



# **Technische Universität Darmstadt**

Fachbereich Elektrotechnik und Informationstechnik

Fachgebiet Echtzeitsysteme

Prof. Dr. rer. nat. Andy Schürr

## **Java Assertions**

Seminararbeit

von

Anton Pussep und Felix Becher

16. Februar 2008

# Inhaltsverzeichnis

1. Einleitung.....	3
2. Allgemein.....	4
2.1. Historischer Einblick.....	4
2.2. Design-by-Contract.....	5
2.3. Assertions.....	6
2.4. Vorteile von Design-by-Contract.....	7
2.5. Kosten der Versäumnisse.....	8
2.6. Akzeptanz von Design-by-Contract.....	9
3. Java 1.4 Assertion Facility.....	10
3.1. Geschichte.....	11
3.2. Spezifikation.....	11
3.2.1. Einführung.....	11
3.2.2. Kompatibilität und Kompilierung.....	12
3.3. Verwendung.....	12
3.3.1. Kontrollfluss-Invarianten.....	13
3.3.2. Klasseninvarianten.....	14
3.3.3. Preconditions.....	14
3.3.4. Postconditions.....	15
3.3.5. Nebeneffekte und komplizierte Assertions.....	15
3.4. Alternativen.....	16
3.5. Bewertung.....	17
4. Fazit.....	18
5. Literaturverzeichnis.....	19

*"How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows" Alan Turing [Turi49]*

## 1. Einleitung

Die Ansprüche an die eingesetzte Software steigen stetig. Zunehmend wird fehlerfreie und sichere Software erwartet. Je komplexer und umfangreicher die Programme werden, um so schwerer wird es die Qualität der Software zu gewährleisten. Doch gerade Qualität ist die wichtigste Voraussetzung für das Vertrauen der Kunden und somit auch ein entscheidendes Kriterium für jedes Unternehmen. Qualitätssicherung in der EDV wird dabei häufig als eigenständiger Prozess am Ende des Entwicklungszyklus gesehen. Allerdings wird diesem Prozess mangels Zeit und Geld oft wenig Aufmerksamkeit geschenkt, sodass Fehlerquellen oft übersehen oder nur mangelhaft behandelt werden. Um dieses Problem in den Griff zu kriegen bedarf es eines ganzheitlichen Qualitätsansatzes. Dadurch können sowohl die Zeit bis zur Inbetriebnahme verkürzt werden, als auch die Kosten der gesamten Qualitätssicherung gesenkt werden.

Design-by-Contract ist ein Konzept, das den Anspruch hat, Teil einer solchen ganzheitlichen Qualitätssicherung zu sein. Es behandelt Klassen wie Parteien, die bei ihrer Kooperation an Kontrakte gebunden sind. Diese Kontrakte dokumentieren somit die Annahmen und die Bedingungen, die für den korrekten Ablauf der Software notwendig sind.

Bereits in den 70er Jahren eingeführt, bilden Assertions die Basis für das Design-by-Contract-Konzept. So erlauben Assertions die Qualität der Software zu steigern, indem sie das Erkennen, Auffinden und Ausmerzen von Fehlern erheblich erleichtern. Assertions sind Zusicherungen im Software-Quellcode, die vom Programmierer festgelegt werden und die wahr sein müssen. Mit Java 1.4 wurden sie in Java eingeführt, jedoch ohne das bisher als das höchste Maß der Dinge geltende Konzept des Design-by-Contract umzusetzen.

Im Rahmen dieser Seminararbeit sollen zunächst in Kapitel 2 die Geschichte, Intention und Bedeutung von Assertions und des Design-by-Contract-Konzeptes vermittelt werden. Anhand von konkreten Beispielen

wird in Kapitel 3 erläutert, wie sich die Java 1.4 Assertion Facility in der Softwareentwicklung einsetzen lässt, um die Qualität von Software zu verbessern. Eine kritische Bewertung zum Abschluss von Kapitel 3 soll dem Leser ein Gefühl für die Möglichkeiten und Unzulänglichkeiten der Java 1.4 Assertion Facility vermitteln. Abschließend wird in Kapitel 4 bewertet, inwiefern die theoretischen Konzepte in Java umgesetzt wurden und dem Ziel der Qualitätssteigerung von Software zuträglich sind.

*„ ... if we don't state what a module should do, there is little likelihood that it will do it.“ Eiffel.com [ES08]*

## 2. Allgemein

Bertrand Meyer - der Entwickler der Programmiersprache Eiffel - nannte das erste Kapitel seines Buches "Object Oriented Software Construction" "Software Quality". Es ist bemerkenswert, welchen Stellenwert er diesem Thema damit einräumt. In seinem viel zitierten Buch präsentiert Meyer den objektorientierten Ansatz, der - man kann es wohl ohne Zweifel behaupten - mittlerweile in den meisten Softwareunternehmen zum Programmierstandard gehört.

Meyer nennt einige Merkmale (Eigenschaften) von Software und hebt besonders die folgenden fünf hervor: Extendibility (Erweiterbarkeit), Reusability (Wiederverwendbarkeit), Compatibility (Kompatibilität), Correctness (Korrektheit) und Robustness (Robustheit). Wegen der Ähnlichkeiten der letzten beiden Begriffe vereinigt Meyer diese häufig unter dem Begriff Reliability (Zuverlässigkeit). Die herausragende Bedeutung, die er dem Thema "Qualität" beimisst, kann man mit einem Satz zusammenfassen. "The purpose of software engineering is to find ways of building quality software". [Meyer97]

### 2.1. Historischer Einblick

Durch die Softwarekrise in den 60er Jahren des letzten Jahrtausends erfuhr der Bereich "Softwareentwicklung" eine Revolution. Der kreative Prozess, der davor praktisch uneingeschränkt im Mittelpunkt der Softwareherstellung stand, wurde nun kritisch betrachtet. Sowohl der informationstechnische als auch der wirtschaftsinformatische Ansatz zur Lösung dieser Krise lagen darin, den Softwareherstellungsprozess möglichst ingenieurmäßig zu gestalten, d.h. eine systematische Verwendung von Prinzipien, Methoden und Werkzeugen zu gewährleisten. So wurde das Konzept der Objektorientierten

Programmierung "geboren". Als eine der wichtigsten Ideen hinter dem Objektorientierten Programmieren steht jedoch der Aspekt der Wiederverwendbarkeit. Dadurch steigen die möglichen Kosten einer für Fehler anfälligen Routine bzw. die Kosten für die Beseitigung der Fehlerquelle um ein x-faches. Es ist also nicht verwunderlich, dass Meyer das Ziel der Softwareentwicklung speziell in der Herstellung von Qualitätssoftware sieht.

## 2.2. Design-by-Contract

Das Konzept zur Herstellung robuster und korrekt arbeitender Software hat Meyer Design-by-Contract genannt. 1986 wurde es erstmals mit der Einführung von Eiffel vorgestellt. Innerhalb dieses Konzepts wird ein Softwaresystem als eine Ansammlung von interaktiven Komponenten aufgefasst, deren Interaktion auf präzisen vordefinierten Spezifikationen beruht – den so genannten Kontrakten. [ES08]

Das Konzept basiert auf früheren Arbeiten von Floyd [Floy67], Hoare [Hoar69], Dijkstra [Dijk76] und Mills [Mill87], die sich schon früh darum bemühten, den kreativen Prozess des Programmierers durch methodische Werkzeuge zu unterstützen. Insbesondere um später sowohl unerwartete Systemabstürze, beispielsweise wegen falscher Input- oder Rückgabewerte zu vermeiden, als auch um spätere Anpassungen und Änderungen leichter und schneller verwirklichen zu können. Indem man sich mit formaler Verifikation, formaler Spezifikation und der so genannten Hoare Logik beschäftigte, legte man das Fundament für moderne Sprachen bzw. Konzepte. Schon zum damaligen Zeitpunkt erschien es den oben genannten Autoren wichtig, die Software auf Korrektheit überprüfen zu können bzw. zu beweisen, dass diese korrekt funktioniert. Doch Korrektheit ist ein wage formulierter Begriff. Um die Herangehensweise von Design-by-Contract zu verstehen, muss man also präzise definieren was darunter verstanden wird. Korrektheit, nach Meyer, kann nur im Sinne von Spezifikation existieren. Assertions, das eigentliche Thema dieser Arbeit, sind nichts anderes als Ausdrücke dieser Spezifikationen. Es ist ein Ausdruck bestehend aus einer Funktionseinheit und einer Eigenschaft dieser Einheit zu einem bestimmten Zeitpunkt während der Laufzeit. Zum Beispiel: die Bedingung, dass ein Integer nicht Null ist, wäre eine typische Assertion.

Es muss kritisch angemerkt werden, dass es schon seit den 60er Jahren Bestrebungen gab, den Herstellungsprozess der Software unter

Anwendung methodologischer Prinzipien zu gestalten. Jedoch dauerte es noch fast zwei Jahrzehnte, bis diese Prinzipien zum integralen Teil einer Programmiersprache wurden. Den möglichen Folgen dieser Entwicklung kann man am besten mit einem Zitat von Whittaker/Voas [WhVo02] Ausdruck verleihen: "Software quality is no better today than it was decades ago. In some cases, it's worse."

## 2.3. Assertions

Man kann Assertions in drei Gruppen einteilen:

- Preconditions – Voraussetzungen, die beim Aufruf einer Routine erfüllt sein müssen,
- Postconditions – Eigenschaften, die erfüllt sind, wenn die Routine abgearbeitet wurde, und
- Invarianten - globale Eigenschaften einer ganzen Klasse, die von allen Routinen dieser Klasse vor und nach jedem Aufruf eingehalten werden müssen (Klasseninvarianten) bzw. Eigenschaften, die beispielsweise nur innerhalb einer Methode und dort nur ab einem bestimmten Zeitpunkt gelten (interne Invarianten). Als Kontrollflussinvarianten werden Annahmen des Entwicklers über den Kontrollfluss bezeichnet, z. B. werden so Stellen "markiert", die nicht erreicht werden sollen.

Im Rahmen von Design-by-Contract besteht die Möglichkeit sowohl Pre- als auch Postconditions zu überschreiben. Damit wurde dem Vererbungsprinzip der Objektorientierung Rechnung getragen. Dem Überschreiben sind allerdings Grenzen gesetzt. So kann man Postconditions einer Unterklasse beispielsweise nicht abschwächen, während für Preconditions der umgekehrte Fall gilt. Diese dürfen nicht strenger sein als bei der Basisklasse. Invarianten der Basisklasse müssen in der Unterklasse erfüllt werden.

An dieser Stelle erscheint uns der Hinweis wichtig, dass Assertions weder einen Mechanismus darstellen, um Sonderfälle abzuarbeiten, noch eine Art Versicherung bei falscher Bedienung seitens der Benutzer sein sollen. Bei Assertions geht es "nur" um die Software-zu-Software-Kommunikation, insbesondere geht es nur darum, Bedingungen für Korrektheit zu schaffen. Falls diese Bedingungen zur Laufzeit verletzt werden, so bedeutet es nichts anderes als ein Vorhandensein eines Fehler in der Software, gemeinhin auch als "Bug" bekannt. Diese Fehler können aus zweierlei Quellen stammen: dem Client oder dem Supplier. Wobei

unter Client die aufrufende Routine und unter dem Supplier die verarbeitende Routine zu verstehen ist. Eine Verletzung der *Precondition* bedeutet, dass der Kontrakt seitens des Clients nicht erfüllt wurde. Umgekehrt deutet eine Verletzung der Postcondition auf einen Bug innerhalb der verarbeitenden Routine. Aus Gründen der Vollständigkeit sei an dieser Stelle angemerkt, dass ein Bughandling (Exception Handling) genauso zum Design-by-Contract Konzept gehört, wie Assertions auch. Es ist deswegen mehr als erstaunlich, dass Exception Handling ein gängiger Begriff in der IT-Welt ist, während Assertions einem eher bescheidenen Kreis von Programmierern ein Begriff sind.

## 2.4. Vorteile von Design-by-Contract

Bevor wir uns dem eigentlichen Thema dieser Arbeit - Assertions in Java - widmen, möchten wir an dieser Stelle nochmal klar und deutlich die Vorteile von Design-by-Contract benennen. Wie so oft in diesem Kapitel, halten wir uns dabei an die Gliederung, die Bertrand Meyer [Mey97] vorgeschlagen hat:

- Hilfe beim Schreiben korrekter Software
- Hilfe bei der Dokumentation
- Hilfe beim Testen, Debuggen und bei der Qualitätssicherung
- Hilfe bei Erstellung von effektiven Ausnahme-Behandlungen

Es ist für den Entwickler von fundamentaler Bedeutung zu wissen, dass seine Software korrekt funktioniert. Die Nutzung von Design-by-Contract bietet ihm eine Möglichkeit, die Software von Grund auf korrekt zu gestalten, anstatt sich einem zufriedenstellenden Resultat durch das Debuggen zu nähern. Auch gegenüber defensiver Programmierung bietet Design-by-Contract einen Vorteil. So besitzen die zusätzlichen Bedingungen bzw. Anweisungen selbst ein Gefahrenpotential. Generell bedeutet jede neue Quellcodezeile jedoch eine Steigerung in der Komplexität. Diese wiederum ist der ärgste Feind von Qualität und sollte nach Möglichkeit gering gehalten werden. Es erscheint daher rational mehr Zeit in die Präzision der formalen Spezifikation zu investieren, um später der undankbaren Aufgabe zu entkommen "die Nadel im Heuhaufen suchen zu müssen", bzw. den Fehler in den zusätzlichen Zeilen. Betrachtet man die Resultate und Erfahrungen (Tabelle 1) aus dem OS/400 Entwicklungsprojekt von IBM, so wird deutlich, dass Assertions einen sinnvollen Beitrag zur Qualitätssicherung leisten, wenn sie denn eingesetzt werden.

Size of delivered code (C++)	2,000,000 LOC
Number of classes	14,000
Number of methods	90,000
% of test code in form of assertions	40%
Assertion effectiveness	Assertions found 14 of the 18 most subtle bugs
Assertion use feasibility	Their cost has, so far, been a non issue
Development activity timeframe	1992-94
Conclusion on assertion usefulness	Helpful for class design, testing; evaluation ongoing

*Tabelle 1: Assertions and the OS/400 development project [HeMZ04]*

Es ist offensichtlich, dass Meyers theoretische Überlegungen durchaus von den praktischen Erfahrungen bei IBM gestützt werden. Aber auch bei Microsoft, einem führenden Anbieter von Software, baut man auf den Einsatz von Assertions, und zwar aus den gleichen Gründen. Beispielsweise soll eine Viertelmillion von Vor- und Nachbedingungen in Microsofts Office 2000 zum Einsatz gekommen sein. [Hoar01]

## 2.5. Kosten der Versäumnisse

Bezüglich der Dokumentationshilfe muss erwähnt werden, dass Meyer hier insbesondere den Aspekt der automatischen Dokumentation in den Vordergrund stellt. Der fehlerhafte Erstflug der Ariane-5 Trägerrakete offenbart die Konsequenzen eines Verzichts auf diese Möglichkeit. 500 Millionen US-Dollar sind beim Absturz der Rakete in der Atmosphäre verbrannt, weil man sich dem Prinzip der Wiederverwendbarkeit verschrieben hat, aber die Vorzüge von Design-by-Contract nicht erkannt bzw. aus Effizienzgründen nicht beachtet hat. Die Designer, die das Prinzip der Wiederverwendbarkeit für sich entdeckt haben, waren sich zu schade "das Rad jedes mal aufs Neue erfinden zu müssen". Also übernahmen sie Teile der Ariane-4 Software. Unglücklicherweise waren darunter auch Codesegmente, die für die Ariane-5 nicht von Bedeutung waren, sich aber bei dem Vorgängermodell bewährt haben und somit von den Entwicklern übernommen wurden. Diese verursachten den Programmfehler im Flugkontrollsystem, sodass der Bordcomputer die Selbstzerstörung einleitete. Das Backup-System schaltete wegen des gleichen Fehlers Sekunden vorher ab.

An diesem Beispiel wird deutlich, wie hoch die Kosten für ein fehlerbehaftetes, aber x-mal wiederverwendetes Modul liegen können.

Es zeigt, dass Wiederverwendbarkeit alleine nicht zur Qualitätssicherung beiträgt. Um diese richtig einzusetzen, bedarf es einer Dokumentation der vorliegenden Ergebnisse. Es bedarf einer klarer Beschreibung der Bedingungen und Rückgabewerte eines Moduls, einer Prozedur oder Komponente. Diese Bedingungen und Rückgabewerte sind nichts anderes als Bestandteile eines Kontraktes – die Pre- bzw. Postconditions, die als Assertions in der Software implementiert sind. Die Dokumentation wird somit Teil des Quellcodes. Bei Vorliegen eines automatischen Dokumentationstools (z. B. in Eiffel) wird die Aufgabe der Dokumentation dazu noch deutlich erleichtert. Bei Änderungen bspw. wird die Beschreibung automatisch angepasst. Somit entfällt eine mögliche Gefahrenquelle, nämlich eine nicht vollständige bzw. nicht aktuelle Beschreibung des vorliegenden Quellcodes. Nach Jézéquel/Meyer [JeMe97] war genau das die Ursache für den Absturz der Ariane-5. Obwohl nicht unumstritten, bleibt ihre Behauptung doch bestehen, dass mit der Verwendung von Design-by-Contract der Fehler in der IRS (Inertial Reference System) Einheit vermieden werden konnte und die Mission möglicherweise erfolgreich wäre.

Ein ähnliches Schicksal hat drei Jahre später, im Jahre 1999, der Mars Climate Orbiter erfahren müssen. Die Kosten der Entwicklung: 193 Millionen US-Dollar. Durch einen Fehler bei der Umrechnung der Einheiten aus dem "imperialen System" in das metrische verglühte die Sonde in der Atmosphäre des "roten Planeten". Und obwohl bei der Mission noch sehr viel mehr schief gelaufen war, wären Assertions möglicherweise ein wirksames Mittel um den Fehler noch vor dem Start der Mission zu diagnostizieren.

## 2.6. Akzeptanz von Design-by-Contract

Um dem Leser eine umfassende Vorstellung des Konzepts von Design-by-Contract bzw. Assertions zu geben, wollen wir an dieser Stelle einen Überblick über die Akzeptanz dieses Verfahrens geben. Diese Aufgabe lässt sich unter anderem dadurch bewältigen, dass dem Leser eine Auswahl der Programmiersprachen präsentiert wird, in denen dem Entwickler eine Möglichkeit geboten wird Assertions zu verwenden, und über Fortschritte bzw. die aktuelle Entwicklung im Bereich Design-by-Contract und speziell im Bereich Assertions berichtet wird.

Obwohl nicht so wie von Meyer skizziert, hat Design-by-Contract Einzug in viele Sprachen gehalten. Neben der schon häufig zitierten

objektorientierten Sprache Eiffel, bieten bspw. Java und C++ aber auch Perl und JavaScript Möglichkeiten Pre- bzw. Postconditions sowie Invarianten zu definieren. Doch nicht nur altbekannte, sondern auch weniger verbreitete und relativ junge Programmiersprachen wie Ruby oder D versuchen auf Design-by-Contract zu achten. An dieser Stelle muss ergänzend hinzugefügt werden, dass mit Ausnahme von Eiffel und seiner Derivate kaum Sprachen existieren, die das Konzept von Design-by-Contract als integralen Bestandteil ansehen. Sowohl für objektorientierte Sprachen wie Ruby oder C++ als auch für Javascript oder Lisp existieren aber Tools, Bibliotheken oder Preprozessoren, die es erlauben, die Funktionalität der Sprache auszudehnen. C# beispielsweise, das 2001 von Microsoft entwickelt wurde, bekam eine Erweiterung der Sprache – Spec#, die es möglich machte Design-by-Contract zu verwenden.

Bevor wir unser Augenmerk auf die Implementation von Assertions in Java richten, erscheint es uns an dieser Stelle wichtig, auf die Anwendung von Assertions auf verwandten Gebieten hinzuweisen. Zu erwähnen sind da Boyapati[BoKM02], Khurshid, und Marinov mit ihrer Arbeit zu automatisierten Tests von Datenstrukturen in Java Programmen und die Arbeit von Baudry, Ouabdesselam, Robach und Le Traon [LORB03], die mit Hilfe von Kontrakten versuchen, die Qualität von komponenten-basierten Systemen zu messen. Dabei achten sie auf Faktoren wie Robustheit und Diagnosefähigkeit. Während die Arbeit von Baudry et al. noch recht theoretisch erscheint und wenig Bezug zur Praxis aufweist, erzeugt Boyapaty et al. ein neuartiges Framework - "Korat", das anhand von Preconditions automatisierte Testfälle erstellt, diese dann anwendet und anhand von Postconditions die Korrektheit des Outputs verifiziert.

### **3. Java 1.4 Assertion Facility**

In Java 1.4 wurden Assertions als neues Konstrukt zur Java-Sprachspezifikation hinzugefügt. Dieser Abschnitt geht auf die Geschichte, Intention und Verwendung von Assertions in Java ein. Dabei wird vor allem aus Sicht der Java-Entwickler argumentiert. In der abschließenden Bewertung soll das Sprachkonstrukt ausführlich bewertet werden und ein Ausblick auf Alternativen gegeben werden.

## 3.1. Geschichte

Ursprünglich sollten Assertions bereits in Oak, dem Vorgänger von Java, implementiert werden. So findet sich in der Oak-Sprachspezifikation 0.2 ein Kapitel zu Assertions, in dem vermerkt ist, dass Assertions noch nicht implementiert sind [GP94]. Es war die Absicht von James Gosling, dem Erfinder von Oak und Java, Assertions in Java zu implementieren: "My major regret about that spec is that the section on assertions didn't survive: I had a partial implementation, but I ripped it out to meet a deadline" [GJ07].

Damit wird deutlich, für wie bedeutend Gosling Assertions erachtete. Sie wären wohl das erst nächste Sprachkonstrukt gewesen, das Gosling zur ersten Java-Version hinzugefügt hätte, wenn mehr Zeit zur Verfügung gestanden hätte. Um so mehr verwundert es, dass sie erst in Java 1.4 Eingang in die Sprachspezifikation gefunden haben.

Ebenfalls interessant ist die Tatsache, dass die Java 1.4 Assertion Facility schließlich in einer solch einfachen Form implementiert wurde. Die Entwickler von Java haben die Implementierung der Java 1.4 Assertion Facility nach dem Design-by-Contract-Konzept erwogen, sich jedoch schließlich für eine weniger riskante und weitaus einfachere Variante entschieden. Die in Oak vorgesehenen Preconditions, Postconditions und echte Klasseninvarianten wurden nicht übernommen, genauso wenig wie auch die Vererbung von Assertions. Auch sollten Assertions in Oak weder beschränkt noch umdefiniert werden können [GP94].

## 3.2. Spezifikation

### 3.2.1. Einführung

Seit Version 1.4 unterstützt Java Assertions durch die Anweisung `assert`. Das bereitgestellte Schlüsselwort `assert` kann in zwei Formen verwendet werden:

```
assert Expression1 ;  
assert Expression1 : Expression2 ;
```

Sind Assertions zur Laufzeit abgeschaltet, so werden Ausdrücke in `assert`-Anweisungen nicht ausgewertet. `assert`-Anweisungen werden in diesem Fall wie leere Anweisungen behandelt. Sind Assertions eingeschaltet, so wird `Expression1` als ein boolescher Ausdruck

ausgewertet, der die vom Programmierer aufgestellte Behauptung enthält. Ist die Behauptung *wahr*, so wird die `assert`-Anweisung beendet. Ist sie falsch, dann wird ein `java.lang.AssertionError` vom Typ `java.lang.Error` geworfen. Bei der Verwendung der zweiten Form wird `Expression2` als Argument an den Konstruktor von `AssertionError` übergeben.

### 3.2.2. Kompatibilität und Kompilierung

Quelldateien, die Assertions enthalten, sind nicht kompatibel zu Interpretern vor Java 1.4. Auch können ältere Quelldateien, die `assert` als Bezeichner verwenden, mit neueren Compilern nicht übersetzt werden. Kompilierte Bytecode-Dateien vor Java 1.4 bereiten dagegen bei der Ausführung mit neueren Interpretern keine Probleme.

Um Quelldateien mit Assertions zu kompilieren, muss im Java 1.4 Compiler der Parameter `"-source 1.4"` an den Compiler übergeben werden, sonst wird bei der Kompilierung die `assert` Anweisung mit dem Fehler `"is not a statement"` quittiert. Dies wurde so spezifiziert, um Programme, die `assert` als Bezeichner einsetzen, eine Übergangszeit zu erlauben und den Quellcode zum Java 1.4 Compiler kompatibel zu halten. Seit Java 1.5 muss die Assertion Facility im Java-Compiler jedoch nicht mehr explizit eingeschaltet werden.

### 3.3. Verwendung

Grundsätzlich können Assertions verwendet werden, um beliebige Annahmen über das Programm zur Laufzeit zu überprüfen. Beachtet werden sollte, dass beim Fehlschlagen einer Assertion ein `Error` geworfen wird, der per Konvention nicht abgefangen werden sollte. Eine nicht erfüllte Bedingung sollte als Ursache nur einen Programmierfehler haben können [Ulle07]. Da Programmierfehler eigentlich nicht sinnvoll behandelt werden können, sollte das Programm beendet werden.

Assertions lassen sich nach Belieben aktivieren und deaktivieren. Und zwar auf der Basis von einzelnen Klassen, ganzen Packages oder bezogen auf Systemklassen. Damit lassen sich gezielte Tests durchführen, die das gesamte System, einzelne Module oder Moduleile überprüfen. Standardmäßig sind Assertions in Suns Java-Interpreter deaktiviert.

Im Folgenden sollen Regeln genannt werden, wie die aus Design-by-Contract bekannten Prinzipien mit Assertions umgesetzt werden. Dazu

skizzieren wir am Anfang eine kleine Anwendung "Kassensystem", die uns als Beispiel dienen soll. Das entsprechende UML-Klassendiagramm ist in Abbildung 1 dargestellt.

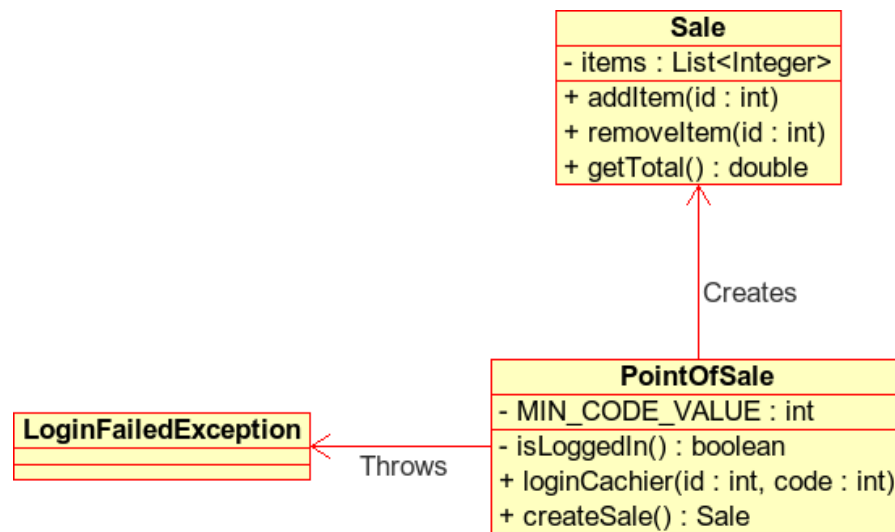


Abbildung 1: Klassendiagramm der Beispielanwendung Kassensystem

Wir stellen uns ein Geschäft vor. In dem Geschäft sind Kassen aufgestellt, die durch die Klasse `PointOfSale` modelliert sind. An den Kassen müssen sich Kassierer mit einer ID und einem Code einloggen. Das geschieht, mit dem Aufruf der Methode `loginCachier(int id, int code)`. Anschließend können Einkäufe von Kunden entgegen genommen werden, indem die Artikelnummern eingegeben werden. Dazu wird für jeden Kaufvorgang eine neue Instanz der Klasse `Sale` erstellt und durch Aufrufe der Methode `addItem(int id)` Artikel hinzugefügt.

### 3.3.1. Kontrollfluss-Invarianten

Ein guter Indikator, wann eine Assertion benutzt werden sollte, ist es wenn ein Punkt eines Programms nur unter bestimmten Umständen oder nie erreicht werden sollte. Oft werden solche Stellen im Code mit Kommentaren versehen. Stattdessen sollten Assertions verwendet werden, weil damit die Lesbarkeit und Korrektheit des Quellcodes verbessert werden kann.

In unserer Beispielanwendung könnte die Methode `Sale.removeItem(int id)` aussehen wie folgt, wobei die letzte Zeile eine Kontrollfluss-Invariante ist:

```
public void removeItem(int id) {
    for (Integer containedId: items) {
        if (containedId.intValue() == id) {
            items.remove(containedId);
            return;
        }
    }
    assert false;
}
```

### 3.3.2. Klasseninvarianten

Mit Java-Assertions lassen sich auch Klasseninvarianten implementieren. Hierzu können am Anfang und am Ende jeder Methode Assertions eingebaut werden, die Klasseninvarianten auf ihre Gültigkeit überprüfen.

In der skizzierten Beispielanwendung wäre eine Klasseninvariante, dass ein Einkauf von einem Kunden nie eine negative Anzahl von Produkten enthalten kann. Sie kann realisiert werden, indem in `Sale.removeItem(int item)` in der ersten Zeile und vor der `return`-Anweisung die folgende Zeile eingefügt wird:

```
assert items.size() >= 0;
```

### 3.3.3. Preconditions

Laut Konvention sollten Preconditions nicht immer durch Assertions sichergestellt werden. Und zwar werden in Java in öffentlichen Methoden per Konvention Exceptions geworfen, wenn Parameter illegale Werte haben [SM07]. Hier bietet sich insbesondere `IllegalArgumentException` im `java.lang` Paket an. Dadurch bekommt der Aufrufer die Möglichkeit seinen Fehler zu erkennen, indem er die Exception abfängt. Bei privaten Methoden wird dagegen oft aus Performancegründen auf die Überprüfung der Parameter verzichtet. Dies ist möglich, da der Programmierer Kenntnis über den Aufrufer (nämlich die aktuelle Klasse) verfügt und Annahmen über die übergebenen Werte treffen kann. Externe Aufrufer, die oft über keine interne Kenntnis verfügen, können dagegen durchaus auch fehlerhafte Werte übergeben, z. B. weil sie den erlaubten Wertebereich evtl. gar nicht kennen. Ob diese Konvention so sinnvoll ist, wird u. a. von [Roge01] in Frage gestellt, wird jedoch bis heute von Java-Entwicklern empfohlen [SM07].

In unserer Beispielanwendung wäre eine Precondition in `PointOfSale.createSale()`, dass ein Kassierer eingeloggt sein muss, bevor er Artikelnummern eingeben kann. Da `createSale()` eine öffentliche Methode ist, sollten laut Konvention Preconditions hier durch Exceptions sichergestellt werden:

```
public Sale createSale() {
    if (! isLoggedIn()) {
        throw new IllegalStateException("Login a cachier first!");
    }
    return new Sale();
}
```

### 3.3.4. Postconditions

Assertions sind dazu geeignet, als Postconditions zu fungieren. Sie sind ein vollwertiger Ersatz für einfache Postconditions, um z. B. zu überprüfen, ob eine bestimmte Variable einen gültigen Wert hat.

Um das oben genannte Beispiel fortzuführen, können wir zu `PointOfSale.loginCachier(int id, int code)` eine Postcondition hinzufügen, die sicherstellt, dass nach dem Beenden der Methode ein Kassierer eingeloggt ist:

```
public void loginCachier(int id, int code)
    throws LoginFailedException {
    // ... Login-Vorgang
    assert loggedIn(); // postcondition
}
```

### 3.3.5. Nebeneffekte und komplizierte Assertions

Java-Assertions müssen nicht frei von Nebeneffekten sein. So können Zuweisungen innerhalb der `assert`-Ausdrücke verwendet werden oder es können Methoden aufgerufen werden, die Daten manipulieren.

Dies ist sehr zweckmäßig, wenn komplizierte Behauptungen realisiert werden müssen, bei denen Daten zur Assertion-Überprüfung nur dann aufbereitet werden, wenn Assertions tatsächlich aktiviert sind. Im Folgenden sei ein Beispiel skizziert, bei dem davon Gebrauch gemacht wird, indem eine Sicherungskopie eines Parameters gemacht wird, an dem keine Manipulationen vorgenommen werden dürfen:

```
public void workOnComplicatedStructure(Object struct) {  
    // Vorbereitung von Daten für Postcondition  
    Object backup = null;  
    assert (backup = struct.clone()) != null;  
  
    // ... Verwendung von struct  
  
    // Postcondition  
    assert backup.equals(struct);  
}
```

Auf der anderen Seite geht eine vermeintliche Fehlerquelle auf, weil das Programm sich je nach aktivierten oder deaktivierten Assertions unterschiedlich verhält. Betrachte folgendes Beispiel:

```
int i = 0;  
assert ++i == 1;
```

Bei deaktivierten Assertions ist der Wert der Variable `i` nach dem Durchlauf dieses Codes 0, mit aktivierten Assertions dagegen 1. Deswegen sollte grundsätzlich auf Datenmanipulation in Assertions verzichtet werden, es sei denn, sie wird nur auf eigens dafür geschaffenen Daten vorgenommen und ist für die Sicherstellung von Behauptungen notwendig.

## 3.4. Alternativen

Es gibt eine Reihe von Alternativen zu der Java 1.4 Assertion Facility. Diese Alternativen setzen oft das Design-by-Contract-Konzept als Ganzes um, verfolgen aber unterschiedliche Ansätze.

Jass wurde vor dem Erscheinen von Java 1.4 in Form einer Diplomarbeit entwickelt, mit dem Ziel, das Design-by-Contract-Konzept komplett umzusetzen. Dabei werden Assertions als Kommentare in den Quellcode geschrieben und vor der Kompilation von einem Precompiler in Java-Code umgewandelt. [JA07]

Die Java Modelling Language (JML) ist eine anerkannte und verbreitete Sprachspezifikation zur Umsetzung des Design-by-Contract-Konzepts in Java. Es verwendet s. g. "auskommentierte Notationen". Mit Tools können diese Kommentare in Assertions überführt werden. Genannt seien ESC/Java2, das eine statische Analyse durchführt und `jmlrec`, das wie Jass einen Precompiler einsetzt, um Kommentare in Java-Code zu überführen und Assertions auch zur Javadoc hinzufügt.

contract4j setzt ebenfalls Design-by-Contract in Java um. Es verwendet Annotationen und bietet mehrere Möglichkeiten, Assertions zur Laufzeit sicherzustellen. Von allen Alternativen, die wir untersucht haben, verwendet diese sicherlich die neuesten Technologien (Annotationen und AspectJ) und wird vermutlich an Bedeutung gewinnen.

### 3.5. Bewertung

Mit Sicherheit entspricht die vergleichsweise sehr rudimentäre Assertion Facility in Java nicht allen Forderungen des Design-by-Contract. Grundsätzlich sind jedoch Pre-, Postconditions und Invarianten mit Assertions umsetzbar, allerdings geht die Verwendung eines einzigen Schlüsselworts auf die Kosten der Lesbarkeit des Codes. So lassen sich Klasseninvarianten nur schwer von Pre- und Postconditions unterscheiden. Dem steht entgegen, dass schon durch einen kurzen Kommentar hinter der Assertion dessen Zweck ersichtlich werden kann (`//class-invariant` oder `//precondition`). Außerdem ist ohnehin fraglich, ob sich ein Entwickler überlegen will, inwiefern es sich um eine Precondition oder Invariante handelt und ob die Verwendung von falschen Schlüsselwörtern nicht mehr Verwirrung als Nutzen stiften würde.

Als negativ anzusehen ist, dass Assertions nicht frei von Seiteneffekten sein müssen. Dadurch können sich Fehler einschleichen, die ohne sie nicht möglich gewesen wären und die schwer aufzufinden sind.

Es lassen sich keine Assertion-Blöcke definieren, die komplett aktiviert oder deaktiviert werden können. Die Java-Entwickler verweisen darauf, dass stattdessen Methoden definiert werden sollen, vor allem um die Lesbarkeit des Codes zu gewährleisten [SM07]. Dennoch bleibt fraglich, ob der Zusatzaufwand für eine neue Methode den Nutzen nicht übersteigt.

Dass Assertions nicht vererbt werden können, kann bei Vererbung zur Duplizierung von Code oder zur Aufweichung von Assertions führen. Somit muss der Entwickler beim Klassenentwurf bedenken, dass die von ihm gemachten Zusicherungen in Unterklassen nicht notwendigerweise gewährleistet sein müssen.

Die Tatsache, dass Assertions beim Kompilationsprozess nicht ausgelassen werden können hat unweigerlich zu Zielkonflikten geführt. So argumentieren die Entwickler, dass Assertions nicht weggelassen

werden können, weil sie gewollte Seiteneffekte haben können. Dafür musste aber auf lesbare `AssertionError`-Fehlermeldungen verzichtet werden (z. B. die Angabe des fehlgeschlagenen Ausdrucks), um die kompilierten Dateien nicht mit Strings aufzublähen. [SM07]

Dadurch, dass Assertions nicht durch Javadoc unterstützt werden, wurde hier eine Möglichkeit zur Verbesserung der Code- und Systemdokumentation verspielt. Es ist aber zu vermuten, dass es die Handhabung und Erlernbarkeit von Assertions erheblich erschweren würde.

## 4. Fazit

Mit der Java 1.4 Assertion Facility wurde ein wichtiges Sprachkonstrukt zur Verfügung gestellt, das von Entwicklern lang ersehnt wurde. Es wurde als ein "Leichtgewicht" implementiert, das einfach zu erlernen und einzusetzen ist. Die Auswirkungen auf die Performance sind vernachlässigbar und die Kompatibilität zu älteren Quelldateien und Interpretern wurde so weit wie möglich gewährleistet.

Java Assertions sind mit Sicherheit dazu geeignet, die Korrektheit der geschriebenen Software entscheidend zu verbessern - eine Anforderung, die von Exceptions nicht erfüllt werden kann [Roge01]. Exceptions verbessern die Robustheit von Programmen, der auch Java Assertions zuträglich sein können. Auch tragen Assertions dazu bei, die Dokumentation des Quellcodes zu verbessern, indem sie die unterstellten Bedingungen aufzeigen.

Der Forderung nach Design-by-Contract für Java wurde nicht entsprochen. Zwar findet sie sich heute auf Platz zwei der Request-for-Enhancement-Liste [SM08], aber konkrete Pläne für Design-by-Contract als Sprachkonstrukt in Java scheint es nicht zu geben. Vielmehr lassen die Java-Entwickler die Option offen, indem sie bewusst erwähnen, dass die Java 1.4 Assertion Facility der zukünftigen Aufnahme einer ausgiebigeren Implementierung nicht im Wege steht [SM07].

Auch lässt die bisherige Implementierung vor allem die Einbindung in vorhandene Java-Sprachkonstrukte wie Vererbung und Javadoc vermissen. Seiteneffekte in Assertions können zu sehr schwer auffindbaren Fehlern führen. Stattdessen haben die Java-Entwickler den Fokus stark auf Einfachheit und Erlernbarkeit gesetzt und damit unserer Meinung nach vorerst einen guten Kompromiss gefunden. Aus den gewonnenen Erfahrungen kann die Java Assertion Facility in Zukunft

ausgebaut werden. Sollte sie dagegen schon in der jetzigen Form keine Akzeptanz finden, ist es auch fraglich, ob eine mächtigere und kompliziertere Version nicht um so mehr dieses Schicksal erleiden würde.

Im Hinblick auf die weitere Entwicklung ist es nicht unwahrscheinlich, dass vor allem die Alternativen zur Java 1.4 Assertion Facility ihre Entwicklung bestimmen werden. Insbesondere JML scheint Akzeptanz zu finden - die neueste Version von Jass soll ebenfalls auf JML basieren. Die Vor- und Nachteile dieser Alternativen bleiben jedoch zukünftigen Untersuchungen überlassen.

## 5. Literaturverzeichnis

- [BoKM02] Boyapati, Chandrasekhar; Khurshid, Sarfraz; Marinov, Darko:  
Korat: Automated Testing Based on Java Predicates.  
ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Juli 2002, Rom, Italien
- [Dijk76] Dijkstra, Edsger W.: A Discipline of Programming.  
Prentice-Hall Series in Automatic Computation, 1976, Englewood Cliffs, NJ
- [ES08] Eiffel Software : Building bug-free O-O software: An introduction to Design by Contract(TM).  
<http://archive.eiffel.com/doc/manuals/technology/contract/>,  
Zugriff am 16.01.2008
- [Floy67] Floyd, Robert W.: Assigning Meaning to Programs.  
in Proceedings of Symposium on Applied Mathematics, 1967, Bd. 19, J.T. Schwartz (Ed.), A.M.S., S. 19-32
- [GJ07] Gosling, James: A Brief History of the Green Project.  
<https://duke.dev.java.net/green/>, Zugriff am 02.12.2007
- [GP94] Green Project: Oak Language Specification 0.2. 1994,  
<http://www.europrog.ru/doc/doc-sun1994-000001e.pdf>,  
Zugriff am 02.12.2007
- [HeMZ04] Henne-Wu, Rachel; Mitchell, William; Zhang, Cui: Support for Design By Contract™ in the C# Programming Language.  
Journal of Object Technology, September/Oktober 2004, Bd. 4, Nr. 7, S. 65-82
- [Hoar01] Hoare, Charles A. R.: Assertions: a personal perspective.  
IEEE Annals of the History of Computing, April/Juni 2003, Bd. 25, Nr. 2, S. 14-25
- [Hoar69] Hoare, Charles A. R.: An axiomatic basis for computer programming.  
Communications of the ACM, October 1969, Bd. 12, Nr. 10, S. 576-585

- [JA07] Jass Homepage:  
<http://csd.informatik.unioldenburg.de/~jass/>, Zugriff am 12.12.2007
- [JeMe97] Jézéquel, Jean-Marc; Meyer, Bertrand: Design by Contract: The Lessons of Ariane. IEEE Computer, Januar 1997, Bd. 30, Nr. 2, S. 129 - 130
- [LORB03] Le Traon, Yves; Ouabdessalam, Farid; Robach, Chantal; Baudry, Benoit: From diagnosis to diagnosability: axiomatization, measurement and application. Journal of Systems and Software, Januar 2003, Band 65 Nr. 1, S. 31-50
- [Meye97] Meyer, Bertrand: Object Oriented Software Construction. 2nd Edition, Prentice Hall, 1997
- [Mill87] Mills, Harlan D.: Principles of Computer Programming: A Mathematical Approach. Allyn and Bacon, Boston, 1987
- [Roge01] Rogers, Paul: J2SE 1.4 premieres Java's assertion capabilities, Part 2. (2001),  
<http://www.javaworld.com/javaworld/jw-12-2001/jw-1214-assert.html>, Zugriff am 02.12.2007
- [SM07] Sun Microsystems: Programming with Assertions.  
<http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html>, Zugriff am 02.12.2007
- [SM08] Sun Microsystems: Top 25 RFE's (Request for Enhancements).  
[http://bugs.sun.com/bugdatabase/top25\\_rfes.do](http://bugs.sun.com/bugdatabase/top25_rfes.do), Zugriff am 02.01.2008
- [Turi49] Turing, Alan: Checking a large routine. Beitrag zur EDSAC Inaugural Conference, 24 June 1949, in: "Report of a Conference on High Speed Automatic Calculating Machines", S. 67-9
- [Ulle07] Ullenbohm, Christian: Java ist auch ein Insel. Galileo Press, 6. Auflage, Dezember 2007
- [WhVo02] Whittaker, James A.; Voas, Jeffrey M.: 50 Years of Software: Key Principles for Quality. iT Professional, November/Dezember 2002, Bd. 4, Nr. 6, S. 28-35