährleistet werden kann.

In [Szyperski97] wird als Ausweg ein Ansatz vorgestellt, der den ursprünglichen Mechanismus um eine Historie ergänzt, auf die innerhalb der Zusicherungen zugegriffen werden kann. Dies scheint eine gangbare Alternative zu sein. Ein Nachteil dieses Ansatzes ist jedoch, daß die Methodik zusätzlich verkompliziert wird.

3.12 Fazit

Design by Contract wurde als praxisorientierte Methode vorgestellt, mit der die Funktionalität von Programmen, bis zu einem gewissen Grad, beschrieben werden kann. Die Korrektheit bezüglich dieser Spezifikation wird durch die Prüfung der Zusicherungen zur Laufzeit sichergestellt. Zwar kann mit Hilfe von Design by Contract keine vollständige Spezifikation der Funktionalität erreicht werden, aber die Sachverhalte, die durch Zusicherungen ausgedrückt werden können, sind bereits sehr hilfreich, um die Semantik von Klassen und Operationen zu formalisieren und deren Korrektheit zur Laufzeit zu prüfen.

[13] Die Charakteristik eines Stacks liegt schließlich im *FIFO-Prinzip* (first-in first-out).

Kapitel 4

Unit-Tests

4.1 Einl■itung

Das vorangegangene Kapitel hat gezeigt, wie Zusicherungen und die Vertragsmethapher dabei helfen, die Funktionalität von Klassen und Operationen zu spezifizieren und die Korrektheit bezüglich dieser Spezifizierungen, zur Laufzeit, sicherzustellen. Das `Stack`-Beispiel des letzten Kapitels hat jedoch gezeigt, wo die Grenzen von Design by ontract liegen.

In diesem Kapitel werden *Unit-Tests* als ergänzende Methode vorgestellt, die ebenfalls helfen, die Korrektheit von Klassen und Operationen sicherzustellen.

4.2 ■Xtr■m■ Programming

Unit-Tests sind ein wesentlicher Bestandteil des *eXtreme Programming (XP)*. Dabei wird durch einen Unit-Test die technische Funktionstüchtigkeit eines Programmteils (die Granularität ist typischerweise auf Klassenebene angesiedelt) gemäß des Entwicklerverständnis ausgedrückt.

eXtreme Programming ist eine relativ neue Softwareentwicklungsmethode, die von *Kent Beck* [Beck99] entwickelt wurde und durch die folgenden 12 Praktiken charackterisiert wird[url XP]:

Das Planungsspiel: Der XP Planungsprozess erlaubt es dem Kunden, zu spezifizieren, welche Funktionalität die Software haben soll und auf welche Funktionalität vorerst verzichtet wird. Die Auswahl trifft der Kunde anhand von Kostenabschätzungen, die ihm für die geforderten Funktionalitäten von den Programmierern gegeben werden.

Kleine Versionen: Es werden sehr früh einfache Produktionsversionen veröffentlicht, die der Kunde ausprobieren kann. Die Versionen werden immer wieder, in sehr kurzen Zeitintervallen, mit neuen Funktionalitäten ergänzt.

Methapern: Es werden einheitliche Namens- und Beschreibungssysteme verwendet, die der Kommunikation und der Entwicklung im Team dienen.

Einfaches Design: Ein Programm, welches mit der XP-Methodik entwickelt wurde, sollte das einfachste Programm sein, das die Anforderungen des Kunden erfüllt. Dabei ist einfaches Design keinesfalls mit schlechtem Design gleichzusetzen. Die gute Qualität eines Designs wird mit Hilfe des *Refactoring* gesichert.

Tests: Die Anforderungen an das Programm werden durch (automatisierte) Tests spezifiziert und sichergestellt. Die Vorgehensweise besteht darin, daß zuerst ein Test geschrieben wird und danach eine Klasse, die die Anforderungen des Tests erfüllt.

Refactoring: Um ein sauberes Design des Programms zu gewährleisten, wird während der gesamten Entwicklung ständig daran gearbeitet, um z.B. Duplikation des Quellcodes zu vermeiden. Die Tests stellen sicher, daß durch das Refactoring kein Schaden angerichtet wird[1].

Paarprogrammierung: Der gesamte Quellcode wird immer in Teams zu jeweils zwei Leuten entwickelt, die zusammen an einem Rechner sitzen. Dadurch wird die Qualität des Quellcodes gesteigert, was sich im Endeffekt durch niedrigere Entwicklungskosten auszahlt.

Kollektiver Quellcodebesitz: Jeder Entwickler darf jeden Quelltext bearbeiten. Das erhöht die Geschwindigkeit, in der Änderungen vorgenommen werden können. Die Tests stellen wiederum sicher, daß kein Schaden angerichtet wird.

Kontinuierliche Integration: Alle Entwickler integrieren neue Funktionalitäten in sehr kurzen Zeitintervallen. Jeder Entwickler baut sich mehrmals täglich die neuste Version des Programms. Dadurch werden typische Integrationsprobleme am Ende der Entwicklung vermieden.

[1] *Martin Fowler* beschreibt in [Fowler et al.99] eine Reihe von Heuristiken, um *bad smelling code* in Quelltexten zu finden und beschreibt Lösungen für die verschiedenen Probleme.

40-Stunden-Woche: Überarbeitete Entwickler machen mehr Fehler. Daher wird in einem XP Team darauf geachtet, übermäßige Arbeitszeiten zu vermeiden.

Kunde vor Ort: Ein XP Projekt wird von jemandem gelenkt, der Entscheidungsbefugnis über die Anforderungen und die Eigenschaften des Programms hat. Er ist der Ansprechpartner, der die Fragen der Entwickler beantwortet. Dadurch ist die Kommunikation direkter und die Erstellung von unnötigen Dokumenten kann in vielen Fällen vermieden werden, was sich in niedrigeren Kosten bemerkbar macht.

Quellcoderichtlinien: Um effektiv paarweise und mit kollektivem Quellcodebesitzt programmieren zu können, gibt es verbindliche Richtlinien, an die sich jeder Entwickler halten muß. Auf diese Weise wird garantiert, daß sich jeder Entwickler schnell in fremden Quelltexten zurecht findet.

4.3 JUnit

JUnit[url JUnit] ist ein in Java implementiertes Testframework, das von *Kent Beck* und *Erich Gamma* entwickelt wurde. Eine ausführliche Beschreibung der Architektur des Frameworks findet sich in [Beck/Gamma99].

Die Kernidee von JUnit ist, daß man Tests schreibt, die wiederholbar sind und deren Ergebnisse maschinell geprüft werden können. Die getestete Einheit ist dabei die Klasse. Im eXtreme Programming geht man sogar soweit, für jede Klasse einen Test vorzuschreiben. Bevor also eine Klasse implementiert wird, wird ein Test geschrieben, der die Anforderungen an die Klasse wiederspiegelt. Durch ein gutes Abdeckungsverhältnis von Tests zu Klassen können bei Änderungen des Programms (z.B. durch Refactoring) sehr genau die Auswirkungen auf das Gesamtverhalten beobachtet werden.

4.3.1 Konzepte des JUnit-Frameworks

Damit ein Test vom JUnit-Framework ausgeführt werden kann, muß der konkrete Test von der Klasse `junit.framework.TestCase` abgeleitet werden. Diese Klasse bietet verschiedene `assert()` Operationen, um innerhalb eines Tests Zustände prüfen zu können.

Die Testfälle einer Klasse werden in Operationen implementiert. Dabei werden per Konvention alle Operationen vom JUnit-Framework aufgerufen, deren Name mit `test` beginnt und deren Signatur wie folgt ist: `public void`

`testXXX()`. Jede Testoperation, die vom Framework aufgerufen wird, wird wie folgt in eine *Template Method*[Gamma et al.94] eingebunden:

```
setUp();
testXXX();
tearDown();
```

Dabei bezeichnet `testXXX()` die eingebettete Operation. Die Operationen `setUp()` und `tearDown()` sind in der Basisklasse `TestCase` als leere Operationen implementiert und können in einer konkreten Testklasse überschrieben werden, um z.B. allgemeine Initialisierungen vor einem Test vorzunehmen (`setUp()`) bzw. nach einem Test Resourcen wieder freizugeben (`tearDown()`). Man kann sich die Operationen analog zu den *Konstruktoren* und *Destruktoren* in ++ vorstellen.

Damit nicht jede Testklasse einzeln getestet werden muß, können verschiedene Testklassen zu *Testsuiten* zusammengeführt werden. Für diesen Zweck bietet das JUnit-Framework die Klasse `TestSuite` an. Da die Klassen `TestCase` und `TestSuite` als *Composite* Muster[Gamma et al.94] implementiert wurden, ist es möglich, einer Testsuite sowohl konkrete Testklassen als auch andere Testsuiten zu übergeben. Das JUnit-Framework sorgt dafür, daß alle Tests innerhalb einer Testsuite ausgeführt werden.

Es sei noch angemerkt, daß die Reihenfolge, in der die Tests ausgeführt werden, bei verschachtelten Testsuiten schwer vorauszusehen ist. Dies sollte jedoch kein Problem darstellen, da ein Test unabhängig von anderen Tests sein sollte. Damit ein Test sich nicht auf einen anderen Test verlassen muß bzw. darauf angewiesen ist, daß er nach einem anderen Test ausgeführt wird (weil er etwa bestimmte Ergebnisse erwartet), stehen schließlich die Initialisierungsoperationen `setUp()` und `tearDown()` zur Verfügung.

4.3.2 Implementierung eines Tests

Im folgenden wird ein Test für das `Stack` Beispiel des letzten Kapitels implementiert. In diesem Test werden die Anforderungen an den Stack geprüft, die sich nicht mit Hilfe der Zusicherungen von Design by ontract ausdrücken ließen.

Zunächst wird jedoch die folgende Implementierung der `Stack`-Schnittstelle betrachtet[2]:

[2] Anmerkung: Die Verträge werden von der `Stack`-Schnittstelle aus Kapitel 3.11 geerbt.

```
public class StackImpl implements Stack
{
   private int      _capacity;
   private int      _count;
   private Object[] _array;


   public StackImpl ( int capacity )
   {
       _capacity = capacity;
       _count    = 0;
       _array    = new Object [ _capacity ];
   }

   public boolean isEmpty  () { return count() == 0;         }
   public boolean isFull   () { return count() == capacity();}
   public int     count    () { return _count;               }
   public int     capacity () { return _capacity;            }
   public Object  top      () { return _array [ count()-1 ]; }


   public Object  pop      ()
   {
       Object item = top();
       _count--;

       return item;
   }


   public void    push     ( Object item )
   {
       for ( int i=0; i <= count(); i++ )
           _array [ i ] = item;

       _count++;
   }

}
```

Die Implementierung erfüllt alle Zusicherungen der `Stack`-Schnittstelle. Trotzdem verhält sich die Implementierung nicht wie ein herkömmlicher Stapelspeicher. Das Problem liegt in der Implementierung der `push()` Operation, denn dort wird nicht nur das übergebene Element auf den Stapel gelegt, sondern es werden zusätzlich alle bisherigen Elemente des Stapels durch das übergebene Element ersetzt.

Da sich das gewünschte Verhalten nicht mit Hilfe der Zusicherungen ausdrücken läßt, wird der folgende Test implementiert:

```
public class StackTest extends junit.framework.TestCase
{
  public StackTest ( String name )
  {
    super ( name );
  }


  public void testPushPop()
  {
    Stack   stack;
    Object  item;
    Integer i1 = new Integer ( 1 );
    Integer i2 = new Integer ( 2 );

    stack = new StackImpl ( 2 );
    assert("CHECK 1: Is empty",     stack.isEmpty());
    stack.push ( i1 );
    assert("CHECK 2: Isn't empty",  !stack.isEmpty());
    assert("CHECK 3: i1 on top",    i1.equals(stack.top()));
    stack.push ( i2 );
    assert("CHECK 4: Is full",      stack.isFull());
    assert("CHECK 5: i2 on top",    i2.equals(stack.top()));
    item = stack.pop();
    assert("CHECK 6: i2 was poped", i2.equals(item));
    assert("CHECK 7: i1 on top",    i1.equals(stack.top()));
    item = stack.pop();
    assert("CHECK 8: i1 was poped", i1.equals (item));
    assert("CHECK 9: Is empty",      stack.isEmpty());
  }

}
```

Im Test werden folgende Schritt vollzogen:

1. Zuerst werden einige Objekte angelegt, die für den Test benötigt werden.

2. Ein Stapel mit der Kapazität 2 wird erzeugt.

   CHECK 1: Is empty Nach Erzeugung muß der Stapel leer sein.

3. Das Element `i1` wird auf den Stapel gelegt.

   CHECK 2: Isn't empty Der Stapel darf nun nicht mehr leer sein.

   CHECK 3: i1 on top Das auf den Stapel gelegte Element muß sich auf der Spitze des Stapels befinden.

4. Das Element `i2` wird auch auf den Stapel gelegt.

   CHECK 4: Is full Da der Stapel mit einer Kapazität von 2 angelegt wurde, muß er nun voll sein.

   CHECK 5: i2 on top Das gerade auf den Stapel gelegte Element `i2` muß sich auf der Spitze des Stapels befinden.

5. Ein Element wird vom Stapel genommen.

   CHECK 6: i2 was poped Das vom Stapel genommene Element, muß das Element `i2` sein, da es zuletzt auf den Stapel gelegt wurde.

   CHECK 7: i1 on top Auf der Spitze des Stapels muß sich nun das Element `i1` befinden.

6. Ein weiteres Element wird vom Stapel genommen.

   CHECK 8: i1 was poped Das vom Stapel genommene Element, muß das Element `i1` sein.

   CHECK 9: Is empty Der Stapel muß leer sein, da die beiden auf den Stapel gelegten Elemente, wieder vom Stapel genommen wurden.

Um den Test zu starten, muß ein *TestRunner* des JUnit-Frameworks aufgerufen werden, dem der Test übergeben werden kann[3]. Da die Operation `testPushPop()` der oben beschriebenen Konvention entspricht (vgl. Kapitel 4.3.1), wird sie automatisch vom TestRunner aufgerufen.

[3] JUnit bietet verschiedene TestRunner an (z.B. den `junit.ui.TestRunner` mit grafischer Benutzeroberfläche).

Bei der Durchführung des `StackTest` meldet der TestRunner ein Fehlschlagen des Tests:

```
There was 1 failure:
1) testPushPop(StackTest) "CHECK 7: i1 on top"
```

Anhand der Meldung kann man erkennen, daß in der Operation `testPushPop()` der Testklasse `StackTest` die Überprüfung, die mit dem String "`CHECK 7: i1 on top`" assoziiert wurde, fehlgeschlagen ist.

Dieses Fehlschlagen ist auf die falsche Implementierung der `push()` Operation zurückzuführen, denn anstelle des Elements `i1` liefert die falsche Implementierung das Element `i2` zurück, da die `push()` Operation immer alle Elemente des Stapels durch das übergebene Element austauscht.

Korrigiert man die `push()` Operation wie folgt und startet danach nochmal den Test, so läuft er fehlerfrei durch.

```
public void push ( Object item )
{
    _array [ count() ] = item;
    _count++;
}
```

4.4 Fazit

Unit-Tests sind ein wesentlicher Bestandteil des eXtreme Programming. Der große Vorteil von Unit-Tests liegt darin, daß große Mengen von Tests maschinell überprüft werden können. Eine hohe Abdeckung mit Testklassen sorgt dafür die Integrität eines Systems, in bezug auf Umbau von bestehendem Quelltext (Refactoring) oder Integration von neuem Quelltext sicherzustellen.

Das Kapitel hat gezeigt, daß Unit-Tests und Design by ontract Methoden sind, die sich nicht im Weg stehen, sondern sich ergänzen.

Die Methoden unterscheiden sich jedoch in ihrem Wesen völlig voneinander. Unit-Tests werden durch Testklassen repräsentiert, die zusätzlich zu bestehenden Klassen implementiert werden und das System *von außen* testen. Design by ontract arbeitet mit Zusicherungen, die direkt im Produktions-Quelltext definiert werden und dann während der Laufzeit des Programms *von innen* geprüft werden.

Kapitel 5

Vergleich vorhandener Tools

5.1 Einl■itung

Im folgenden werden verschiedene Tools beschrieben, die Design by ontract für Java unterstützen. Die Tools (*JWAMContract, iContract, JMSAssert* und *jContractor*) unterscheiden sich zum Teil sehr wesentlich in ihrer Philosophie, so daß es interessant ist, die Stärken und Schwächen der einzelnen Tools zu untersuchen, um später in einer eigenen Implementierung davon profitieren zu können.

5.2 JWAMContract

5.2.1 Beschreibung

Bei *JWAMContract* handelt es sich um ein sehr minimalistisches Framework, das innerhalb des *JWAM*-Projekts [1] verwendet wird. Im wesentlichen besteht JWAM ontract aus den Klassen `Contract, ContractBrokenException` und dem Interface `Invariant`.

Zusicherungen werden unter Verwendung von JWAM ontract explizit als Java- ode in den Quelltext eingefügt. Dazu bietet die Klasse `Contract` verschiedene statische Operationen. Um z.B. die Vorbedingung einer Operation zu prüfen, kann man sich der statischen Operation `require()` bedienen. In einem Aufruf wird der Operation die Vorbedingung in Form eines boolschen Ausdrucks übergeben. Ein zusätzlicher String dient als Label. Eine mögliche Überprüfung sähe dann folgendermaßen aus:

[1] *JWAM* ist ein Java-basiertes Framework für interaktive Geschäftsanwendungen [url JWAM]

```
Contract.require( x >= 0, "negative argument" );
```

Dabei ist `x` ein Parameter, der in der Signatur der Operation spezifiziert ist. Die obige Vorbedingung wäre z.B. sinnvoll für eine Operation, welche die Quadratwurzel zu einem gegebenen `x` berechnet.

Wird nun die Vorbedingung verletzt, so wird eine `ContractBrokenException` (abgeleitet vom Typ `RuntimeException`) geworfen. Durch den Stacktrace kann nun genau nachvollzogen werden, wo die Vertragsverletzung aufgetreten ist. Außerdem erscheint auch das angegebene Label der verletzten Zusicherung.

Will man zusätzlich zu Vor- und Nachbedingungen auch Invarianten verwenden, so muß die betreffende Klasse das Interface `Invariant` implementieren. Dieses Interface definiert lediglich eine Operation `invariant()`, die ein boolschen Wert zurückgibt.

Die Prüfung der Invarianten geschieht durch Verwendung spezieller `require()` bzw. `ensure()` Operationen, denen man zusätzlich zur Prüfbedingung und zum Label noch eine Referenz auf das Objekt, welches die Invariante implementiert, übergeben kann. In der Praxis sieht dies dann folgendermaßen aus:

```
public class Foo implements Invariant
{
    private String _member = "Hello world.";

    public boolean invariant() {
        return _member != null;
    }

    public void bar(int i) {
        Contract.require( i != 0, this, "zero not valid" );
        // ... do something
    }
}
```

Die Verletzung einer Invariante verursacht, genau wie die Verletzung einer Vor- oder Nachbedingung, eine `ContractBrokenException`, die dann zur Laufzeit geworfen wird.

5.2.2 Fazit

Die Stärke von JWAM ontract liegt ganz eindeutig in der einfachen Benutzung und in der Übersichtlichkeit des Frameworks. Es wird kaum Einarbeitungszeit benötigt, um das Framework zu erlernen. Gerade bei der Einführung einer Technologie wie Design by ontract in einen eventuell schon bestehenden Softwareentwicklungsprozeß hilft eine leichte Handhabbarkeit mit Sicherheit, die Akzeptanz für "das Neue" zu steigern.

Ein weiterer Pluspunkt für JWAM ontract ist die freie Verfügbarkeit des Frameworks im Quelltext. Dieser darf sowohl verändert als auch weitergegeben werden, solange man die opyright-Klausel des Frameworks ebenfalls unverändert weitergibt.

Von der Möglichkeit, den Quelltext zu verändern, sollte man beim Einsatz von JWAM ontract auf jeden Fall Gebrauch machen, denn mit sehr wenig Aufwand läßt sich die Qualität des Frameworks deutlich steigern. Zuerst sollte man für Vorbedingungen, Nachbedingungen und Invarianten eigene Exceptiontypen definieren. Da im Stacktrace lediglich eine `ContractBrokenException` erscheint, ist meistens nicht klar, welche Art von Vertragsverletzung aufgetreten ist. Spezielle Exceptiontypen wie etwa `PreconditionException, PostconditionException` und `InvariantException` könnten von `ContractBrokenException` abgeleitet werden und dann an den entsprechenden Stellen geworfen werden. Dann ginge unmittelbar aus dem Stacktrace die Art der Vertragsverletzung hervor.

Ein weiterer Kritikpunkt an der Implementierung ist die Überprüfung der Invarianten. Diese Überprüfungen werden -wie oben gezeigt- mittels spezieller `require()` bzw. `ensure()` Operationen durchgeführt. Die Operationen haben die folgende Signatur:

```
ensure(boolean postcond, Object contractor, String description)
```

Geschickter wäre sicherlich folgende Signatur:

```
ensure(boolean postcond, Invariant contractor, String description)
```

Das hätte zum einen den Vorteil, daß die Typüberprüfung zur ompilezeit und nicht zur Laufzeit stattfinden würde, zum anderen würde sich die Operation dadurch besser selber dokumentieren.

Mit diesen Verbesserungsvorschlägen stößt man aber auch schon an die Grenzen dessen, was JWAM ontract zu leisten vermag.

Da Design by ontract nur sinnvoll ist, wenn die Verträge auch nach außen hin sichtbar sind, ist man als Programmierer natürlich zu "doppelter

Buchführung" verpflichtet. Die Verträge müssen sowohl im Quelltext implementiert werden als auch in der Dokumentation der Operationen bzw. Klassen. Dieser zweistufige Mechanismus ist sehr anfällig für Fehler, so daß es leicht passieren kann, daß Verträge vergessen werden bzw. falsch dokumentiert werden.

Weitere Probleme, die der Programmierer "von Hand" lösen muß, sind die Vermeidung von endlosen Rekursionen (vgl. Kapitel 3.10.1), etwa durch Zusicherungen, die sich gegenseitig aufrufen, und die Nachbildung des *old*-Mechanismus für Nachbedingungen.

Das größte Problem wird jedoch sein, darauf zu achten, daß Verträge von Basisklassen richtig geerbt bzw. eingehalten werden, wenn Operationen in abgeleiteten Klassen überschrieben werden.

5.3 iContract

5.3.1 Beschreibung

iContract ist ein Präprozessor, der den Einsatz von Invarianten, Vor- und Nachbedingungen unterstützt. Die verschiedenen Zusicherungen werden innerhalb eines Javadoc-Kommentars einer Operation bzw. einer Klasse durch spezielle Tags (`@invariant, @pre, @post`) gekennzeichnet, denen die zu überprüfende Bedingung als boolscher Ausdruck angehängt wird. Zusätzlich können innerhalb der Zusicherungen auch einige O L[2]-ähnliche Ausdrücke benuzt werden.

Folgende Ausdrücke stehen dafür zur Verfügung:

- **forall** bietet die Möglichkeit, eine Bedingung für eine Kollektion von Elementen zu überprüfen
- **exist** kann verwendet werden, um die Existenz eines Elements in einer Kollektion von Elementen zu überprüfen
- **implies:** mit diesem Ausdruck können bedingte Überprüfungen formuliert werden

Auch an die Implementierung des `old`-Mechanismus für Nachbedingungen wurde gedacht. Mittels des Schlüsselwortes `$pre` kann der Wert des Attributes referenziert werden, den das Attribut bei Eintritt in die Operation hatte.

[2]die Object Constraint Language (OCL) ist eine Spezifikation der Unified Modeling Language (UML), die es erlaubt, Verträge für UML-Modelle zu beschreiben, die nicht visualisiert werden können.

Im folgenden wird beispielhaft eine Nachbedingung für eine Operation, unter Verwendung des `old`-Mechanismus, definiert.

```
/**
 * @post list.size() == list.size()@pre + 1
 */
void append(Vector list, Object obj);
```

Des weiteren ist es möglich, Zusicherungen auch für Interfaces zu definieren. Diese werden dann entsprechend in den Klassen eingefügt, die diese Interfaces implementieren. Auch Zusicherungen, die in Basisklassen definiert wurden, werden in den abgeleiteten Klassen eingefügt.

Um die Zusicherungen zu aktivieren, ist es nötig, eine Konfigurationsdatei zu erstellen, in der die zu instrumentierenden Quelldateien angegeben werden. Dabei ist es möglich, für jede einzelne Quelldatei anzugeben, welche Vor- und Nachbedingungen geprüft werden sollen und ob die Prüfung der Invarianten erfolgen soll. i ontract sucht dann anhand der Konfiguarationsdatei die entsprechenden Quelldateien zusammen, instrumentiert diese und kompiliert dann die instrumentierten Dateien. Die kompilierten Dateien werden dann in einem Verzeichnisbaum abgelegt, den man ebenfalls in der Konfigurationsdatei spezifizieren kann.

Tritt zur Laufzeit eine Vertragsverletzung auf, so wird eine Exception geworfen. Der Stacktrace der Exception beschreibt präzise, welche Art von Vertragsverletzung aufgetreten ist.

5.3.2 Fazit

Die Art und Weise, wie Zusicherungen in i ontract formuliert werden, ist sehr intuitiv. Durch die Spezifikation der Verträge innerhalb der Javadoc-Kommentare bleibt der eigentliche Quelltext sehr übersichtlich. Ein weiterer Vorteil dieser Vorgehensweise besteht darin, daß man mit Hilfe des Javadoc-Tools automatisch eine Dokumentation des Quelltextes erzeugen kann (ähnlich der *short-form* in Eiffel), in der auch die Verträge beschrieben sind[3].

Leider befindet sich i ontract noch in einem sehr frühen Betastadium (derzeit Version 0.3d2), was sich an vielen Stellen bemerkbar macht. Insbesondere die Dokumentation des Tools ist an einigen Stellen fehlerhaft bzw. inkonsistent in bezug auf das ausgelieferte jar-Paket. So war es beispielsweise

[3] Javadoc kann über s.g. Doclets parametrisiert werden

nötig, das jar-Paket auszupacken und zu analysieren, um herauszufinden wie das Tool gestartet wird.

Die Erstellung der Konfigurationsdatei ist ebenfalls noch weit davon entfernt, "einfach benutzbar" zu sein. Dieser Teil des Tools scheint auch noch fehlerhaft zu sein, da es nicht möglich war, daß sowohl Vor- als auch Nachbedingungen in den Quelltext instrumentiert werden. Es war immer nur möglich, entweder Vorbedingungen oder Nachbedingungen zu aktivieren.

Der wohl gravierendste Fehler, daß i ontract nur Operationen instrumentieren kann, die lediglich ein einziges `return`-Statement enthalten, disqualifiziert das Tool jedoch gänzlich für einen ernsthaften Einsatz in größeren Projekten. Es ist auch fraglich, wie schnell die Schwächen dieses Tools behoben werden, da i ontract lediglich von einem einzigen Autor gepflegt wird und der Quelltext nicht frei verfügbar ist.

Abschließend ist noch zu bemerken, daß aufgrund der spröden Handhabung des Tools leider nicht alle oben beschriebenen Features getestet werden konnten. Sowohl die Vererbung von Zusicherungen als auch die Benutzung der O L-Ausdrücke wurden hier nur unter Berufung auf [Kramer98] beschrieben.

5.4 JMSAss■rt

5.4.1 Beschreibung

Bei *JMSAssert* handelt es sich ebenfalls um einen Präprozessor, der i ontract sowohl in der Benutzung als auch im Funktionsumfang sehr ähnlich ist. Auch hier werden Zusicherungen mittels bestimmter Tags (`@inv, @pre, @post`) innerhalb der entsprechenden Javadoc-Kommentare spezifiziert. Diesen Tags folgt dann ein boolscher Ausdruck, der die Zusicherung beschreibt.

JMSAssert ist jedoch an die Microsoft Windows Plattform gebunden. Anders als bei i ontract wird kein instrumentierter Java- ode generiert, der dann kompiliert wird, sondern die mit Zusicherungen bestückten Klassen werden in eine eigene Skriptsprache (*JMScript*) transformiert. Die Syntax dieser Skriptsprache ist stark an die Java-Syntax angelehnt. Mit Hilfe der Skriptsprache können auch Makros definiert werden, die dann innerhalb der Zusicherungen verwendet werden können.

Um ein Programm zu starten, dessen Quelltexte zuvor mit JMSAssert bearbeitet wurden, muß der Java-Interpreter mit einer speziellen dynamischen Bibliothek[4] gestartet werden. In dieser Bibliothek ist der Interpreter für *JMScript* enthalten, der dann die Ausführung des Programms mit übernimmt.

[4]das ist auch der Grund, weshalb JMSAssert plattformspeziefisch ist, da diese Bibliothek (*jmsdll.dll*) derzeit nur für Windows verfügbar ist

5.4.2 Fazit

Da JMSAssert in der Benutzung fast identisch mit i ontract ist, bietet es natürlich die gleichen Vorteile.

Der wesentliche Nachteil von JMSAssert ist wohl die Bindung an die Microsoft Windows Plattform, da hier die Vorteile der Plattformunabhängigkeit von Java nicht mehr genutzt werden können.

Zur Evaluierung des Tools wurde JMSAssert auf einem P mit Windows NT 4.0 installiert. Um die Installation durchzuführen, waren Administratorrechte notwendig.

Leider funktionierte die Ausführung des mitgelieferten Beispielprogramms nicht wie erwartet. Der Quelltext ließ sich zwar mit Hilfe des Präprozessors instrumentieren, so daß die entsprechenden Skripte erzeugt wurden, aber bei der Ausführung des Programms wurden nicht, wie erwartet, die provozierten Vertragsverletzungen angezeigt.

5.5 jContractor

5.5.1 Beschreibung

jContractor wird in [Karaorman et al.98] als "rein bibliotheks-" und "designpattern"-basierter Ansatz beschrieben. Zusicherungen werden hier in eigenen Operationen implementiert, die einer bestimmten Namenskonvention folgen müssen. Im folgenden wird beispielhaft die Vorbedingung für eine Operation zur Berechnung der Quadratwurzel angegeben:

```
class Foo
{
    double sqrt(double x) {
        return Math.sqrt( x );
    }

    protected boolean sqrt_PreCondition(double x) {
        return x >= 0;
    }
}
```

Ein spezieller `ClassLoader` sorgt dann zur Laufzeit dafür, daß beim Laden der Klassen die entsprechenden Operationen mit den relevanten Zusicherungen instrumentiert werden. Möchte man keinen speziellen `ClassLoader`

verwenden, so können Objekte auch mit Hilfe einer speziellen Factory instanziiert werden. Diese Factory instrumentiert dann, ähnlich wie der `ClassLoader` von j ontractor, die entsprechenden Objekte. In der Praxis sieht die Verwendung der Factory dann folgendermaßen aus:

```
Foo f = (Foo) jContractor.New("Foo");
```

Die Verwendung des `old`-Mechanismus wird über eine Klasse mit Namen `OLD` realisiert, die eine statische Operation `value()` besitzt. j ontractor generiert zu diesem Zweck ode, um alle Attribute, die in der Nachbedingung verwendet werden, zwischenzuspeichern.

j ontractor bietet auch die Möglichkeit, Exception-Handler für Operationen zu definieren, um z.B. bei auftretenden Exceptions die Invarianten wieder herstellen zu können. Hiermit wird der `rescue`-Mechanismus von Eiffel nachgebildet. Die Definition eines Exception-Handlers muß ebenfalls einer bestimmten Namenskonvention folgen, damit der `ClassLoader` bzw. die Factory von j ontractor die Operationen entsprechend instrumentieren kann. Es ist ferner möglich für eine Operation mehrere Exception-Handler zu spezifizieren, die sich dann um unterschiedliche Exceptions kümmern können.

5.5.2 Fazit

Leider gibt es derzeit noch keine Implementierung von j ontractor[5], so daß über die Verwendung des Tools im praktischen Einsatz nur spekuliert werden kann.

Die Benutzung von j ontractor wird wahrscheinlich relativ einfach sein, da Zusicherungen mit Hilfe einer Namenskonvention definiert werden und dann zur Laufzeit von einem speziellen `ClassLoader` eingebaut werden. Durch den Einsatz dieses `ClassLoaders` entfällt natürlich der zusätzliche Aufwand, die Quelltexte mit einem Tool nachzubearbeiten. Dies ist, nach den Erfahrungen mit i ontract und JMSAssert, mit Sicherheit positiv zu bewerten.

Leider geht durch die Verwendung von Zusicherungen mittles einer Namenskonvention die Struktur einer Klasse etwas verloren. Hier spiegelt sich nämlich die Verwendung der Technologie Design by ontract im Design der Klasse wieder. Wenn für jede Operation sowohl eine Vorbedingung als auch eine Nachbedingung angegeben wird, hat die Klasse dreimal soviele Operationen wie ohne die Verwendung von Zusicherungen. Werden zusätzlich noch Exception-Handler definiert, so verschlimmert sich die Situation weiter. Bei einem solchen "Aufblähen" des Quelltextes sind Akzeptanzprobleme

[5]Die Beschreibung des Tools basiert daher lediglich auf [Karaorman et al.98]

gegenüber der Verwendung einer solchen Technologie natürlich vorprogrammiert.

Ein weiterer Nachteil ist die "doppelte Buchführung", zu der man auch hier verpflichtet ist. Zwar sieht man in der Klassendokumentation (beispielsweise Javadoc), daß es Zusicherungen für bestimmte Operationen gibt, doch die Semantik ergibt sich natürlich nicht daraus. Daher ist eine zusätzliche Beschreibung der Zusicherungen notwendig, was die schon beschriebenen Nachteile mit sich bringt.

Kapitel 6

Der *sushi* -Präprozessor

6.1 Einleitung

In diesem Kapitel wird die Implementierung des Präprozessors *sushi* beschrieben, der im Rahmen der vorliegenden Arbeit entwickelt wurde. Die Implementierungsentscheidungen und Konzepte stützen sich dabei auf die Erfahrungen, die mit anderen Design by Contract Werkzeugen gesammelt wurden (vgl. Kapitel 5).

6.2 Ziele der Entwicklung

Die Evaluierung der verschiedenen Design by Contract Werkzeuge hat gezeigt, daß die ausgefeiltesten Konzepte wertlos sind, wenn sich das Werkzeug nicht auf einfache Weise benutzen läßt. An dieser Stelle muß klar gesagt werden, daß JWAM Contract (vgl. Kapitel 5.2) das einzige Werkzeug war, welches sich für den professionellen Einsatz eignet. Leider handelt es sich bei JWAM Contract aber auch um das Werkzeug mit der geringsten Funktionalität.

1. Ziel: Die einfache Benutzbarkeit des Werkzeugs.

Ein weiterer Pluspunkt für JWAM Contract ist die Verfügbarkeit des Werkzeugs im Quelltext. Dadurch ist es möglich, durch Eigeninitiative erkannte Fehler und Schwächen des Werkzeugs selber zu beheben. Ist der Quelltext nicht frei verfügbar, so ist das insbesondere dann ärgerlich, wenn das Werkzeug nur von sehr wenigen Leuten gepflegt wird, da die Behebung von Fehlern dann typischerweise länger dauert (als Beispiel sei hier iContract genannt, das nur von einem einzigen Autor gepflegt wird).

2. Ziel: Die freie Verfügbarkeit des Quelltextes.

Da es sich bei Java um eine plattformunabhängige Programmiersprache handelt, sollte auch das Werkzeug plattformunabhängig sein, um den Entwickler bei der Benutzung des Werkzeugs nicht an eine bestimmte Plattform zu binden (die Abhängigkeit von der Microsoft Windows Plattform war ein wesentlicher Nachteil von JMSAssert).

3. Ziel: Die Plattformunabhängigkeit des Werkzeugs.

Das Werkzeug muß sich für den professionellen Einsatz eigenen. Es soll kein akademisches Projekt im Sinne einer Machbarkeitsstudie sein, sondern ein Werkzeug, das der Entwickler im Alltag einsetzen kann.

4. Ziel: Die Praxistauglichkeit des Werkzeugs.

Die gesamte Implementierung des Werkzeugs orientiert sich an diesen Zielen, so daß alle wichtigen Entscheidungen gegen diese Ziele abgewogen werden.

6.3 Allgemeine Anforderungen

Im folgenden werden die allgemeinen Anforderungen und die grundlegenden technologischen Entscheidungen an das Werkzeug erläutert.

6.3.1 Präprozessor

Das Werkzeug wurde in Form eines Präprozessors implementiert, der die Verträge aus den Javadoc-Kommentaren liest und die Zusicherungen dann an den entsprechenden Stellen in den Quelltext instrumentiert. Die Unterbringung der Verträge in den Javadoc-Kommentaren wurde während der Evaluierung der anderen Werkzeuge als am natürlichsten empfunden.

Vorteile

- Die Verträge müssen nicht extra dokumentiert werden, da sie Teil der Dokumentierungskommentare sind.
- Die Synchronität von Verträgen und deren Dokumentation wird automatisch gewährleistet.

- Die Quelltexte lassen sich auch ohne die Verwendung des Werkzeugs kompilieren, da Kommentare für die Kompilierung keine Bedeutung haben.
- Auch innerhalb von Schnittstellen können Verträge definiert werden.

Nachteile

- Der Quelltext muß vor der Kompilierung mit dem Präprozessor bearbeitet werden.

Mögliche Alternativen wären Implementierungen im Stil von JWM ontract bzw. j ontractor gewesen, doch beide Alternativen haben den Nachteil, daß die Synchronität zwischen Verträgen und deren Dokumentation nicht sichergestellt ist. Der Ansatz von j ontractor hat den weiteren Nachteil, daß sich die Verwendung der Methodik massiv im Design der Klassen wiederspiegelt. Eine Implementierung in der Art von JWAM ontract hätte den zusätzlichen Nachteil gehabt, daß die Vererbung von Verträgen nicht durch das Werkzeug hätte sichergestellt werden können, sondern vom Entwickler übernommen werden müßte (insbesondere hätte sich ein Problem bei der Schnittstellenvererbung ergeben, da Schnittstellen keine Implementierungen besitzen und somit nicht mit Zusicherungen bestückt werden können).

Eine proprietäre Erweiterung der Programmiersprache Java wurde von vornherein ausgeschlossen, da der Aufwand den Rahmen der vorliegenden Arbeit gesprengt hätte. Es ist aber auch fraglich, ob eine proprietäre Erweiterung nicht zu enormen Akzeptanzproblemen geführt hätte.

6.3.2 Implementierungssprache Java

Das Werkzeug wurde in der Programmiersprache Java entwickelt. Dadurch war es möglich, eine Version des Werkzeugs zu bauen, die auf allen Plattformen mit Java-Unterstützung lauffähig ist. Verschiedene Binärversionen, die für die unterschiedlichen Plattformen hätten kompiliert werden müssen, entfielen somit.

Alternativ zu Java hätte man Skriptsprachen, wie z.B. Python[url Python] oder Ruby[url Ruby] verwenden können, da diese Sprachen auch für alle wichtigen Plattformen zur Verfügung stehen und ebenfalls interpretiert werden. Da ein Ziel jedoch die freie Verfügbarkeit des Quelltextes ist, bot sich Java eher als Implementierungssprache an, da die Benutzer des Werkzeugs Java-Entwickler sind. Sie sind bestens mit der Programmiersprache Java vertraut, so daß es ihnen relativ einfach fallen sollte, etwaige Modifikationen am Werkzeug vorzunehmen.

6.3.3 Syntax der Zusicherungen

Die Zusicherungen werden in Javadoc-Kommentaren definiert. Dabei werden Invarianten im Javadoc-Kommentar, der die Klasse beschreibt, untergebracht; Vor- und Nachbedingungen werden in die zu den Operationen gehörenden Javadoc-Kommentare geschrieben.

Die Syntax der Zusicherungen lehnt sich an die aus Eiffel bekannte Syntax an. Für die Invarianten wird das Tag[1] `@invariant`, für Vorbedingungen das Tag `@require` und für Nachbedingungen das Tag `@ensure` benutzt. Dem Tag kann optional eine Bezeichnung folgen, die ausgegeben wird, wenn die Zusicherung verletzt wird. Der optionalen Bezeichnung folgt die Zusicherung in Form eines Java Ausdrucks, der einen boolschen Wert zurückliefert. Tag, Bezeichnung und Zusicherung werden jeweils durch einen Doppelpunkt voneinander getrennt.

6.3.4 Der Parsergenerator javacc/jjtree

Ein wesentlicher Bestandteil des entwickelten Präprozessors ist ein Parser, der Java Quelltexte analysiert und die benötigten Informationen aus den Quelltexten extrahiert (z.B. die Zusicherungen aus den Kommentaren).

Da die Implementierung eines Parsers von Hand sehr aufwendig und fehleranfällig ist, wurde auf einen Parsergenerator zurückgegriffen. Die Wahl fiel dabei auf das Werkzeug *javacc* in Verbindung mit *jjtree*[url javacc/jjtree]. Bei javacc handelt es sich um den eigentlichen Parsergenerator. Das Werkzeug jjtree ist eine Erweiterung, basierend auf javacc, die es ermöglicht, abstrakte Syntaxbäume zu erzeugen. Die Entscheidung für javacc/jjtree wurde aufgrund der guten Dokumentation der Werkzeuge getroffen. Weitere Vorteile sind der hohe Verbreitungsgrad der Werkzeuge und deren Fähigkeit, einen Parser in der Programmiersprache Java zu generieren.

6.3.5 Dokumentation

Da ein Werkzeug nur dann einen Wert hat, wenn der Benutzer weiß, wie das Werkzeug zu benutzen ist, wurde eine Internetseite[2][url sushi] erstellt, die eine englischsprachige Dokumentation enthält. Die Dokumentation erläutert anhand eines einfachen Beispiels die Benutzung des Präprozessors und weist

[1] Ein Tag besteht aus einem Klammeraffen ("`@`") gefolgt von einer Zeichenkette. Das Javadoc Werkzeug zur Erzeugung der Dokumentation kann über s.g. Doclets parametrisiert werden, so daß die Verträge dann auch in der generierten Dokumentation sichtbar sind.

[2] Die Dokumentation findet sich auch auf der beiliegenden CD-ROM

zusätzlich auf die besonderen Eigenschaften und Limitierungen des Werkzeugs hin.

6.4 Implementierung

Nachdem die allgemeinen Anforderungen und technologischen Entscheidungen erläutert wurden, soll im folgenden die Implementierung des Werkzeugs genauer betrachtet werden.

6.4.1 Überblick über die Funktionsweise

Zunächst wird der einfachste Fall der Benutzung des Präprozessors betrachtet, um die Funktionsweise zu erläutern: *Der Benutzer ruft den Präprozessor auf und übergibt den Dateinamen des zu instrumentierenden Quelltextes.*

Schritt 1: Parsen Der Quelltext wird durch einen Parser analysiert. Dabei werden alle Informationen gesammelt, die später für das Instrumentieren von Bedeutung sind.

Schritt 2: Instrumentieren Die gesammelten Informationen werden vom Instrumentierer benutzt, um an entsprechenden Stellen im Quelltext Quellcode einzubauen, der die Zusicherungen prüft.

Als Resultat liefert der Präprozessor eine instrumentierte Version des Quelltextes, der an den benötigten Stellen Quellcode zur Überprüfung der Zusicherungen enthält.

6.4.2 Parsen

Mit Hilfe der Werkzeuge javacc und jjtree wurde ein Parser generiert, der einen abstrakten Syntaxbaum für Java-Quelltexte erzeugt. Dabei wurde die den Werkzeugen beiliegende Grammatik so erweitert, daß auch die Zusicherungen innerhalb der Javadoc-Kommentare beachtet werden.

Das jjtree Werkzeug generiert auch die Schnittstelle für einen *Visitor* [Gamma et al.94], der für jeden Knoten des abstrakten Syntaxbaums eine entsprechende `visit()` Operation besitzt. Mit Hilfe des Visitors kann der Baum dann traversiert werden.

Jeder `visit()` Methode kann neben einem konkreten Knoten auch ein Exemplar vom Typ `Object` übergeben werden. Damit ist es auf einfache Art und Weise möglich, die in einem Knoten gesammelten Informationen,

in diesem Exemplar zu speichern und sie an den nächsten Knoten weiterzureichen[3]. In [Beck96] ist diese Vorgehensweise als Entwurfsmuster *Collecting Parameter* beschrieben.

Kernabstraktion

Da der Präprozessor auf einem Java-Quelltext arbeitet und ein Java-Quelltext immer genau eine Klasse (bzw. Schnittstelle) repräsentiert[4], ist die Klasse (bzw. die Informationen über die Klasse) die Kernabstraktion der Architektur. Die konkrete Ausprägung dieser Kernabstraktion ist die Klasse `ClassInfo`. Ein Exemplar vom Typ `ClassInfo` wird daher dem Visitor übergeben, der die Informationen einsammelt, um sie dann in das `ClassInfo` Exemplar einzutragen. Dieser Visitor wird durch die Klasse `CollectInfoVisitor` implementiert.

`ClassInfo` - Informationen über eine Klasse

Im folgenden werden die Informationen beschrieben, die zum späteren Instrumentieren benötigt werden. Diese Informationen entsprechen daher den Attributen der Klasse `ClassInfo`.

package: Das *package* der Klasse wird benötigt, um die Basisklasse und die Schnittstellen zu finden, die evtl. aus dem gleichen package geerbt werden.

import: Die *import*-Ausdrücke werden zum Auffinden von Schnittstellen (resp. der Basisklasse) aus anderen packages benötigt.

isAbstract: Diese Information ist zum Generieren eines Konstruktors wichtig (vgl. Kapitel 6.4.3).

isClass: Gibt an, ob der Quelltext eine Klasse oder eine Schnittstelle repräsentiert. Dies ist wichtig, da Schnittstellen nicht instrumentiert werden.

name: Der Name der Klasse ist für mehrere Aspekte des Instrumentierens von Interesse, z.B. bei der Generierung eines Konstruktors (vgl. Kapitel 6.4.3).

---

[3]Die Informationen werden in einem Exemplar der Klasse `ClassInfo` gespeichert. Jede `visit()`-Operation führt daher einen *Cast* von `Object` auf `ClassInfo` durch.

[4]Aus Zeitgründen wurde bei der Implementierung des Präprozessors darauf verzichtet auch innere- und anonyme Klassen bearbeiten zu können. Verträge für innere- und anonyme K

superclass: Der Name der Basisklasse, von der möglicherweise Verträge geerbt werden.

interfaces: Eine Liste der Schnittstellen, von denen möglicherweise Verträge geerbt werden.

invariants: Eine Liste der Invarianten, die für diese Klasse definiert wurden (exklusive der Invarianten von Schnittstellen bzw. Basisklassen).

constructors: Eine Liste der Konstruktoren der Klasse.

methods: Eine Liste von `MethodInfo` Objekten, welche die Operationen repräsentieren.

`MethodInfo` - Informationen über eine Operation

Die Informationen über eine Operation werden in Exemplaren der Klasse `MethodInfo` konserviert. Folgende Informationen sind dabei wichtig:

signature: Die Signatur der Operation wird benötigt, um feststellen zu können, ob die Operation eine Operation aus einer Basisklasse überschreibt bzw. die Operation einer Schnittstelle implementiert.

preconditions: Eine Liste der Vorbedingungen, die für die Operation definiert wurden.

postconditions: Eine Liste der Nachbedingungen, die für die Operation definiert wurden.

returns: Wenn eine Methode in der Signatur einen Rückgabetyp spezifiziert, besitzt sie mindestens eine *return*-Anweisung. Diese Anweisung(en) müssen gespeichert werden, da sich die Nachbedingung möglicherweise darauf bezieht (mittels `$return`).

signature wurde hier zur Vereinfachung als Attribut aufgeführt. Tatsächlich wurde jedoch keine eigene Klasse definiert, da sich die Signatur sehr gut durch wenige Attribute in der Klasse `MethodInfo` modelieren ließ (**signature** steht hier stellvertretend für diese Attribute).

Die *return*-Anweisungen einer Operation werden in Exemplaren vom Typ `ReturnInfo` gespeichert.

`CollectInfoVisitor` - Das Einsammeln der Informationen

Das Einsammeln der Informationen geschieht durch einen Visitor vom Typ `CollectInfoVisitor`, der den abstrakten Syntaxbaum traversiert. Allerdings ist es nicht nötig, alle Knoten des Baums zu besuchen, da nur die beschriebenen Informationen von Interesse sind und diese relativ nah an der Wurzel des Baums liegen. Daher wurde eine Klasse `ConfigurableVisitor` implementiert, welche die generierte Visitor Schnittstelle implementiert. Exemplare der Klasse können derart konfiguriert werden, daß nur bestimmte Knoten des Baums traversiert werden.

Die Klasse `CollectInfoVisitor` wurde von `ConfigurableVisitor` abgeleitet und konfiguriert die Basisklasse so, daß nur die benötigten Knoten traversiert werden. Für diese Knoten implementiert sie die entsprechenden `visit()`-Operationen, die dann die Informationen aus den Knoten extrahieren.

Die Beziehung zwischen der generierten Visitor Schnittstelle (`JavaDbCParserVisitor`) und der Klasse `ConfigurableVisitor` ist eine Variation des Entwurfmusters *Generation Gap*[5][Vlissides98].

Um die Informationen aus den Knoten des abstrakten Syntaxbaums zu extrahieren, greift die Klasse `CollectInfoVisitor` auf die Funktionalität der Klasse `TreeFunctions` zurück. In diese Klasse wurden Operationen verlagert, welche die Details des abstrakten Syntaxbaums kennen.

Die Abbildung 6.1 zeigt einen Überblick über die Architektur des Präprozessors. Dabei befinden sich im *Package* `datastructures` die Klassen, in deren Exemplaren die Informationen gesammelt werden. In der rechten Hälfte des Diagramms sind die Pakete dargestellt, deren Klassen zum Einsammeln der Informationen benötigt werden.

6.4.3 Instrumentieren

Das Instrumentierersubsystem ist der andere wesentliche Teil des Präprozessors. Der Instrumentierer benutzt die Informationen, die während des Parsens gewonnen wurden, um im Quelltext an entsprechenden Stellen ode einzufügen, der die Zusicherungen aus den Verträgen prüft.

Stellen im Quelltext, an denen Code eingefügt wird

Der ode, der zum Überprüfen der Zusicherungen generiert wird, muß an verschiedenen Stellen im Quelltext eingefügt werden.

[5] *"Modify or extend generated code just once no matter how many times it is regenerated."*

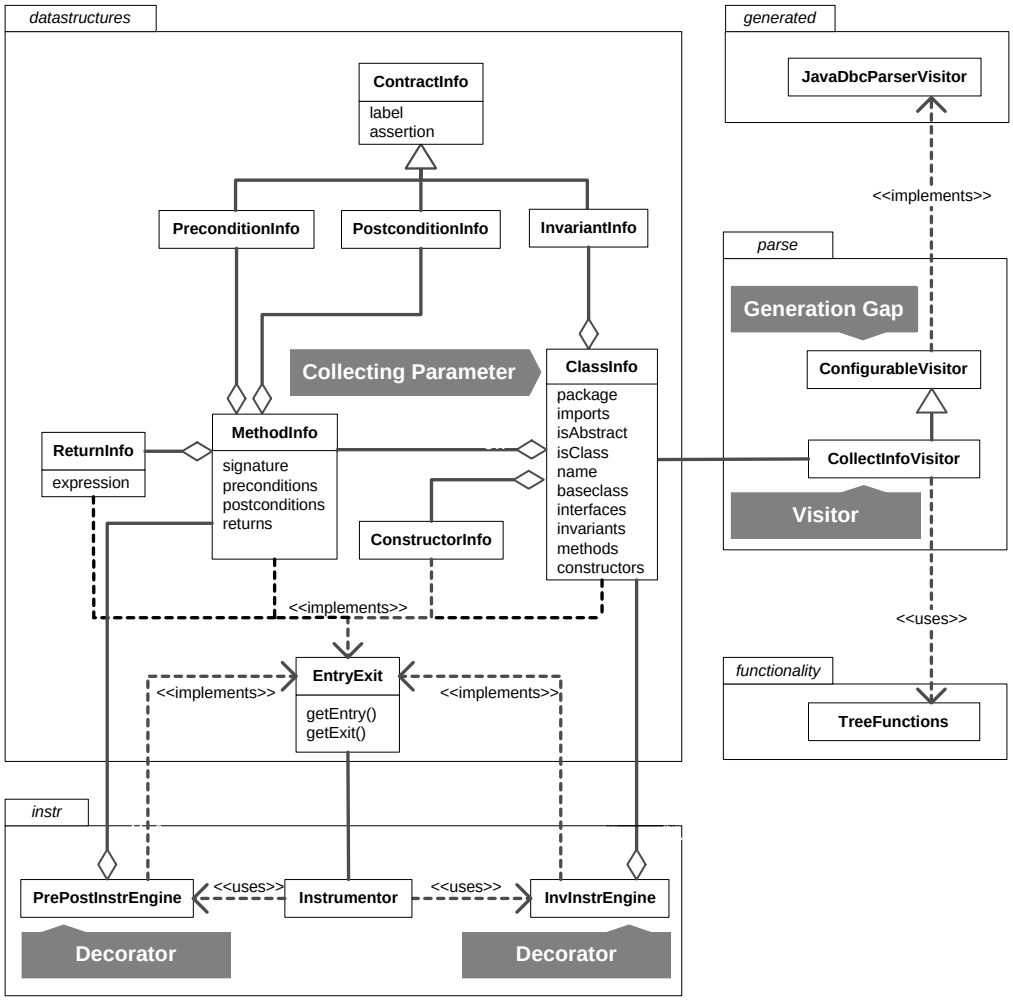

Abbildung 6.1: Überblick über die Architektur

Vorbedingungen: Vorbedingungen beziehen sich immer auf genau eine Operation. Die Vorbedingungen einer Operation müssen geprüft werden, bevor die Operation mit ihrer eigentlichen Arbeit beginnt. Konkret bedeutet das, daß der Prüfcode direkt hinter die erste geschweifte Klammer ("{") einer Operation instrumentiert werden muß.

Nachbedingungen: Nachbedingungen beziehen sich ebenfalls immer auf genau eine Operation. Sie müssen geprüft werden, sobald die Operation ihre Arbeit beendet hat. Bei Operationen, die nichts zurückliefern, können sämtliche Nachbedingungen direkt vor die letzte geschweifte Klammer ("}") der Operation instrumentiert werden. Liefert die Operation jedoch ein Ergebnis zurück, so müssen die Nachbedingungen vor

jeder *return*-Anweisung geprüft werden.

Invarianten: Invarianten müssen vor und nach Beendigung jeder Operation gelten. Daher wird ihr Prüfcode sowohl an den Stellen, an denen die Vorbedingungen instrumentiert werden, als auch an den Stellen, an denen die Nachbedingungen instrumentiert werden, eingefügt. Eine Ausnahme bilden jedoch Operationen, die als `static` oder `private` deklariert sind. Da Invarianten sich auf den gültigen Zustand eines Exemplars beziehen, ist es nicht möglich, diesen in statischen Operationen zu überprüfen. Private Operation dürfen die Invarianten verletzen, da sie nur innerhalb der Klasse aufgerufen werden können und die aufrufende Operation die Verantwortung für die Einhaltung der Invariante übernehmen kann.

Da eine Invariante direkt nach der Erzeugung eines Exemplars gelten muß, wird vor der letzten geschweiften Klammer jedes Konstruktors ("}") Prüfcode eingefügt.

Der folgende Pseudoquelltext veranschaulicht, an welchen Stellen der generierte Prüfcode in den Quelltext instrumentiert wird:

```
public class Foo
{

    public Foo()
    {
        init();

        // Pruefung der Invarianten
    }

    public void bar ()
    {
        // Pruefung der Invarianten
        // Pruefung der Vorbedingungen

        doSomething();

        // Pruefung der Nachbedingungen
        // Pruefung der Invarianten
    }
```

```
    private void doSomething()
    {
        // Keine Invariantenpruefung, da private Operation
        // Pruefung der Vorbedingungen

        doSomethingElse();

        // Pruefung der Nachbedingungen
    }
}
```

Welcher Code muß generiert werden?

Für jeden Typ von Zusicherung wurde eine entsprechende Exception-Klasse implementiert (`InvariantError`, `PreconditionError` und `PostconditionError`), um im Falle der Verletzung einer Zusicherung feststellen zu können, welche Art der Vertragsverletzung aufgetreten ist. Die Gemeinsamkeiten der drei Klassen sind in der Basisklass `AssertionError` gekapselt, die wiederum von der Klasse `Error` abgeleitet ist (vgl. Abbildung 6.2). Alternativ hätte man für die Klasse `AssertionError` auch `RuntimeException` als Basisklasse wählen können. Funktionell wäre das kein Unterschied gewesen, allerdings hat die Klasse `Error` eine für diesen Verwendungszweck angemessenere Semantik[6].

Ein Vorteil der Klassen `Error` und `RuntimeException` gegenüber "normalen" Exception-Klassen ist, daß weder `Error`- noch `RuntimeException`-Klassen in der *throws*-Klausel einer Operation angegeben werden müssen. Daher kann in den Quelltext einer Operation ode instrumentiert werden, der eine Exception wirft, ohne daß die Signatur der Opertion verändert werden muß.

Code zum Prüfen der Zusicherungen

Die einzelnen Zusicherungen werden durch *if*-Abfragen überprüft. Schlägt eine Zusicherung fehl, so wird die entsprechende Exception geworfen. Dieser Prüfcode ist in einen *try-catch*-Block eingebettet, so daß andere Exceptions, die während der Prüfung der Zusicherungen auftreten, gefangen werden können. Sie werden dann in die entsprechende Zusicherungs-Exception ver-

[6] Aus der Klassendokumentation von `Error`: *"An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions."*

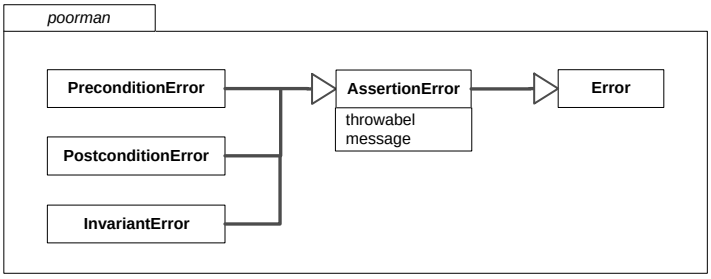

Abbildung 6.2: Exception-Klassen der Zusicherungen

packt und dann ebenfalls geworfen. Der folgende Quelltext zeigt, wie die Prüfung einer Vorbedingung aussieht:

```
public class Person {
  private int _age;
  /**
   * @require: age not negative : age >= 0;
   */
  public void setAge ( int age )
  {
  /********** PRECONDITION ** CHECK * BEGIN **********/
  try {
    boolean __PRE_0_Person_setAge__ = (  age >= 0 );

    if ( !( ((__PRE_0_Person_setAge__)) ))
      throw new de.sushi3003.dbc.poorman.PreconditionError(
        "The following precondition(s) failed: "
        +(__PRE_0_Person_setAge__ ? "":"[age not negative]")
       );
  }
  catch ( de.sushi3003.dbc.poorman.PreconditionError
          __PRE_ERROR__ ) {
    throw __PRE_ERROR__;
  }
  catch ( Throwable __THROWABLE__ ) {
    throw new de.sushi3003.dbc.poorman.PreconditionError
            ( "during require", __THROWABLE__ );
  }
  /********** PRECONDITION ** CHECK * END ***********/
      _age = age;
```

```
  }
}
```

Auffallend sind die beiden *catch*-Blöcke. Dabei scheint insbesondere der catch-Block, der den `PreconditionError` auffängt, unsinnig zu sein, da die gefangene Exception direkt wieder geworfen wird. Dies ist jedoch notwendig, da die Exception andernfalls vom zweiten catch-Block gefangen würde und der gefangene `PreconditionError` seinerseits in einen neuen `PreconditionError` verpackt würde.

Werden für eine Operation mehrere Zusicherungen definiert, so werden diese für den Fall, daß es sich um Nachbedingungen oder Invarianten handelt, nacheinander geprüft. Auch die geerbten Nachbedingungen und Invarianten werden in diesem sequentiellen Ablauf überprüft.

Für die Zusicherungen aus den Vorbedinungen gilt, daß alle Zusicherungen, die lokal für die Operation definiert wurden, "verundet" werden, denn alle Vorbedingungen müssen erfüllt sein, damit die Operation ihre Arbeit beginnen kann. Werden zusätzlich Vorbedingungszusicherungen aus einer Schnittstelle (oder Basisklasse) geerbt, so werden die dort definierten Zusicherungen zuerst "verundet" und dann mit den "verundeten" Zusicherungen der abgeleiteten Klasse "verodert".

Vermeidung von endlosen Rekursionen

In Kapitel 3.10.1 wurde die Problematik der endlosen Rekursionen erläutert. Damit der Entwickler sich nicht selber darum kümmern muß, endlose Rekursionen, die durch Zusicherungen verursacht werden können, zu vermeiden, generiert der Präprozessor Prüfcode, der automatisch endlose Rekursionen vermeidet.

Das folgende Beispielprogramm verdeutlicht das Konzept, welches zur Vermeidung von endlosen Rekursionen eingesetzt wird:

```
import java.util.Vector;

public class Recursion
{
    private Vector threads = new Vector();


    public void beginRecursion()
    {
        System.err.println ( "begin" );
```

```
        if ( ! threads.contains ( Thread.currentThread() ) ) {
            try {
                threads.add ( Thread.currentThread() );
                beginRecursion();
            }
            finally {
                threads.remove ( Thread.currentThread() );
            }
        }

        System.err.println ( "end" );
    }

}
```

Wird die Operation `beginRecursion()` aufgerufen, so passiert folgendes:

1. Die Zeichenkette “begin” wird ausgegeben.
2. Es wird nachgeschaut, ob sich der gegenwärtige Thread in der Liste befindet. Da bisher jedoch noch nichts in die Liste eingetragen wurde, wird in den if-Block verzweigt.
3. Der gegenwärtige Thread wird in die Liste eingetragen.
4. Die Operation ruft sich selbst auf.
5. Die Zeichenkette “begin” wird wieder ausgegeben.
6. Es wird wieder nachgeschaut, ob sich der gegenwärtige Thread in der Liste befindet. Da der Thread zuvor in die Liste eingetragen wurde, wird nicht mehr in den if-Block verzweigt.
7. Die Zeichenkette “end” wird ausgegeben.
8. Der rekursive Operationsaufruf kehrt zurück.
9. Der gegenwärtige Thread wird aus der Liste entfernt.
10. Die Zeichenkette “end” wird abermals ausgegeben.
11. Der Operationsaufruf terminiert.

Durch das Eintragen des Threads in die Liste und durch die Überprüfung vor dem rekursiven Aufruf wird die Rekursionstiefe auf 1 beschränkt. Alle Zusicherungen werden daher in ein solches ode-Fragment eingebettet, um somit endlose Rekursionen zu vermeiden.

Um die Threads in eine Liste eintragen zu können, wird in jede Klasse ein privates Attribut instrumentiert, das zum Speichern der Threads vom Prüfcode benutzt werden kann.

Generieren eines Konstruktors

Alle Invarianten müssen direkt nach der Erzeugung eines Exemplars gültig sein. In der Programmiersprache Java ist es jedoch so, daß man nicht zwingend einen Konstruktor für eine Klasse implementieren muß.

Hat eine Klasse keinen Konstruktor, so wird vom Präprozessor ein *Default-Konstruktor* generiert. Innerhalb des Konstruktors werden dann die Invarianten geprüft. Die Generierung eines Konstruktors geschieht allerdings nur, wenn die folgenden drei Bedingungen erfüllt sind:

1. Die Klasse besitzt keinen Konstruktor.

2. Für die Klasse (bzw. die Basisklasse oder Schnittstelle) wurde zumindest eine Invariante definiert.

3. Die Klasse ist keine abstrakte Klasse.

Funktionsweise des Instrumentierer-Subsystems

Nachdem nun beschrieben wurde, welcher ode an welchen Stellen im Quelltext eingefügt werden muß, wird die Funktionsweise des Instrumentierers beschrieben.

Der Instrumentierer erhält den Namen einer Quelldatei zusammen mit einem `ClassInfo`-Exemplar, das der Parser für diesen Quelltext ausgefüllt hat, und einem Ausgabestrom, in den der instrumentierte Quelltext geschrieben wird.

`EntryExit` - Positionen der Entitäten

Die für das Instrumentieren wichtigen Informationen sind in das `ClassInfo`-Exemplar eingetragen. An der Klasse `ClassInfo` "hängen" jedoch weitere Klassen, die ebenfalls Informationen zur Verfügung stellen, wie z.B. `ConstructorInfo, MethodInfo` oder (indirekt) `ReturnInfo` (vgl. Abbildung 6.1).

Diese Objekte repräsentieren die Stellen im Quelltext, an denen ode eingefügt werden muß, und beinhalten zu gleich die Informationen, um den entsprechenden Prüfcode zu generieren.

Um dem Instrumentierer eine abstraktere Sicht über den Aufbau des Quelltextes zu geben, implementieren die Informationsobjekte `ClassInfo`, `ConstructorInfo`, `MethodInfo` und `ReturnInfo` die Schnittstelle `EntryExit`. Über diese Schnittstelle kann die Position einer Entität innerhalb des Quelltextes erfragt werden (die Schnittstelle stellt dafür die Operationen `getEntry()` und `getExit()` bereit). Dabei können verschiedene Entitäten in einander verschachtelt sein. Die Abbildung 6.3 veranschaulicht die Sichtweise, die der Instrumentierer auf den Quelltext hat[7].

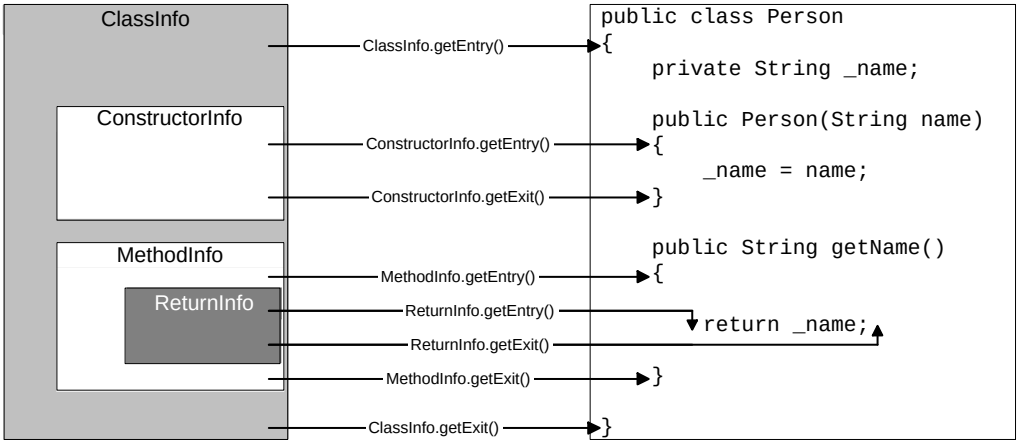

Abbildung 6.3: Sicht des Instrumentierers auf den Quelltext

Verpacken der Informationsobjekte

Der Instrumentierer nimmt das `ClassInfo`-Objekt, welches ihm zu Anfang übergeben wird, und extrahiert aus diesem die `ConstructorInfo`- und `MethodInfo`-Objekte. Danach wird das `ClassInfo`-Objekt in ein `InvInstrEngine`-Objekt verpackt und die `MethodInfo`-Objekte werden in `PrePostInstrEngine`-Objekte gehüllt. Sowohl das `InvInstrEngine`-Objekt als auch die `PrePostInstrEngine`-Objekte können als eine Art *Decorator*[Gamma et al.94] verstanden werden.

Das `InvInstrEngine`-Objekt ist in der Lage, ode zur Prüfung der Invarianten zu erzeugen. Ein Exemplar der Klasse `PrePostInstrEngine` kann

[7]Die Beschriftung der Pfeile in der Abbildung 6.3 läßt vermuten, daß es sich bei den Operation `getEntry()` und `getExit()` um statische Methoden handelt. Dem ist allerdings nicht so. Der Klassenname wurde lediglich der Übersichtlichkeit halber ergänzt.

für das *dekorierte* `MethodInfo`-Objekt den Prüfcode für die Vor- und Nachbedingungen generieren.

Der Instrumentierer nimmt nun alle verpackten Informationsobjekte und sortiert sie nach ihrem Eintrittspunkt (wird von der Operation `getEntry()` geliefert). Dadurch ergibt sich eine Struktur wie auf der linken Seite von Abbildung 6.3 dargestellt.

Der Instrumentierer nimmt nun der Reihe nach alle Informationsobjekte und kopiert dabei die ursprüngliche Quelldatei solange in den Ausgabestrom, bis die Position in der Quelldatei mit der Eintrittsposition des aktuellen Informationsobjekts übereinstimmt. An dieser Stelle muß dann, in Abhängigkeit von der Art des Informationsobjektes, unterschiedlich verfahren werden.

InvInstrEngine: Handelt es sich bei dem aktuellen Informationsobjekt um ein Exemplar der Klasse `InvInstrEngine`[8], so werden zunächst zwei private Attribute für die Klasse deklariert, in denen die Threads während der Prüfung der Zusicherungen gespeichert werden können, um endlose Rekursionen zu vermeiden. Dabei ist ein Attribut `static` deklariert, damit es innerhalb von statischen Operationen verwendet werden kann.

Danach wird, falls nötig, ein Konstruktor generiert, in dem die Invarianten geprüft werden.

ConstructorInfo: Die Quelldatei wird weiterhin in den Ausgabestrom geschrieben bis die Position in der Quelldatei mit der Austrittsposition des `ConstructorInfo`-Exemplars übereinstimmt. Am Ende des Konstruktors wird dann der ode zum Prüfen der Invarianten eingefügt.

PrePostInstrEngine: Bei Operationseintritt wird der Prüfcode für die Vorbedingungen und Invarianten eingefügt. Danach wird, für den Fall, daß die Operation nichts zurückliefert, die Quelldatei wieder solange in den Ausgabestrom geschrieben, bis die Austrittsposition der Operation erreicht ist. Dort wird dann der Prüfcode für die Nachbedingungen und Invarianten in die Ausgabe geschrieben.

Liefert die Operation etwas zurück, so wird anders verfahren. Das `PrePostInstrEngine`-Exemplar liefert eine nach Eintrittsposition sortierte Liste von `ReturnInfo`-Objekten zurück. Die Quelldatei wird dann immer solange in die Ausgabe geschrieben, bis eine *return*-Anweisung erreicht ist. Dort wird dann jeweils der Prüfcode für die Nachbedingungen und Invarianten in die Ausgabe geschrieben.

[8] Wenn es sich um einen gültigen (kompilierbaren) Quelltext handelt ist dies immer das erste Informationsobjekt in der Liste.

Nachdem der Instrumentierer das letzte Informationsobjekt erreicht hat, wird einfach der "Rest" der Quelldatei in die Ausgabe kopiert. Damit ist der Vorgang des Instrumentierens abgeschlossen.

6.5 Ausblick und Zusammenfassug

Im folgenden werden einige Dinge beschrieben, welche die Qualität des *sushi*-Präprozessors zusätzlich verbessern würden. All diese Dinge sind jedoch nicht essentiell für die Funktionstüchtigkeit des entwickelten Programms, sondern laufen lediglich unter der Rubrik "nice-to-have".

Die meisten dieser Dinge konnten aus Zeitgründen nicht im Rahmen der vorliegenden Arbeit realisiert werden. Bei anderen Dingen (z.B. Reviews) liegt es in der Natur der Sache begründet, daß sie erst nach Beendigung der Implementierung durchgeführt werden können.

6.5.1 Review und Refactoring

Die Architektur und Implementierung eines Programms ist in den wenigsten Fällen auf Anhieb perfekt. Daher sollte als nächster Schritt ein kritisches *Review* des Projekts erfolgen. Die dabei zu Tage tretenden Schwächen in der Architektur und der Implementierung sollten dann durch ein gründliches *Refactoring* (vgl. Kapitel 4.2) iterativ beseitigt werden.

6.5.2 Dokumentation der Verträge

Da Verträge für Klassen und Schnittstellen nur Sinn machen, wenn diese auch dokumentiert sind, sollte ein *Doclet* erstellt werden, um das Javadoc-Tool so zu erweitern, daß die Verträge auch in der automatisch generierten Dokumentation sichtbar sind. Zur Zeit muß der Entwickler immer noch in die Javadoc-Kommentare des Quelltextes sehen, um die Verträge zu lesen.

6.5.3 Erweiterte Funktionalität

Während der Implementierung wurde aus Zeitgründen darauf verzichtet, daß auch für Konstruktoren definierte Zusicherungen vom Präprozessor beachtet werden. Auch der aus Eiffel bekannte *old*-Mechanismus wurde nicht implementiert.

Das Fehlen dieser Funktionalitäten scheint jedoch für den praktischen Einsatz eher marginal zu sein, da während der Entwicklung des *sushi*-Präpro-

zessors mit den zur Verfügung stehenden Mitteln sehr gut gearbeitet werden konnte.

Das Ergänzen der Funktionalitäten ist daher eher unter dem Gesichtspunkt der Vollständigkeit zu sehen.

6.5.4 Automatische Generierung einer Makedatei

Bei der Verwendung der Programmiersprache Java wird meistens keine *Makedatei* benötigt, da sich die Abhängigkeiten unmittelbar aus den Quelltexten der Klassen ergeben[9]. Daher stellt der Einsatz eines Präprozessors eine zusätzliche Last dar, denn der Entwickler muß alle Klassen, die er seit der letzten Kompilierung verändert hat, zuerst mit dem Präprozessor bearbeiten.

Um den Entwickler zu entlasten, sollte ein Werkzeug (eine *Front-end* für den Präprozessor) implementiert werden, das für ein Projekt eine *Makedatei* erstellt, die zuerst alle benötigten Klassen instrumentiert und danach kompiliert.

6.5.5 Fazit

Das implementierte Programm wurde getestet und erwies sich als zuverlässig. Anhand verschiedener Indikatoren läßt sich auch die Qualität des *sushi*-Präprozessors belegen:

1. Während der gesamten Entwicklung wurde *Design by Contract* verwendet, um die Korrektheit der Implementierung sicherzustellen.

2. Der *sushi*-Präprozessor wurde in Übungen zum Thema *Design by Contract* von *Prof. Dr. Manfred Kaul* an der *Fachhochschule Rhein-Sieg* eingesetzt.

3. Der Softwareentwicklungsprozeß der Firma *Coodex Consulting GmbH* wird in naher Zukunft um die Methodik *Design by Contract* ergänzt. Das dabei zum Einsatz kommende Werkzeug ist der *sushi*-Präprozessor.

[9]Der Einsatz von Makedateien macht jedoch Sinn, wenn sich die Abhängigkeiten nicht mehr alleine aus den Quelltexten ergeben. Das ist z.B. der Fall bei der Verwendung von Java -RMI (Remote-Method-Invocation), da die *Stubs* und *Skelettons* nicht automatisch vom Java-Compiler erzeugt werden, sondern mit Hilfe des *rmic* Tools generiert werden müssen.

Internet Resourcen

[url prover] *Database of Existing Mechanized Reasoning Systems.* `http://www-formal.stanford.edu/clt/ARS/systems.html`

[url Heisenbug] unningham & unningham, Inc. *Heisen Bug.* `http://c2.com/cgi/wiki?HeisenBug`

[url XP] Jeffries Ron. *What is eXtreme Programming.* `http://www.xprogramming.com/what_is_xp.htm`

[url JUnit] Object Mentor Incorporated. *JUnit, Testing Resources for Extreme Programming.* `http://www.junit.org/`

[url JWAM] Apcom Workplace Solutions. *JWAM Home.* `http://www.jwam.de/`

[url Python] *Python Language Website.* `http://www.python.org/`

[url Ruby] *Ruby Home Page.* `http://www.ruby-lang.org/`

[url javacc/jjtree] Metamata, Inc. *Metamata.* `http://www.metamata.com/`

[url sushi] Hermanns Jan. *sushi Home.* `http://www.sushi3003.de/`

Literaturverzeichnis

[Balzert99] Balzert Helmut. *Lehrbuch Grundlagen der Informatik.* Spektrum Akademischer Verlag, 1999.

[Beck99] Beck Kent. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[Beck96] Beck Kent. *Smalltalk Best Practice Patterns.* Prentice Hall, 1996.

[Fowler et al.99] Fowler Martin, Beck Kent, Brant John, Opdyke William, Roberts Don. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Gamma et al.94] Gamma Erich, Helm Richard, Johnson Ralf, Vlissides John. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Beck/Gamma99] Beck Kent, Gamma Erich. *JUnit: A Cook's Tour.* Java Report Volume 5, Number 5, SIGS 1999.

[Hunt/David00] Hunt Andrew, Thomas David. *The Pragmatic Programmer.* Addison-Wesley, 2000.

[Karaorman et al.98] Karaorman Murat, Hölzle Urs, Bruno John. *jContractor: A Reflective Java Library to Support Design by Contract.* Department of omputer Science, University of alifornia, Technical Report TR S98-31, 1998.

[Kramer98] Kramer Reto. *iContract - The Java Design by Contract Tool.* ambridge Technology Patterns, 1998.

[Liskov/Wing94] Liskov Barbara, Wing Jeanette. *A Behavioral Notion of Subtyping.* A M Transactions on Programming Languages and Systems, Vol 16, No 6, November, 1994.

[Meyer97] Meyer Bertrand. *Object Oriented Software Construction.* Prentice Hall, second edition, 1997.

[Szyperski97] Szyperski lemens. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 1997.

[Vlissides98] Vlissides John. *Pattern Hatching - Design Patterns Applied.* Addison-Wesley, 1998.