

F. Steimann
Mitarbeit: D. Keller

Moderne Programmiertechniken und -methoden

Kurseinheiten 1 - 7

Fakultät für
Mathematik und
Informatik

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Der Inhalt dieses Studienbriefs wird gedruckt auf Recyclingpapier (80 g/m², weiß), hergestellt aus 100 % Altpapier.

Inhaltsverzeichnis

Inhaltsverzeichnis	3
Vorwort (F. Steimann Sommer 2005).....	8
Übersicht.....	9
1 Interfacebasierte Programmierung	11
1.1 Der Begriff des Interfaces.....	11
1.2 Interfaces als Typen	13
1.2.1 Explizite Interfaceimplementierung	16
1.2.2 Nominale vs. strukturelle Typkonformität	17
1.2.3 Interfaces vs. abstrakte Klassen	18
1.3 Eigenschaften von Interfaces.....	19
1.3.1 Aufrufen und aufgerufen werden: die zwei Seiten eines Interfaces	19
1.3.2 Totale und partielle Interfaces	20
1.3.3 Öffentliche vs. veröffentlichte Interfaces	22
1.4 Anzeichen interfacebasierter Programmierung	23
1.5 Arten des Gebrauchs von Interfaces.....	28
1.5.1 Übersicht.....	28
1.5.2 Anbietende Interfaces.....	29
1.5.3 Allgemeine Interfaces	29
1.5.4 Idiosynkratische Interfaces	30
1.5.5 Familieninterfaces	31
1.5.6 Kontextspezifische Interfaces	31
1.5.7 Client/Server-Interfaces	32
1.5.8 Ermöglichende Interfaces.....	33
1.5.9 Server/Client-Interfaces	34
1.5.10 Server/Item-Interfaces	36
1.5.11 Gebrauch von Interfaces in der Praxis.....	38
1.6 Dependency injection	39
1.6.1 Constructor injection	40
1.6.2 Setter injection	41
1.6.3 Interface injection	41
1.6.4 Assembler.....	42
1.6.5 Einschränkungen	43
1.6.6 Alternativen.....	43

1.6.7 Fazit	44
1.7 Umkehrung von Abhängigkeiten mit Interfaces	45
1.8 Interpretation von Interfaces als Rollen	48
1.9 Werkzeugunterstützung für das interfacebasierte Programmieren	49
1.10 Weiterführende Literatur	51
1.11 Lösungen zu den Selbsttestaufgaben.....	52
2 Design by contract	53
2.1 Verhaltensspezifikation durch Zusicherungen: Vor- und Nachbedingungen, Invarianten	55
2.2 Ein paar einfache Beispiele	57
2.3 Design by contract in der Analysephase	59
2.4 Design by contract in der Programmierung	60
2.4.1 Zeitpunkt der Überprüfung von Zusicherungen.....	61
2.4.2 Beeinflussung der Programmierung	63
2.5 Zusicherungen und Vererbung	64
2.6 Spezifikationssprachen für das Design by contract	66
2.6.1 EIFFEL	67
2.6.2 JAVA	70
2.6.3 JML	73
2.6.4 Grenzen der Ausdrucksstärke	77
2.7 Design by contract als Form des Testens	77
2.8 Vor- und Nachteile des Design by contract.....	78
2.9 Zusammenfassung und Ausblick.....	80
2.10 Weiterführende Literatur	81
2.11 Lösungen zu den Selbsttestaufgaben.....	82
3 Unit-Testen	84
3.1 Der Begriff des Testfalls	84
3.1.1 Was testet ein Testfall?.....	86
3.1.2 Organisation von Testfällen	87
3.2 JUNIT	87
3.2.1 Interner Aufbau des JUNIT-3.8-Frameworks.....	88
3.2.2 Die Anwendung von JUNIT.....	100
3.2.3 Änderungen in JUNIT 4	102
3.2.4 Änderungen mit JUNIT 4.4	104
3.2.5 Probleme von JUNIT	106

3.3	Vererbung von Testfällen	107
3.4	Das Testen von Interfaces	107
3.5	Testen mit Mock-Objekten.....	111
3.5.1	Ausprogrammierte Mock-Objekte	111
3.5.2	Mock-Frameworks	115
3.5.3	Grenzen der Einsetzbarkeit von Mock-Objekten	117
3.5.4	Mock-Objekte bei ermöglichenen Interfaces	117
3.6	Auswertung von Unit-Tests zur Fehlerlokalisierung.....	117
3.6.1	Abdeckungsbasierte Fehlerlokalisierung	118
3.6.2	Modellbasierte Fehlerlokalisierung	120
3.6.3	Andere Arten von Fehlerlokatoren	120
3.6.4	Kombination von Fehlerlokatoren	120
3.7	Kontinuierliches Testen	121
3.8	Wer testet die Tests?	121
3.9	Unit-Testen, Design by contract, Typprüfung – drei Wege, ein Ziel	123
3.10	Weiterführende Literatur	126
3.11	Lösungen zu den Selbsttestaufgaben	127
4	Entwurfsmuster	128
4.1	Historisches	128
4.2	Übergeordnete objektorientierte Programmierprinzipien	129
4.2.1	Offene Rekursion und das Vererbungsinterface.....	130
4.2.2	Vererbung vs. Komposition	133
4.2.3	Forwarding vs. Delegation	133
4.3	Definition	134
4.4	Wichtige Entwurfsmuster.....	135
4.4.1	COMPOSITE Pattern.....	135
4.4.2	OBSERVER Pattern.....	143
4.4.3	TEMPLATE METHOD Pattern	148
4.4.4	STRATEGY Pattern.....	149
4.4.5	ROLE OBJECT Pattern.....	150
4.4.6	FACTORY METHOD Pattern	152
4.4.7	ADAPTER Pattern	157
4.4.8	FACADE Pattern	160
4.4.9	VISITOR Pattern	161

4.5	Bewertung	168
4.6	Ausblick	168
4.7	Weiterführende Literatur	169
4.8	Lösungen der Selbsttestaufgaben	169
5	Refactoring	171
5.1	Einordnung	172
5.1.1	Katalogisierung	172
5.1.2	Refaktorisierungen als Algorithmen	173
5.1.3	Refactoring to patterns.....	174
5.1.4	Werkzeugunterstützung	174
5.1.5	Ein Beispiel	176
5.2	Eine Auswahl von Refactorings.....	180
5.2.1	Bedingungen vereinfachen	180
5.2.2	Lesbarkeit verbessern.....	189
5.2.3	Daten organisieren	196
5.2.4	Generalisierung einsetzen.....	208
5.2.5	Methoden organisieren	219
5.3	Zusammenfassung und Ausblick.....	230
5.4	Weiterführende Literatur	231
5.5	Lösungen der Selbsttestaufgaben	232
6	Metaprogrammierung	234
6.1	Metaprogrammierung auf sich selbst: Reflektion	236
6.1.1	Reflektieren ohne zu verändern: Introspektion	236
6.1.2	Introspektion in JAVA	237
6.1.3	Interzession	239
6.1.4	Modifikation.....	239
6.1.5	Bewertung der Reflektion	239
6.2	Programmieren mit Metadaten: Annotationen und Attribute	240
6.2.1	Annotationstypen.....	241
6.2.2	Annotationsinstanzen und deren Verwendung	243
6.2.3	Annotationsverarbeitung zur Übersetzungszeit	244
6.3	Aspektorientierte Programmierung	244
6.3.1	Entwicklungsgeschichtliche Einordnung.....	245
6.3.2	Inhalte von Aspekten.....	247

6.3.3 Charakterisierung der aspektorientierten Programmierung	248
6.3.4 Aspektorientierte Programmierung und Modularisierung	253
6.3.5 Aspektorientierte Programmierung und Lesbarkeit	255
6.3.6 ASPECTJ	257
6.4 Zusammenfassung und Ausblick	263
6.5 Weiterführende Literatur	264
6.6 Lösungen der Selbsttestaufgaben	265
7 Extreme Programming	266
7.1 Geschichte des Extreme Programming	267
7.2 Ziele des Extreme Programming	269
7.3 Der Test-first-Ansatz	269
7.4 Das Programmieren in Paaren	271
7.5 Keine Planung	273
7.6 Die Kundin vor Ort	275
7.7 Gemeinsame Verantwortung	276
7.8 Der Prozess des Extreme Programming	277
7.9 Voraussetzungen für den Einsatz von Extreme Programming	279
7.10 Werkzeuge des Extreme Programming	281
7.11 Extreme Programming als risikogetriebene Methode	282
7.12 Zusammenfassung	284
7.13 Übergang zu agilen Prozessen	285
7.14 Weiterführende Literatur	286
Index	286

Vorwort (F. Steimann Sommer 2005)

Was ist modern? Das, was gerade angesagt ist? Dann müsste diese Vorlesung jedes Semester neu geschrieben werden. Was dann?

Manches Wissen der Informatik hat eine erschreckend kurze Halbwertszeit. Das gilt auch für den Bereich der Softwareentwicklung und der Programmiersysteme: Die Programmiersprache der Wahl scheint heute JAVA zu sein, die dazugehörige Entwicklungsumgebung vielleicht ECLIPSE. Extreme Programming und agile Softwareentwicklung erschienen uns gestern als die Zukunft, heute ist es darum schon deutlich ruhiger geworden. Was also gehört in eine Vorlesung, die das Attribut „modern“ trägt?

Meine Antwort darauf heißt: Wissen, das man vor Jahren noch nicht hatte, das Sie aber voraussichtlich auch noch nutzen können, wenn Sie Ihr Studium abgeschlossen haben und wenn Sie dann (wieder) mitten in einem Programmierprojekt stecken. Ihr hier erworbenes Wissen entspricht dann sicher nicht dem neuesten Hype, aber es hat hoffentlich noch eine gewisse Aktualität (während der Hype von heute vielleicht längst als „ganz nette Idee, aber letztlich doch untauglich“ abgeschrieben ist). Um konkreter zu werden: Das Wissen dieses Kurses sollte eine Halbwertszeit von zehn Jahren haben, d. h., die Hälfte dessen, das Sie heute lernen, sollte in zehn Jahren noch gültig und verwertbar sein.

Modern heißt aber auch immer: Noch nicht vor der Geschichte bewährt. Und so habe ich mir erlaubt, in einem Kurstext das Ideal der Einheit von Forschung und Lehre beim Wort zu nehmen und das eine oder andere an meinem Lehrgebiet erzielte Forschungsergebnis in den Text einfließen zu lassen. Da diese Ergebnisse allesamt jüngeren Ursprungs sind, gilt für sie natürlich ganz besonders, dass sie sich noch nicht bewährt haben. Auf der anderen Seite finden Sie ja darin vielleicht einen interessanten Denkansatz und idealerweise sogar ein Thema für eine eigene Abschlussarbeit.

Noch ein Wort zur Sprache: Aufgrund eines Rektoratsbeschlusses der Fernuniversität bin ich gehalten, eine geschlechtsneutrale Sprache zu verwenden oder, wo nicht möglich, beide Geschlechter anzusprechen. Ich kann nicht sagen, ob dies überhaupt praktikabel ist — Proponentinnen und Proponenten mögen sich „Programmierer- und Programmiererinnenproduktivität“ zu Gemüte führen oder versuchen, einen Satz wie „Wer glaubt, das sei einfach, ohne es probiert zu haben, den kann ich nicht ernstnehmen.“ geschlechtsneutral zu formulieren, ohne daraus ein Ungetüm zu machen —, aber eine verständliche Sprache auf dem Altar der Gleichstellung zu opfern war für mich keine Option. Ich habe mich deshalb dazu entschlossen, ausschließlich die weibliche Form zu verwenden. Diese Entscheidung führt hier und da zu unerwarteten Wendungen, die zeigen, wie sehr das männliche Geschlecht in unserer Sprache verankert ist, vor allem aber dazu, dass einer die direkte Ansprache eines Geschlechts überhaupt erst auffällt. Ich hoffe, dass sich dadurch niemand, die diesen Text liest, diskriminiert fühlt.

Übersicht

Der Kurs beginnt mit einem etwas eigenwilligen Thema, nämlich der sog. *interfacebasierten Programmierung*. Diese propagiert die Verwendung von Interfaces (als Typen wie in JAVA oder C#) anstelle von Klassen bei der Typisierung von Variablen (also in Variablendeklarationen). Das Konzept und die Verwendung von Interfaces ist ein immer wiederkehrendes Thema in den folgenden Kurseinheiten; die Einführung der interfacebasierten Programmierung gleich zu Anfang scheint daher gerechtfertigt, selbst wenn es sich bei ihr ausdrücklich nicht um ein Standardthema handelt.

Interfaces à la JAVA und C# sind unvollständig. Was ihnen fehlt, ist eine (formale) Beschreibung dessen, was die Einhaltung des Interfaces über die rein syntaktischen Methodensignaturen hinaus verlangt, gewissermaßen eine Semantik der Methoden oder, anders gesagt, eine genaue, damit verbundene Verhaltensspezifikation. *Design by contract* ist ein besonders griffig formuliertes Prinzip, dieses Defizit auszugleichen: Ihm zufolge wird über ein Interface ein Vertrag geschlossen, nach dem beide Seiten — gewissermaßen über Kreuz — gegenseitige Verpflichtungen und Nutzen haben. Die Einhaltung dieses Vertrages kann über sog. Zusicherungen zur Laufzeit und — in ausgewählten Fällen — auch statisch, also zur Übersetzungszeit, geprüft werden. All dies ist Gegenstand der zweiten Kurseinheit.

Die Überprüfung der Einhaltung von Verträgen sowie allgemeiner der Korrektheit von Code über Zusicherungen ist nur eine Möglichkeit, für Qualität in der Programmierung zu sorgen. Die andere ist das Testen. Zwar ist das Testen nicht besonders beliebt (es hat gewissermaßen destruktiven Charakter — man macht die großen Würfe anderer kaputt, ohne jemals selbst brillieren zu können), doch ist es nach wie vor unverzichtbar. Nicht zuletzt nutzt einer das ganze schöne Design by contract nichts, wenn dessen Zusicherungen bei der Kundin das erste Mal zur Anwendung kommen und dann dort eine Vertragsverletzung melden. Sog. *Unit-Tests*, insbesondere die, die auf dem Framework JUNIT basieren, haben das Testen unter Programmiererinnen etwas populärer gemacht, weswegen ihnen auch eine ganze Kurseinheit gewidmet wird.

Die vierte Kurseinheit widmet sich dann der Idee der *Entwurfsmuster* und damit einem deutlich populäreren Thema. Entwurfsmuster bieten zunächst einen Katalog von Standardlösungen für häufig wiederkehrende Entwurfsprobleme; sie bilden aber ganz nebenbei auch ein Vokabular, das es Softwareentwicklerinnen erlaubt, sich kurz und prägnant über Software zu unterhalten, und zwar ohne sich in Details zu verlieren, nämlich unter Verweis auf bekannte Konzepte (ganz so, wie sich Drehbuchautoren und Produzenten in Robert Altmans „The Player“ über Filme unterhalten). Beinahe überflüssig zu sagen, dass Interfaces in Entwurfsmustern eine wichtige Rolle spielen.

Auch wenn es viele nicht wahrhaben wollen: Die Programmierung ist ein evolutionärer Prozess, bei dem einmal getroffene Entscheidungen ständig auf die Probe gestellt werden und gegebenenfalls wieder geändert werden müssen. Die Änderung von Code ist aber in der Regel eine verzweigte und entsprechend verzwickte Angelegenheit: Nur selten ist es mit einer Änderung an einer Stelle getan. Das führt dann häufig zu der Erkenntnis, dass Software unter Änderung verdirbt. Dem sollen sog. *Refactorings*, standardisierte und teilweise auch automatisierte Änderun-

gen von Code, die dessen Bedeutung nicht verändern, entgegenwirken. Ja noch viel mehr: Mit Hilfe von Refactorings soll sich der Entwurf (und damit die Qualität) existierender Software durch gezielte Änderungen nachträglich verbessern lassen.

Als Abschluss der Programmietechniken wird noch ein anderes Thema aufgegriffen, das — unter einem neuen Deckmäntelchen namens *aspektorientierte Programmierung* — vor einigen Jahren für Aufsehen sorgte: die Metaprogrammierung. Unter *Metaprogrammierung* versteht man zunächst die Erstellung von Programmen, die Programme erzeugen oder ändern. Besonders interessant wird die Metaprogrammierung dann, wenn ein Programm sich selbst zu verändern in der Lage ist. Dies ist in gewisser Weise bei der aspektorientierten Programmierung der Fall. Mit einer Behandlung dieser und den immer aktueller werdenden Annotationen schließt diese Kurseinheit.

Unit-Tests und Refactorings sind zwei Programmietechniken, die im Rahmen einer bestimmten Programmiermethode größere Bekanntheit erlangt haben, nämlich des Extreme Programming. Extreme Programming vereint eine Vielzahl relativ unorthodoxer Herangehensweisen zu einem Gesamtkunstwerk, dessen Praxistauglichkeit in der Vergangenheit viel diskutiert wurde. Ohne dass heute ein abschließendes Ergebnis vorläge, haben die „agilen Methoden“ oder Prozesse, zu denen Extreme Programming gehört, zu einer immer stärker werdenden Abkehr von schwergewichtigen Softwareentwicklungsansätzen und damit, zumindest aus Sicht der Programmierinnen, zu einer Art Befreiung geführt. Auch wenn die agile Softwareentwicklung längst nicht auf alle Projekte und Organisationen passt, so lässt sich doch sicher die eine oder andere Anregung daraus mitnehmen. Deshalb endet das Kapitel mit einer kurzen Einführung in die agilen Methoden Scrum, Kanban und Feature Driven Development (FDD).

1 Interfacebasierte Programmierung

Program to an interface, not an implementation.

Erstes Prinzip wiederverwendbaren objektorientierten Designs, aus [1]

Interfacebasierte Programmierung (engl. interface-based programming) ist kein feststehender Begriff wie etwa objektorientierte Programmierung oder Design by contract. Man kann noch nicht einmal davon sprechen, dass er sich eingebürgert hätte; dazu wird er, zumindest in der Literatur, viel zu wenig verwendet. Es handelt sich vielmehr um einen Nischenbegriff, der im Umfeld der objektorientierten Programmierung (vor allem bei MICROSOFT) geprägt wurde und der eine bestimmte Alternative (oder Ergänzung) dazu darstellt. Bevor sich die eine oder andere Leserin jetzt abwendet: An der interfacebasierten Programmierung ist nichts proprietär und sie wird bereits vielfach eingesetzt, ohne dass das jeweils so deklariert wäre.

Unter interfacebasierter Programmierung versteht man in erster Linie die Verwendung von Interfaces à la JAVA und C# anstelle von Klassen und deren Superklassen in Variablen Deklarationen. In JAVA und C# definieren Interfaces genau wie Klassen Typen (vgl. Kurs 01814), geben aber — anders als Klassen — keine Implementierungen vor. Die interfacebasierte Programmierung bietet damit *Polymorphismus* und *dynamisches Binden* (also den Umstand, dass die Implementierung einer aufgerufenen Methode von der konkreten Klasse des Empfängerobjekts abhängt), ohne dies mit dem Konzept der *Vererbung* zu verquicken. Zwar war die Vererbung früher eines der Hauptargumente für die Einführung und rasche Verbreitung der objektorientierten Programmierung, aber inzwischen, nachdem die teilweise recht subtilen Probleme der Vererbung offenbar wurden (Stichwort *Fragile base class problem*; s. Kurs 01814), hat sich die Begeisterung ziemlich gelegt. Die interfacebasierte Programmierung hingegen konzentriert sich in gewisser Weise auf einen unstrittigen Aspekt der objektorientierten Programmierung. Doch der Reihe nach.

1.1 Der Begriff des Interfaces

Interfaces sind ein sehr allgemeines Konzept der Informatik. Zunächst vor allem von Bedeutung für die Entwicklung von Hardware, wurde der Interface-Begriff recht bald auf die Softwareentwicklung übertragen und dort spätestens durch die Arbeiten von Dijkstra [2] und Parnas [3] bekanntgemacht.

Die IEEE definiert ein Interface als „eine gemeinsame Grenze, über die hinweg Information gebracht wird“ [4]. Im Software Engineering ist mit Grenze praktisch immer *Modulgrenze* gemeint; tatsächlich war, zumindest in der Vergangenheit, der Begriff des Interfaces praktisch unauflöslich mit dem des *Moduls* verbunden (die ACM listet in ihrer Begriffsklassifikation Interface und Modul gemeinsam unter dem Eintrag D.2.2, „Design Tools and Techniques“, auf [5]).

Die bekanntesten frühen Arbeiten zu Modulen und Interfaces sowie dem damit verbundenen Geheimnisprinzip („Information Hiding“) stammen von Parnas. Parnas argumentiert, dass jede

Entwurfsentscheidung in einem Modul gefasst und damit hinter einer Schnittstelle (Interface) verborgen werden soll, die sich bei einer (späteren) Änderung der Entwurfsentscheidung nicht mit ändert. Die Entwurfsentscheidung wird somit zum Geheimnis des Moduls und die Auswirkungen der Entwurfsentscheidung bleiben lokal begrenzt. Die Vorteile liegen auf der Hand: Änderungen, die sich in der Praxis niemals ausschließen lassen, wirken sich, bei einer vorausschauenden Modularisierung des Systems, immer nur auf Teile dessen aus; der Änderungsaufwand bleibt damit beschränkt, die unerwünschten Nebeneffekte werden minimiert. Auch wenn es in der Praxis nicht immer ganz so einfach ist, so hat der Parnassche Modularisierungsansatz doch unbestreitbar Vorteile.

Welche Bedeutung Interfaces und das Modulkonzept im Folgenden für die Programmierung hatten, kann man schon daran erkennen, dass bedeutende Programmiersprachen wie MODULA und ADA mit eigenen Sprachkonstrukten dafür ausgestattet wurden. So kann die Schnittstelle eines MODULA-2-Moduls beispielsweise wie folgt aussehen:

Beispiel

```

1 DEFINITION MODULE A;
2 PROCEDURE m();
3 PROCEDURE n();
4 VAR b : BOOLEAN;
5 END A.

```

Auf das **DEFINITION MODULE** folgt dann ein gleichnamiges **IMPLEMENTATION MODULE**¹, das außer den Deklarationen der exportierten Prozeduren auch noch deren Implementierungen sowie private Programmelemente enthält. Man beachte, dass von Modulen neben Prozeduren auch Typen, Variablen und Konstanten exportiert werden können.

Mit Hilfe der Sprachkonstrukte zur Modulbildung lassen sich übrigens auf einfache und technisch saubere Art und Weise abstrakte Datentypen implementieren, die später (zumindest den Theoretikerinnen) als Grundlage der objektorientierten Programmierung dienen sollten. Man kann also durchaus auch schon in Sprachen wie MODULA oder ADA *objektbasiert*, d. h. objekt-orientiert ohne Vererbung, programmieren.

Klassen als Module

Mit dem Aufkommen der *objektorientierten* Programmierung wurde das Modulkonzept weitgehend durch den Klassenbegriff ersetzt². Eine Klasse hat eine **Schnittstelle**, hinter der sie ihre Entwurfsentscheidung, im Wesentlichen die Implementierung der Methoden, verbirgt (bzw. verbergen kann). Während jedoch in Sprachen wie ADA und MODULA die Schnittstellenspezifikation von der Implementierung getrennt (etwa in verschiedenen Abschnitten des Quelltextes oder sogar in verschiedenen Dateien) vorgenommen wurde, wird in Programmiersprachen wie JAVA oder C# die **Schnittstelle einer Klasse** gemeinsam mit

¹ In Modula-3 heißt ersteres INTERFACE, letzteres nur noch MODULE.

² Sog. Pakete wie die Packages in JAVA werden gelegentlich auch als Module bezeichnet. Es ist jedoch fraglich, inwieweit eine bloße Sammlung von Untermodulen, den enthaltenen Klassen, selbst dem Modulbegriff gerecht wird.

ihrer Implementierung spezifiziert. Dazu werden lediglich den Attributen bzw. Methodennamen sog. *Access modifier* vorangestellt, die angeben, ob ein Element der Klasse öffentlich zugänglich und damit Teil ihrer Schnittstelle sein soll oder ob der Zugriff nur eingeschränkt (im Rahmen der Möglichkeiten der jeweiligen Programmiersprache) erfolgen können soll. Im Folgenden nennen wir Schnittstellenspezifikationen, die durch öffentliche Access modifier (in JAVA **public**) an Ort und Stelle der Implementierung deklariert werden, **Klasseninterfaces**.

Beispiel

Das Klasseninterface der JAVA-Klasse

```
6  public class A {  
7      int i;  
8      public m() {...}  
9      public n() {...}  
10     protected o() {...}  
11     private p() {...}  
12     q() {...}  
13 }
```

besteht aus den Methoden m() und n().

1.2 Interfaces als Typen

Die erste (bekanntere) Programmiersprache, die Interfaces als eigenständige Typen einföhrte (so dass Variablen mit Interfaces statt mit Klassen als Typ deklariert werden konnten), war vermutlich CLU[6]. Durch Interfacetypen war es in CLU möglich, *Module* getrennt voneinander zu kompilieren, was nicht nur die Entwicklungszeit deutlich verkürzte (früher waren Compiler noch wesentlich langsamer als heute), sondern auch die getrennte Auslieferung von Softwarekomponenten (eben diesen Modulen) erlaubte. Quasi nebenbei war es dadurch auch möglich geworden, dass der (Implementierungs-)Typ eines Objektes, das einer Variablen zugewiesen wurde, zur Laufzeit variierte (*Polymorphismus*). Voraussetzung hierfür war lediglich, dass es erlaubt war, zwei oder mehr alternative Implementierungen eines Interfaces gleichzeitig in einem Programm zu haben. Dies wird dadurch möglich, dass Interface und Implementierungen verschiedene Bezeichner haben.

CLU mag als akademische Programmiersprache einige Aufmerksamkeit erfahren haben, aber der breiten Masse der Programmiererinnen blieben sie und ihre Konzepte doch aber wohl unbekannt. So wurde erst Jahre später unabhängig von einer konkreten Programmiersprache untersucht, inwieweit sich Interfacespezifikationen als Typen eignen[7]. Das so entstandene Interface-als-Typ-Konzept erfuhr jedoch erst mit dem Aufkommen der Programmiersprache JAVA größere Popularität: In JAVA können ja tatsächlich Variablen aller Art (Felder, Parameter, temporäre Variablen und Rückgabewerte von Methoden) mit Interfaces als Typ deklariert werden. Wie es die Interfaces-als-Typen von ihren Vorläufern in JAVA geschafft haben, ist mir nicht genau bekannt: William Cook will James Gosling eine Kopie seiner Arbeit [7] in die Hand gedrückt haben (persönliche Kommunikation), offiziell hört man jedoch eher, dass JAVA seine Interfaces von der Pro-

grammiersprache OBJECTIVE-C übernommen hat, wo sie allerdings *Protokolle* heißen, was wiederum von SMALLTALK bzw. dessen Nachfolger STRONGTALK übernommen worden sein dürfte.

Interfaces als Ersatz für fehlende Mehrfachvererbung?

Bedauerlicherweise ist in der JAVA -Dokumentation wenig zum Grund der Einführung von Interfaces in die Sprache zu finden. Man liest dort lediglich, dass sie als Ersatz für die fehlende *Mehrfachvererbung* dienen sollen. Das allerdings ist schwach, denn Interfaces vererben ja nichts — sie zwingen vielmehr ihren implementierenden (konkreten) Klassen auf, für die im Interface genannten Methoden Implementierungen anzugeben. Vererbt wird also allenfalls die Schnittstelle, weswegen man gelegentlich auch von *Interface inheritance* — im Gegensatz zu *Implementation inheritance* — spricht. Eigentlich sollten aber die Zuweisungskompatibilität und die damit verbundene *Substituierbarkeit*, eben das interfacebasierte Programmieren, bei der Vorstellung von Interfaces im Mittelpunkt stehen.

Entwicklung der Wahrnehmung und des Gebrauchs von Interfaces

Tatsächlich schien zunächst nicht ganz klar, wie Interfaces in der JAVA-Programmierung zu verwenden wären. Dies spiegelt sich recht anschaulich in Abbildung 1.1 wider, der der Verlauf des Einsatzes von Interfaces im JDK über die Version 1.0 bis 1.4 zu entnehmen ist. So gab es anfangs (Version 1.0) nur wenige Interface-Implementierungen (ca. 0,3) pro Klasse, in Version 1.1 dann schon immerhin fast 0,8 und in Version 1.2 schon fast 1,8. Der erste Sprung lässt sich auf die Einführung von `java.io.Serializable`, einem klassischen sog. *Tagging-* oder *Marker-Interface* (s. Abschnitt 1.5.10), zurückführen, der zweite hingegen auf die Einführung von SWING mit seinen zahlreichen Listenern. Man beachte, dass Tagging- oder Marker-Interfaces lediglich zum Markieren von Klassen verwendet werden, eine Funktion, die heute weitgehend durch *Annotationen* (Attribute im C#-Jargon; s. Abschnitt 6.2) ersetzt wird.

Eine weitere interessante Entwicklung ist ebenfalls Abbildung 1.1 zu entnehmen: das Verhältnis von interfacetypisierten zu klassentypisierten Variablen. Dieses hat sich nämlich nur einmal sprunghaft verändert, und zwar (wiederum) beim Wechsel von Java 1 auf Java 2. Neben Swing ist dafür diesmal auch das Java-2-Collection-Framework verantwortlich, dessen Interfaces (wie beispielsweise `List`) zwar nur selten implementiert werden, die jedoch sehr häufig in Variablen-deklarationen auftauchen. Diese Beobachtung deutet bereits darauf hin, dass es durchaus unterschiedliche Arten der Verwendung von Interfaces gibt, ein Umstand, der Gegenstand von Abschnitt 1.5 sein wird.

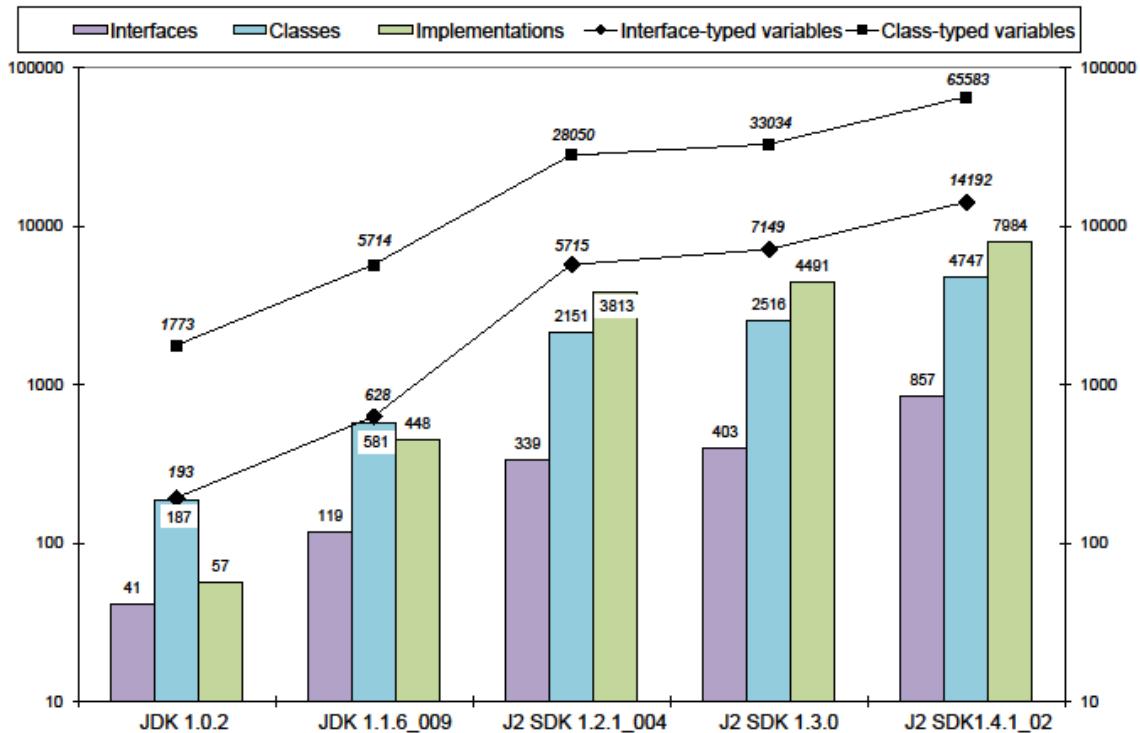


Abbildung 1.1: Entwicklung des Einsatzes von Interfaces im JDK über die Zeit. Man beachte die logarithmische Skala: gleiche Differenzen in der Höhe einzelner Säulen entsprechen gleichen Verhältnissen. So ist das Verhältnis von Klassen zu Interfaces über alle Versionen in etwa konstant geblieben (ca. 5,5 : 1), während sich das Verhältnis von mit Klassen typisierten zu mit Interfaces typisierten Variablen beim Übergang von Java 1 zu Java 2 fast halbiert hat (von ca. 9 : 1 zu ca. 5 : 1). Gleichzeitig ist die durchschnittliche Anzahl implementierter Interfaces pro Klasse von 0,8 auf 1,8 angestiegen. Woran mag das liegen? Antworten im Text.

Selbsttestaufgabe 1.1

Schätzen und notieren Sie, bevor Sie weiterlesen, welche die am häufigsten implementierten und welche die am häufigsten referenzierten Interfaces des JDK 1.4 sind. Welche Interfaces, meinen Sie, tauchen in beiden Listen auf?

java.awt.event.ActionListener	java.util.Enumeration	javax.accessibility.Accessible
java.awt.image.ImageObserver	java.util.EventListener	javax.swing.Action
java.awt.MenuContainer	java.util.Iterator	javax.swing.Icon
java.awt.Shape	java.util.List	javax.swing.text.AttributeSet
java.io.Serializable	java.util.Map	javax.swing.text.Element
java.lang.CharSequence	java.util.Set	org.omg.CORBA.Object
java.lang.Cloneable	javax.accessibility.Accessible	org.omg.CORBA.portable.IDLEntity
java.security.PrivilegedAction	javax.swing.Action	org.w3c.dom.Node
java.util.Collection	javax.swing.Icon	javax.accessibility.Accessible

1.2.1 Explizite Interfaceimplementierung

C# hat das Interfacekonzept von JAVA übernommen und weiterentwickelt. So ist es in C# möglich, dass eine Klasse für Methoden gleichen Namens und gleicher Signatur, die sie aus verschiedenen Interfaces „erbt“, unterschiedliche Implementierungen anbietet. Diese Implementierungen sind dann nur über das jeweilige Interface zugreifbar, was so viel heißt wie dass eine Variable mit dem Typ des Interfaces deklariert sein muss, um auf die jeweilige Methodenimplementierung zugreifen zu können.

Beispiel

Im folgenden Codebeispiel deklarieren zwei verschiedene Interfaces dieselbe Methode(nsignatur).

```

14 interface I {
15     void m();
16 }

17 interface J {
18     void m();
19 }

20 class A : I, J {
21     ...
22 }
```

Die Klasse A implementiert beide Interfaces, I und J; (in C# steht der Doppelpunkt sowohl für das aus JAVA bekannte **implements** als auch für das **extends**).

In JAVA kann für **void** m() in A nur eine Implementierung angegeben werden. Da die Interfaces I und J aber verschieden sind (im gegebenen Beispiel unterscheiden sie sich zwar nur durch den Namen, aber in der Praxis könnten sie auch weitere Methoden haben, deren Signaturen sich nicht gleichen), ist davon auszugehen, dass ihre Methoden auch verschiedenen Zwecken dienen und somit Verschiedenes tun sollen. (Gerade im gegebenen Beispiel würde man sonst ja nur ein Interface benötigen, oder die Interfaces könnten voneinander erben.) In C# ist es nun möglich, dem Rechnung zu tragen, indem man die Klasse beide Interfaces parallel implementieren lässt:

```

23 public class A : I, J {
24     void I.m() {...}
25     void J.m() {...}
26 }
```

Im C#-Jargon nennt man das **explizite Interfaceimplementierung** (engl. explicit interface implementation). Der Aufruf der Methoden eines so implementierten Interfaces muss dann über das Interface erfolgen, also z. B. durch

```
27 A a = new A();  
28 I i = (I) a;  
29 i.m();  
30 ((J) a).m();
```

Die Methoden von expliziten Interfaceimplementierungen gehören jedoch nicht zum *Klassen-interface*. So würde ein Aufruf wie

```
31 a.m();
```

zu einem Compiler-Fehler führen. Es ist somit möglich, dass das Klasseninterface einer Klasse leer ist, auf die Eigenschaften der Instanzen der Klassen also nur über Variablen, die ein entsprechendes Interface zum Typ haben, zugegriffen werden kann (vgl. das Beispiel aus Abschnitt 1.5.6).

Beide Eigenschaften von C#, die mehrfache Implementierung der gleichen Signatur, wenn verschiedene Interfaces einer Klasse dies verlangen, und der Zugriff auf Methoden ausschließlich über Interfacetypen, stehen für das **Prinzip des interfacebasierten Programmierens**, das von MICROSOFT schon früh (als *Interface inheritance*) über die objektorientierte Programmierung (hier insbesondere die *Vererbung zwischen Klassen*, die *Implementation inheritance*) gestellt wurde. So basiert beispielsweise der COM-Standard auf Interfaces, ohne objektorientiert zu sein, und VISUAL BASIC kam — mit Interfaces — lange ohne Klassen aus.

Zusammenfassung: Interface-als-Typ-Konzept

Auch wenn der Begriff des Interfaces traditionell nicht mit einem Typ verbunden ist, so haben doch Programmiersprachen wie C# und JAVA mit ihrem Interface-als-Typ-Konzept der Programmierung attraktive Möglichkeiten eröffnet, die es zuvor nicht gab. Diese sollen im Folgenden genauer untersucht werden. Wann immer also im Folgenden von Interfaces die Rede ist, sind damit Interfaces als Typen gemeint.

Doch zunächst noch ein weiterer interessanter Punkt.

1.2.2 Nominale vs. strukturelle Typkonformität

Während in JAVA und C# ein Interface von einer Klasse nur dann implementiert wird, wenn die Klasse eben dies deklariert (die sog. **nominale Typkonformität**; s. Kurs 01814), ist es auch denkbar, dass sich die Implementierung aus syntaktischer Gleichheit von Methodendeklaration in Interface und Klasse automatisch ergibt (sog. strukturelle Typkonformität). Dies hat insbesondere den Vorteil, dass bei offenen Systemen, bei denen Systemteile dynamisch und von verschiedenen Quellen nachgeladen werden können, die Typkompatibilität nicht im Vorhinein deklariert sein muss. Eine Klasse kann (bzw. deren Instanzen können) so an den *Plug points* beliebiger (unbekannter) Frameworks eingesetzt werden. Dieser Vorteil ist aber auch ein Nachteil, nämlich dann, wenn sich eine strukturelle Typkonformität rein zufällig ergibt und Objekte dadurch für Aufgaben verwendet werden können, für die sie gar nicht gedacht/geeignet sind.

Um diesem Problem abzuhelpfen, müsste neben der syntaktischen Übereinstimmung der Schnittstellen (Gleichheit von Methodensignaturen) eigentlich auch eine semantische Übereinstimmung (Gleichheit von mit den Methoden verbundenem Verhalten) erzwungen werden. Das scheitert aber in der Praxis schon daran, dass zurzeit noch keine abstrakten (das heißt, von der konkreten Implementierung losgelösten) Formen der Verhaltensspezifikation zur Verfügung stehen, mit denen normale Programmiererinnen zurechtkämen. Und selbst wenn man eine solche Spezifikationsform verwendet (wie in Kurseinheit 2), so ist doch die automatische Überprüfung der Typkonformität (im Falle der Verwendung logischer Ausdrucksformen: die logische Äquivalenz) zweier Spezifikationen nur extrem rechenaufwendig nachzuweisen.

Im Folgenden gehen wir immer von nomineller Typkonformität aus, und zwar schon allein deswegen, weil die in den meisten Programmiersprachen so vorgesehen ist³. Man beachte, dass die nominale die strukturelle Typkonformität erzwingt.

1.2.3 Interfaces vs. abstrakte Klassen

Man kann darüber diskutieren, ob Interfaces-als-Typen zwingende Voraussetzung für die interfacebasierte Programmierung sind. Tatsächlich würde dies Programmiersprachen wie C++ von der interfacebasierten Programmierung ausschließen. Dies ist schon allein deswegen nicht sinnvoll, weil es in C++ abstrakte Klassen gibt, die — genau wie Interfaces in JAVA und C# — keine Implementierungen ihrer Methoden vorgeben müssen, und weil C++ Mehrfachvererbung erlaubt, eine Klasse also insbesondere mehrere abstrakte Klassen direkt erweitern kann (was dann ja der Implementierung mehrerer Interfaces entspräche). Der **einzigste Nachteil** ist, dass hier nicht auf Sprachebene unterschieden werden kann, ob eine abstrakte Klasse ohne jede Implementation die Funktion einer Superklasse (*Generalisierung*) oder eines Interfaces hat.

Dieser Einschränkung stand bis Java 1.8 jedoch ein **gewichtiger Nachteil von Interfaces à la JAVA** bei der praktischen Verwendung gegenüber: Wenn man sich in einem JAVA -Programm dazu entscheidet, ein Interface durch Aufnahme einer weiteren Methode zu erweitern (was im Rahmen der Weiterentwicklung regelmäßig vorkommt), dann läuft man Gefahr, dass die Typkorrektheit des Programms verlorengeht, nämlich dann, wenn nicht alle Klassen, die dieses Interface implementieren, auch über die zusätzliche Methode verfügen. Dies ist z. B. dann ein Problem, wenn das Interface zu einem Framework gehört und man selbst, als Entwicklerin des Frameworks, keine Kontrolle über dessen Verwendungen durch andere hat. Die Verwendung einer abstrakten Klasse anstelle des Interfaces hätte hingegen erlaubt, die neue Methode mit einer Default-Implementierung zu versehen, die von allen „implementierenden“ Klassen geerbt würde. **Diese Default-Implementierungen sind jetzt auch ab JAVA 1.8 in Interfaces möglich.**

³ Skriptsprachen wie Python, PHP, Groovy, Ruby sehen stattdessen das sogenannte „Duck Typing“ vor.

1.3 Eigenschaften von Interfaces

Mit Interfaces sind eine Reihe von Eigenschaften verbunden, die für die Untersuchung ihres Gebrauchs im Abschnitt 1.5 von Bedeutung sein werden. Diese Eigenschaften werden im Folgenden kurz behandelt.

1.3.1 Aufrufen und aufgerufen werden: die zwei Seiten eines Interfaces

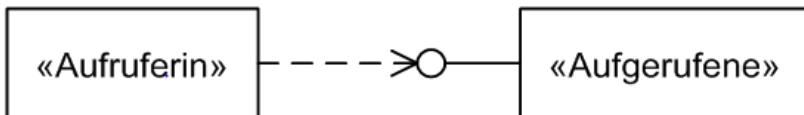


Abbildung 1.2: Aufruferin und Aufgerufene (engl. caller und called oder callee) eines Interfaces. Die Aufrufabhängigkeit wird hier (wie in UML) durch einen gestrichelten Pfeil dargestellt, die Implementierung eines Interfaces durch den Kreis mit Verbindungsleitung (sog. Lollipop-Notation; das Interface ist der Kreis).

Jedes Interface hat zwei Seiten: Es trennt die *Aufrufende* (oder *Aufruferin*) von der *Aufgerufenen*. Dabei sind Aufrufende und Aufgerufene zwei *Rollen* einer Beziehung, nämlich der Beziehung des Aufrufens (vgl. Abschnitt 1.8).

Da die Aufruferin von der Aufgerufenen programmiertechnisch abhängig ist, spricht man auch von einer *Abhängigkeitsrelation* (engl. dependency), genauer von einer **Aufrufabhängigkeit** (engl. call dependency); wie wir allerdings noch sehen werden, kann die mit der Aufrufabhängigkeit einhergehende inhaltliche Abhängigkeit durchaus auch umgekehrt bestehen, weswegen wir den Begriff nachfolgend vermeiden wollen. Ähnlich ist es mit der Bezeichnung *Client* für die Aufruferin und *Server* für die Aufgerufene: Auch hier kann das inhaltliche Verhältnis umgekehrt, also die Aufrufende der Server und die Aufgerufene der Client sein (s. Abschnitt 1.5.8).

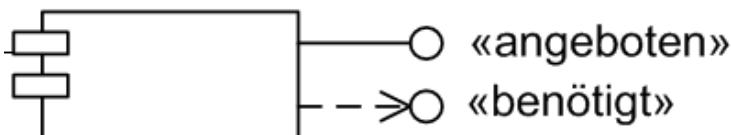


Abbildung 1.3: Angebotene (engl. provided) und benötigte (engl. required) Interfaces einer Komponente. Die Komponente ist einmal Anruferin und einmal Angerufene

Im Kontext der *komponentenbasierten Programmierung* spricht man häufig von *angebotenen* (engl. provided) und *benötigten* (engl. required) Interfaces (Abbildung 1.3).

Über angebotene Interfaces wird eine Klasse oder Komponente aufgerufen, bei benötigten Interfaces ist sie selbst die Aufruferin. Aus der Sicht der Komponente bezeichnen angebotenes und benötigtes Interface also zwei verschiedene Interfaces; allerdings ist das benötigte Interface einer Komponente in der Regel das angebotene einer anderen.

In einem objektorientierten Programm finden sich die beiden Seiten eines Interfaces auf verschiedene Weise wieder. Die Aufgerufene (genauer: die Klasse eines aufgerufenen Objektes) implementiert ein Interface und bietet somit alle Dienste an, die in dem Interface spezifiziert sind. Die Aufruferin (die ja schließlich das Objekt benennen muss, das sie aufrufen möchte) deklariert eine Variable mit dem Interface als Typ. Dazu muss sie (genauer: ihre Klasse⁴) das Interface zunächst importieren. Das Objekt, das durch die Variable referenziert wird, bietet dann alle Services des Interfaces an; auf diesem Objekt können die benötigten Funktionen aufgerufen werden. Das folgende Beispiel veranschaulicht den Sachverhalt.

Beispiel

```

32 interface Interface {
33     void aufruf();
34 }

35 class Aufgerufene implements Interface {
36     ...
37     public void aufruf() {...}
38 }

39 import Interface;
40 class Aufruferin {
41     Interface a = new Aufgerufene();
42     ...
43     a.aufruf();
44     ...
45 }
```

1.3.2 Totale und partielle Interfaces

Der klassische Interface-Begriff, wie er von Dijkstra, Parnas et al. geprägt wurde, umfasst alle öffentlichen, d. h. von außen zugänglichen Eigenschaften eines Moduls. Ein solches Interface nennen wir im Folgenden ein **totales Interface**. Bei der Verwendung totaler Interfaces wird nicht berücksichtigt, dass verschiedene Aufruferinnen unter Umständen verschiedene Dienste der Aufgerufenen benötigen — alle Aufruferinnen haben dieselbe Sicht auf die Aufgerufenen.

Häufig möchte man den Zugriff auf ein Objekt für verschiedene Aufruferinnen differenzieren. In solchen Fällen benötigt eine Aufruferin nur einen Teil der angebotenen Funktionen. Ein entsprechendes Interface, das nicht den vollen Funktionsumfang der Aufgerufenen abdeckt, nennen wir

⁴ Wenn wir im Folgenden von Aufruferinnen und Aufgerufenen sprechen, dann sind damit in der Regel Objekte gemeint. Da die Definition von Objekten aber in Sprachen wie Java und C# immer auf Typeebene, also der von Klassen und Interfaces-als-Typen, stattfindet, ergibt sich gelegentlich, dass der dazugehörige Typ (Klasse oder Interface) gemeint ist. Dies sollte jedoch immer aus dem Kontext heraus klar sein.

im Folgenden ein **partielles Interface**. Partielle Interfaces eines Objektes können disjunkt sein (sich nicht überlappen), müssen dies aber nicht.

Beispiel

Um den lesenden und den schreibenden Zugriff auf ein Objekt voneinander zu trennen, ist es beispielsweise sinnvoll, für dieses Objekt zwei Interfaces, Read und Write, einzuführen, die jeweils nur Getter bzw. nur Setter-Methoden enthalten. So können sich zum Beispiel zwei Objekte einen Puffer teilen, in den das eine nur schreiben und aus dem das andere nur lesen kann. Mit einem totalen Interface wäre eine solche Differenzierung nicht möglich.

Man beachte, dass ob ein Interface total oder partiell ist, nicht vom Interface alleine abhängt, sondern genauso von der das Interface implementierenden Klasse. Tatsächlich kann zum Beispiel ein Interface von mehreren Klassen implementiert werden, die einen unterschiedlichen Funktionsumfang haben, wobei es dann möglich ist, dass dasselbe Interface einmal ein partielle und einmal ein totales ist. Wenn wir also von einem partiellen oder einem totalen Interface sprechen, dann immer in Bezug auf eine bestimmte implementierende Klasse.

Partielle Interfaces enthalten also nicht den gesamten Funktionsumfang einer Klasse, eine Eigenschaft, die sie mit abstrakten Klassen teilen. Während aber abstrakte Klassen der *Generalisierung*, also der Verallgemeinerung mehrerer Klassen unter einer Superklasse dienen (vgl. Kurs 01814), *fokussieren* partielle Interfaces auf einen bestimmten Aspekt, also einen Teil der sie implementierenden Klassen, den sie dafür aber vollständig (ohne Abstraktion) darstellen. Partielle Interfaces sind also zumindest konzeptuell nicht gegen abstrakte Klassen auszutauschen, ein weiterer Punkt, der die Darstellung von Interfaces als Ersatz für die fehlende *Mehrfachvererbung* (Einleitung zu Abschnitt 1.2) als schwach erscheinen lässt. Dennoch kann es manchmal aus praktischen Überlegungen sinnvoll sein, abstrakte Klassen mit der Funktion eines Interfaces einzusetzen (s. Abschnitt 1.2.3).

Generalisierung vs. Fokussierung

Partielle Interfaces bedienen das sog. **Interface Segregation Principle** [9]. Es besagt, dass die Abhängigkeit einer Klasse von einer anderen auf die Teile des Interfaces beschränkt sein soll, die tatsächlich benötigt werden. Dadurch soll vermieden werden, dass alle Benutzerinnen einer Klasse über diese Klasse miteinander verkoppelt werden: Wenn eine Benutzerin eine Änderung des Interfaces erfordert, so die Befürchtung, müssen alle anderen Benutzerinnen entsprechend angepasst werden, auch wenn sie diese Änderung eigentlich gar nicht betrifft. Das Interface Segregation Principle liest sich wie folgt:

das Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use.

Interface Segregation Principle, Robert C. Martin, [9]

1.3.3 Öffentliche vs. veröffentlichte Interfaces

Die Implementierung eines Interfaces macht den durch das Interface abgedeckten Teil einer Klasse anderen Klassen, die das Interface verwenden, zugänglich. Unabhängig von separat definierten, implementierten Interfaces kann aber auch schon das *Klasseninterface*, das mittels der Zugriffsmodifizierer **public** etc. bei der Klassendefinition deklariert wird, einen solchen Zugriff ermöglichen. Beiden ist jedoch gemeinsam, dass die Gewährung des Zugriffs nicht dediziert erfolgen kann, dass also nicht angegeben werden kann, wer genau den Zugriff erhält. Aus Sicht der Klasse gibt es nur eine Öffentlichkeit⁵.

Berücksichtigung eines verteilten und asynchronen Softwareentwicklungsprozesses

Aus Prozesssicht ist es mit einer Öffentlichkeit jedoch nicht getan, wie folgende Überlegung zeigt. Klassen, die gemeinsam entwickelt werden, unterliegen häufig der gemeinsamen Änderung. Sollte dabei eine Änderung der Schnittstelle einer Klasse notwendig werden, z. B. weil eine andere Klasse einen zusätzlichen Dienst benötigt, weil er umbenannt oder mit anderen Parametern versehen werden soll oder weil ein Dienst gelöscht werden soll, so ist das kein Problem: Das Interface kann entsprechend angepasst werden, solange man nur Zugriff auf alle davon betroffenen Klientinnen hat. Ist das jedoch nicht der Fall ist, hat man ein ernsthaftes Problem.

Letzteres ist immer dann der Fall, wenn man für Wiederverwendung durch Dritte entwickelt, wenn also die betroffenen Klassen nicht aus einer Hand stammen. Die Öffentlichkeit besteht dann aus einer unbekannten Menge von Klassen, von denen man nichts weiß, außer dass sie mit den alten Schnittstellen funktionierten. Diese alten Schnittstellen zu ändern ist dann schlichtweg unmöglich (vgl. dazu die Ausführungen zu Interfaces vs. abstrakte Klassen in Abschnitt 1.2.3)⁶. Gleichwohl können die Schnittstellen von Klassen, die ausschließlich voneinander abhängen, immer noch geändert werden. Die Frage ist nur, wie man feststellen kann, ob diese Ausschließlichkeit besteht.

Differenzierung der Öffentlichkeit

Was man dazu benötigt, ist eine Differenzierung der Öffentlichkeit, und zwar in eine, unter die die Klassen fallen, die man selbst kontrolliert, und in eine, unter die die Klassen fallen, auf die man keinen Zugriff hat. Genau das ist mit der Unterscheidung von **öffentliche** und **veröffentlichtem Interface** (engl. **public** bzw. **published interface**) [8] gemeint: Während die öffentliche Schnittstelle im Wesentlichen dem entspricht, was man in JAVA und ähnlichen Sprachen mit dem Schlüsselwort **public** herstellt, deklariert die veröffentlichte Schnittstelle eine unveränderliche (die immer auch öffentlich ist, denn sonst hätte die Unveränderlichkeit kaum einen Nutzen). Ein spezielles Schlüsselwort gibt es dafür jedoch nicht⁷. Man ist aber frei, den Quellcode einer veröffentlichten Schnittstelle entsprechend zu kennzeichnen, z. B. mittels eines entsprechenden Kommentars oder einer *Annotation*.⁸

⁵ Dies ist in Eiffel — wie so vieles — anders: Hier wird grundsätzlich nicht importiert, sondern immer nur exportiert, und in einer Export-Klausel kann genau angegeben werden, an welche Klassen exportiert werden soll.

⁶ Damit verwandt ist die im Microsoft-Umfeld bekannte sog. *DLL hell*

⁷ Das liegt schon daran, dass die Entwicklungszeit außerhalb des Regelungsbereichs der Sprachdefinition liegt (sieht man einmal von den Möglichkeiten der Metaprogrammierung ab; s. Kurseinheit 6). So wird

Nun mag man einwenden, JAVA besäße schon ein Sprachmittel, um verschiedene Grade von Öffentlichkeit herzustellen. So gibt es dort das Paket-Konstrukt, das es erlaubt, den Zugriff auf Elemente des gleichen Pakets zu beschränken, also paketlokale Öffentlichkeit zu vereinbaren (mittels des Default access modifiers, der aus dem Weglassen der anderen besteht). Da Pakete (bzw. die damit verbundenen Sichtbarkeiten) aber nicht schachtelbar sind, eröffnet das einem lediglich die Möglichkeit, alle Klassen, die untereinander Öffentlichkeit benötigen, aber nicht veröffentlicht werden sollen, in jeweils ein Paket zu verfrachten und darin mit Standardsichtbarkeit (eben paketlokal) zu versehen. Aufgrund der Transitivität der Sichtbarkeitsbeziehung (eine erste Klasse, die eine zweite sieht, die eine dritte sieht, sieht auch die dritte) ergibt sich daraus nicht selten ein großes Paket, das alles enthält, so dass der zweite Zweck von Paketen, ein Programm zu strukturieren, der Veröffentlichung von Schnittstellen geopfert werden muss. Das gegenwärtige Paketkonzept bietet also keine praktikable Unterstützung zur Unterscheidung von öffentlichen und veröffentlichten Schnittstellen.⁹

JAVAs Paket-Konstrukt zur Abgrenzung nur beschränkt geeignet

Es ist also wichtig, dass man sich im Klaren darüber ist, welche der öffentlichen Schnittstellen eines Projekts man veröffentlichen will, und dass man dann darüber Klarheit herstellt. Dabei sollte man bei der Veröffentlichung von Schnittstellen eher zurückhaltend agieren, da diese schnell zu Altlästen werden können, die man nicht so leicht (im Extremfall gar nicht) wieder loswird, die sich bei der weiteren Entwicklung aber ziemlich einschränkend auswirken können.

1.4 Anzeichen interfacebasierter Programmierung

In der objektorientierten Programmierung rufen Objekte (Instanzen von Klassen) sich gegenseitig auf, um gemeinsam einen bestimmten Zweck zu erfüllen. Damit sich Objekte gegenseitig aufrufen können, müssen sie einander „kennen“. Dabei hat ein Objekt Kenntnis von einem anderen, wenn es eine Instanzvariable (ein Feld) besitzt, deren Inhalt eben dieses andere Objekt ist. Da (in Sprachen wie JAVA oder C#) die Variable das Objekt nicht selbst enthält, sondern lediglich einen Verweis (Pointer) darauf, spricht man auch davon, dass das andere Objekt referenziert wird. Alternativ kann ein Objekt (bzw. eine Methode dieses Objekts) auch vorübergehend von einem anderen Objekt Kenntnis erlangen, indem es (den Verweis auf) dieses andere Objekt als aktuellen Methodenparameter oder als Ergebnis eines Methodenaufrufs übergeben bekommt. In allen Fällen, also sowohl bei der Instanzvariable als auch beim formalen Parameter als auch beim Rückgabewert, ist der Verweis typisiert, so dass nur Objekte des verwendeten Typs und seiner Subtypen referenziert werden können. Im nicht interfacebasierten Programmieren ist dieser Typ

Quellcode in Java et al. immer noch in Dateien gespeichert, auf deren Änderung ein Compiler keinen Einfluß hat, so dass eine Überprüfung der Einhaltung der Unveränderlichkeit kaum stattfinden kann.

⁸ Ich schlage die Annotation @API dafür vor. Im Java SDK gilt übrigens alles als veröffentlicht, für das es einen Javadoc-Eintrag gibt. Man behält sich gleichwohl vor, mittels Deprecated-Annotierung langfristig angekündigte Änderungen vorzunehmen. In Eiffel wiederum kann man export {ANY} als eine Veröffentlichung ansehen.

⁹ Im Eclipse-Projekt gibt es eine Namenskonvention, die öffentliche, aber nicht veröffentlichte Schnittstellen kennzeichnet: Alle öffentlichen Elemente aus Paketen, die das Namenssegment `internal` enthalten, gehören nicht zum API und können von den Eclipse-Entwicklerinnen jederzeit und ohne weiteres geändert werden.

charakteristischerweise eine Klasse, im interfacebasierten ein Interface. Abbildung 1.4 gibt ein Beispiel für die nicht interfacebasierte Programmierung.

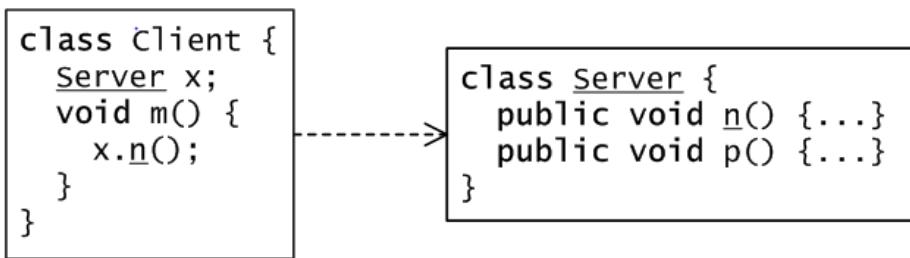


Abbildung 1.4: Die Klasse Client hängt unmittelbar von der Klasse Server ab, da die Variable x in Client als vom Typ Server deklariert ist. Zudem verwendet Client auch noch eine Methode, die in Server definiert ist. Der programmiersprachliche Niederschlag der Abhängigkeiten ist durch Unterstrichen hervorgehoben. S. a. Fußnote 10.

Man sieht hier, wie eine Klasse `client` von einer Klasse `server` abhängt¹⁰ (dargestellt durch den gestrichelten Pfeil), weil Instanzen von `client` eine Variable (namens `x`) haben, deren Inhalt ein Objekt vom Typ (der Klasse) `Server` ist. Wegen dieser Abhängigkeit spricht man von einer starken Kopplung zwischen `client` und `server`; der `client` kann nicht funktionieren, solange keine Klasse mit dem Namen „`Server`“ zur Verfügung steht. Solche starken Kopplungen sind häufig unerwünscht, weil sie die Flexibilität von Software einschränken. Auch wenn sie sich nicht immer vermeiden lassen (im Gegenteil — eine starke Kopplung ist oft natürlich und sie zu vermeiden wäre dann unsinnig; s. u.), geht es bei der interfacebasierten Programmierung vor allem darum, wie man sie umgeht.

Der klassische Weg, die Kopplung zwischen den beiden Klassen zu vermeiden, ist, bei der Klasse `Server` das Interface von der Implementierung zu trennen, den `client` vom Interface abhängig und damit die Implementierung austauschbar zu machen. In JAVA kann man dazu alle öffentlichen Methodendeklarationen in ein Interface übertragen, die `Server`-Klasse dieses Interface implementieren lassen und beim `client` die Variable `x` mit diesem Interface typisieren. Das Ergebnis ist Abbildung 1.5 zu entnehmen.

Die Klasse `Client` ist jetzt nur noch vom Interface `IServer` und damit von keiner konkreten Implementierung mehr abhängig. Die Implementierung des Interfaces, hier durch die Klasse `Server1`, kann im Rahmen der Wartung nach Belieben ausgetauscht werden — es ist aber auch möglich, dass in einem Programm mehrere Implementierungen gleichzeitig zur Verfügung stehen. Das Diagramm täuscht allerdings darüber hinweg, dass eine gewisse Restabhängigkeit von der Klasse des Server-Objekts bestehen bleibt: Das Server-Objekt muss nämlich irgendwann einmal erzeugt und dem Client bekannt gemacht werden. Wenn dies auf Seiten des Clients selbst erfolgt (der Konstruktorauftrag `new Server1()` also beim Client durchgeführt wird), dann besteht die Abhängigkeit des Clients vom Server natürlich trotz des Interfaces weiter. Dies lässt

¹⁰ In diesem Abschnitt spreche ich — entgegen vorheriger Ankündigung — doch noch einmal von Abhängigkeit, allerdings nur, um die Darstellung flüssiger formulieren zu können.

sich aber mit der sog. *Dependency injection* (Abschnitt 1.6) oder durch den Einsatz von *Factory-Methoden* (Abschnitt 4.4.6) vermeiden.

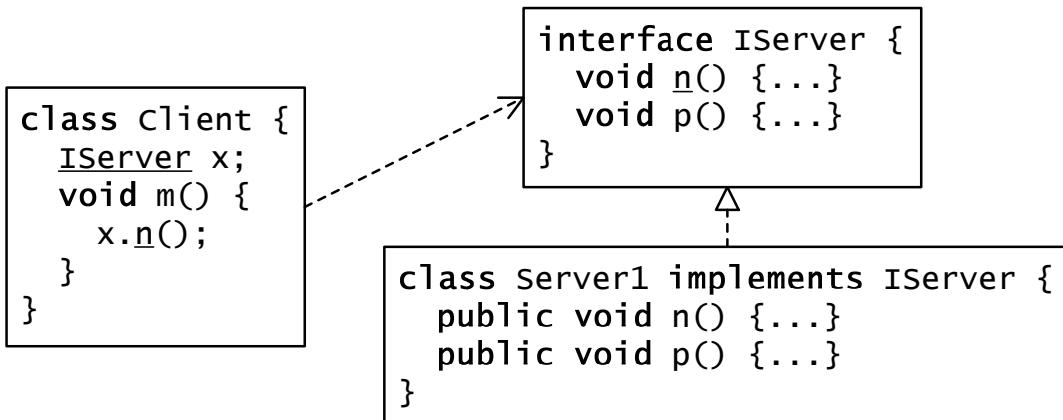


Abbildung 1.5: Entkopplung durch Verwendung eines Interfaces. Die Klasse Client ist jetzt direkt nur noch von Interface `IServer` abhängig. Die Klasse des Servers, hier in „Server1“ umbenannt, kann durch eine andere ersetzt werden, ohne dass Client davon betroffen wäre.

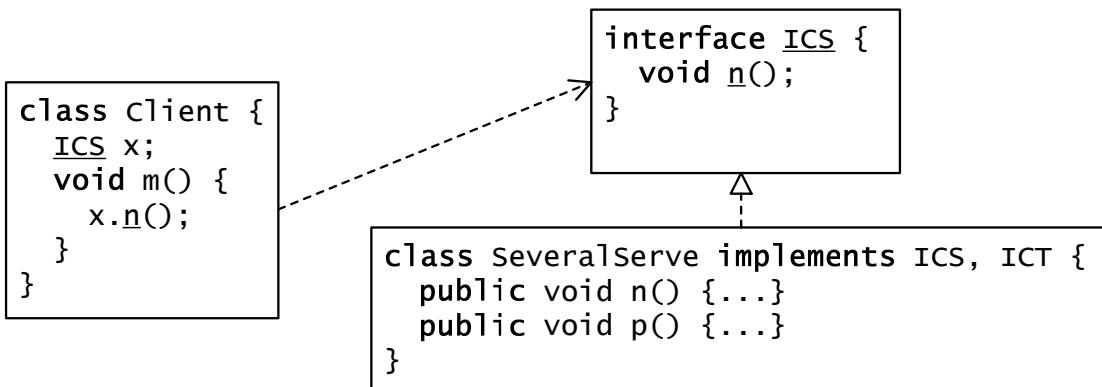


Abbildung 1.6: Zusätzliche Entkopplung des Clients von den Funktionen des Servers, die der Client nicht benötigt. Das Interface `ICS` (kurz für `InterfaceClientserver`) enthält jetzt nur noch die Methode `n()`; es kann deshalb leichter von mehreren Servern angeboten werden als das umfangreichere Interface `IServer`. Der Code wird dadurch flexibler. Die Klasse `SeveralServe` kann natürlich noch weitere Interfaces implementieren, die dann anderen Clients dienen (`ICT` oben).

Wie man dem Beispiel weiter entnehmen kann, sind die Anforderungen, die durch die Deklaration der Variable `x` als vom Typ `IServer` an die Server-Objekte gestellt werden, unnötig groß: Neben der Methode `n()`, die (aus der Methode `client.m()` heraus) tatsächlich auf `x` aufgerufen wird, müssen die Objekte auch noch eine Methode `p()` zur Verfügung stellen, obwohl diese (zumindest im Beispiel) von den Client-Objekten gar nicht benötigt wird. Die Abhängigkeit lässt sich also weiter reduzieren, indem man im Interface nur die Methoden aufführt, die von Clients tatsächlich benötigt werden (in Abbildung 1.6 dargestellt). Dadurch wird die Zahl der möglichen Server, die die Clients bedienen können, größer, denn sie müssen ja nun weniger Bedingungen erfüllen. Sollte die Methode `p()` von einem anderen Client benötigt werden, kann der Server dafür ein weiteres, separates Interface zur Verfügung stellen (`ICT` in Abbildung 1.6).

Anhand dieses Beispiels lässt sich erahnen, warum es als vorteilhaft angesehen wird, wenn Variablen mit Interfaces anstelle von Klassen als Typen deklariert werden. Allerdings, und das muss deutlich gesagt werden, ist es nicht immer sinnvoll, dies zu tun: Wie bereits oben erwähnt, ist die starke Kopplung zwischen Klassen häufig natürlich und sie aufzuheben wäre widersinnig. So ist es beispielsweise in der Regel nicht sinnvoll, anstelle des Klassentyps `String` ein entsprechendes Interface zu verwenden. Interfaces dienen in erster Linie der Entkopplung — sie sollen daher nur dort eingesetzt werden, wo eine Entkopplung auch vonnöten ist. Dies ist vor allem an Modulgrenzen, also an den Schnittstellen von größeren Programmeinheiten, der Fall. Das eingangs zitierte erste Prinzip wiederverwendbaren objektorientierten Designs, „program to an interface, not an implementation“, ist also nicht sklavisch überall anzuwenden.

Selbsttestaufgabe 1.2

Versuchen Sie, das Beispiel aus Abbildung 1.4 bis Abbildung 1.6 in einer Entwicklungsumgebung Ihrer Wahl, ggf. unter Zuhilfenahme von geeigneten *Refactorings*, nachzuvollziehen.

Anzeichen der interfacebasierten Programmierung

Das Vorliegen interfacebasierter Programmierung ist demnach hauptsächlich an zwei Anzeichen erkennbar:

1. Klassen implementieren Interfaces und
2. Variablen werden mit Interfaces als Typ deklariert.

Man kann sich also einfach ein Bild davon machen, ob (zu welchem Grad) interfacebasiert programmiert wird, indem man die Häufigkeit des Auftretens dieser Anzeichen zählt. Die Ergebnisse einer solchen Zählung für die jeweils häufigsten Interfaces in beiden Kategorien sind in den folgenden Tabellen wiedergegeben.

Tabelle 1.1: Die am häufigsten implementierten Interfaces im JDK 1.4

#	Name des Interfaces	# Implementierungen	# Referenzierungen
1	java.io.Serializable	1975	140
2	java.util.EventListener	584	92
3	java.lang.Cloneable	535	0
4	java.awt.event.ActionListener	235	79
5	javax.accessibility.Accessible	210	194
6	java.awt.image.ImageObserver	209	43
7	java.awt.MenuContainer	209	8
8	org.omg.CORBA.portable.IDLEntity	183	1
9	javax.swing.Action	178	175
10	java.security.PrivilegedAction	158	8

Tabelle 1.2: Die Interfaces, mit denen im JDK 1.4 die meisten Variablen deklariert wurden

#	Name des Interfaces	# Referenzierungen	# Implementierungen
1	org.w3c.dom.Node	715	65
2	javax.swing.text.Element	525	8
3	javax.swing.text.AttributeSet	486	20
4	java.util.Iterator	446	49
5	java.util Enumeration	423	36
6	javax.swing.Icon	379	65
7	org.omg.CORBA.Object	364	77
8	java.awt.Shape	316	33
9	java.util.Map	248	30
10	java.util.List	234	29
13	java.util.Set	197	29
18	java.util.Collection	158	69
...			
29	java.lang.CharSequence	95	14

Tabelle 1.3: Zum Vergleich die Klassen, mit denen im JDK 1.4 die meisten Variablen deklariert wurden

#	Name der Klasse	# Referenzierungen
1	java.lang.String	16143
2	java.lang.Object	5684
3	java.awt.Component	1610
4	java.lang.Class	1342
5	javax.swing.JComponent	1077
6	java.awt.Rectangle	1006
7	java.awt.Dimension	997
8	java.awt.Color	900
9	org.omg.CORBA.TypeCode	749
10	java.awt.Graphics	704
11	java.util.Vector	635

1.5 Arten des Gebrauchs von Interfaces

Syntaktisch gibt es nur ein Interfacekonstrukt. Allerdings ist dieses Konstrukt in seiner Verwendung nicht festgelegt — wie die folgenden Ausführungen zeigen, lässt es sich auf recht verschiedene Arten einsetzen. Dabei ist die Art des Gebrauchs zunächst nur aus dem Kontext heraus ersichtlich; es gibt aber bestimmte Merkmale wie die Benennung von Interfaces sowie die Anzahl der implementierenden Klassen oder der Variablen, die mit dem Interface als Typ deklariert wurden, die auf die Art des Gebrauchs schließen lassen (vgl. Abschnitt 1.3).

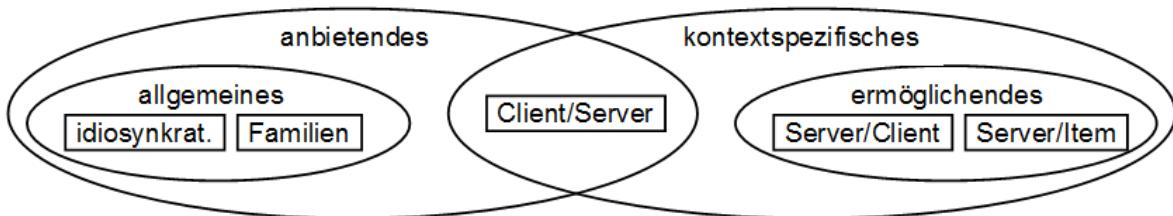


Abbildung 1.7: Klassifikation der verschiedenen Arten (des Gebrauchs von) von Interfaces: jedes fällt in eine der durch Rechtecke markierten Kategorien. Man beachte, dass anbietend und ermöglichen sowie allgemein und kontextspezifisch einander jeweils ausschließen.

Wenn wir die Interfaces eines Programms wie nachfolgend nach ihrem Gebrauch klassifizieren, dann unterstellen wir damit, dass ein gegebenes Interface in einem Programm nur auf eine Art gebraucht wird. Tatsächlich ist das in der Praxis aber längst nicht immer der Fall — vielmehr müsste man sich jede einzelne Variablen Deklaration bzw. Implements-Beziehung dazu ansehen. Das würde jedoch an dieser Stelle zu weit gehen. Wenn wir also im Folgenden von der Art des Gebrauchs eines Interfaces sprechen, dann ist damit immer der spezielle Gebrauch an einer Stelle oder der überwiegende gemeint.

1.5.1 Übersicht

Die nachfolgend hergeleitete Klassifizierung des Gebrauchs von Interfaces unterscheidet primär zwei Kriterien: Allgemeinheit und Nutzen. Bei der Allgemeinheit wird zwischen *allgemeinen* und *kontextspezifischen* Interfaces unterschieden, beim Nutzen zwischen *anbietenden* und *ermöglichen*. Beide Kriterien sind unabhängig voneinander. Von den resultierenden vier möglichen Kategorien sind jedoch nur drei besetzt: Es gibt nämlich keine allgemeinen ermöglichen Interfaces. Bei den übrigen drei Kategorien ergeben sich zum Teil noch weitere Differenzierungsmerkmale. Abbildung 1.7 bietet eine Übersicht; die dort enthaltenen neun Benennungen von Interfaces finden sich in den Überschriften der nachfolgenden Abschnitte wieder. Sie sollten jedoch im Sinn behalten, dass die Beschriftungen der Ovale lediglich Kriterien bezeichnen und die der Rechtecke die endgültigen Kategorien.

1.5.2 Anbietende Interfaces

Bei der Unterscheidung der Arten von Interfaces differenzieren wir zunächst danach, wer die Nutznießerin eines Aufrufs ist: die Aufruferin oder die Aufgerufene.

Gewöhnlich geht man davon aus, dass die Aufruferin sich über ein Interface an die Aufgerufene wendet, weil sie etwas von ihr will. Man könnte auch sagen: Die Aufruferin nimmt eine Dienstleistung der Aufgerufenen in Anspruch. Umgekehrt bietet die Aufgerufene über das Interface ihre Dienstleistung an. Das Interface (bzw. der damit verbundene *Vertrag*; vgl. Kurseinheit 2) hat somit den Charakter eines Dienstangebots; wir sprechen daher von einem **anbietenden Interface**.¹¹

1.5.3 Allgemeine Interfaces

Eine weitere mögliche Art der Differenzierung des Gebrauchs von Interfaces ergibt sich aus der Frage, ob ein Interface für spezielle Aufruferinnen entworfen wurde oder ganz allgemein zur Verfügung steht. Sog. **allgemeine Interfaces** enthalten in der Regel alles, was eine Aufgerufene anzubieten hat (es handelt sich also um totale Interfaces); wäre das nicht der Fall, müsste man davon ausgehen, dass es auch andere Verwendungen der Aufgerufenen gibt, die durch das Interface nicht abgedeckt sind, weswegen es wiederum nicht allgemein wäre. Allgemeine Interfaces sind also in der Regel *totale Interfaces* der Klassen, die sie implementieren. (Dies gilt in der Regel nicht für abgeleitete Klassen, die ihre Interfaces von den Klassen, von denen sie ableiten, erben, insbesondere dann nicht, wenn sie die geerbten Klassen erweitern.)

Aufgerufene haben in der Regel nicht mehrere allgemeine Interfaces. Es wäre schließlich wenig sinnvoll, mehrere allgemeine Interfaces, die ja alle denselben Funktionsumfang haben müssten, für verschiedene Aufruferinnen oder Zwecke vorzusehen. Tatsächlich ist der einzige Grund für die Existenz allgemeiner Interfaces, die Spezifikationen einer Aufgerufenen von ihrer Implementierung zu trennen, so dass letztere ausgetauscht werden kann, ohne dass die Aufruferinnen davon Notiz nehmen müssen (Abbildung 1.5).

In Abhängigkeit davon, ob alternative Implementierungen gleichzeitig oder lediglich im Zuge der Evolution (Wartung) eines Systems angeboten werden, unterscheiden wir bei allgemeinen Interfaces weiter zwischen idiosynkratischen und Familieninterfaces.

¹¹ Auch wenn dies das natürliche Verhältnis zwischen Aufruferin und Aufgerufener zu sein scheint, so wird dieses Verhältnis häufig (wie wir noch sehen werden) umgekehrt. In diesem Fall ist die Aufruferin diejenige, die einen Dienst erbringt, und die Aufgerufene die Dienstempfängerin. Da in der Regel ein solcher Aufruf der Aufgerufenen etwas ermöglicht, wozu sie sonst selbst nicht in der Lage wäre, sprechen wir dann von einem *ermöglichenen Interface* (s. u.). Wie bereits erwähnt unterscheiden sich anbietendes und ermöglichenes Interface syntaktisch nicht — die Differenzierung erfolgt ausschließlich auf inhaltlicher Basis, aus der Beantwortung der Frage, wer vom Aufruf profitiert.

1.5.4 Idiosynkratische Interfaces

Ein allgemeines Interface, das lediglich von einer konkreten (also nicht abstrakten) Klasse implementiert wird, wobei diese Implementierung über die Zeit geändert werden darf, aber nie zwei alternative Implementierungen gleichzeitig in einem Projekt existieren, nennen wir ein idiosynkratisches Interface. Die Situation ist in Abbildung 1.8 dargestellt. **Idiosynkratische Interfaces** tragen häufig die Namen der Klassen, die sie implementieren; die Klasse trägt dann einen Namenszusatz, der sie als Implementierung ausweist.

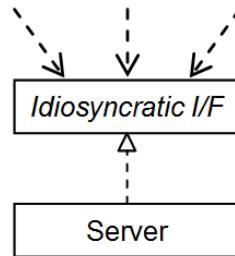


Abbildung 1.8: Ein idiosynkratisches Interface wird von genau einer Klasse des Programms implementiert.

Da es sich um ein *anbietendes Interface* handelt, ist diese Klasse immer Dienstanbieterin (Server) für eine oder mehrere andere. Das idiosynkratische Interface umfaßt alle öffentlichen Funktionen der Klasse; es kann daher die Klasse in allen Variablendeklarationen vollständig ersetzen. Dies ist die klassische Form eines Interfaces; sie wird aber in Programmiersprachen wie JAVA oder C# kaum verwendet, da dort das *Klasseninterface* den gleichen Zweck erfüllt.

Beispiel

Das idiosynkratische Interface der Klasse

```

46. public class StackImpl<E> {
47.     public boolean isEmpty() {...}
48.     public E peek() {...}
49.     public void pop() {...}
50.     public void push(E element) {...}
51. }
```

ist

```

52. interface Stack<E> {
53.     boolean isEmpty();
54.     E peek();
55.     void pop();
56.     void push(E element);
57. }
```

Alternativ wird auch gern dem Namen der Klasse ein „I“ vorangestellt und das Ergebnis als Name des Interfaces verwendet, im gegebenen Fall also etwa **IStack**. In beiden Fällen ergibt sich aus der Namenskonvention, dass das Interface speziell für die Klasse und unabhängig von einer konkreten Verwendung entworfen wurde.

Während idiosynkratische Interfaces dem ursprünglichen Sinn des Schnittstellenkonzepts entsprechen, wird man sie in aktuellen Sprachen wie JAVA oder C# kaum finden, da durch das *Klasseninterface* (spezifiziert durch die Verwendung des `public` Schlüsselworts in den Methodendefinitionen einer Klasse) bereits ein idiosynkratisches (eben das Klassen-) Interface zur Verfügung steht. Die zusätzliche Bereitstellung eines separaten Interfaces wäre in der Tat sinnlos, wollte man nicht mehrere alternative Implementierungen innerhalb eines Projektes vorsehen. Das Interface wäre damit aber nicht mehr idiosynkratisch. Wir können daher nicht erwarten, viele idiosynkratische Interfaces in JAVA- oder C#-Programmen zu finden. Dies steht im Gegensatz zu vielen Lehrbüchern, in denen man öfter beispielhaft Interfaces wie `IDog` oder `IPerson` findet, deren Natur idiosynkratisch ist.

1.5.5 Familieninterfaces

Wenn dagegen ein *allgemeines Interface* von mehr als einer Klasse gleichzeitig (d.h. innerhalb desselben Projekts) implementiert wird, nennen wir es ein Familieninterface (Abbildung 1.9). Die verschiedenen Klassen bieten häufig alternative Implementierungen, die verschiedene technische Eigenschaften aufweisen; gleichwohl erfüllt jede dieselbe Interfacespezifikation. Im Gegensatz zu idiosynkratischen Interfaces, deren Implementierung lediglich zur Entwurfs-(Programmier-)Zeit geändert werden kann, erlauben Familieninterfaces die Auswahl zwischen Implementierungsalternativen zur Laufzeit, weswegen sie auch häufig gemeinsam mit sog. *Factories* (siehe Abschnitt 4.4.6) auftreten.

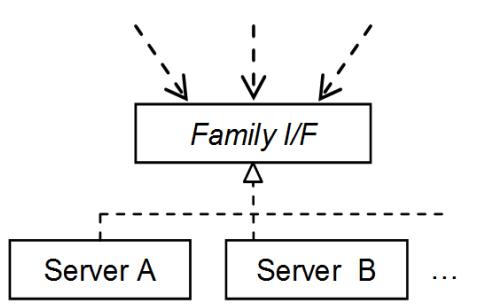


Abbildung 1.9: Ein Familieninterface ist gleichzeitig (*totales*) Interface mehrerer Klassen, die alle dasselbe Protokoll anbieten. Klientinnen können so mit unterschiedlichen Dienstanbietern arbeiten, ohne dass sie etwas davon wissen müssen.

Weil Familieninterfaces allgemein und damit kontextunabhängig sind, spezifizieren sie die Natur der aufgerufenen Objekte und nicht ihre *Rolle* in einem bestimmten Kontext. Familieninterfaces tragen daher oft typische Klassennamen (wie z. B. „Number“ oder „Interval“) und könnten genauso gut durch abstrakte Klassen ersetzt werden. Tatsächlich werden in C# und JAVA, denen beiden die *Mehrfachvererbung* fehlt, Familieninterfaces anstelle von abstrakten Klassen eingesetzt, wenn die sie implementierenden Klassen zugleich von anderen Klassen erben können sollen.

1.5.6 Kontextspezifische Interfaces

Interfaces, die speziell für bestimmte Verwendungen von Objekten einer Klasse entworfen wurden und die deswegen nicht das gesamte *Klasseninterface* abdecken, die also nicht allgemein

(und nicht *total*, sondern *partiel*) sind, nennen wir **kontextspezifisch**. Ein kontextspezifisches Interface fasst all die Eigenschaften einer Aufgerufenen zusammen, die von einer oder mehreren Aufruferinnen aus einem bestimmten Kontext heraus benötigt werden. Dabei ist ein kontextspezifisches Interface in der Praxis nicht einer speziellen Aufruferin zugeordnet — diese Zuordnung, die bedeuten würde, dass ein Interface-als-Typ nur bestimmten Klassen zugänglich gemacht würde, ist in Sprachen wie JAVA oder C# auch gar nicht vorgesehen —; vielmehr ist es die Funktion (oder *Rolle*; vgl. Abschnitt 1.8), die die Aufgerufene in dem gegebenen Kontext spielt, die darüber entscheidet, was in das Interface gehört. Es ist somit möglich, dass dieselbe Aufruferin dieselbe Aufgerufene über verschiedene kontextspezifische Interfaces anspricht, nämlich genau dann, wenn sie dies aus verschiedenen Kontexten heraus tut.

Beispiel

Eine Klasse **RingBuffer** kann je nach Kontext nur lesend oder nur schreibend eingesetzt werden. Diese Kontexte werden beispielsweise durch die Interfaces **ReadStream** und **WriteStream** bedient:

```
58. interface ReadStream<T> {
59.     T read();
60. }

61 interface WriteStream<T> {
62     void write(T object);
63 }
```

Wenn die Klasse **RingBuffer** nun beide Interfaces implementiert, kann ein und dieselbe Instanz der Klasse in einem Kontext nur lesend und in einem anderen nur schreibend eingesetzt werden.

In C# lässt sich der kontextspezifische Zugriff auf Objekte per Verwendung von Interfaces in Variablen Deklarationen erzwingen, indem man die Methoden **T read()** und **void write(T)** in **RingBuffer** als die in Abschnitt 1.2.1 beschriebenen *expliziten Interfaceimplementierungen* deklariert. Dazu muss dem Methodennamen bei der Deklaration in der Klasse der jeweilige Interfacename vorangestellt werden — ein Zugriff über mit der Klasse typisierter Variablen ist dann nicht mehr möglich.

1.5.7 Client/Server-Interfaces

Bei der offensichtlichen Form eines kontextspezifischen Interfaces ist die Aufgerufene die Anbieterin und die Aufruferin die Profiteurin. Es handelt sich also zugleich um ein *anbietendes Interface*. Da die Aufgerufene in diesem Fall als Server (Dienstleisterin) für bestimmte Clients fungiert, nennen wir ein solches Interface ein **Client/Server-Interface** (Abbildung 1.10).

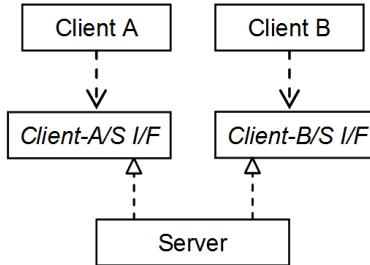


Abbildung 1.10: Ein Client/Server-Interface ist ein *kontextspezifisches, anbietendes Interface*, das speziell für eine oder mehrere Klientinnen (bzw. Kontexte, in denen diese Klientinnen den Server benötigen) entworfen wurde. Auf ein Serverobjekt kann vom gleichen oder von verschiedenen Klientinnen über verschiedene Client/Server-Interfaces gleichzeitig zugegriffen werden.

Client/Server-Interfaces wird man vorzugsweise in geschlossenen Applikationen antreffen, wo das Interface speziell für das Zusammenspiel von Client und Server entworfen wurde. Es ist aber auch denkbar, dass ein solches Interface in einer Bibliothek oder in einem Framework für Klientinnen zur Verfügung gestellt wird, die noch gar nicht (alle) existieren. So sind beispielsweise die Interfaces `ReadStream` und `WriteStream` der Klasse `RingBuffer` aus obigem Beispiel recht allgemein und nicht für eine bestimmte Klientin von `RingBuffer` (wohl aber für bestimmte, eben entweder nur lesende oder nur schreibende, nicht aber für alle Kontexte, die durch die Klientinnen hergestellt werden) entworfen.

1.5.8 Ermöglichende Interfaces

Bei den in den vorangegangenen Abbildungen vorgestellten Arten von Interfaces (idiosynkratisches, Familien- und Client/Server-Interface) handelt es sich durchweg um anbietende Interfaces: Die Aufgerufene stellt ihre Dienste zur Verfügung und welches diese Dienste sind, wird in dem Interface festgehalten. Es gibt aber auch den umgekehrten Fall, nämlich dass ein Interface den Zweck hat, der Aufgerufenen einen Service zuteilwerden zu lassen, den sie selbst nicht leisten (allenfalls unterstützen) kann. In einem solchen Fall ist das Nutzenverhältnis umgekehrt: Die Aufruferin ermöglicht der Aufgerufenen etwas, weswegen wir solche Interfaces **ermöglichende Interfaces** nennen. Konkrete Beispiele für ermöglichende Interfaces aus der JAVA -API sind `Runnable` und `Comparable`; beide unterscheiden sich jedoch in einem wichtigen Detail, weswegen wir die Kategorie der ermöglichenen Interfaces noch weiter unterteilen müssen. Vorab sei jedoch schon vermerkt, dass ermöglichende Interfaces im Englischen häufig auf „able“ oder „ible“ enden, was schon ausdrückt, dass mit den Aufgerufenen (die ja den Typ des Interfaces haben) etwas gemacht werden kann, ihre Rolle also eine passive ist. Wie bereits in Abschnitt 1.5.1 erwähnt, sind ermöglichende Interfaces immer kontextspezifisch (pathologische Ausnahmen bestätigen die Regel).

Ermöglichende Interfaces findet man häufig in Frameworks, bei denen eine *Umkehrung der Ausführungskontrolle* (engl. *inversion of control*) stattfindet: Anstatt wie bei der Benutzung von Programmbibliotheken üblich selbst die Ausführungssteuerung des Programms in die Hand zu nehmen und andere Klassen bei Bedarf aufzurufen, werden die Anwendungsklassen bei der Verwendung eines Frameworks in das Framework eingeklinkt (über sog. *Hooks* oder *Plug points*;

s. Abschnitt 4.4.3) und durch das Framework aufgerufen. Man nennt dies gelegentlich auch das *Hollywood-Prinzip*, wegen der dort üblichen Ansage „don't call us, we call you“. Allerdings finden ermögliche Interfaces vornehmlich bei sog. *Black-Box-Frameworks* Verwendung, bei denen das Aufrufen über Komposition und Delegation bzw. Forwarding erfolgt; die besser bekannten (und weiter verbreiteten) *White-Box-Frameworks* werden im Gegensatz dazu über *Vererbung* und *offene Rekursion* erweitert (s. a. Abschnitt 4.2.1).

1.5.9 Server/Client-Interfaces

Wir kommen nun zu den beiden zu unterscheidenden Fällen. Im ersten Fall ist die Aufgerufene selbst die Nutznießerin der Aufruferin und die Aufruferin die Erbringerin der Dienstleistung; wir nennen daher ein solches Interface ein **Server/Client-Interface** (Abbildung 1.11).

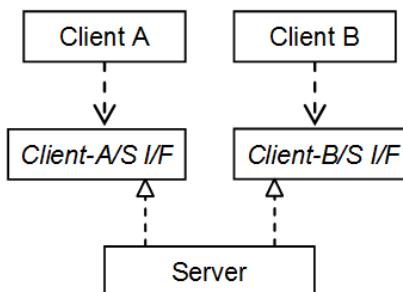


Abbildung 1.11: Bei einem Server/Client-Interface ist das Verhältnis von Aufruferin zu Aufgerufener umgekehrt: Die Aufruferin bietet einen Dienst, von dem die Aufgerufene profitiert. Man findet solche Interfaces häufig in Frameworks, in denen die Umkehrung der Ausführungskontrolle (engl. inversion of control) das dominierende Prinzip ist. Das Interface spezifiziert dann die Anforderungen eines Plug points.

Beispiel: Runnable

Ein typisches **Beispiel** für ein Server/Client-Interface ist `java.lang.Runnable`: hier ist die Klasse `Thread` der Server, der es einem Client ermöglicht, Methoden in einem eigenen Thread laufen zu lassen. Der Client muss dazu lediglich die Methode `void run()` implementieren, die der Server aufruft; das Vorhandensein dieser Methode wird über die deklarierte Implementierung des Interfaces `Runnable` sichergestellt:

```

64 public interface Runnable {
65     void run();
66 }
67 class Client implements Runnable {
68     ...
69     public void run() {...}
70 }
71 Client client = new Client();
72 new Thread(client).start();
  
```

Man beachte, dass es sich hier um ein Beispiel eines (gewissermaßen minimalen, da aus nur einer Klasse bestehenden) **Black-Box-Frameworks** mit *Umkehrung der Ausführungskontrolle* handelt: Die Frameworkklasse `Thread` ruft als Server (in Reaktion auf den Aufruf von `start()` von woher auch immer) die Methode `run()` des Clients auf, der dem Server (`Thread`) dazu als Parameter übergeben wird und der zu diesem Zweck das Server/Client-Interface `Runnable` implementiert. Zugleich bietet `Thread` auch **White-Box-Frameworkfunktionalität** an: Wenn eine (Client-) Klasse von `Thread` ableitet, wird die Methode `start()` von `Thread` geerbt und ruft, per *offener Rekursion*, die Methode `run()` auf, die dazu in der Klasse überschrieben wird. Ein spezielles Interface (außer dem *impliziten Vererbungsinterface*) ist dazu nicht notwendig (s. Abschnitt 4.2.1).

Die symmetrische Benennung der Verwendungsarten Server/Client- und Client/Server-Interfaces legt bereits nahe, dass sie paarweise auftreten können. So etwas kommt zum Beispiel vor, wenn in einer Client/Server-Konstellation der Server für die Ausführung seiner Dienste Information benötigt, die ihm nicht beim Aufruf übergeben wurde. In solchen Fällen wird der Server zur gegebenen Zeit beim Client rückfragen, um diese Information zu erhalten. Es ist dazu allerdings notwendig, dass der Server den Client kennt — in der Regel wird das dadurch erreicht, dass der Client sich selbst (in JAVA durch die Variable `this` repräsentiert) beim Aufruf an den Server übergibt¹². Je nachdem, wie langfristig die Beziehung zwischen Client und Server ist, kann der Server auch einen Verweis auf den Client (ggf. auf alle seine Clients) halten, was einer Registrierung des (der) Clients beim Server entspricht.

paariges Auftreten von Server/Client- und Client/Server-Interfaces

Beispiel

```
73 interface ClientServer<T> {  
74     void serve(ServerClient<T> caller);  
75 }  
  
76 interface ServerClient<T> {  
77     T support();  
78 }  
  
79 class Server<T> implements ClientServer<T> {  
80     ...  
81     public void serve(ServerClient<T> caller) {  
82         ...  
83         T additionalInfo = caller.support();  
84         ...  
85     }  
86 }  
  
87 class Client<T> implements ServerClient<T> {  
88     ...
```

¹² In Sprachen wie JAVA ist der Absender einer Nachricht dem Empfänger sonst nicht bekannt.

```

89   public T support() { ... }

90   ...
91   (new Server<T>()).serve(this);
92   ...
93 }
```

Ein weiteres Beispiel für paarig auftretende Client/Server-Server/Client-Interfaces ist die asynchrone Kommunikation, bei der das Ergebnis (der Rückgabewert) eines Methodenaufrufs durch einen speziellen Rückruf (engl. *callback*¹³) übergeben werden muss.

Obwohl Server/Client-Interfaces häufig mit Client/Server-Interfaces gepaart vorkommen, gibt es auch zahlreiche sinnvolle Verwendungen für erstere allein. Ein neben dem oben bereits diskutierten **Runnable** weiteres gutes Beispiel ist der **Event-Listener-Mechanismus**, der in JAVA besonders in den (Framework-)Klassen zum grafischen Benutzer-Interface (AWT und SWING) sattsam Verwendung findet (auch als *OBSERVER Pattern* bekannt; s. Abschnitt 4.4.2). Hierbei registrieren sich all die Objekte bei einem Server-Objekt, die von dessen *Zustandsänderungen* in Kenntnis gesetzt werden wollen¹⁴. Die zu informierenden Objekte (Clients) implementieren dazu ein Listener-Interface, über das sie vom Server aufgerufen werden können. Gleichzeitig bezeichnet dieses Interface den Typ der Objekte, die der Server zur Notifikation registrieren kann. Nur wenn die Clients vom Server zusätzliche Informationen benötigen (die bei der Notifikation nicht übergeben wurden), wird auch ein Client/Server-Interface benötigt¹⁵.

1.5.10 Server/Item-Interfaces

Bei der zweiten Art ermöglichernder Interfaces ist nicht die Aufgerufene selbst die unmittelbare Nutznießerin des Dienstes der Aufrufenden, sondern eine Dritte, die zu der Aufgerufenen in Beziehung steht. Ein typisches Beispiel ist das Interface **Comparable**, mit Hilfe dessen zwei Objekte miteinander verglichen werden können. Nutznießerin des Vergleichs, der von einem Server (unter Zuhilfenahme der Aufgerufenen) vorgenommen wird, ist aber nicht die Aufgerufene selbst (das vergleichbare Objekt — was hätte das von dem Vergleich?), sondern eine Dritte, die die Aufgerufene verglichen haben möchte. Dieses dritte Objekt ist der eigentliche Client des Servers; die Aufgerufene ist lediglich ein Element oder eine Position (engl. item) des Clients. Wir nennen solche Interfaces deswegen **Server/Item-Interfaces** (Abbildung 1.12).

¹³ Genaugenommen bezeichnet Callback eine Funktion, die als Parameter an eine andere, aufgerufene übergeben wird und die diese dann aufruft. In der objektorientierten Programmierung reicht dafür jedoch häufig (ein Pointer auf) das Objekt, das die Methode zur Verfügung stellt (nämlich wenn der Name und die Signatur der Methode bekannt sind). C# und das Common Type System von .NET stellen dafür sog. *Delegates*, das sind Methodenpointerklassen, zur Verfügung.

¹⁴ Dieser Mechanismus wird in Abschnitt 4.4.2 noch ausführlicher behandelt.

¹⁵ In C# kämen hier wieder *Delegates* zum Einsatz.

Beispiel

```

94 interface Comparable {
95     int compareTo(Object o);
96 }

97 class Item implements Comparable {
98     ...
99     public int compareTo(Object o) {...}
100 }

101 class Server {
102     ...
103     void sort(List<? extends Comparable> items) {
104     ...
105     if (a.compareTo(b) > 0) {...}
106     ...
107 }
108 }

109 class Client {

110     Server server;
111     List<Comparable> items;
112     ...
113     server.sort(items);
114     ...
115 }

```

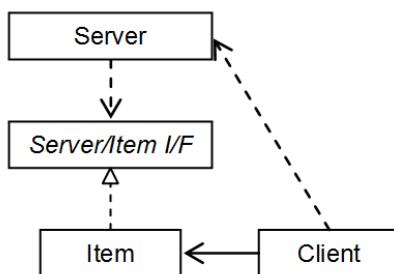


Abbildung 1.12: Bei einem Server/Item-Interface hält der Client Objekte (engl. items), die ein Server für ihn bearbeiten soll. Damit die Bearbeitung unabhängig von der Art der Objekte durchgeführt werden kann, müssen diese das Server/Item-Interface implementieren. Der Server ruft dann die Items über das Server/Item-Interface auf — Nutznießer ist aber der Client (und nicht die Items).

Ein anderes Beispiel für einen Server/Item-Interface ist **Printable**. Ein Client übergibt an einen Server (**Printer**) ein Objekt, das der Client gedruckt haben möchte. Dazu muss dieses Objekt das Interface **Printable** implementieren, so dass der Server dem Objekt die zum Drucken notwendigen Informationen entnehmen kann.

Wenn man im Kontext eines Server/Item-Interfaces den Client auch noch vom Server abkoppeln möchte, dann kann man zusätzlich noch ein Client/Server-Interface einbauen, wie dies in Abbildung 1.13 zu sehen ist. Dieses Client/Server-Interface ist dann u. U. selbst vom Server/Item-Interface abhängig, da die an den Server übergebenen Objekte letzteres implementieren müssen, was der Server durch einen entsprechenden Parametertypen verlangen kann.

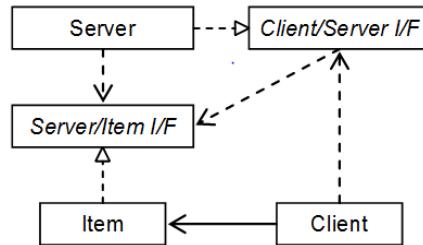


Abbildung 1.13: Zusätzliche Entkopplung des Clients vom Server durch ein Client/Server-Interface. Da der Client seine Items an den Server übergeben muss, damit dieser seine Dienste darauf verrichten kann, ist das Client/Server-Interface vom Server/Item-Interface abhängig. Diese Abhängigkeit äußert sich in der Verwendung des Server/Item-Interfaces bei der Deklaration eines oder mehrerer formaler Parameter der Methoden im Interface des Servers.

Tagging- oder Marker-Interfaces

Eine besondere Form der Server/Item-Interfaces sind übrigens die sog. **Tagging-** oder **Marker-Interfaces**. Marker-Interfaces sind oft leer; ihre einzige Funktion besteht darin, Objekte mit einem zusätzlichen Typ zu versehen, der zur Übersetzungszeit vom Compiler beziehungsweise zur Laufzeit durch eine Typabfrage geprüft werden kann. Ein Objekt erfährt dann eine spezielle Behandlung, wenn es (bzw. seine Klasse) das Interface implementiert.

Das wohl bekannteste Tagging-Interface ist **java.io.Serializable**: Es enthält keinerlei Funktionen, die von den implementierenden Klassen anzubieten wären. Stattdessen erlaubt es dem Serialisierungsframework von JAVA, zur Laufzeit (mittels des Operators `instanceof`) für ein Objekt zu bestimmen, ob seine Klasse als serialisierbar gekennzeichnet („getagt“) wurde. In C# ist **Serializable** übrigens ein Attribut (s. Abschnitt 6.2).

1.5.11 Gebrauch von Interfaces in der Praxis

Anhand der vorangegangenen Darstellung ergibt sich die in Abbildung 1.7 wiedergegebene Klassifikation der verschiedenen Arten von Interfaces. Jeder Gebrauch eines Interfaces fällt in genau eine durch ein Rechteck gekennzeichnete Kategorie. In der Praxis ergibt sich jedoch, insbesondere durch die Kombination mit *Subclassing*, dass ein Interface in mehrere Kategorien fallen kann: So kann beispielsweise ein Familieninterface auch ein kontextspezifisches sein, nämlich wenn einzelne Klassen der Familie Subklassen haben, die mehr Methoden veröffentlichen, als dies durch das Familieninterface vorgegeben wäre, so dass deren Verwendung als Mitglied der Familie durch bestimmte Kontexte vorgegeben ist (und in anderen Kontexten anders ausfällt).

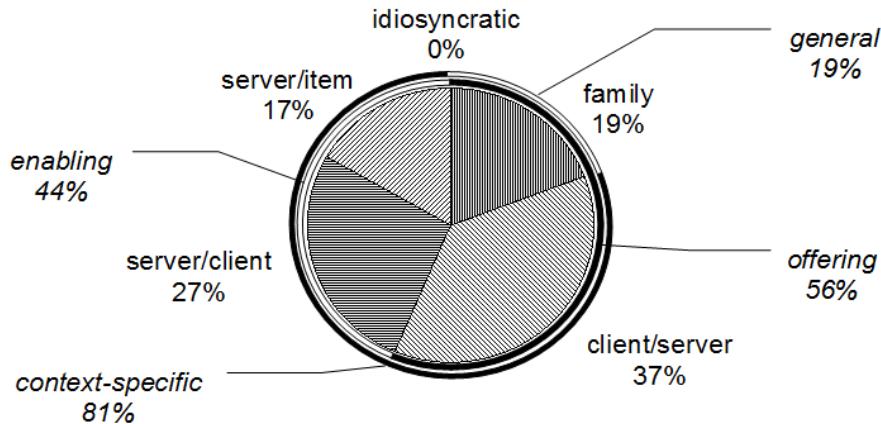


Abbildung 1.14: Relative Verteilung der verschiedenen Arten (des Gebrauchs) von Interfaces, ermittelt anhand der 100 am häufigsten implementierten und 100 am häufigsten referenzierten Interfaces (Überlappung von 43 Interfaces) innerhalb des JDK 1.4.

Man könnte nun meinen, dass bestimmte Arten der Verwendung von Interfaces häufiger vorkommen als andere. Dass sich das nicht so allgemein sagen lässt, zeigt Abbildung 1.14, die eine Stichprobe von Interfaces aus dem JDK auswertet. Nun ist es allerdings so, dass das JDK vieles zugleich ist: eine Sammlung von allgemeinen Klassen (vergleichbar mit einer gewöhnlichen Bibliothek), eine Sammlung von Frameworks (AWT, SWING, etc.) sowie die Implementierung verschiedener Middleware-Standards (CORBA etc.). In Bibliotheken sollten anbietende Interfaces dominieren (da hier insbesondere keine *Plug points* o. *Hooks* vorgesehen sind; vgl. Abschnitt 1.5.8); in einem Framework wiederum wird man vermehrt ermöglichte Interfaces vorfinden.

1.6 Dependency injection

Die konsequente Verwendung von Interfaces in Variablen- und Methodendeklarationen erlaubt es, die Anzahl der Referenzierungen anderer Klassen und damit die Abhängigkeit von diesen (bzw. die damit verbundene Kopplung) zu verringern. Es bleibt jedoch die Abhängigkeit von einer Klasse, die durch den Aufruf des Konstruktors zum Zwecke der Erzeugung einer Instanz dieser Klasse entsteht. Diese lässt sich durch die sog. **Dependency injection** eliminieren.

Konstrukturaufruf erzeugt Abhängigkeit

Sehr häufig findet man in Klassen, die sich Instanzen einer anderen Klasse als Server halten, Code der Art

```
116 Server server = new Server();
```

Dabei nützt es dann nichts, wenn man den Typ der Variable in ein geeignetes Client/Server-Interface, also etwa wie in

```
117 IServer server = new Server();
```

abändert, denn dann wird ja immer noch die Klasse direkt referenziert. Es ist auch der Konstruktorauf调用 zu entfernen.

Die Lösung der Dependency injection besteht nun darin, die benötigte Instanz nicht von dem von der Abhängigkeit zu befreien Objekt selbst erzeugen zu lassen, sondern sie von außen in dieses hineinzubringen, eben zu injizieren. Wenn die Instanzvariable nicht direkt von außen zugänglich ist (was in der Regel der Fall sein wird), dann geht das über eine temporäre Variable, einen formalen Parameter, der dann natürlich nicht den Typ der Klasse hat (denn sonst würde ja nur die eine Abhängigkeit durch eine andere ersetzt), sondern den des Interfaces.

unglückliche Namenswahl

Die Bezeichnung Dependency injection ist nicht unumstritten. Tatsächlich wird eigentlich keine Abhängigkeit injiziert (die dann ja nach der Injektion bestehen müsste), sondern ein Objekt. Zwar ist die Klientin von diesem Objekt abhängig, aber diese Abhängigkeit wurde nicht entfernt (und sollte auch gar nicht entfernt werden) — die Variable, die auf das Objekt verweist, besteht ja weiter. Entfernt werden sollte vielmehr die Abhängigkeit von der Klasse des Objekts. Dies geschieht auch tatsächlich mittels Dependency injection, die Abhängigkeit wird aber dadurch auch nicht injiziert (denn die Klasse des Objekts bleibt dem Client weiterhin unbekannt). An der Bezeichnung Dependency injection soll hier aber festgehalten werden, selbst wenn sie falsche Assoziationen auslöst — sie ist einigermaßen etabliert und das alternative Inversion of control (zu Deutsch: *Umkehrung der Ausführungskontrolle*), das sich eigentlich auf den Kontrollfluss im Kontext von Frameworks bezieht, ist noch weniger passend.

In Abhängigkeit davon, wie das Objekt injiziert wird, unterscheidet man verschiedene Arten von Dependency injection. Die gebräuchlichsten nennen sich Constructor injection, Setter injection und Interface injection.

1.6.1 Constructor injection

Bei der Constructor injection wird das Objekt, zu dem eine Abhängigkeit aufgebaut werden soll, dem abhängigen beim Konstruktorauf调用 übergeben:

```
118 class Client {
119     IServer server;
120     Client(IServer aServer) {
121         server = aServer;
122         ...
123     }
124     ...
125 }
```

Das ist in aller Regel nicht nur ausreichend, sondern sogar die eleganste und sicherste Lösung. Insbesondere wird so sichergestellt, dass die Initialisierung des Clients, bei der die Verknüpfung (die Konfiguration) hergestellt wird, nicht vergessen werden kann.

Der offensichtliche Nachteil der Constructor injection ist, dass der Konstruktorauf调用 pro Abhängigkeit, die injiziert werden soll, um ein Argument länger wird. Wenn zudem schon so mehrere

alternative Konstruktoren zur Verfügung gestellt werden sollen und sich durch die zusätzliche Dependency injection auch noch mehrere Varianten ergeben, kann die Zahl der benötigten Konstruktoren exponentiell ansteigen. In diesem Fall ist die sog. Setter injection vorzuziehen.

1.6.2 Setter injection

Bei der Setter injection wird die Variable, die die Abhängigkeit darstellt, separat, eben durch einen Setter, mit einem Objekt versorgt (die Injektion). Der entsprechende client-seitige Code sieht in etwa so aus:

```
126 class Client {  
127     IServer server;  
128     void setServer(IServer aServer) {  
129         server = aServer;  
130     }  
131     ...  
132 }
```

Der offensichtliche Nachteil ist, dass der Aufruf vergessen werden kann (was dann zu einer Nullpointer-Exception führen kann). Er kann aber auch, anders als bei der Constructor injection, mehrfach ausgeführt werden, wodurch sich die Abhängigkeit im Laufe der Lebenszeit des Clients ändern kann; die Zahl der Fälle, in denen das sinnvoll ist, hält sich aber in Grenzen (s. u.).

1.6.3 Interface injection

Bei der Interface injection schließlich wird die Methode, mittels derer die Abhängigkeit injiziert werden soll, durch ein entsprechendes Interface vorgeschrieben, das die Client-Klasse implementieren muss. Dies kann eine Setter-Methode wie bei der Setter injection sein, die Methode kann aber auch beliebig anders heißen. In der Regel wird sie jedoch nicht viel mehr tun, als eben diese Abhängigkeit durch die Zuweisung ihres formalen Parameters an das jeweilige Feld herzustellen:

```
133 interface ServerInjected {  
134     void injectServer(IServer aServer);  
135 }  
  
136 class Client implements ServerInjected {  
137     IServer server;  
138     void injectServer(IServer aServer) {  
139         server = aServer;  
140     }  
141     ...  
142 }
```

Insofern bleibt zur Setter injection nur der Unterschied, dass die Assembler-Klasse zur Herstellung der Abhängigkeit (Konfiguration; s. u.) nicht die Client-Klasse selbst kennen muss, sondern nur das zur Herstellung der Abhängigkeit benötigte Interface.

Selbsttestaufgabe 1.3

Um was für ein Interface handelt es sich bei `ServerInjected`?

1.6.4 Assembler

Es bleibt natürlich die Frage, wer die Injektion durchführt, also wer die entsprechenden Methoden aufruft und wo dabei die Objekte, zu denen eine Beziehung hergestellt werden soll, herkommen. Dies ist Aufgabe eines Assemblers.

Der Assembler ist ein Programmstück (häufig eine eigenständige Klasse), das aufgerufen wird, um die Objekte (oder Komponenten), die miteinander kooperieren sollen, zu verdrahten. Der Assembler stellt also eine Konfiguration aus voneinander unabhängigen (in dem Sinne, dass sie keine expliziten Abhängigkeiten besitzen) Objekten her. Eine Assemblerklasse könnte beispielsweise wie folgt aussehen:

```
143 class Assembler {
144     Client client = new Client();
145     Server server = new Server();
146     {client.setServer(server);}
147 }
```

Dieser Assembler verwendet Setter injection; die Variante mit Interface injection könnte dagegen so aussehen:

```
148 class Assembler {
149     ServerInjected client = new Client();
150     Server server = new Server();
151     {client.injectServer(server);}
152 }
```

Der Vorteil ist jedoch, durch die explizite Erzeugung eines Clients per Konstruktorauftrag, begrenzt.

Nun sprechen zwei Gründe gegen Assemblerklassen der obigen Art:

1. Der Code fällt häufig gleich oder zumindest sehr ähnlich aus, weswegen man ihn nicht jedes Mal gern von neuem schreibt.
2. Die Konfiguration ist hier hart (im Quellcode) verdrahtet. Im Gegensatz dazu möchte man in der Praxis aber die Konfigurationen häufig gern von Aufruf zu Aufruf (Programmstart zu Programmstart) variieren, ohne dazu das Programm ändern zu müssen.

Aus beiden Gründen erscheint es sinnvoll, die Konfiguration nicht auszuprogrammieren, sondern in einer **Konfigurationsdatei** (gern auch im XML-Format) zu hinterlegen und die Konfiguration von einem universellen Assembler, der eine solche Datei lesen und interpretieren kann, durchführen zu lassen. Da hierfür aber wieder Konzepte der *Metaprogrammierung* (s. Kursein-

heit 6) erforderlich sind (unter anderem müssen Instanzen von Klassen erzeugt werden, deren Namen lediglich in einer Datei hinterlegt sind), gehen wir hier nicht weiter darauf ein.

1.6.5 Einschränkungen

Die Verwendung der Dependency injection kommt immer dann nicht infrage, wenn die Erzeugung der Abhängigkeit (also die Zuweisung des Objekts, zu dem die Abhängigkeit besteht, an eine Variable) von **Bedingungen** abhängig ist, deren Erfüllung nur die abhängige Klasse selbst erkennen kann. So ist zum Beispiel unklar, wie bei Vorliegen der Codestrecke

```
153 if (...)  
154     server = Server();  
155 else  
156     server = Server("www.fernuni-hagen.de");
```

eine der obengenannten Formen der Dependency injection eingesetzt werden soll, ohne dass sich daraus eine Änderung der Programmlogik ergäbe. Die Bedingung, die in Programmzeile 153 überprüft wird, müsste dazu auch vom Assembler überprüft werden können, was aber keineswegs immer möglich ist.

Ein ähnliches Problem tritt auf, wenn die Abhängigkeit nicht zu einem genau definierten, von außen feststellbaren **Zeitpunkt** (wie beispielsweise dem der Erzeugung des Client-Objekts) eingericichtet wird. Es ist dann für den Assembler nahezu unmöglich, genau diesen Zeitpunkt abzupassen und zu diesem die Abhängigkeit herzustellen. Hierfür sind schon Techniken der Metaprogrammierung, wie sie in Kurseinheit 6 beschrieben werden, notwendig. Dasselbe gilt auch für wiederholte Zuweisungen der Variable zu für den Assembler nicht vorhersagbaren Zeitpunkten.

Gleichermaßen unmöglich ist die Dependency injection für **temporäre Variablen**: Da ihre Belegung flüchtig und praktisch immer von der Programmlogik bestimmt ist, kann ein Assembler schon deswegen nicht eingreifen, weil er gar nicht weiß, wann die temporäre Abhängigkeit benötigt wird. Das gleiche gilt natürlich auch für durch formale Parameter hergestellte Abhängigkeiten.

1.6.6 Alternativen

Dependency injection ist nicht die einzige Möglichkeit, Abhängigkeiten durch Konstruktoraufruufe zu beseitigen — die Verwendung von *Factories* (Abschnitt 4.4.6) und (den im Zusammenhang mit Dependency injection häufig angeführten) sog. *Service locators* sind gebräuchliche Alternativen. Dabei wird zur Erzeugung einer Instanz der Serverklasse kein Konstruktor aufgerufen, sondern eine standardisierte Methode einer Factory oder eines Service locators, die eine entsprechende Instanz zurückgibt. In der einfachsten Form könnte das wie folgt aussehen:

```
157 class Client {  
158     IServer server = ServiceLocator.newServer();  
159     ...  
160 }  
  
161 class ServiceLocator {  
162     static IServer newServer() {  
163         return new Server();  
164     }  
165 }
```

Allerdings bezieht die Dependency injection ihren großen Reiz aus der vollständigen Lösung einer Klasse aus Abhängigkeiten und der externen Konfiguration: Während bei der Verwendung von Service locators oder Factories die Service-locator- bzw. Factory-Klasse bekannt sein und aus der Client-Klasse heraus aufgerufen werden muss, bleibt bei der Dependency injection nichts in der Klasse zurück, das ihre Verwendung an konkrete Voraussetzungen knüpfen oder ihre Konfiguration fixieren würde. Es werden insbesondere keine alten durch neue Abhängigkeiten ersetzt.

1.6.7 Fazit

Die Dependency injection ergänzt die interfacebasierte Programmierung um die Möglichkeit, Konstruktoraufrufe zu eliminieren und damit auch noch den letzten Rest der Abhängigkeit einer Klasse von anderen zu beseitigen. Dependency injection ohne Interfaces ist zwar ebenfalls möglich, ihr Nutzen beschränkt sich dann aber auf die flexible Konfiguration von Objekten — eine Entkopplung wird dadurch nicht erreicht.

Aufgrund ihrer schematischen Umsetzung wird die Dependency injection häufig auch als *Entwurfsmuster* bezeichnet. Entwurfsmuster werden in Kurseinheit 4 ausführlich behandelt. Zugleich ist die Dependency injection Ziel von Refactorings; diese sind Gegenstand von Kurseinheit 5.

1.7 Umkehrung von Abhängigkeiten mit Interfaces

- a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
- b. Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle, Robert C. Martin [9].

In traditioneller, prozeduraler Programmierung gilt es als gutes Design, wenn Prozeduren und Funktionen nach Abstraktionsgrad hierarchisch angeordnet sind. Eine solche Ordnung entsteht im allgemeinen automatisch aus der sog. *funktionalen Dekomposition*, bei der ein Gesamtproblem rekursiv so lange in Teilprobleme zerlegt wird, bis diese nicht weiter sinnvoll zerlegbar sind oder deren Lösungen bereits existierenden Programmbibliotheken entnommen werden können. Für ein solches Design sind *Aufrufabhängigkeiten* von oben nach unten charakteristisch.

In der objektorientierten Programmierung gibt es diese Abhängigkeit natürlich auch. Sie äußert sich aber nicht nur im Aufruf von primitiveren Methoden, sondern auch in der Referenzierung von Klassen, deren Instanzen die Methoden zugeordnet sind und über die sie aufgerufen werden (s. Abbildung 1.15a)).

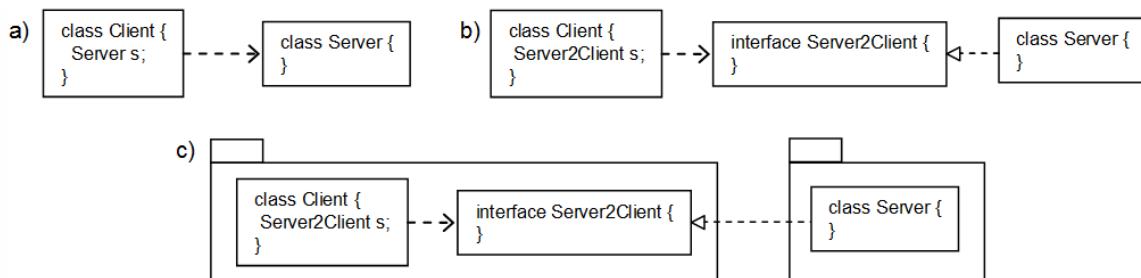


Abbildung 1.15: Schrittweise Herleitung der Abhängigkeitsumkehrung.

- konventionelle Aufrufabhängigkeit des Clients vom Server
- Aufrufabhängigkeit des Clients von einem Interface und Vererbungsabhängigkeit des Servers davon
- Umkehrung der Abhängigkeit gegenüber a) durch Betrachtung der Pakete, in denen Client, Server und Interface liegen

Da die Methoden einer Klasse nicht unbedingt alle das gleiche Abstraktionsniveau haben, lässt sich die hierarchische Ordnung der Methoden (die sich aus der funktionalen Dekomposition ergibt) nicht auch auf die Klassen übertragen. Dies wäre in einer geschichteten objektorientierten Architektur der Fall, auf die hier jedoch, da sie zur Erläuterung des Sachverhalts nicht notwendig ist, nicht eingegangen werden soll (die interessierte Leserin schaue bei [9] nach).

Wenn man nun die Klassen mit den aufgerufenen Methoden gegen andere austauschen möchte, muss man die Referenzen in den aufrufenden Klassen ändern. Die aufrufenden Klassen sind

damit, im Gegensatz zu den aufgerufenen, nicht ohne Modifikationen wiederverwendbar. Dies ist beispielsweise für den Entwurf von Frameworks, bei denen die darin codierte Ausführungs-kontrolle wiederverwendet werden soll (die *Umkehrung der Ausführungskontrolle*; vgl. Abschnitt 1.5.8), hinderlich (aber nicht nur dort), weswegen man diese Form der Abhängigkeit gern vermeiden würde.

Vererbungs-abhängigkeit

Wenn man nun die Referenz auf die Klasse durch eine Referenz auf ein geeignetes Interface, das von der Klasse implementiert wird, ersetzt, bleibt die Abhängigkeit, wenngleich zu einem anderen Typ, so doch zunächst bestehen (Abbildung 1.15 b)). Es kommt sogar eine neue hinzu, nämlich die von der (ursprünglich referenzierten) Klasse zu dem Interface, das sie implementieren muss, damit ihre Instanzen über die neuen Referenzen angesprochen werden können (eine sog. Vererbungs-, Spezialisierungs- oder Subtyp-Abhängigkeit). Was also ist gewonnen und vor allem: Worin besteht die Umkehrung der Abhängigkeit?

Blickwechsel durch Übergang zu größeren Organisations-einheiten

Die Beantwortung der Frage ergibt sich erst aus einer etwas distanzierteren Betrachtung, und zwar wenn man von den Klassen zu den sie enthaltenden Organisationseinheiten (in JAVA den Paketen) übergeht. Wenn man dann nämlich das Interface im Paket des Clients unterbringt und der Server in einem anderen Paket liegt, dann wird aus Paketsicht die Aufrufabhängigkeit in eine Vererbungsabhängigkeit in umgekehrter Richtung überführt und damit eine Umkehrung der Abhängigkeit erzielt (Abbildung 1.15 c)). Die Platzierung des Interfaces in der Nähe des Clients ist dabei gar nicht so gekünstelt, wie es auf den ersten Blick scheinen mag — es handelt sich dabei nämlich um das *benötigte Interface* (s. Abschnitt 1.3.1), das durch den Client, nicht den Server, bestimmt wird. Das eingeführte Interface macht dieses benötigte Interface (das sich zuvor allenfalls in einer Import-Klausel ausdrückte) explizit.

das Dependency Inversion Principle

Diese Umkehrung der Abhängigkeiten wird durch das sog. **Dependency Inversion Principle** [9] zum Grundsatz erhoben. Es besagt, dass man die Aufrufabhängigkeit von einer konkreten Klasse aufheben soll, indem man von der Klasse abstrahiert und die Abhängigkeit von der Klasse in eine von der Abstraktion umwandelt. Die Klasse realisiert (implementiert) dann die Abstraktion und ist dadurch ebenfalls von ihr abhängig. Alle Abhängigkeiten bestehen also von einer Abstraktion, was grundsätzlich gut ist, da die Abstraktionen eines Systems in der Regel das stabilste an ihm sind.

Schönheitsfehler

So reizvoll diese Betrachtung scheint, sie greift leider mindestens in zweifacher Hinsicht zu kurz. Zunächst wäre da die Abhängigkeit, die durch die Objekterzeugung (das Server-Objekt muss ja schließlich irgendwo herkommen) entsteht und die sich nur mit einem Aufwand beseitigen lässt (z. B. der *Dependency injection*; vgl. Abschnitt 1.6). Viel schwerer wiegt jedoch der Umstand, dass das Interface, das als Abstraktion dient, nicht auf Basis der Client-Klasse allein bestimmt werden kann: Wenn der Client noch andere Server hat und einem oder mehreren dieser das Server-Objekt, von dem es die Abhängigkeit auflösen will, als Parameter übergibt, dann verlangt das Typsystem, dass die Abstraktion auch die benötigten Interfaces der anderen Server (als Clients des übergebenen Server-Objekts) berücksichtigt.

Das Problem wird durch das folgende **Beispiel** verdeutlicht:

```
166 package a;
167 import b.B;
168 import c.C;
169 class A {
170     B b;
171     C c;
172     ...
173     b.m(c);
174     c.n();
175     ...
176 }
```

Die benötigten Interfaces der Klasse **A** von den Klassen **B** und **C** scheinen auf den ersten Blick klar zu sein: Das von **B** benötigte umfasst **m(C c)**, das von **C** umfasst **n()**. Nur leider wird der Code

```
177 package a;
178 interface IB { void m(C c); }
179 interface IC { void n(); }

180 class A {
181     IB b;
182     IC c;
183     ...
184     b.m(c);
185     c.n();
186     ...
187 }
```

nicht kompilieren, da die Methode **m** in der Klasse **B** einen Parameter vom Typ **C** erwartet und **IC** ein Supertyp, kein Subtyp von **C** ist. Außerdem ist hier die Abhängigkeit der Klasse **A** von der Klasse **C** auf das Interface **IB** übertragen, das per formalem Parameter **c** von **C** abhängt. Aus Sicht des Pakets **a** ändert sich also an der Abhängigkeit von **C** nur wenig.

Man könnte nun auf die Idee kommen, **C** in **IB** und in **B** durch **IC** zu ersetzen. Das würde das obige Typproblem und die Abhängigkeit von **C** gleichermaßen beheben, setzt jedoch voraus, dass der Klasse **B** das Interface **IC** als Typ von **c** genügt. Diese Frage lässt sich jedoch i. a. nicht durch eine Analyse von **B** allein beantworten, da die Klasse **B** den Parameter **c** an andere Klassen weiterreichen kann, so dass sich das Problem erneut stellt. Tatsächlich ist eine Verfolgung aller möglichen Zuweisungen von **c** notwendig, um zu bestimmen, welchen Typ **c** mindestens haben muss (vgl. das INFER TYPE Refactoring in Abschnitt 5.2.4.6). Dieser Typ wäre dann auch in **A** zu verwenden, selbst wenn er nicht nur das

**Inferenz des
transitiv benötigten
Interfaces**

Interface, das **A** von **C** benötigt, sondern das aller Klassen, die Zugriff auf den Parameter **C** erlangen, repräsentiert. Die Abhängigkeit von **C** wird damit durch ziemlich undurchsichtige Abhängigkeiten von **B** und ggf. weiteren Klassen ersetzt, was kaum der Sinn der Sache sein kann.

Cross casts an den Modulgrenzen

Eine einfache, aber brutale Lösung dieses Problems ist, **IC** für **A** tatsächlich wie oben (in Zeile 179) beschrieben zu implementieren und für **B** und alle anderen Klassen, die **C** referenzieren, ein eigenes Interface für **C** jeweils so, wie sie es brauchen. Es hätte dann jeder Client von **C** genau sein benötigtes Interface und an den Stellen, an denen eine Referenz auf eine Instanz der Klasse **C** an eine andere Klasse übertragen wird (per Parameter eines Methodenaufrufs wie in Zeile 184 oder per Zuweisung an ein Feld), müsste eine Typumwandlung, ein sog. **Cross cast**¹⁶, stattfinden. Ein solcher Cross cast entzieht sich zwar der *statischen Typprüfung* (und kann deswegen grundsätzlich zu *Laufzeittypfehlern* führen, weswegen er i. a. nicht gern gesehen wird), aber solange dieser Cross cast nur auf Variablen, die vorher mit **C** typisiert waren, durchgeführt wird, können dabei keine *Typumwandlungsfehler* (Type cast errors) auftreten.¹⁷ Ob diese Lösung aber praktikabel ist (oder praktiziert wird), darüber liegen mir keine Erkenntnisse vor.

1.8 Interpretation von Interfaces als Rollen

Man mag sich vielleicht fragen, welche Bedeutung Interfaces auf konzeptueller Ebene haben. Klassen stehen ja für Mengen gleichartiger Objekte, Methoden für die Funktionen dieser Objekte und Attribute für deren Eigenschaften bzw. die Verknüpfungen zwischen ihnen. Wofür könnte also ein Interface stehen?

Die Antwort ergibt sich in gewisser Weise aus der Verwendung von Interfaces als Typen von Variablen. Eine Variable drückt ja häufig (als Instanzvariable oder als formaler Parameter einer Methode) eine Beziehung eines Objektes zu einem anderen aus. Nun definiert jede Beziehung Rollen, die die Funktionen der involvierten Objekte an ihren jeweiligen Positionen festschreiben. So definiert die Mutter-Tochter-Beziehung die Rolle der Mutter und die der Tochter, die Angestelltenbeziehung die der Arbeitgeberin und die der Arbeitnehmerin sowie ganz allgemein das Dienstleistungsverhältnis die der Dienstanbieterin und die der Dienstnehmerin. Die Deklaration einer Variablen in einer Klasse definiert eine binäre (zweistellige) Beziehung zwischen dem Objekt, zu dem die Variable gehört, und dem, das die Variable enthält. Während der Typ des ersten Objekts durch die Klasse, in der die Variable deklariert ist, festgelegt ist, besteht für den des zweiten eine gewisse Freiheit, die in der Variablen Deklaration ausgenutzt werden kann:

- Verwendet man eine Klasse, dann ist damit die Art der Objekte, zu denen die Beziehung aufgenommen werden kann, festgelegt. Das gilt sogar, wenn auch mit Einschränkungen,

¹⁶ Ein Cross cast ist eine Typumwandlung, deren Ziel weder ein Super- noch ein Subtyp des Ausgangstyps, sondern ein „Geschwistertyp“ ist. Man kann sich ihn als eine Kombination aus Up cast und Down cast vorstellen.

¹⁷ Dies deswegen nicht, weil der Cross cast durch einen Down cast auf den Ausgangstyp (**C** im Beispiel), der in diesem Fall sicher ist, und einen anschließenden Up cast (der immer sicher ist) realisiert werden kann (s. vorangegangene Fußnote).

für abstrakte Klassen, denn auch diese können (Teile der) Implementierung vorgeben, und es ist schließlich die Implementierung, die die Art (das Genus) der Objekte ausmacht.

- Verwendet man hingegen ein Interface, so ist damit nicht die Art, sondern lediglich die Rolle der Objekte, die den Platz der Variable einnehmen, festgelegt. Um eine Rolle spielen zu können, müssen die Objekte das mit der Rolle verbundene Verhalten mitbringen — sie müssen die Rolle (das Interface) implementieren.

Die Interpretation von Interfaces als Rollen wird auch vielfach durch die Namensgebung unterstützt: Viele der ermöglichen Interfaces, die auf „able“ oder „ible“ enden, drücken eine Rolle aus, so z. B. die des Serialisierbaren (engl. `serializable`) oder die des Vergleichbaren (engl. `comparable`). Aber auch typische Client/Server-Interfaces tragen Rollennamen: `MenuContainer` beispielsweise drückt eine Rolle von Objekten so unterschiedlicher Klassen wie `Button`, `CheckBox` etc. aus. Lediglich die allgemeinen Interfaces drücken in der Regel keine Rollen aus; sie stehen in Konkurrenz zu abstrakten Klassen, die die inhaltliche Verwandtheit in den Vordergrund rücken.

Der Interpretation von Interfaces als Rollen wird manchmal entgegen gehalten, dass ein Objekt dieselbe Eigenschaft je nach Rolle unterschiedlich realisiert. Z. B. hat eine Person zuhause und im Büro zwei verschiedene Telefonnummern, und welche davon auf die Anfrage `getTelefonnummer()` zurückgegeben wird, hängt davon ab, in welcher Rolle sie sich gerade befindet (das die Person repräsentierende Objekt angesprochen wird). Zumaldest in C# ist das zunächst kein Problem, da es dort ja die explizite Interfaceimplementierung gibt (vgl. Abschnitt 1.2.1). Problematisch wird es allerdings, wenn eine Person mehrere Anstellungen und damit auch mehrere Büronummern hat; anstatt jedoch eine Person in verschiedenen Rollen durch verschiedene Instanzen zu repräsentieren, wäre zu überlegen, ob die Telefonnummer nicht besser Merkmal eines Objekts einer eigenständigen Klasse `Anstellung`, die die Beziehung von Angestellter und der Firma repräsentiert, wäre.

1.9 Werkzeugunterstützung für das interfacebasierte Programmieren

Die Erstellung, Verwendung und Pflege von Interfaces ist bei der Programmierung mit einem Aufwand verbunden. Besonders lästig ist, wenn sich eine Methode ändert oder eine neue hinzukommt: Diese Änderungen müssen dann sowohl im Interface als auch in der es implementierenden Klasse durchgeführt werden. Es ist sogar zu vermuten, dass dieser zusätzliche Aufwand einer der Hauptgründe ist, warum Interfaces in der Programmierung eher sparsam eingesetzt werden. Ein anderer Grund, nämlich dass die Verwendung von Interfaces die Performance bei der Programmausführung verschlechtert, wird immer wieder angeführt, ist jedoch nicht unbedingt stichhaltig; s. z. B. „Java subtype tests in real-time“ von Krzysztof Palacz und Jan Vitek, Proc. of ECOOP 2003. Der dritte Grund, die Probleme der Erweiterung von Interfaces bei Frameworks (in Abschnitt 1.2.3 diskutiert), dürfte für die meisten Programmiererinnen nicht relevant sein.

Glücklicherweise gibt es mittlerweile einige Werkzeuge, mit deren Hilfe das Erstellen und Verwenden von Interfaces bis zu einem gewissen Grad automatisiert erfolgen kann. Diese Werk-

zeuge, die auch unter dem Namen *Refactorings* bekannt sind und die in Kurseinheit 5 ausführlicher behandelt werden, sind heute bereits Bestandteil populärer Entwicklungsumgebungen wie ECLIPSE, NETBEANS oder INTELLIJ IDEA. Speziell für das interfacebasierte Programmieren können beispielsweise die ECLIPSE-Refactorings EXTRACT INTERFACE (bzw. das intelligentere INFER TYPE, das die Methoden eines Interfaces automatisch bestimmt; s. Abschnitt 5.2.4.6), GENERALIZE TYPE und USE SUPERTYPE WHERE POSSIBLE verwendet werden. Das Umbenennen von Methoden bzw. das Ändern von Signaturen eines Interfaces und aller es implementierenden Klassen wird einer durch das Refactoring RENAME METHOD abgenommen.

Das Bewusstsein, dass man Interfaces in Variablen Deklarationen verwenden sollte, ist eine Sache, beim Programmieren immer daran zu denken eine andere. Mit dem DECLARED TYPE GENERALIZATION CHECKER (DTGC)¹⁸ existiert ein Plugin für ECLIPSE, das für jede Variable im Programm überprüft, ob nicht vielleicht ein Interface existiert, mit dem sie besser deklariert werden könnte. Der DTGC greift dazu auf das ECLIPSE-interne Refactoring (s. Kurseinheit 5) GENERALIZE DECLARED TYPE zurück und erbt damit dessen Unzulänglichkeiten (insbesondere die mangelnde transitive Verfolgung von Zuweisungen). Alternativ kann zur Prüfung möglicher Generalisierungen auch eine Typinferenz (die von INFER TYPE; s. o.) verwendet werden, doch das dauert ziemlich lange und schlägt zudem eine kaum zu überblickende Vielzahl von verschiedenen Interfaces vor.

Die Darstellung und Umkehrung von Paketabhängigkeiten wie in Abschnitt 1.7 (Umkehrung von Abhängigkeiten mit Interfaces) beschrieben wird komfortabel durch den PACKAGE DEPENDENCY INVERTER¹⁹ erledigt. Er erlaubt es, für alle Verwendungen einer Klasse eines Pakets aus einem anderen Paket heraus ein gemeinsames benötigtes Interface des anderen Pakets von der Klasse zu berechnen und einzusetzen. Dabei werden auch die Abhängigkeiten berücksichtigt, die sich transitiv, aus der Weiterreichung von Instanzen der benutzten Klasse an andere Pakete, ergeben. Die praktische Verwendung des PACKAGE DEPENDENCY INVERTER zeigt aber auch, dass sich längst nicht alle Abhängigkeiten zwischen Paketen umkehren lassen: Die Verwendung eines Typs, zu dem eine Aufrufabhängigkeit besteht, als formaler Parametertyp kann dazu führen, dass das benötigte Interface eine neue Abhängigkeit zu ihm einführt, die sich zwar auch umkehren lässt, aber nur um den Preis einer neuen in die andere Richtung. Wie so oft widersetzt sich die Realität einer schönen Idee.

¹⁸ <http://www.fernuni-hagen.de/ps/prjs/DTGC/>

¹⁹ <http://www.fernuni-hagen.de/ps/prjs/PDI/>

1.10 Weiterführende Literatur

Die Verwendung von Interfaces in Variablendeklarationen ergibt sich ganz allgemein aus den Grundsätzen der objektorientierten Programmierung (so z. B. das eingangs erwähnte „program to an interface, not an implementation“ [1]) sowie speziell aus verschiedenen Prinzipien, so u. a. aus dem *Dependency Inversion Principle* und dem *Interface Segregation Principle* [9]. Erich Gamma erläutert seine Gedanken zum Thema Interfaces in der Programmierung in einem Interview, das unter <http://www.artima.com/lejava/articles/designprinciples.html> nachzulesen ist. Eine Übersicht zum sog. *Fragile base class problem* der objektorientierten Programmierung, das zu einer Art Ächtung der Vererbung geführt hat, findet sich in [10]; es wird in Kurs 01814 ausführlicher behandelt.

Der klassische Text von Martin Fowler zur Dependency injection findet sich unter <http://www.martinfowler.com/articles/injection.html>. (Dort kann man auch nachlesen, was ein Service locator ist.) Ein relativ neues Dependency-injection-Framework ist GUICE von GOOGLE (<http://code.google.com/p/google-guice/>). (Version 4.1, 27. Juni 2016)

Einige der hier wiedergegebenen Definitionen, Abbildungen und Ergebnisse stammen aus [11].

- [1] E Gamma, R Helm, R Johnson, J Vlissides Design Patterns — Elements of Reusable Software (Addison-Wesley, 1995).
- [2] EW Dijkstra „The structure of "THE"-multiprogramming system“ CACM 11:5 (1968) 341–346.
- [3] DL Parnas „On the criteria to be used in decomposing systems into modules“ CACM 15:12 (1972) 1053–1058.
- [4] IEEE Standard Computer Dictionary (IEEE, 1991).
- [5] <https://dl.acm.org/> “The Encyclopedia of Computer Science”
- [6] B Liskov, A Snyder, R Atkinson, C Schaffert „Abstraction mechanisms in CLU“ CACM 20:8 (1977) 564–576.
- [7] PS Canning, WR Cook, WL Hill, WG Olthoff „Interfaces for strongly-typed object-oriented programming“ in: Proc. of OOPSLA (1989) 457–467.
- [8] M Fowler „Public vs. published interfaces“ IEEE Software 19:2 (2002) 18–19.
- [9] RC Martin Agile Software Development. Principles, Patterns, and Practices (Prentice Hall International, 2003).
- [10] L Mikhajlov, E Sekerinski „A study of the fragile base class problem“ in: 12th European Conference on Object-Oriented Programming (ECOOP'98) Springer LNCS 1445 (1998) 355–382.
- [11] F Steimann, P Mayer „Patterns of interface-based programming“ Journal of Object Technology 4:5 (2005) 75–94.
- [12] K. Palacz, J. Vitek „Java subtype tests in real-time“ in Proc. Of ECC008 (2003) 378-404

1.11 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 1.1 (Seite 15)

s. Tabelle 1.1 und Tabelle 1.2

Selbsttestaufgabe 1.2 (Seite 26)

Für diesen Zweck stehen in manchen integrierten Entwicklungsumgebungen spezielle Refactorings zur Verfügung, die einem bei der Erstellung des Interfaces auf Basis einer existierenden Klasse (im gegebenen Beispiel `Server`) und dessen Verwendung helfen. Es sind dies z. B. in ECLIPSE EXTRACT INTERFACE (zur Erzeugung eines Interfaces) und GENERALIZE TYPE (zur Verallgemeinerung des Typs einer Variable, hier hin zum Interface, wenn es denn verwendbar ist, also die von der Variablen benötigten Methoden auch enthält). Unter Umständen müssen Sie jedoch das neu erzeugte Interface noch von Hand in die Typhierarchie einordnen, damit das Programm typkorrekt bleibt. Auch bleibt die Auswahl der Methoden, die in das Interface sollen, Ihnen überlassen.

Ein Refactoring, das all das automatisch vornimmt, ist unter dem Namen INFER TYPE als ECLIPSE-Plugin verfügbar (<http://www.fernuni-hagen.de/ps/prjs/InferType>).

Selbsttestaufgabe 1.3 (Seite 42)

Ein Server/Client-Interface, da die Instanzen der implementierenden Klasse die Nutznießerinnen sind: Sie bekommen jeweils ein Objekt, von dem sie abhängig sind, injiziert.

2 Design by contract

How to write correct programs and know it.

HD Mills , ACM SIGPLAN Notices 10:6 (1975) 363-370.

Korrektheit ist ein relativer Begriff:
Ein Programm ist korrekt, wenn es seine Spezifikation erfüllt.
Ohne Spezifikation ist jedes Programm korrekt!

nach Bertrand Meyer

Interfaces, wie sie in den vorangegangenen Kurseinheiten vorgestellt und verwendet wurden, haben einen entscheidenden Nachteil: Sie sind rein syntaktischer Natur, erlauben also nicht, ein bestimmtes erwartetes Verhalten²⁰ mit einer Schnittstelle zu verbinden. Das kann im Extremfall dazu führen, dass aufgrund zufälliger Namensgleichheit von Methoden Klassen mit völlig verschiedenen Funktionen, die nicht gegeneinander austauschbar sind, dasselbe Interface implementieren könnten.²¹ Dass eine Klasse ein Interface, das sie angibt zu implementieren, auch inhaltlich ausfüllt, also das erwartete Verhalten erbringt, liegt in der Verantwortung der Programmiererin und kann zunächst nicht automatisch sichergestellt werden.

Wie im richtigen Leben auch hat das Zusammenspiel zwischen Aufruferin und Aufgerufener nur dann das gewünschte Ergebnis, wenn sich beide an bestimmte Vereinbarungen halten. So kann beispielsweise von der Benutzerin (Aufruferin) eines Stacks erwartet werden, dass sie sich vergewissert, dass der Stack nicht leer ist, bevor sie sein oberstes Element anfordert. Die Benutzerin

²⁰ Unter Verhalten versteht man die dynamischen Aspekte eines Objekts oder eines ganzen Systems, also die Veränderungen über die Zeit, wobei mit Veränderungen in der Regel Zustandsänderungen gemeint sind. Veränderungen des Zustands vollziehen sich in kleinen, elementaren und diskreten Schritten, in der objektorientierten Programmierung in der Regel durch die Änderung von Variablenwerten. Wollte man Verhalten als Mengen möglicher Folgen von solchen Elementaränderungen beschreiben, so würde die Verhaltensspezifikation nicht nur äußerst umfangreich (und vermutlich, was Kompaktheit und Lesbarkeit angeht, nicht besser verständlich als der Code/das Programm selbst, dessen Verhalten es zu spezifizieren gilt), sondern auch übermäßig spezifisch: Auf die Reihenfolge der Veränderungen kommt es in der Regel gar nicht an, sondern nur auf das Endergebnis. Man spricht daher eigentlich besser von *Funktionalität*, aber Verhalten hat sich im Kontext der objektorientierten Programmierung eingebürgert.

²¹ Schon aus diesem Grund ist, wie bereits in Abschnitt 1.2.2 diskutiert, der *nominalen Typkonformität*, bei der eine Klasse explizit angeben muss, dass sie ein Interface implementiert (und man diese Deklaration als eine Art ernstgemeintes Versprechen interpretieren kann, all das zu tun, was das Interface verheißen) gegenüber der *strukturellen Typkonformität*, so wünschenswert sie auch manchmal sein mag, der Vorzug zu geben.

kann vom Stack (Aufgerufenen) im Gegenzug erwarten, dass ein auf ihm abgelegtes Element wieder abrufbar ist (also insbesondere nicht verlorengeht). Bildlich gesprochen könnte man sagen, dass die beiden, Aufruferin und Aufgerufene, einen **Vertrag** eingehen, der mit gegenseitigen Nutzen und Obliegenheiten verbunden ist. Die Art des Softwareentwurfs, die diese Verträge explizit macht, wurde von Bertrand Meyer **Design by contract** genannt [13] [14].²²

Das Prinzip des Design by contract betrachtet ein Softwaresystem als eine Menge kommunizierender Komponenten, deren Interaktion auf einer genauen Spezifikation der gegenseitigen Obliegenheiten beruht, nämlich Verträgen (Contracts).

Übersetzung aus [13]

Nach der Philosophie des Design by contract ist der erste Schritt in Richtung zuverlässiger Software, für jedes Softwareelement explizit zu sagen, was es tun soll. Zwar bewirkt die Formulierung einer Vorgabe nicht automatisch, dass sie auch eingehalten wird; lässt man es jedoch, dann ist die Chance, dass die (nicht gemachten, impliziten) Vorgaben eingehalten werden, außerordentlich gering. Zudem hat die ausführliche Befassung mit dem Was (der Vorgabe) selbst dann günstige Auswirkungen auf die Umsetzung in das Wie (die Implementierung), wenn beide sprachlich kaum etwas gemeinsam haben (so dass sich kein Effekt der Wiederverwendung des Was ergibt): Schon allein das Hinschreiben des Was hilft einem dabei, ein Verständnis dafür zu entwickeln, was die möglichen Probleme und Fehlerquellen der Implementierung sind, und nicht zuletzt können auf Basis einer umfassenden Spezifikation leicht (teilweise sogar automatisch) Testfälle generiert werden.

If you can't write it down in English, you can't code it.

Aus: JL Bentley "Bumper-sticker computer science" CACM 28:9 (1985) 896-901

²² Die technische Basis des Design by contract ist alles andere als neu. Jedoch ist der Begriff des Design by contract inzwischen so weitverbreitet (und die damit verbundene Metapher so eingängig), dass ich es für gerechtfertigt halte, ihn hier als stellvertretend für das ganze Thema zu verwenden.

2.1 Verhaltensspezifikation durch Zusicherungen: Vor- und Nachbedingungen, Invarianten

Tabelle 2.1: Vertrag zwischen Passagierin (Kundin) und Fluglinie (Lieferantin) über Beförderung (angelehnt an[13])

	VERPFLICHTUNGEN	NUTZEN
KUNDIN	<p><i>muss Vorbedingung einhalten</i></p> <ul style="list-style-type: none"> • rechtzeitig am Flughafen erscheinen • nur zulässiges Gepäck mitbringen • Ticket bezahlen 	<p><i>profitiert von Nachbedingung</i></p> <ul style="list-style-type: none"> • erreicht Zielort
LIEFERANTIN	<p><i>muss Nachbedingung einhalten</i></p> <ul style="list-style-type: none"> • Passagierin an Zielort bringen 	<p><i>profitiert von Vorbedingung</i></p> <ul style="list-style-type: none"> • muss niemanden befördern, die nicht pünktlich ist, unzulässiges Gepäck mitführt oder nicht bezahlt hat

Wie kann man aber das Was formulieren? Jede Aktivität, die nicht bedeutungslos ist, hat einen Effekt: Sie bewirkt eine Änderung, genauer die Änderung eines Zustandes. Die Betonung liegt dabei auf dem unbestimmten Artikel: Auf welchem Zustand die Aktivität ausgeführt wird, ist mit der Aktivität allein nicht festgelegt, weswegen die Änderung nur relativ dargestellt werden kann. Nach Tony Hoare [15] verwendet man dazu Tripel der Form

P {A} Q

(gelesen als „wenn P vor der Ausführung von A wahr ist, dann ist Q danach wahr“; man findet gelegentlich auch eine umgekehrte Klammerung der Ausdrücke vor, was jedoch nicht der ursprünglichen Form entspricht), wobei P und Q logische Ausdrücke, **Zusicherungen** genannt, sind und A eine (eine Änderung bewirkende) Anweisung oder Folge von Anweisungen (Programm) ist. P nennt man aufgrund seiner Position relativ zu A **Vor-**, Q **Nachbedingung** (engl. precondition und postcondition oder kurz *pre* und *post*). Um den Effekt, also die Veränderung relativ zum vorher vorherrschenden Zustand, ausdrücken zu können, benötigt die Nachbedingung eine Möglichkeit, auf den Zustand vor der Veränderung zuzugreifen. Mehr dazu in Abschnitt 2.5 Zusicherungen und Vererbung.

Vor- und Nachbedingungen lassen sich wie in Tabelle 2.1 verdeutlicht auf die Metaphorik des Design by contract übertragen: Die Aktion A entspricht der Dienstleistung (Beförderung von Passagieren), die Vorbedingung P den Verpflichtungen der Kundin und die Nachbedingung Q denen der Lieferantin.

Insbesondere ergeben sich aus Tabelle 2.1 die folgenden möglichen Vertragsverletzungen und deren Konsequenzen:

**Vor- und
Nachbedingungen
im Design by
contract**

1. Wenn die Vorbedingung nicht eingehalten wird, dann ist dies der Fehler der Kundin. Die Lieferantin ist damit frei von allen Verpflichtungen.
2. Wenn hingegen die Vorbedingung eingehalten wird, die Nachbedingung aber nicht, dann ist dies der Fehler der Lieferantin.

Übertragen auf ein Programm bedeutet dies: Eine Verletzung der Vorbedingung spricht für einen Fehler auf Seiten der Aufruferin, eine Verletzung der Nachbedingung für einen Fehler auf Seiten der Aufgerufenen.

Auslotung der Extreme

Ganz offensichtlich sind stärkere Vorbedingungen (also Vorbedingungen, die mehr Fälle ausschließen) schlechter für die Kundin und besser für die Lieferantin; umgekehrt sind stärkere Nachbedingungen besser für die Kundin und schlechter für die Lieferantin. So hat die Lieferantin das verständliche Bestreben, Vorbedingungen möglichst streng und Nachbedingungen möglichst schwach zu formulieren: Wenn man sie ließe, würde sie die Vorbedingung `false` (also immer falsch, so dass sie alle Fälle ausschließt) und die Nachbedingung `true` (also immer richtig, so dass sie erfüllt ist, egal was passiert) wählen. Zusicherungen dieser Art sind aber bedeutungslos: Sie dürfen in einem Programm, das ohne explizite Vor- und Nachbedingungen spezifiziert wurde, als *trivialerweise* gegeben angesehen werden (aus `false` folgt immer `true`).

Neben Vor- und Nachbedingungen verwendet man zusätzlich **Invarianten** genannte Zusicherungen, um Bedingungen zu spezifizieren, die *durchgängig erfüllt* sein müssen. Im Kontext von Design by contract kommen Invarianten vor allem als **Klasseninvarianten** vor, also als Bedingungen, die für alle Objekte einer Klasse *ständig* erfüllt sein müssen. Die aus der Programmverifikation bekannten Schleifeninvarianten zählen üblicherweise nicht zum Repertoire des Design by contract.

Beispiel

Ein Stack ist genau dann leer, wenn die Anzahl seiner Elemente gleich Null ist. Entsprechende Deklarationen vorausgesetzt, ergibt sich folgende Invariante:

`leer = (anzahl = 0)`

Man beachte, dass hier `=` für die Gleichheit, nicht für die Zuweisung steht.

Einschränkung der Erfüllung von Invarianten

Aufgrund eines fehlenden Transaktionskonzepts (also der blockweise atomaren Ausführung von Anweisungen) kann „durchgängige Erfüllung“ nur heißen, dass die Klasseninvariante vor und nach einem Methodenaufruf gelten muss. Während eines Methodenaufrufs (genauer: während der Abarbeitung einer Methode) können Klasseninvarianten temporär verletzt werden. So wird beispielsweise im folgenden JAVA-Code die obige Klasseninvariante vorübergehend verletzt (= bedeutet dabei Wertzuweisung!):

```
188 anzahl = anzahl - 1;
189 if (anzahl == 0)
190     leer = true;
```

Würden die drei Zeilen hingegen wie ein atomarer Block ausgeführt (als Transaktion), würde die Invariante auch nicht temporär verletzt.

Klasseninvarianten spezifizieren das Verhalten (die Semantik) einer ganzen Klasse, nicht einzelner Methoden. Sie tragen wesentlich zur Definition und — im Idealfall — auch zum Verständnis der Klasse bei. Wenn die Invarianten einer Klasse einmal richtig erfasst worden sind (richtig in dem Sinne, dass sie die Anforderungen der Kundin an die Klasse beschreiben), dann sollten sie gegenüber Änderungen (Korrekturen) der Klasse invariant sein²³. Invarianten liefern damit einen wertvollen Beitrag zur Sicherstellung der Tatsache, dass ein Programm nach einer Änderung noch korrekt ist, eine Eigenschaft, die normalerweise durch sog. Regressionstesten (s. a. Kurseinheit 3 und Abschnitt 7.3) sichergestellt wird.

Klasseninvarianten und Semantik einer Klasse

Klasseninvarianten können auf private Eigenschaften einer Klasse (bzw. deren Instanzen) zugreifen. Sie beziehen sich dann auf den internen Zustand von Objekten und damit deren Implementationsgeheimnis; man nennt sie deswegen auch Implementationsinvarianten. Da Interfaces (einschließlich Klasseninterfaces) keine Implementierungen vorschreiben, finden sich in Interfaces auch keine **Implementationsinvarianten**. Umgekehrt können sich Vor- und Nachbedingungen, solange sie Teil einer Interfacespezifikation sind, nur auf öffentlich sichtbare Eigenschaften einer Klasse beziehen (s. Abschnitt 2.4 Design by contract in der Programmierung).

2.2 Ein paar einfache Beispiele

In der einfachsten Form des Design by contract übernehmen Vor- und Nachbedingungen Aufgaben eines Typsystems: Sie überprüfen die Zulässigkeit der Ein- und Ausgabewerte einer Methode. So könnte beispielsweise eine Methode

```
191 void setzeUhrzeit(int stunde, int minute, int sekunde) {...}
```

mit den Vorbedingungen versehen werden, dass der Wert von stunde (genauer: der bei der Parameterübergabe an den formalen Parameter stunde zugewiesene Wert) zwischen 0 und 23 und die Werte von minute und sekunde zwischen 0 und 59 liegen müssen:

$$0 \leq \text{stunde} \leq 23 \quad \wedge \quad 0 \leq \text{minute} \leq 59 \quad \wedge \quad 0 \leq \text{sekunde} \leq 59$$

Genau diese Vorbedingungen ließen sich jedoch auch durch die **Verwendung entsprechender Typen** direkt im Code ausdrücken:

```
192 void setzeUhrzeit(Stunde stunde, Minute minute, Sekunde sekunde) ...
```

Der Vorteil dieser Variante wäre zudem, dass der Compiler die Einhaltung der Invarianten statisch (d. h. zur Übersetzungszeit) überprüft, solange ein entsprechendes Typsystem existiert (vgl. Kurs 01814; s. a. Abschnitt 3.9). Dem entgegen werden Zusicherungen à la Design by contract

²³ Das gilt auch für die Vererbung: Subklassen einer Klasse müssen deren Invarianten erfüllen. Mehr dazu in Abschnitt 2.5 Zusicherungen und Vererbung.

meist dynamisch, d. h. zur Laufzeit, überprüft, was zum einen teuer ist und zum anderen auch immer nur eine fallweise Sicherstellung (nämlich mit den aktuellen Werten) der Einhaltung ist (s. Abschnitt 2.4 Design by contract in der Programmierung).

Ausdruck von Abhängigkeiten zwischen Parametern

Ein Vorteil von Vor- und Nachbedingungen gegenüber den simplen Bereichsangaben, die mittels Typen möglich sind, liegt darin, dass auch Abhängigkeiten zwischen den Parametern ausgedrückt werden können. So kann beispielsweise für eine Methode zur Berechnung der Steigung zwischen zwei Punkten a und b

```
193 double steigung(Punkt a, Punkt b) {
194     return (double) (a.y - b.y) / (a.x - b.x)
195 }
```

die Vorbedingung

$$a.x \neq b.x$$

angegeben werden, was einem Typsystem wie dem JAVAs ein größeres Problem bereiten würde.

Spezifikation von Zusammenhängen zwischen Ein- und Ausgabe

Noch interessanter wird die gegenüber den einfachen Wertebereichsüberprüfungen eines Typsystems vergrößerte Ausdrucksstärke, wenn es gilt, Zusammenhänge zwischen Ein- und Ausgabe zu spezifizieren. Dies ist insbesondere für solche Probleme interessant, bei denen eine Lösung vergleichsweise schwierig zu finden, aber einfach zu überprüfen ist, wie etwa bei der Berechnung der Quadratwurzel.

Beispiel

```
196 double sqrt(double a) {
197     double x = 0;
198     double xn = 1;
199     while (Math.abs(x - xn) > 1/1000) do {
200         x = xn;
201         xn = (x + a/x)/2
202     }
203     return xn;
204 }
```

Als Vorbedingung würde man hier

$$a \geq 0$$

angeben, als Nachbedingung

$$|a - \text{sqrt}\cdot\text{sqrt}| < \varepsilon$$

wobei ε eine Konstante ist, die die gewünschte Genauigkeit des Verfahrens angibt.

Man beachte, dass in diesem Beispiel das Abbruchkriterium für die Iteration ein anderes ist als das, das in der Nachbedingung ausgedrückt wird. Wäre es das gleiche, dann wäre die Einhaltung der Nachbedingung *trivial* — die Methode könnte allenfalls nicht terminieren. Die Absicherung durch solche Nachbedingungen ist jedoch nicht sinnvoll — wie schon bei *Typsystemen*, kommt der Mehrwert durch *Redundanz*, also dadurch, dass das Gleiche auf verschiedene Weisen ausgedrückt wird und somit ein Widerspruch einen Fehler aufdecken kann.

Mehrwert nur durch Redundanz

Den bisherigen Beispielen ist übrigens gemein, dass sie keinen Bezug auf den Zustand von Objekten nehmen (vgl. die Ankündigungen des vorherigen Abschnitts). Das macht die Sache relativ einfach; insbesondere reichen so einfache logische oder mathematische Ausdrücke, um die Zusammenhänge herzustellen. Schwieriger wird es jedoch, wenn auch die zeitliche Komponente ins Spiel kommt, wenn also die Zustandsänderungen von Objekten, die eine Methode bewirken kann, in die Formulierung insbesondere der Nachbedingung einbezogen werden sollen. Dann sind nämlich erweiterte Möglichkeiten des Ausdrucks erforderlich, wie das folgende Beispiel zeigt:

Spezifikation der Abhängigkeit vom Zustand

Beispiel

```
205 boolean toggle() {  
206   toggle = ! toggle;  
207   return toggle;  
208 }
```

Als Nachbedingung würde man hier

$\text{toggle} = \neg \text{toggle}'$

angeben, wobei toggle' für den Wert von toggle vor der Ausführung der Methode stehen soll.

Selbsttestaufgabe 2.1

Ein Standardbeispiel für Objekte mit Zustand ist die Klasse `stack`. Formulieren Sie für diese Klasse formlos die Vor- und Nachbedingungen für die Methoden `push`, `pop` und `top`. Versuchen Sie, möglichst vollständig zu sein.

2.3 Design by contract in der Analysephase

Wie der Name schon suggeriert, handelt es sich beim Design by contract zunächst um eine Art des Entwurfs. Oftmals fällt es jedoch leichter, eine genaue Spezifikation dessen, was eine Klasse tun soll, zu erstellen, solange man noch keine konkrete Umsetzung vor Augen hat: Nur allzu leicht lässt man sich von seinen Plänen zu Design und Implementierung leiten und verliert dabei vor lauter Programmiererifer das eigentliche Ziel, das ja mit der Kundin gemeinsam erarbeitet werden sollte (vgl. auch Abschnitt 7.6), aus den Augen. Vorbedingungen, Nachbedingungen

und Invarianten erstellt man deswegen, zumindest in einer ersten Version, zweckmäßigerweise bereits während der Analysephase eines Projekts.

Vor- und Nachbedingungen als hartes Ergebnis der Analysephase

Ein weiterer triftiger Grund, Design by contract bereits während der Analysephase zu verwenden, ist, dass die Endprodukte dieser Phase sonst häufig nur aus Kästchen- und Liniendiagrammen bestehen, auf die es zwar leicht ist, sich zu einigen, von denen aber jede eine andere Interpretation mit nach Hause nimmt. So passiert es leider häufig, dass aus der frühen Euphorie (angesichts der schönen Bildchen, auf die sich ja alle geeinigt hatten) eine späte Enttäuschung wird (angesichts der Software, die nicht das tut, was die Kundin eigentlich wollte). Insbesondere die Spezifikation von Nachbedingungen, also dessen, was eine bestimmte Systemfunktion bewirken soll, kann dabei helfen, solche Enttäuschungen zu vermeiden.

Übrigens: Wenn man die gefundenen Bedingungen gleich in Form von Zusicherungen den Methoden und Klassen, die in der Analysephase identifiziert wurden, zuordnet, dann kann man diese in Entwurf und Programmierung übernehmen. Design by contract unterstützt die vielgerühmte Durchgängigkeit der Objektorientierung.

The hardest single part of building a software system is deciding precisely what to build. [...] No other part of the work so cripples the resulting system if done wrong; no other part is more difficult to rectify later.

FP Brooks „No Silver Bullet“ in: *The Mythical Man Month* (Jubiläumsausgabe, 1995).

2.4 Design by contract in der Programmierung

Trotz seines unbestreitbaren Nutzens in der Analyse- und Entwurfsphase hat das Design by contract in der Praxis seinen wirksamsten Niederschlag in der Programmierung gefunden: Vor- und Nachbedingungen sowie Invarianten werden in Programme eingebaut, als wären sie Teil der Implementierung. Dabei können die Zusicherungen direkt auf den sie umgebenden Code (die sichtbaren Programmelemente) zugreifen und sie zum Teil ihrer Bedingungen machen. Das erleichtert die praktische Umsetzung des Design by contract erheblich, birgt aber auch Nachteile.

Beispiel

Eine Vorbedingung der Methode `pop` der Klasse `Stack` ist, dass der Stack nicht leer sein darf. Wenn die Klasse `Stack` eine Methode `isEmpty()` anbietet, dann lässt sich die Vorbedingung als `! isEmpty()` formulieren.

geringe Redundanz bei Verwendung von Programm-elementen

Ein erster offensichtlicher Nachteil der Verwendung von „Nutzcode“ für die Formulierung von Zusicherungen ist die fehlende Redundanz: Anstatt unabhängig zu formulieren, was Leersein für einen Stack bedeutet, wird hier auf eine Methode zurückgegriffen, die selbst Bestandteil des (durch Zusicherun-

gen abzusichernden) Programms ist. Eine derart formulierte Vorbedingung versagt dann ihren Dienst, wenn die Methoden, auf die sie zurückgreift, selbst fehlerhaft sind. Dies kann natürlich ausgeschlossen werden, indem man `isEmpty()` selbst durch eine entsprechende Nachbedingung absichert, etwa der Form, dass `isEmpty()` dann und nur dann wahr liefert, wenn die Anzahl der Elemente auf dem Stack gleich Null ist. Spätestens dann, wenn man es mit zirkulären Abhängigkeiten zu tun hat, wenn also die Korrektheit einer in einer Zusicherung herangezogenen Methode selbst direkt oder indirekt von der abzusichernden Methode abhängt, wird die Sache aber unübersichtlich.

Ein zweiter Nachteil besteht darin, dass Methoden, die zur Überprüfung einer Zusicherung ausgeführt werden, einen (Neben)Effekt haben können. Das Programm würde sich dann anders verhalten, wenn die Überprüfung nicht durchgeführt würde. Die (Überprüfung der) Zusicherung würde damit zu einem Bestandteil der Programms; sie dürfte selbst dann nicht entfernt werden, wenn erwiesen wäre, dass das Programm funktioniert. Auch stellt sich die Frage, wie die so erzielte (indirekte) Wirkung der Zusicherung überprüft werden soll (gibt es Vor- und Nachbedingungen für Zusicherungen?) und ob die Reihenfolge der Überprüfungen von Zusicherungen einen Einfluss auf das Ergebnis der Überprüfung hat. Da zudem eine Wirkung mit der Idee einer Zusicherung unvereinbar ist, sollten (neben)effektvolle Methoden von der Verwendung in Zusicherungen ausgeschlossen werden. Das automatisch sicherzustellen ist aber nicht ganz einfach (vgl. auch Abschnitt 2.6 Spezifikationssprachen für das Design by contract, insbesondere 2.6.3JML).

Gefahr der Abhängigkeit von Nebeneffekten

2.4.1 Zeitpunkt der Überprüfung von Zusicherungen

Die obigen Ausführungen zu den Nebeneffekten legen nahe, dass zur Überprüfung der Zusicherungen der Code, der zu ihrer Formulierung herangezogen wird, ausgeführt werden muss. Tatsächlich sehen die heute üblichen Instrumentierungen des Design by contract vor, dass die Einhaltung von Vor- und Nachbedingungen zur Laufzeit überprüft wird und nicht durch einen Theorembeweiser interpretiert wird. Anders als bei der klassischen Programmverifikation wird also kein allgemeingültiger Beweis der Korrektheit durchgeführt, sondern lediglich eine fallweise Überprüfung. Es gilt daher auch für das Design by contract der Leitspruch des Testens, nämlich dass es nur Fehler, keine Fehlerfreiheit nachweisen kann (s. Kurseinheit 3). Da die Überprüfung zur Laufzeit auch noch teuer ist (sie benötigt zumindest zusätzliche Rechenzeit), muss sie abgeschaltet werden können, was im Falle (neben)effektvoller Methoden als Basis der Zusicherungen auch noch zu einer Änderung des Programmverhaltens führen kann.

Überprüfung der Zusicherungen nur zur Laufzeit

Zweifellos sind es gerade die Möglichkeit der Überprüfung von Zusicherungen durch das Programm selbst und damit der beherrschbare zusätzliche Aufwand, die dem Design by contract zu seinem vergleichsweise (gegenüber klassischen Verfahren der Programmverifikation) großen Erfolg verholfen haben. Die oben angeführten Einschränkungen relativieren das Ergebnis jedoch erheblich. Fernziel des Design by contract ist daher eine statische Programmverifikation, die zur Übersetzungszeit stattfindet und die allgemeingültige

Verified design by contract

Aussagen über die Einhaltung der Zusicherungen macht. Dies wird auch **Verified design by contract** genannt, ist aber derzeit leider immer noch Zukunftsmusik²⁴.

Invarianz bei externen und internen Aufrufen einer Methode

Während der Zeitpunkt der Überprüfung von Vor- und Nachbedingungen klar ist (Vorbedingungen müssen unmittelbar vor, Nachbedingungen unmittelbar nach der Ausführung einer Methode überprüft werden), ist es der für die die ganze Klasse betreffenden Invarianten nicht; sie sind nämlich gewissermaßen Vor- und Nachbedingung aller Methoden einer Klasse zugleich. Da Invarianten durch Ausführung jeder einzelnen Methode verletzt werden können, müssen sie zumindest *nach* jedem externen Methodenaufruf (das sind Aufrufe, bei denen Aufruferin und Aufgerufene verschiedene Objekte sind) überprüft werden. Solange nach dem externen Methodenaufruf vor dem externen Methodenaufruf ist (also zwischen zwei externen Methodenaufrufen auf einem Objekt nichts anderes passiert), ist die Prüfung der Invarianten vor einem Methodenaufruf überflüssig. Dass das aber nicht automatisch der Fall ist, dazu unten mehr.

Im Gegensatz zu externen Aufrufen dürfen interne Aufrufe einer Methode Klasseninvarianten nach Belieben verletzen. Interne Aufrufe erfolgen nämlich praktisch immer als Resultat von externen Aufrufen und dienen dann ausschließlich der Erfüllung dieser. Da während der Ausführung einer Methode Invarianten temporär verletzt werden dürfen, gilt dies auch für im Zuge der Ausführung aufgerufene weitere Methoden. Daraus folgt übrigens, dass nach Aufrufen von Methoden, die nur intern aufgerufen werden können, grundsätzlich keine Invarianten geprüft werden müssen. In JAVA beispielsweise gilt das aber nicht automatisch für als **private** deklarierte Methoden, da diese ja auch von anderen Objekten derselben Klasse, und damit extern, aufgerufen werden können.

Invarianten und Aliasing

Die Überprüfung der Invarianten scheint sich also auf den Zeitpunkt nach externen Methodenaufrufen einschränken zu lassen. Es gibt aber ein recht subtiles Problem, das durch das der objektorientierten Programmierung immer wieder zu schaffen machende und bislang verschwiegene sog. **Aliasing** (mehrere Variablen enthalten — oder vielmehr: zeigen auf — dasselbe Objekt; s. Kurs 01814) verursacht wird: Wenn die Invariante einer Klasse K den *Zustand* von Objekten einbezieht, die selbst Instanzen anderer Klassen sind (was typischerweise bei Attributwerten der Fall ist), dann ist es möglich, dass der Zustand dieser (fremden) Objekte durch ihre eigenen Methoden (von Dritten aufgerufen) geändert wird, ohne dass diese Änderungen unter der Kontrolle der Klasseninvarianten von Klasse K stünden. Aus Sicht der Klasse, die die Invariante definiert hat (Klasse K), ändern sich diese Objekte (Attributwerte) völlig unkontrollierbar — quasi zufällig —, so dass es keinen definierten Zeitpunkt gibt, zu dem die Invariante überprüft werden müsste. Augenfällig wird eine so bewirkte Verletzung der Klasseninvariante erst, nachdem wieder eine Methode der Klasse selbst von extern aufgerufen wurde. Dann aber ist es zu spät.

Dies lässt sich kaum verhindern. Um aber wenigstens die Last der Schuld für die Nichteinhaltung der Klasseninvariante von der Methode, nach deren Ausführung die Verletzung entdeckt wurde,

²⁴ In die Richtung des Verified design by contract gehen die Code Contracts von Microsoft.

zu nehmen, sollten die Klasseninvarianten auch vor jeder Methodenausführung überprüft werden. Wenn eine Invariante vor einer Methodenausführung bereits verletzt ist, dann kann dies nur das Resultat einer Fremdverletzung sein, da ja jede eigene Methode am Ende ihrer Ausführung wieder alle Varianten einhalten musste.

2.4.2 Beeinflussung der Programmierung

In der Programmierung wird stets *Modularität* angestrebt. Das bedeutet, dass die Abhängigkeit von Eigenschaften des Moduls (in der Objektorientierung in der Regel eine Klasse) möglichst gering gehalten werden soll. So soll unter anderem ein hoher Grad an Wiederverwendbarkeit sichergestellt werden. Dazu gehört auch, dass das Modul selbst die Einhaltung seiner Nutzungsbedingungen sicherstellt: Insbesondere sollten vor jeder Ausführung einer Methode deren Vorbedingungen überprüft und bei Nichteinhaltung die Ausführung der Methode verweigert werden. Man nennt dies *defensive Programmierung*; sie galt jahrzehntelang als vorbildlich.

Das Design by contract verlangt jedoch etwas anderes: Vorbedingungen drücken Verpflichtungen der Aufruferin aus, die diese einhalten muss, damit die Aufgerufene ihren Teil des Vertrages erfüllen kann. Dabei ist es die Aufruferin, die sicherzustellen hat, dass die Vorbedingung eingehalten wird. Im Beispiel der Klasse **Stack** und der Operation **push** beispielsweise hat die Nutzerin des Stacks sicherzustellen, dass der Stack nicht voll ist. Wenn alle Aufruferinnen die Vorbedingungen einhalten, ist eine weitere Absicherung des Stacks überflüssig. Sie müssen es nur alle tun.

Für diese (gegenüber der traditionellen Aufteilung) umgekehrten Verantwortlichkeiten gibt es neben bloßer Ideologie einen handfesten **praktischen Grund**: Die Aufgerufene kann zwar einen Dienst ablehnen (also nicht verrichten), wenn die Vorbedingung nicht eingehalten wurde, aber sie weiß nicht, was die Aufruferin stattdessen machen möchte. Sie muss also der Aufruferin die Verletzung signalisieren und diese muss dann entscheiden, was sie stattdessen zu tun gedacht. Dies kommt aber der Abfrage eines Fehlercodes auf Seiten der Aufruferin und einer davon abhängigen Verzweigung im Programm gleich (vgl. dazu auch die Diskussion in Abschnitt 5.2.2.4). Stattdessen kann die Aufruferin aber auch gleich selbst prüfen, ob die Vorbedingung eingehalten ist, und daraufhin entsprechend verzweigen. Dabei tut es nichts zur Sache, ob die Aufruferin zur Überprüfung der Vorbedingung auf die Aufgerufene zurückgreifen muss; im Beispiel des Stacks kann die Aufruferin ja selbst gar nicht wissen, ob der Stack vielleicht schon voll ist. Die notwendige Voraussetzung dieser Aufgabenteilung ist jedoch, dass die Aufgerufene ihre Eigenschaften, die zur Überprüfung der Einhaltung der Vorbedingung notwendig sind, nicht vor der Aufruferin verbirgt, sondern ihr direkt zugänglich macht. Dies kann z. B. durch Verwendung des Access modifiers **public** oder durch die Bereitstellung eines entsprechenden Interfaces geschehen und harmoniert mit der Feststellung vom Ende des Abschnitts 2.1 Verhaltensspezifikation durch Zusicherungen: Vor- und Nachbedingungen, Varianten, nach der sich Vor- und Nachbedingungen einer Klasse auf öffentliche Eigenschaften beziehen müssen sind. Die Modularität wird dadurch jedoch möglicherweise etwas geringer.

2.5 Zusicherungen und Vererbung

Das *Prinzip der Substituierbarkeit* verlangt, dass die Instanzen von Subtypen verhaltenskonform zu den Instanzen der Typen sind, von denen die Subtypen ableiten (s. Kurs 01814). Bislang war es der Verantwortung der Programmiererinnen überlassen, die verlangte Verhaltenskonformität zu gewährleisten. Mit Hilfe des Design by contract ist es nun möglich, Verhalten zu spezifizieren; damit sind die Voraussetzungen dafür geschaffen, dass Verhaltenskonformität in gewissen Grenzen auch automatisch überprüft werden kann.

Die Überprüfung basiert auf den folgenden Überlegungen. Substituierbarkeit ist nur dann gegeben, wenn das substituierende Objekt den Vertrag des substituierten einhält. Das tut das substituierende Objekt genau dann, wenn es nicht mehr an Voraussetzungen (Vorbedingungen) verlangt und nicht weniger an Leistung (Nachbedingungen) liefert. Mit anderen Worten: Die Vorbedingungen der Methoden eines Subtyps müssen gleich oder schwächer sein, die Nachbedingungen gleich oder stärker.

Überprüfung der Einhaltung

Wie aber lässt sich überprüfen, ob Zusicherungen stärker oder schwächer sind? Logisch wird ein entsprechender Sachverhalt durch die Implikation (\rightarrow) ausgedrückt: Stärkere Bedingungen implizieren schwächere, was so viel heißt wie wenn die stärkeren Bedingungen erfüllt werden, sind automatisch auch die schwächeren erfüllt. Auf Vor- und Nachbedingungen im Kontext der *Vererbung* übertragen heißt das:

$$\text{Vorbedingung}_{\text{super}} \rightarrow \text{Vorbedingung}_{\text{sub}}$$

$$\text{Nachbedingung}_{\text{super}} \leftarrow \text{Nachbedingung}_{\text{sub}}$$

Leider lässt sich die Frage, ob zwei logische Ausdrücke einander implizieren, rechnerisch nur höchst aufwendig beantworten. Stattdessen kann man sich aber eines einfachen Tricks bedienen: Es gelten nämlich immer die sog. Tautologien

$$A \rightarrow A \vee B$$

$$A \wedge B \rightarrow A$$

womit auch

$$\text{Vorbedingung}_{\text{super}} \rightarrow \text{Vorbedingung}_{\text{sub}} \vee \text{Vorbedingung}_{\text{super}}$$

$$\text{Nachbedingung}_{\text{super}} \leftarrow \text{Nachbedingung}_{\text{sub}} \wedge \text{Nachbedingung}_{\text{super}}$$

gilt. Wenn man also im Falle der Vorbedingungen die Vorbedingung des substituierenden Objekts (des Subtyps) mit der Vorbedingung des substituierten Objekts (des Supertyps) disjunktiv (per logischem Oder) verknüpft, wird dadurch die Vorbedingung des substituierten Objekts höchstens aufgeweicht, so dass die Vorbedingung tatsächlich wie verlangt abgeschwächt wird. Im Falle der Nachbedingung ist es genau umgekehrt: Wenn man hier die beiden Nachbedingungen konjunktiv (per logischem Und) verknüpft, wird dadurch die Nachbedingung des substituierten Objekts höchstens verschärft. Es entspricht dies genau den Anforderungen der *Substituierbarkeit*.

In Sprachen wie JAVA oder C# und zu einem gewissen Grad auch EIFFEL ist die Substituierbarkeit (das *Subtyping*) an die Klassenhierarchie und damit an die Vererbung geknüpft (das sog. *Subclassing*; vgl. Kurs 01814).²⁵ Es liegt daher nahe, auch von einer *Vererbung von Vor- und Nachbedingung* zu sprechen.

Interpretation als Vererbung von Vor- und Nachbedingungen

Allerdings fällt das Erbe nach obigem Prinzip jeweils unterschiedlich aus: Es muss einmal disjunktiv, einmal konjunktiv eingebaut werden, und zwar je nachdem, ob es sich um Vor- oder Nachbedingung handelt. In EIFFEL (und auch in JML; s. Abschnitt 2.6.3 JML) muss sich die Programmiererin darum allerdings nicht kümmern: Dort werden die geerbten Zusicherungen nach dem eben beschriebenen Prinzip automatisch mit den neu gemachten verknüpft. Die für das mit der Vererbung einhergehende Subtyping notwendige Aufweichung bzw. Verschärfung (s. Kurs 01814) ergibt sich damit in EIFFEL automatisch; im schlimmsten Fall wird die Vorbedingung logisch äquivalent zu `true` und die Nachbedingung äquivalent zu `false`.

Gegen diese pragmatische Vorgehensweise, die immer richtig ist, gibt es aber auch handfeste Vorbehalte. So kann beispielsweise für eine in einer Subklasse überschriebene Methode die Vorbedingung zunächst verschärft werden, was dann aber bei der Überprüfung zur Laufzeit (wegen der automatischen, aber unsichtbaren Disjunktion) keine Konsequenzen hat. Wenn die Methode dann trotzdem so implementiert wird, dass die verschärzte Vorbedingung Voraussetzung für die erfolgreiche Durchführung ist, dann stellt die Vorbedingung keinen Schutz dar, auch wenn dies bei Inspektion der Klasse zunächst so aussah.

Vorsicht Falle!

Aus der Substituierbarkeit folgt ferner, dass alle Eigenschaften, die für die Objekte einer Klasse gelten, auch für die Objekte ihrer Subklassen gelten müssen. Das bedeutet für Eigenschaften, die als Invarianten ausgedrückt werden: Die Invarianten der Subklasse müssen die der Superklasse implizieren. Aus denselben Gründen wie oben wird die Implikation aber nicht rechnerisch überprüft, sondern durch Erben der Invarianten und deren konjunktive Verknüpfung mit den spezifischen der Subklasse (falls vorhanden) sichergestellt. So kann beispielsweise die Invariante für ein Konto besagen, dass der Kontostand immer über dem Dispositionslimit liegen muss. Die Invariante eines Sparbuchs (als ein spezielles Konto) kann dem hinzufügen, dass das Dispositionslimit null ist. Die logische Verknüpfung ergibt dann, dass bei einem Sparbuch der Kontostand immer größer als Null sein muss.

Vererbung von Invarianten

Wie bereits am Beispiel der Uhrzeit in Abschnitt 2.2 bemerkt, gibt es eine aufschlussreiche Analogie zwischen den Regeln der *Typprüfung* und den Regeln für die *Vererbung von Vor- und Nachbedingungen*. Die *Substituierbarkeit* von Objekten verlangt nämlich (zumindest nach landläufiger Ansicht), dass die Parameter einer überschriebenen Methode nur **kontravariant** verändert werden dürfen, während der Rückgabewert **kovariant** verändert werden kann. Dabei versteht man unter Kovarianz die Veränderung eines Parameter- oder Rückgabetypen in dieselbe Richtung wie die der definierenden Klasse, unter Kontravarianz hingegen eine Veränderung in die entgegengesetzte Richtung (s. Kurs 01814). Da Parametertypen Vorbedingungen ausdrücken, entspricht die Kontravarianz der Parametertypen

Vorbedingungen, Nachbedingungen und das Typsystem

²⁵ In EIFFEL können geerbte Methoden gelöscht oder ihre Eingabeparameter kovariant redefiniert werden, womit die Substituierbarkeit verlorengeht.

einer Abschwächung der Vorbedingung; der Typ des Rückgabewerts hingegen ist eine Nachbedingung, die durch die Kovarianz verschärft wird.

Selbsttestaufgabe 2.2

Versuchen Sie, am Beispiel eines in einer Subklasse überschriebenen symmetrischen Getter-/Setter-Paars die Regeln zur Ko- und Kontravarianz bzw. zur abgeschwächten Vor- und verschärften Nachbedingung durchzuspielen. Was ist Ihr Lösungsvorschlag?

Kontravarianz ist unnatürlich

Bezeichnenderweise ist Bertrand Meyer, der Promoter des Design by contract schlechthin, ein großer Verfechter der kovarianten Parameterredefinition, also der Tatsache, dass eine Methode beim Überschreiben die Typen ihrer Parameter einschränken darf. Gerade dadurch wird aber die Vorbedingung, die durch die Parametertypen ausgedrückt wird, nicht aufgeweicht, sondern verschärft! Wie dieser Konflikt technisch aufgelöst werden kann, ist immer noch Gegenstand aktiver Forschung; dass so viel Energie hineingesteckt wird, ist dadurch gerechtfertigt, dass es so gut wie keine praktischen Beispiele gibt, in denen die Kontravarianz von Parametertypen sinnvoll (der Realität entsprechend) wäre. Die Lösung JAVAs, keine Parametervarianz zuzulassen (sog. No- oder *Invarianz*), ist vor diesem Hintergrund gar nicht mal die schlechteste; dagegen steht jedoch, dass die Kovarianz von Parametertypen in der Welt recht häufig vorkommt.

Selbsttestaufgabe 2.3

Überlegen Sie sich Beispiele für Ko- und Kontravarianz von Parametertypen und Rückgabewerten. Konstruieren Sie Ihre Beispiele so, dass sie in einer realen Anwendung (oder einem realen Problem) eine Entsprechung haben.

2.6 Spezifikationssprachen für das Design by contract

Klassische Spezifikationssprachen basieren auf Logik, genauer auf *Prädikatenlogik* erster Stufe. Nun ist Prädikatenlogik leider atemporal, d. h., sie hat keinen eingebauten Zeitbegriff. Das ist deswegen ein Problem, weil Verhalten mit Veränderung verbunden ist und sich Veränderung nun mal als Unterschied von vorher und nachher definiert. Insbesondere dann, wenn der Zustand (Wert einer Variable oder mehrerer Variablen) vorher einen Einfluss auf den Zustand nachher hat, muss man in der Spezifikation auf beide Werte zurückgreifen können.

Das mindeste, das eine Spezifikationssprache also braucht, ist eine Möglichkeit, auf den „alten“ Wert einer Variablen zurückgreifen zu können. Angelehnt an die *Spezifikationssprache Z* wird das häufig durch einen Strich (wie in x') getan.

Beispiel

```
209 void stepRight(Point p) {
210   p.x = p.x + 1;
211 }
212 // @post: p.x = p.x' + 1
```

(Man bemerkt einmal mehr, wie unglücklich die Wahl des Gleichheitszeichens als Zuweisungsoperator in JAVA, C, etc. ist. In der Nachbedingung soll es nämlich tatsächlich für Gleichheit stehen.) Auch auf den Rückgabewert einer Methode, falls vorhanden, sollte man aus einer Nachbedingung heraus zugreifen können.

Wie das obige Beispiel zeigt, gleichen Zusicherungen manchmal bis auf den Gleichheits-/Zuweisungsoperator Anweisungen. Trotz dieser Ähnlichkeit gibt es einen entscheidenden Unterschied zwischen den beiden: Zusicherungen beschreiben etwas, sie sind *deskriptiv* — Anweisungen schreiben hingegen vor, wie etwas zu verlaufen hat, sie sind *präskriptiv*.²⁶ Die obige Nachbedingung ist übrigens ein Beispiel für eine triviale (vgl. Abschnitt 2.2): Sie kann höchstens einen Fehler im Compiler oder im Annotationsprozessor feststellen.

**präskriptiv/
deskriptiv**

Um Problembeschreibung und -lösung nicht miteinander zu verquicken, sondern stattdessen größtmögliche Redundanz (die ja zur Fehleraufdeckung unverzichtbar ist) zu erzielen, wäre es sinnvoll, die dazugehörigen Sprachen klar voneinander zu trennen. Die Praxis hat jedoch gezeigt, dass Programmiererinnen sich schwertun, eine zusätzliche Spezifikationssprache zu lernen — es fällt ihnen schon schwer genug, überhaupt den Übergang von der präskriptiven zur deskriptiven Formulierung eines Problems bzw. seiner Lösung zu leisten. Die meisten praktischen Ansätze zur Instrumentierung des Design by contract sind deswegen darum bemüht, Zusicherungen soweit wie möglich in der Syntax der Programmiersprache auszudrücken, in die sie eingebettet sind. Der Preis dafür ist eben die Gefahr der Verquickung und der zu geringen Redundanz.

2.6.1 EIFFEL

Tabelle 2.2: Vertrag zwischen der Klasse `Dictionary` (Lieferantin) und seiner Benutzerin (Kundin)

	VERPFLICHTUNGEN	NUTZEN
KUNDIN	<p><i>muss Vorbedingung einhalten</i></p> <ul style="list-style-type: none"> • sicherstellen, dass <code>Dictionary</code> nicht voll ist • sicherstellen, dass der Schlüssel nicht Null ist 	<p><i>profitiert von Nachbedingung</i></p> <ul style="list-style-type: none"> • erhält <code>Dictionary</code>, in dem das Element unter dem Schlüssel eingetragen ist
LIEFERANTIN	<p><i>muss Nachbedingung einhalten</i></p> <ul style="list-style-type: none"> • sicherstellen, dass das Element unter dem Schlüssel in <code>Dictionary</code> eingetragen ist 	<p><i>profitiert von Vorbedingung</i></p> <ul style="list-style-type: none"> • muß sich nicht darum kümmern, wenn <code>Dictionary</code> voll oder der Schlüssel Null ist

Natürgemäß hat EIFFEL, die Programmiersprache von Bertrand Meyer [14], unter den objektorientierten Sprachen die Metapher des Design by contract am ehesten verinnerlicht. So wird der Vertrag über die Leistung „Element mit Schlüssel in Dictionary eintragen“, der, in Analogie zu Tabelle 2.1, in Tabelle 2.2 dargestellt ist, wie folgt in EIFFEL -Code übersetzt:

²⁶ Diese Unterscheidung zieht sich durch den gesamten Softwareentwicklungsprozess: Analysemodelle beispielsweise sind deskriptiv (beschreiben, was das Problem ist), Entwurfsmodelle hingegen sind präskriptiv (sie beschreiben die Lösung).

```

213 put (x: ELEMENT; key: STRING) is
214   -- Insert x so that it will be retrievable through key.
215   require
216     count < capacity
217     not key.empty
218   do
219     ... Some insertion algorithm ...
220   ensure
221     has (x)
222     item (key) = x
223     count = old count + 1
224 end

```

Wie man sieht, verwendet EIFFEL für den Zugriff auf alte Werte das Schlüsselwort **old**; für den Zugriff auf den Rückgabewert verwendet man **result**.

**Probleme des
Design by contract
in EIFFEL
nebeneffektvolle
Zusicherungen**

Es soll jedoch nicht verschwiegen werden, dass die Implementierung des Design by contract selbst in EIFFEL nicht ohne Probleme ist. So ist es zwar nicht erlaubt, wird aber durch den Compiler nicht verhindert, wenn in Zusicherungen Funktionsaufrufe (Methodenaufrufe) eingebaut werden, die Nebeneffekte haben.²⁷ Es macht dann einen Unterschied, ob das Programm mit oder ohne Überprüfung der Zusicherungen ausgeführt wird — ein Programm kann also im „gesicherten“ Modus (z. B. während der Testphase bei der Entwicklerin) funktionieren, im Produktivbetrieb aber (wenn die Prüfung der Zusicherungen aus Effizienzgründen ausgeschaltet ist) nicht mehr! Konstrukte, die eigentlich der Korrektheit eines Programms dienen, können somit selbst der Grund für Programmierfehler sein.

Ein weiteres Problem ergibt sich (wie so oft) aus der **Vererbung**, hier in Verbindung mit dem **Aliasing**: Meyer fordert,

1. dass die Invariante einer Klasse impliziter Bestandteil jeder Vor- und Nachbedingung dieser Klasse ist, insbesondere mit der Vor- und mit der Nachbedingung konjunktiv (mit logischem Und) verknüpft wird,
2. dass die Invarianten einer Klasse von jeder ihrer Subklassen eingehalten werden müssen (die Invarianten also höchstens verschärft werden dürfen) und
3. dass Vorbedingungen in Subklassen nicht verschärft und Nachbedingungen nicht gelockert werden dürfen.

²⁷ Eiffel unterscheidet zu diesem Zweck zwischen *Queries* (Abfragen) und *Commands*. Im Gegensatz zu Commands dürfen Queries den Zustand des Systems nicht verändern, weswegen nur sie in Zusicherungen verwendet werden sollten.

Zusammengenommen würde das aber heißen, dass Invarianten in Subklassen gar nicht geändert werden dürfen, denn sonst würde mit der allein möglichen Verschärfung auch die Vorbedingung jeder ihrer Methoden verschärft! Nun zählen die Invarianten eigentlich nicht zu den vertraglichen Verpflichtungen, die von Klientinnen vor der Nutzung einer Methode sichergestellt werden müssen (man denke etwa an Implementierungsvarianten, die zum Geheimnis einer Klasse gehören und die die Klientinnen gar nichts angehen). Es ist also eigentlich die Klasse selbst, die dafür sorgen muss, dass die Zustände ihrer Objekte ihren Invarianten genügen. (Der Grund, warum Invarianten vor externen Methodenaufrufen geprüft werden, war ja ein anderer: durch Aliasing ermöglichte Änderungen von Objekten, die außerhalb der Kontrolle der Zusicherungen einer Klasse stehen; s. Abschnitt 2.4.1 Zeitpunkt der Überprüfung von Zusicherungen.) Gleichwohl kann es zwischen schwächeren (oder gleichen) Vorbedingungen und stärkeren Invarianten zu Konflikten kommen, wie das folgende Beispiel zeigt.

**stärkere Invarianten
in Subklassen
kollidieren mit
schwächeren
Vorbedingungen**

Beispiel

Die Klasse **Quadrat** werde als Subklasse der Klasse **Rechteck** definiert mit der (zusätzlichen) Invariante, dass die beiden Seitenlängen, *a* und *b*, gleich sein müssen. Um der Regel der schwächeren Vorbedingung zu folgen, deklariert die Klasse Quadrat ihre Methode **setzes Seitenlängen(x, y)** ohne stärkere Vorbedingungen für die beiden Parameter. Wenn diese Methode nun für ein **Quadrat** mit zwei unterschiedlichen Seitenlängen aufgerufen wird, kommt es zu einer Verletzung der Invariante. Aus der Tatsache, dass die Invariante nicht Teil der Vorbedingung werden kann, folgt, dass **Quadrat** nicht Subklasse von Rechteck sein kann.

Als Pragmatiker, der er ist, bietet Meyer für dieses und ähnliche Probleme die folgende Lösung an. Man ersetze einfach in den betroffenen Zusicherungen den problematischen Ausdruck durch einen Funktionsaufruf mit dem Rückgabetyp Boolean. Die aufgerufene Funktion kann dann in jeder Klasse individuell so implementiert werden, dass sie deren Bedingung der Sachlage entsprechend (und ohne Berücksichtigung der Subklassenbeziehung) ausdrückt. Auf der Strecke bleibt zwar die logische Implikation, also dass die (eigentlich stärkere) Invariante der Subklasse die ihrer Superklasse impliziert bzw. die (eigentlich schwächere) Vorbedingung einer Methode der Subklasse von der ihrer Superklasse impliziert wird, dennoch wird die *Substituierbarkeit* zumindest dann nicht gefährdet, wenn die Klientinnen der Klasse (bzw. ihrer Subklassen) zur Überprüfung der Einhaltung der Vorbedingung selbst diese Funktion aufrufen (und den Dienst nicht in Anspruch nehmen, wenn die Funktion das nicht ausdrücklich erlaubt, sie also nicht erfüllt ist).

**Zusicherungen mit
polymorphem Inhalt
als Lösung**

Beispiel

Im gegebenen Beispiel könnte das so aussehen, dass sowohl **Rechteck** als auch **Quadrat** eine Methode **kantenlängeOK(x, y)** implementieren, die die Werte von *x* und *y* entsprechend prüft. Sowohl die Klassenvarianten als auch die Vorbedingungen zu **setzes Seitenlängen(x, y)** würden dann jeweils **kantenlängeOK** involvieren (wodurch beide in dieselbe Richtung — im gegebenen Fall verschärft — werden, was aber, wenn man sich nur die Zusicherungen ansieht, nicht offenbar wird).

Voraussetzung dafür, dass dies funktioniert, ist die Überprüfung der Vorbedingung auf Seiten der KlientInnen (an jeder Aufrufstelle); da diese aber nicht vom Compiler erzwungen werden kann, sind mögliche Laufzeitfehler nicht zu verhindern.

**Design by contract
in EIFFEL spezifiziert
das Klassen-
interface einer
Klasse**

Bedauerlicherweise hat EIFFEL kein von Klassen getrenntes Interfacekonzept — Interfaces müssen durch abstrakte Klassen emuliert werden. Ansonsten wäre es guter Stil, Vor- und Nachbedingungen sowie öffentliche (Klassen)Invarianten mit Interfaces zu spezifizieren; lediglich die internen Invarianten wären ein Spezifikum der implementierenden Klassen und sollten auch dort spezifiziert werden. So aber bezieht sich Design by contract immer auf das Klasseninterface einer Klasse.

2.6.2 JAVA

JAVA hat mit der Version 1.4 ein Schlüsselwort **assert** spendiert bekommen. Mit ihm können beliebige boolesche JAVA-Ausdrücke zu Zusicherungen (engl. assertions) gemacht werden. Allerdings können die Zusicherungen auf alle Methoden des Programms zurückgreifen und können somit dessen Ablauf entscheidend verändern, wie das folgende Beispiel zeigt:

Beispiel

```
225 boolean ping() {
226   assert pong();
227   return true;
228 }

229 boolean pong() {
230   assert ping();
231   return true;
232 }
```

Bei eingeschalteter Überprüfung der Zusicherungen terminiert das Programm mit einem Stapelüberlauf, obwohl (bei ausgeschalteter Überprüfung) die Zusicherungen erfüllt sind.

Auch kann die Auswertung einer Zusicherung am Zustand des Programms selbst etwas ändern, was wiederum Auswirkungen auf das Ergebnis der Auswertung haben könnte. Im Extremfall würde das bedeuten, dass beispielsweise eine Vorbedingung als erfüllt geprüft wird, sie aber bei ihrer Überprüfung (als Nebenwirkung) einen Zustand hergestellt hat, in dem sie gar nicht mehr erfüllt ist:

Beispiel

```
233 boolean isFirst = true;

234 void firstAccess() {
235   assert isFirst();
```

```

236 assert isFirst();
237 init();
238 }

239 boolean isFirst() {
240     boolean answer = isFirst;
241     isFirst = false;
242     return answer;
243 }

```

Hier muss die zweite Überprüfung derselben Vorbedingung scheitern, obwohl sich das logisch verbietet.

Anders als in EIFFEL ist die mangelnde Nebeneffektfreiheit in JAVA aber unerlässlich, wenn man in einer als Nachbedingung verstandenen Assert-Klausel einen alten Wert referenzieren will: Dann muss dieser nämlich als Nebenwirkung einer ebenfalls mit Hilfe von assert formulierten Zusicherung, die bereits zuvor (als Vorbedingung) überprüft wurde, explizit zwischengespeichert werden. Die JAVA -Dokumentation schlägt als Speicherort für solche Werte übrigens eine innere Klasse vor²⁸, deren Existenz dann jedoch nicht an das Ein- und Ausschalten der Überprüfung der Zusicherungen gebunden ist.

Nachbedingungen mit assert

Wen die stiefmütterliche Behandlung von Zusicherungen in JAVA wundert, die führe sich die folgende Einleitung zu Gemüte:

An assertion is a statement in the Java™ programming language that enables you to test your assumptions about your program. For example, if you write a method that calculates the speed of a particle, you might assert that the calculated speed is less than the speed of light.

Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error. By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors. (Quelle: s. Fußnote 28)

Das liest sich nicht gerade wie eine Werbeschrift für die Verwendung von Asserts oder gar für das Design by contract.

Da die in JAVA eingebauten Möglichkeiten der Zusicherung von Programmeigenschaften eher dürftig sind (außer dem Schlüsselwort **assert** ist da nichts), Zusicherungen aber auch beim programmgesteuerten Testen (insbesondere dem Unit-Testen; s. Kurseinheit 3) benötigt werden, hat man sich im Rahmen dessen eigene Gedanken gemacht. So wurde mit dem Unit-Testframework JUNIT die Klasse **Assert** eingeführt,

Entlehnungen aus dem Unit-Testen: Assert-Methoden

28 <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>

die eine ganze Reihe von (statischen) Methoden zur Verfügung stellt, die eigentlich dem Überprüfen von Testergebnissen dienen, die aber genauso gut zur Überprüfung beliebiger Bedingungen genutzt werden können (`assertTrue`, `assertFalse`, `assertEquals` etc.).²⁹ Und tatsächlich handelt es sich bei der Überprüfung des Testergebnisses ja auch um nichts anderes als um die Überprüfung einer Nachbedingung, nämlich der der Ausführung der getesteten Methode unter den Bedingungen des Tests. Der Verwendung von Assert-Methoden, die eigentlich zum Testen gedacht sind, im Rahmen des Design by contract steht also auch konzeptuell nichts entgegen. Da die Assert-Methoden aber alle keinen Rückgabewert haben und stattdessen bei ihrem Scheitern eine eigene Runtime-Exception werfen, können sie nicht in einem Assert-Statement verwendet werden.³⁰

HAMCREST-Bibliothek

Mit der Version 4.4 von JUNIT wurden dann die sprachlichen Möglichkeiten, Nachbedingungen für Tests auszudrücken, deutlich erweitert. Dazu wurde aus JMOCK, einem Framework, das eigentlich der Erzeugung von „unechten“ Hilfsobjekten zur Testung unter Laborbedingungen dient (vgl. Abschnitt 3.5), eine Bibliothek von Funktionen, die jeweils einfache logische Ausdrücke implementieren, die aber geschachtelt zu Ausdrücken erheblicher Komplexität kombiniert werden können, übernommen (die sog. HAMCREST -Bibliothek). Damit, und mit einer neuen, universellen Assert-Methode `assertThat(.,.)` mit dem zu testenden Objekt und einem Testausdruck als Argumenten, lassen sich dann Ausdrücke wie die folgenden formulieren:

```
244 assertThat(Objekt1, not(eq("Hello")));
245 assertThat(Objekt2, not(contains("Cheese")));
246 assertThat(Objekt3, or(contains("color"), contains("colour")));
```

Dabei ist die Methode `assertThat(.,.)` wie folgt implementiert:

```
247 protected void assertThat(Object something, Constraint matches) {
248     if (!matches.eval(something)) {
249         StringBuffer message = new StringBuffer("\nExpected: ");
250         matches.describeTo(message);
251         message.append("\nbut got : ").append(something).append("\n");
252         fail(message.toString());
253     }
254 }
```

`fail()` ist eine Methode aus dem JUNIT-Framework, deren Aufruf man aber auch durch das direkte Werfen einer entsprechenden Exception ersetzen kann. Wesentlich ist hier, dass es sich bei den Methoden `eq()`, `not()` usw. um Factory-Methoden (vgl. Abschnitt 4.4.6) handelt,

²⁹ Die Klasse `Assert` und ihre Methoden stammen noch aus der Zeit vor JAVA 1.4, zu der man auch `assert` noch als Methodenname verwenden durfte; mit der Einführung von `assert` als Schlüsselwort musste die so genannte Methode dann aber umbenannt werden.

³⁰ Seit JUNIT 4 kann man aber immerhin ein Assert-Statement anstelle einer Assert-Methode in einem Unit-Test verwenden (nur sollte die Ausführung von Asserts dann auch angeschaltet sein).

die jeweils ein neues Objekt vom Typ `constraint` zurückgeben, das die Methoden `eval()` und `describeTo()` ihrer Bedeutung entsprechend implementiert. Wenn Ihnen die Aufrufe dieser Methode fremd anmuten, liegt das am statischen Import, der es erlaubt, die Empfängerklasse (in diesem Fall die Klasse `Matchers`, in der alle Factory-Methoden hinterlegt sind) wegzulassen. Richtig interessant wird diese alternative Syntax aber erst dann, wenn man die Möglichkeit, seine eigenen Operatoren (auch Matcher genannt) zu definieren, nutzt; es besteht dann kein Unterschied mehr in der Verwendung von Standard- (wie den Und-, Oder- und Nicht-Operatoren, die ja in JAVA ihre eigenen Syntax haben) und speziellen, selbst definierten Operatoren. Ansonsten erinnern die Zwiebelschalen doch schon stark an LISP.³¹

2.6.3 JML

JML steht für JAVA MODELLING LANGUAGE und ist in erster Linie als Sprache konzipiert, die die formale, semantische (im Sinne von auf das Verhalten und nicht die Syntax abzielende) Spezifikation von JAVA -Interfaces erlaubt. Ihr Name drückt aus, dass der Hintergrund von JML zunächst ein anderer ist: der der Modellierung. JML soll erlauben, Elemente der mathematischen Modellierung von Software mit JAVA zu verbinden. Dabei ist JML viel mächtiger als vergleichbare Design-by-contract-Sprachen (einschließlich der von EIFFEL) — die grundlegende Syntax von JML ist aber, zumindest was die Wahl der Schlüsselwörter angeht, an EIFFEL angelehnt. [18] [19]

Die Spezifikation einer Methode, die die Quadratwurzel berechnen soll, sieht in JML wie folgt aus:

```

255 /*@ requires x >= 0.0;
256   @ ensures
257   @  JMLDouble.approximatelyEqualTo(x, \result * \result, 0.001);
258   @*/
259 public static double sqrt(double x) { ... }
```

Wie man hieran schon sieht, kommt JML mit einer eigenen Bibliothek von zur Spezifikation hilfreichen Funktionen, die aus Zusicherungen heraus aufgerufen werden dürfen. Die Syntax von Ausdrücken ist die Standardsyntax von JAVA; JML-eigene Schlüsselwörter wie `forall` oder `old` und spezielle Variablen wie `result` (für das Ergebnis einer mit Zusicherungen annotierten Methode) werden durch einen Backslash eingeleitet. Ähnlich wie in EIFFEL hat man Zugriff auf die Parameter einer Methode, für die die Vor- und Nachbedingungen gelten sollen:

³¹ Beinahe ironischerweise wird die HAMCREST-Bibliothek so ausgebaut, dass sich damit die aus SMALLTALK bekannten Collection-Methoden zum einfachen Iterieren (`select:`, `detect:` etc.; vgl. Kurs 01814) simulieren lassen. Dies ist deswegen ironisch, weil damit JUNIT, eine Entwicklung von zwei alten Smalltalkern (Kent Beck und Erich Gamma), über den Umweg von JMock und Assertions für die Erweiterung von JAVA um die Konstrukte, die die SMALLTALK-Gemeinde von Anfang an an JAVA vermisste, gesorgt hat. Unter Smalltalkern ist Unit-Testen aber gar nicht beliebt.

Beispiel

```

260 /*@ requires a != null
261   @  && (\forall int i; 0 < i && i < a.length; a[i-1] <= a[i]);
262   @*/
263 int binarySearch(int[] a, int x) {
264 // ...
265 }
```

Anders als in EIFFEL darf man in JML jedoch nur nebenwirkungsfreie Ausdrücke und Methoden (in EIFFEL auch *Queries* genannt; vgl. Fußnote 27) in seinen Zusicherungen verwenden. Die Verwendung von Zuweisungsoperatoren (in JAVA -Asserts der einzige Weg, alte Werte zwischenzuspeichern) ist also verboten. Da der JML-Compiler nicht selbst entscheiden kann, welche Methoden einer Klasse nebenwirkungsfrei sind, muss die Programmiererin sie durch einen speziellen Tag, *pure*, kennzeichnen. Alle Methoden, die diesen Tag nicht tragen, dürfen in Zusicherungen nicht aufgerufen werden. Genaugenommen ist JML damit die einzige sichere Design-by-contract-Sprache (selbst wenn Fehler auch hier, durch fahrlässige Vergabe von *pure*, leicht möglich sind).

Modellvariablen und -methoden

Der wichtigste Unterschied zwischen JML und EIFFEL ist aber ein anderer: JML erlaubt in seinen Spezifikationen die Einführung und Verwendung sog.

Modellvariablen und **-methoden** (daher auch der Name *JML*), also von Variablen und Methoden, die lediglich dem Ausdrücken von Zusicherungen dienen. Damit wird es möglich, vollständig abstrakte Spezifikationen zu schreiben, selbst wenn diese sich auf einen *Objektzustand* (Instanzvariablenwerte) beziehen, der nicht über entsprechende Methoden zugänglich gemacht wird.

Das nachfolgende **Beispiel** zeigt dies anhand der Spezifikation eines JAVA -Interfaces (das ja selbst keine Instanzvariablen vorsehen kann):

```

266 public interface SortedIntListType {
267   /*@ public model instance non_null JMLEqualsSequence theList;
268   @ public initially theList.isEmpty();
269   @ public instance invariant
270   @  (\forall int i; 0 <= i && i < theList.size());
271   @    theList.itemAt(i) instanceof Integer
272   @  && (\forall int i; 0 < i && i < theList.size());
273   @    ((Integer)theList.itemAt((int)(i - 1)))
274   @     .compareTo(theList.itemAt(i)) <= 0);
275   @*/
276
277 //@ ensures \result == theList.length();
278 /*@ pure */
279 int size();
```

279 /*@ requires 0 <= index && index < theList.length();

```

280  @ ensures
281  @ \result == ((Integer)theList.itemAt(index)).intValue();
282  @*/
283 /*@ pure */
284 int get(int index);

285 //@ ensures \result == theList.has(new Integer(elem));
286 /*@ pure */
287 boolean contains(int elem);

288 /*@ ensures
289  @ theList.isInsertionInto(\old(theList), new Integer(elem));
290  @*/
291 void add(int elem);
292 }
```

Man beachte, dass das Interface vollständig in terminis einer (gedachten) Repräsentation, **theList**, erfolgt, die jedoch selbst kein Teil des Interfaces ist, sondern eines Modells davon. So wird es möglich, die Semantik von Methoden wie **size** () zu spezifizieren, ohne dazu auf die (interne) Repräsentation einer konkreten, das Interface implementierenden Klasse zurückgreifen zu müssen. Die zu diesem Zweck eingeführte Modellinstanzvariable **theList** (Variablen in Interfaces sind normalerweise Klassenvariablen — deswegen die spezielle Kennzeichnung mittels **instance**) ist vom JML-eigenen Typ **JMLEqualsSequence** und trägt den Zusatz **non_null**, der ausdrückt, dass die Variable nie den Wert **null** haben darf. Die Zusicherungen erfolgen allesamt durch Zugriff auf (die Implementierung von) **JMLEqualsSequence**, also gewissermaßen auf eine Referenzimplementierung, deren Korrektheit vorausgesetzt wird. Die konkreten Klassen, die das Interface **SortedIntListType** implementieren, können sich von dieser Implementation vollständig lösen; die Einhaltung der Verträge von **SortedIntListType** wird trotzdem — und unabhängig von der eigenen Implementierung — überprüft.

Die Sache hat jedoch noch einen kleinen Schönheitsfehler: In der obigen Spezifikation erhält **theList** niemals einen Wert und Elemente werden auch keine eingefügt. Die Invariante **non_null** und die Nachbedingung zu **add** können damit aber nicht erfüllt werden, es sei denn, die konkreten Implementierungen des Interfaces erhielten Zugriff auf **theList** und würden es entsprechend pflegen. Dann müssten sich die Implementierungen aber um zwei interne Repräsentationen kümmern, die eigene und die des Modells. Abgesehen von dem Mehraufwand besteht dabei keine Garantie, dass die Pflege der eigenen Repräsentation der des Modells entspricht, so dass die sich auf das Modell beziehenden Invarianten zwar erfüllt sein könnten, die eigene Implementierung sie aber verletzt (was aufgrund der Tatsache, dass sich die Zusicherungen nur auf das Modell beziehen, gar nicht auffiele).

Um dies zu verhindern, sieht JML den Begriff der Abstraktionsfunktion vor, die die (interne) Repräsentation einer konkreten Implementierung auf die des Modells abbildet. Bei der Implementierung einer konkreten Klasse, die die mit dem Interface verbundene Spezifikation erfüllen soll, muss man dann eine solche Funktion angeben. Sie gestattet es, vor der Überprüfung der Zusicherungen die Modellrepräsentation zu

**Synchronisierung
von Modell und
Implementierung**

aktualisieren. Die folgende Implementierung des Interfaces `SortedIntListType` durch einen binären Baum verdeutlicht das Vorgehen:

```

293 public class SortedIntList implements SortedIntListType {
294     private boolean isEmpty;
295     private int val;
296     private SortedIntList left, right;

297     //@ private invariant isEmpty == (left == null && right == null);

298     //@ private represents theList <- abstractValue();

299     /*@ private model pure JMLEqualsSequence abstractValue() {
300         @ JMLEqualsSequence ret = new JMLEqualsSequence();
301         @ if (!isEmpty) {
302             @ ret = left.abstractValue().insertBack(new Integer(val))
303             @     .concat(right.abstractValue());
304         @ }
305         @ return ret;
306     @ }
307     @*/
308
309     //@ ensures theList.isEmpty();
310     public SortedIntList() { isEmpty = true; }

311     public void add(int elem) { ... }
312 }
```

In Zeile 298 wird festgelegt, wie der Wert der Modellvariable `theList` definiert ist: durch Berechnung der Abstraktionsfunktion, vertreten durch die Modellmethode `abstractValue()`. Diese traversiert den Binärbaum in einer Links-rechts-Tiefensuche und macht daraus ein Repräsentationsobjekt gleichen Inhalts, aber vom Typ `JMLEqualsSequence`. Wie man sieht, hat die abstrakte Spezifikation (genauer: die Überprüfung von deren Einhaltung) ihren Preis.

Entkopplung der Spezifikation bei konkreten Klassen

Ein weiterer Nutzen von Modellvariablen und -methoden ist, dass die Spezifikation des Verhaltens einer konkreten Klasse von deren Umsetzung durch die Klasse getrennt werden kann. Wenn beispielsweise eine Vorbedingung auf eine Variable `x` Bezug nimmt, die aber aus irgendeinem Grund in eine Variable `y` überführt werden oder ganz verschwinden muss, kann man `x` immer noch als Modellvariable weiterpflegen. Deren Inhalt muss dann mittels einer entsprechenden Abstraktionsfunktion auf dem Laufenden gehalten werden.

2.6.4 Grenzen der Ausdrucksstärke

Prädikatenlogik hat gegenüber den hier vorgestellten Sprachen zur Formulierung von Zusicherungen den Vorteil, dass sie nicht der Anforderung der maschinellen Ausführung genügen muss. Die Ausführbarkeit erlegt den Sprachen gewisse Beschränkungen auf, die zu Lasten der Ausdrucksstärke gehen. So lässt sich zum Beispiel die Nachbedingung

$$\text{remove}(\text{put}(s, x)) = s$$

der Klasse `Stack` für die Methode `remove` in den gängigen Design-by-contract-Sprachen nicht ausdrücken. Das sollte eine allerdings nicht davon abhalten, diese Nachbedingung dennoch im Programm zu erwähnen, wenn auch nur als Kommentar — liefert sie doch eine gute Basis für das Schreiben eines Testfalls (vgl. dazu Selbsttestaufgabe 3.1 und Abschnitt 3.2.4).

2.7 Design by contract als Form des Testens

In den heute üblichen Instrumentierungen des Design by contract werden ja erst während der Programmausführung Vorbedingungen, Nachbedingungen sowie Invarianten überprüft und entsprechende Fehlermeldungen ausgelöst, wenn sie verletzt werden sollten. Die Verletzung einer solchen Zusicherung weist nach, dass im Programm ein Programmierfehler (oder ein Fehler in der Formulierung der Bedingung) vorliegt. Ähnlich wie das Testen (Kurseinheit 3) kann diese Verfahrensweise jedoch nur durch vollständiges Durchspielen aller möglichen Fälle nachweisen, dass ein Programm in Hinsicht auf die gegebene Spezifikation korrekt ist.

Tatsächlich handelt es sich bei dieser Art der Anwendung des Design by contract in der Programmierung lediglich um eine etwas fortschrittlichere Form des Testens, bei der das *Testorakel* (also was das Programm auf eine gegebene Eingabe hin liefern soll) in Form der Formulierung der Nachbedingung direkt ins Programm eingebaut ist. Die Definition der Testfälle und damit die Sicherstellung, dass die fehlerprovokierenden Eingaben auch gemacht werden bzw. Situationen auch zustande kommen, bleiben jedoch zunächst weiter in der Verantwortung der Testerin.

Damit ist das Potential von Design by contract jedoch noch nicht ausgeschöpft: Mögliche Testfälle lassen sich nämlich aus den Vorbedingungen ableiten. Und so gibt es beispielsweise mit JCONTRACT ein kommerzielles Werkzeug der Firma PARASOFT, mit dem sich nicht nur Zusicherungen à la Design by contract (ähnlich wie in JML) in JAVA -Programme einbauen und zur Laufzeit prüfen, sondern auch aus den Zusicherungen automatisch Tests generieren lassen, die versuchen zu zeigen, dass die Zusicherungen (nicht) eingehalten werden. Dazu werden mittels JTEST, einem Werkzeug des gleichen Herstellers, die Design-by-contract-Klauseln interpretiert und daraus (nach einem nicht näher beschriebenen Verfahren) Testfälle abgeleitet, die angeblich echte Funktionstests (sog. *Black-Box Tests*) ermöglichen.

Ein (vermutlich) anderer Testansatz auf Basis von Zusicherungen wurde übrigens im Zusammenhang mit JML entwickelt: Da in einem korrekten Programm bei Einhaltung der Vorbedingungen weder Nachbedingungen noch Invarianten verletzt werden dürfen, kann man das Testorakel, also die Vorgabe des richtigen Ergebnisses, durch die Prüfung auf das Nichtauftreten von Zusicherungsverletzungen ersetzen [20] : Wenn keine Zusicherungen verletzt wurden, dann war das

Ergebnis wohl korrekt (oder die Spezifikation zu unspezifisch). Wird hingegen eine Invariante oder Nachbedingung verletzt, dann kann bei der Berechnung des Ergebnisses irgendetwas nicht gestimmt haben, weswegen man diesem nicht trauen sollte (auch wenn man nicht weiß, welches denn das richtige Ergebnis gewesen wäre). Dies gilt übrigens auch für die Verletzung von Vorbedingungen, wenn diese zu Methoden gehören, die nur indirekt durch den Test aufgerufen wurden, da die aufrufende Methode die Verantwortung für die Einhaltung aller Vorbedingungen der von ihr aufgerufenen Methoden trägt. Verletzungen der Vorbedingungen unmittelbar beim Testaufruf hingegen besagen lediglich, dass der Testaufruf ungültig war — das Ergebnis ist also ohne Bedeutung.

Es bleibt wiederum die Frage, wo die Daten (Empfänger- und Parameterobjekte), anhand derer die Tests durchgeführt werden, herkommen sollen. Diese können im Prinzip zufällig generiert werden: Wenn die Vorbedingungen nur streng genug formuliert sind, werden alle ungültigen Eingaben zurückgewiesen (per Verletzung der Vorbedingungen). Ein etwas informierteres Verfahren permutiert als Eingaben alle Objekte aus allen bestehenden Testsuiten (*Test fixtures*; s. Abschnitt 3.2.1) typgerecht durch.

2.8 Vor- und Nachteile des Design by contract

In einem vieldiskutierten Artikel zum Absturz des Jungfernfluges der Ariane-5-Rakete behauptet Bertrand Meyer, die Katastrophe hätte verhindert werden können, wenn die Steuerungssoftware nach dem Prinzip des Design by contract programmiert worden wäre [21]. Sicher gibt es viele gute Gründe, die für eine genaue Spezifikation der Funktion von Methoden sprechen. Dabei hilft die Metapher des Design by contract, festzulegen, was genau zu spezifizieren ist, und die Last, die Spezifikationen einzuhalten, auf Nutzer und Erbringer einer Funktionalität gleichermaßen zu verteilen. Jedoch ist dieser Ansatz nicht ohne praktische Probleme.

1. Die Überprüfung der Einhaltung der Spezifikation ist alles andere als trivial. So kann zwar bei Aufruf einer Methode festgestellt werden, ob die Vorbedingung eingehalten wurde, und entsprechend bei Beendigung der Methode überprüft werden, ob sie die Nachbedingung erfüllt, aber diese Überprüfung erfolgt eben immer nur fallweise und kann — genau wie das Testen (Kurseinheit 3) — praktisch nie die vollständige Korrektheit einer Implementierung beweisen, denn dazu müssten erst alle möglichen Aufrufe durchgespielt werden. Derer sind es in der Regel aber zu viele. Es wäre zwar theoretisch auch möglich, per formaler Verifikation zu beweisen, dass die Methode die Vorbedingung in die Nachbedingung überführt, aber formale Verifikationsverfahren sind immer noch höchst rechenaufwendig und schon deshalb in der Praxis kaum im Einsatz.
2. Vor- und Nachbedingungen richtig zu formulieren ist eine Kunst für sich. Jede, die sich einmal darin versucht hat, weiß, wie schwierig es ist, eine formale Spezifikationen selbst eines einfachen Sachverhalts auf Anhieb richtig hinzubekommen. Ganz und gar ausgeschlossen scheint es, formale Spezifikationen ohne entsprechende Werkzeugunterstützung im großen Stil (routinemäßig) einzusetzen — wer will durch Hinsehen sagen können, ob eine logische Formel, die sich über mehrere Zeilen erstreckt, das Richtige ausdrückt? Tatsächlich müssen Vor- und Nachbedingungen debugt werden, was sie nicht viel besser macht als die Pro-

gramme, deren Richtigkeit sie sicherstellen sollen. Es läuft also in der Praxis darauf hinaus, zwei potentiell fehlerhafte Artefakte, die Spezifikation und das dazugehörige Programm, durch wiederholte Programmläufe aneinander anzulegen, wobei selbst dann nicht garantiert ist, dass beide hinterher das Richtige tun. Dennoch ist natürlich die Formulierung von Vor- und Nachbedingungen grundsätzlich dazu geeignet, Fehler aufzudecken, und deswegen auch nützlich. Nur Garantien gibt sie keine.

3. Spätestens seit Barry Boehms Auflistung der zehn größten Risiken der Softwareentwicklung ([22] ; s. Abschnitt 7.11) weiß man, dass die Entwicklung der falschen Anforderungen ein gravierendes Problem darstellt. Für die Praxis heißt das: Selbst wenn es einer gelungen ist, verbal formulierte Anforderungen korrekt in eine formale Spezifikationen zu überführen, ist damit noch lange nicht gesagt, dass damit die tatsächlichen Anforderungen der Kundinnen erfüllt werden. Vielmehr sind häufig schon diese Anforderungen falsch formuliert. (Den Prozess, die Anforderungen auf ihre Richtigkeit, d. h. im Wesentlichen die tatsächlichen Bedürfnisse der Kundinnen, hin zu überprüfen, nennt man *Validierung* oder *Validation*. Den Prozess der Überprüfung, ob die Anforderungen durch das Programm richtig umgesetzt wurden, nennt man hingegen *Verifikation*.)

The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.

FP Brooks „No Silver Bullet“ in: *The Mythical Man Month* (Jubiläumsausgabe, 1995).

2.9 Zusammenfassung und Ausblick

Das Design by contract ist geeignet, die Schwachstelle der Interfacespezifikationen à la JAVA und C# zu beheben: Vor- und Nachbedingungen können die Deklaration von Methodensignaturen (Syntax) um die Spezifikation von Verhalten (Semantik) ergänzen. Die Implementierung eines Interfaces muss dann lediglich sicherstellen, dass sie die Zusicherungen einhält, genauer, dass sie — genau wie bei der Vererbung zwischen Klassen — die Vorbedingungen nicht verschärft und die Nachbedingungen nicht abschwächt.

Derzeitige Instrumentierungen des Design by contract erlauben die Überprüfung der Einhaltung von Zusicherungen nur zur Laufzeit. Verletzungen werden angezeigt und führen regelmäßig zu einem Programm- oder zumindest Funktionsabbruch, der meistens als Aufforderung zur Nachbesserung verstanden werden muss. Während dieses Verhalten während der Testphase hochwillkommen ist, ist es im Produktivbetrieb auch nicht viel besser als ein gewöhnlicher Programmabsturz. Was man stattdessen bräuchte, ist ein Verified design by contract, also ein formaler Beweis, dass eine Verletzung von Zusicherungen nicht auftreten kann.

Korrektheitsbeweise von Programmen sind natürlich nach wie vor ein großes Thema und Gegenstand aktiver Forschung. Selbst Firmen wie MICROSOFT haben ein vitales Interesse daran, ihre Software so fehlerfrei wie möglich zu machen. Tatsächlich arbeiten dort inzwischen hochkarätige Teams (einschließlich Tony Hoare höchstselbst) unter anderem daran, Erkenntnisse aus der Theorie in die Programmierpraxis, also insbesondere in Werkzeuge für gängige Programmiersprachen, einzubringen.

Bis aber etwas wie Verified design by contract für die Allgemeinheit zur Verfügung steht, müssen wir uns mit (abschaltbaren) Überprüfungen der Zusicherungen zur Laufzeit und (automatischer) Ableitung von Testfällen aus den Zusicherungen begnügen.

2.10 Weiterführende Literatur

Das Hauptwerk zum Thema Design by contract ist sicherlich das Buch von Meyer [14] , selbst wenn die Grundideen viel älter sind und in unzähligen Werken aufbereitet werden. Wer Freude an Originalliteratur hat, kann natürlich auch die Arbeiten von Hoare et al. lesen ([15] etc., gibt's im Internet).

Die Ausdehnung und tiefergehende Behandlung der Idee von Verträgen (Contracts) in der Programmierung kann man auch in [23] [24] [25] nachlesen.

- [13] Building bug-free O-O software: An introduction to Design by Contract(TM) (<http://archive.eiffel.com/doc/manuals/technology/contract/>)
- [14] B Meyer Object-Oriented Software Construction 2. Ausgabe (Prentice Hall 1997).
- [15] CAR Hoare „An axiomatic basis for computer programming“ CACM 12:10 (1969) 576–580.
- [16] Enseling „ICONTRACT: Design by Contract in JAVA“ JAVA World 2001.
- [17] R Kramer „ICONTRACT - the JAVA™ design by contract™ tool“ in: TOOLS 26: Technology of Object-Oriented Languages and Systems (1998) 295–307.
- [18] GT Leavens AL Baker, C Ruby "JML: A notation for detailed design" in: H Kilov, B Rumpe, I Simmonds Behavioral Specifications of Businesses and Systems (Kluwer Academic Publishers 1999) 175–188.
- [19] GT Leavens, Y Cheon Design by Contract with JML (<ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf>)
- [20] Y Cheon, GT Leavens „A simple and practical approach to unit testing: the JML and JUNIT way“ ECOOP (2002) 231–255.
- [21] JM Jézéquel, B Meyer „Design by contract: the lessons of Ariane“ IEEE Computer 30:1 (1997) 129–130.
- [22] B Boehm „Software risk management: principles and practices“ IEEE Software 8:1 (1991) 32–41.
- [23] IM Holland „Specifying reusable components using contracts“ in: Proceedings of the 6th European Conference on Object Oriented Programming (Springer-Verlag 1992) 287–308.
- [24] A Beugnard, JM Jézéquel, N Plouzeau, D Watkins „Making components contract aware“ IEEE Computer (1999) 38–45.
- [25] RB Findler, M Felleisen „Contract soundness for object-oriented languages“ in: OOPSLA'01 (2001) 1–15.

2.11 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 2.1 (Seite 59)

Vorbedingung für die Methode **push**: Der Stack darf nicht voll sein.

Nachbedingung: Der Stack ist nicht (mehr) leer, er kann sogar voll sein, seine Größe ist um 1 angewachsen, das mit **push** auf den Stack gelegte Element liegt oben, ansonsten hat sich nichts verändert.

Vorbedingung für die Methode **pop**: Der Stack darf nicht leer sein.

Nachbedingung: Der Stack ist nicht (mehr) voll, er kann sogar leer sein, seine Größe ist um 1 geschrumpft, das zuletzt mit **push** auf den Stack gelegte Element liegt nicht mehr darauf.

Vorbedingung für die Methoden **top**: Der Stack darf nicht leer sein.

Nachbedingung: Das zuletzt mit **push** auf den Stack gelegte Element wird zurückgegeben, der Stack hat sich nicht verändert.

Selbsttestaufgabe 2.2 (Seite 66)

Angenommen, Sie haben die folgende Situation

```
313 class A {  
314     private X x;  
315     public void setX(X x) { this.x = x; }  
316     public X getX() { return x; }  
317 }
```

```
318 class Y extends X {...}
```

```
319 class B extends A {  
320     private Y x;  
321 }
```

Wie wollen Sie dann die Getter/Setter für **x** in B implementieren? Der einzige Ausweg ist die „In-“ oder „Novarianz“: **x** muss seinen Typ beibehalten (und damit auch die Parameter).

Selbsttestaufgabe 2.3 (Seite 66)

Für die Kontravarianz von Parametertypen fällt mir kein praktisch sinnvolles Beispiel ein. Wenn Sie eines haben, immer her damit!

Ein Beispiel für die Kovarianz von Parametertypen liegt vor, wenn ein spezialisiertes (d. h., von einem anderen abgeleitetes) Subjekt sich selbst an einen zu ihm passenden spezialisierten Observer übergibt (vgl. Selbsttestaufgabe 4.6).

3 Unit-Testen

Program testing can be used to show the presence of bugs,
but never to show their absence!

Edsger W. Dijkstra „Structured Programming“ EWD268 (1969).

Im Prinzip erlaubt das Design by contract bereits eine sehr komfortable Form des Testens: Wenn die Überprüfung der Zusicherungen eingeschaltet ist, wird in jedem Programmlauf, egal ob zu Testzwecken oder im Routinebetrieb, ständig geprüft, ob bestimmte Eingaben (ausgedrückt durch die Vorbedingungen) zu bestimmten Ausgaben (ausgedrückt durch die Nachbedingungen) führen. Anders als beim konventionellen Testen, bei dem jeder Testfall einzeln und vorab spezifiziert werden muss, wird einfach jede Eingabe, die da kommt, genommen und gegen ein berechnetes Ergebnis geprüft. Dabei gibt es jedoch mehrere Probleme:

1. Es gibt keine Garantie, dass alle Fälle — oder auch nur ein bestimmter Fall — durchgespielt werden. Im schlimmsten Fall weiß man also nicht, ob ein bestimmter, problematischer Pfad im Programm überhaupt ausgeführt wurde.
2. Die Möglichkeiten des Design by contract, das erwartete Ergebnis als Funktion der Eingabe darzustellen, sind praktisch beschränkt (siehe das Beispiel zu Abschnitt 2.6.4 Grenzen der Ausdrucksstärke).
3. Wenn die Überprüfung der Zusicherungen einer Klasse auf deren Funktionen (Methoden mit Rückgabewerten) zurückgreift, dann können Fehler verborgen bleiben, z. B. wenn die Korrektheit einer Methode durch die Methode selbst zugesichert wird.

Separat erstellte (d. h. von der tatsächlichen Verwendung eines Programms unabhängige) Testfälle hingegen können so entworfen werden, dass sie die Ausführung eines bestimmten Pfads im Programm erzwingen. Außerdem hat man, wenn der Testfall selbst als Programm implementiert wurde, die Möglichkeit, die Korrektheit der Ausgaben durch beliebige Verfahren (anstelle der Auswertung der logischen Ausdrücke in Form von Vor- und Nachbedingungen) zu überprüfen. Das Schreiben expliziter Testfälle kann also die Möglichkeiten des Design by contract wirkungsvoll ergänzen.

3.1 Der Begriff des Testfalls

Unter einem **Testfall** versteht man die Spezifikation einer bestimmten Eingabe und der dazu von einem Programmteil oder ganzen Programm erwarteten Ausgabe. So besteht beispielsweise ein Testfall für die Nachfolgefunktion aus der Zahl 0 als Eingabe und der Zahl 1 als Ausgabe. Testfälle prüfen die Korrektheit des Programms oder seines Teils, indem es mit den spezifizierten Eingaben aufgerufen und sein Ergebnis mit dem erwarteten verglichen wird. Liefert beispielsweise die Nachfolgefunktion für die Eingabe 0 ein anderes als das erwartete Ergebnis, 1, ist sie fehlerhaft.

Testfälle sind zunächst nicht mehr als hingeschriebene partielle Spezifikationen (der Ein- und Ausgabe) eines Programms. Um ein Programm mithilfe von Testfällen zu überprüfen, kann man es von Hand mit den Eingaben der Testfälle füttern und das Ergebnis ebenfalls von Hand mit der erwarteten Ausgabe vergleichen. Wenn ein solcher Test nur einmal oder nur sehr selten durchgeführt werden soll, ist das ausreichend (und aufgrund des geringen Aufwands sogar sinnvoll). In der Praxis wird man aber die Tests, selbst wenn sie einmal erfolgreich waren, immer wieder durchführen wollen, ganz einfach um zu sehen, ob das Programm „noch läuft“ — dies ist nicht nur nach Wartungsarbeiten sinnvoll, sondern auch schon, wenn sich an der Umgebung etwas (z. B. die Version des Laufzeitsystems) geändert hat. Solche wiederholten Tests nennt man **Regressionstests**; sie sind idealerweise automatisiert.

Regressionstests

Es stellt sich dann die Frage, wie man die Ein- und Ausgaben der Testfälle für eine automatische Überprüfung hinterlegt. Ist die Zahl der Testfälle für eine bestimmte Funktion des Programms groß, wird man dazu eine Datenbank o. ä. verwenden wollen und die Funktion mit deren Einträgen der Reihe nach testen. In der Regel ist aber die Zahl der substantiell verschiedenen Testfälle einer Funktion (so verschieden, dass man mit einem Versagen in einem Fall und einem Gelingen in allen anderen rechnen kann) nicht sonderlich groß, so dass der Aufwand der Verwendung einer Datenbank nicht gerechtfertigt erscheint. Zugleich wird durch die Beschränkung auf Werte, die in einer Datenbank hinterlegt werden können, nicht jeder zu testenden Funktion Rechnung getragen: Nicht selten verlangt die Überprüfung des richtigen Ergebnisses mehr als den Vergleich mit einem vorgegebenen Wert.

Hinterlegung von Testfällen

Selbsttestaufgabe 3.1

Überlegen Sie sich, wie Sie die Methode zur Bestimmung der Quadratwurzel aus Abschnitt 2.2 Ein paar einfache Beispiele(Programmzeilen 196 -204) testen würden. Können Sie mit demselben Verfahren auch eine Implementierung der Klasse **Stack** testen?

Wenn die Zahl der zum Testen einer Funktion herangezogenen Ein-/Ausgabe-kombinationen gering ist oder wenn die Ausgabe sich nicht durch eine einfache Zielwertvorgabe überprüfen lässt, empfiehlt es sich, den Testfall zu programmieren. Damit wird aber faktisch ein Programm geschrieben, das ein anderes testet, ein Aufwand, den so manche scheut. Um diesen Aufwand möglichst gering zu halten, wurden sog. **Testframeworks** entwickelt; sie erlauben es, den Programmieraufwand für einen Testfall weitgehend auf Spezifizieren der Eingabe, Aufruf der zu testenden Funktion (Methode) und Vergleich mit dem erwarteten Ergebnis (das zu diesem Zweck ebenfalls spezifiziert werden muss) zu reduzieren. Die Verwaltung der Testfälle und deren Ausführung sowie die Protokollierung und Anzeige der Ergebnisse wird der Testerin vom Framework abgenommen. Die Beschäftigung mit einem dieser Frameworks, JUNIT, wird einen Gutteil des Rests dieser Kurseinheit ausmachen.

programmierte Tests und Testframeworks

Es bleibt die Frage, wie man geeignete Testfälle auswählt. Dazu gibt es eine einfache Regel:

Auswahl der Testfälle

Ein guter Testfall ist einer, der einen Fehler entdeckt.

Um dieser Regel zu genügen, muss man ein Gespür dafür haben, wo die zu testende Funktion versagen könnte. Neben typischen Eingabewerten sind dies insbesondere Grenzfälle (also größter und kleinster Wert oder Werte am Übergang einer Fallunterscheidung, wobei man für letztere mitunter schon Kenntnis der Implementierung haben muss). Prinzipiell kann ein Fehler aber überall lauern und der eingangs zitierte Leitsatz des Testens lässt sich durch noch so geschickt gewählte Testfälle nicht aushebeln.

Aus Sicht der Testerin ist es also positiv, wenn ihr Test einen Fehler entdeckt. Es folgt, dass Testen eine destruktive Tätigkeit ist; entsprechend niedrig ist seine Reputation. Mehr dazu, und wie man das schlechte Image des Testens mit einem einfachen Handgriff aufpolieren kann, in Abschnitt 7.3.

3.1.1 Was testet ein Testfall?

Wie der Name bereits nahelegt, testet man beim Unit-Testen die Einheiten eines Programms. Im Fall der objektorientierten Programmierung sind diese Einheiten konkret Methoden, Klassen oder im Extremfall auch ganze Pakete. Die getestete Eigenschaft der Einheit ist die Korrektheit im Sinne der Erfüllung der Abbildung, die durch die Spezifikation der Ein- und Ausgabewerte vorgegeben wird.

Wie werden Testfälle den zu testenden Einheiten zugeordnet?

Grundsätzlich reicht ein Testfall nicht aus, um eine funktionale Einheit (in der Regel eine Methode) ausreichend zu testen. Entsprechend gilt natürlich für Klassen und ganze Pakete, dass zum umfassenden Testen immer mehrere Testfälle nötig sind. Man könnte also erwarten, dass die Beziehung von Einheiten zu Testfällen $1:n$ ist.

Nun testet ein Testfall nur im Idealfall genau eine Einheit. In der Praxis werden dagegen durch einen Testfall häufig, direkt oder indirekt, mehrere Einheiten eines Programms gleichzeitig getestet. Dies ergibt sich aus der Tatsache, dass Funktionen (Methoden) in der Regel andere aufrufen und von deren Korrektheit abhängen, und gilt natürlich umso mehr, je umfassender die zu testende Funktion bzw. der dazu aufzubauende Apparat ist. Daraus ergibt sich wiederum, dass die Beziehung von getesteten Einheiten zu Testfällen im allgemeinen $m:n$ ist. Dies macht die Zuordnung eines fehlgeschlagenen Testfalls zu seiner Fehlerquelle nicht immer ganz einfach und nicht selten löst das Fehlschlagen eines Testfalls eine längere Suche nach der Ursache aus.

Beispiel

So testet z. B. Testfall 1 in Gestalt von

```
322 Stack stack = new Stack();  
323 assert stack.isEmpty();
```

sowohl den Konstruktor von `Stack` als auch dessen Methode `isEmpty()` und Testfall 2 in Gestalt von

```
324 Stack stack = new Stack();
325 stack.push(new Element());
326 assert ! stack.isEmpty();
```

neben Konstruktor und `isEmpty()` auch noch die Methode `push()`.

Aus dieser Mehrdeutigkeit der Beziehungen ergibt sich wiederum das Problem der Organisation von Testfällen. Die enge inhaltliche Verbindung von Testfällen und getesteten Funktionen würde wünschen lassen, dass die Organisation diese Verbindung widerspiegelt, dass also die Ordnung der Tests der Ordnung der getesteten Software gleicht. Dies ist jedoch bei einer echten $m:n$ -Beziehung nicht möglich. Stattdessen ist die strukturelle Organisation der Testfälle häufig ein eigenes Entwurfsproblem und bedarf unter anderem der gelegentlichen Refaktorisierung (Kurseinheit 5).

3.1.2 Organisation von Testfällen

Je komplexer die zu testende Funktion ist, desto mehr Testfälle wird man in der Regel formulieren wollen, um möglichst viele potentielle Fehler auszuschließen. Schon aus diesem Grund ist es sinnvoll, Testfälle zu größeren Einheiten, sog. **Testsuites**, zusammenzufassen. Dabei ist jedoch zu beachten, dass die Tests voneinander unabhängig sein müssen, so dass sie sich in ihrem Ausgang nicht gegenseitig beeinflussen können. Daraus folgt, dass das Ergebnis jeder Testsuite stets unabhängig von der Reihenfolge ist, in der ihre Tests ausgeführt werden.

Wie können Tests als Programm organisiert werden?

Wenn die Testfälle selbst programmiert werden (also in Form eines Programms vorliegen) sollen, sind die Möglichkeiten der (programmtechnischen) Organisation durch die verwendete Programmiersprache vorgegeben. In JAVA beispielsweise stehen hierfür dieselben sprachlichen Mittel wie für die getesteten Einheiten zur Verfügung, nämlich Methoden, Klassen und Pakete. Doch während einzelne Testfälle als Methoden implementiert werden können, stellt sich die Frage, welche Rollen Klassen beim Testen spielen sollen — die Klassen der zu testenden Objekte existieren ja bereits und nur für Testfälle braucht man in der Regel keine speziellen Objekte, Klassen oder Pakete. Und so verwendet JUNIT, eines der am besten bekannten Frameworks zum Unit-Testen in JAVA, Klassen lediglich als ein Hilfsmittel, das die Testerin davor bewahrt, eigene Datenstrukturen für die Organisation von Testfällen sowie, was schwerer wiegt, einen eigenen Interpreter für deren Ausführung definieren zu müssen.

3.2 JUNIT

JUNIT ist gleich aus mehreren Gründen interessant. Zum einen stellt es ein recht einfaches und zugleich wirksames Werkzeug zum Unit-Testen von JAVA-Programmen dar. Da es frei zugänglich ist, verursacht sein Einsatz über den Einarbeitungsaufwand hinaus praktisch keine Kosten. Zum anderen wurde JUNIT von zwei Stars der objektorientierten Programmierung entwickelt (Kent Beck und Erich Gamma) und stellt eine Art Lehrstück derselben dar. Insbesondere kommen im Design von JUNIT eine ganze Menge sog. *Entwurfsmuster* zur Anwendung, einem Konzept, das von den beiden Autoren von JUNIT wesentlich geprägt wurde. Entwurfsmuster werden in Kurs-

einheit 4 behandelt; wer sie noch nicht kennt und beim Lesen dieses Abschnitts deswegen Schwierigkeiten hat, möchte vielleicht diese Kurseinheit vorziehen. Aus Gründen der inhaltlichen Kohärenz (von Interfaces zum Design by contract zum Unit-Testen) habe ich jedoch diese Reihenfolge gewählt.

Es soll daher zunächst der innere Aufbau von JUNIT untersucht werden. Wer sich lieber zuerst ein Bild von dessen Anwendung machen möchte (oder zwischendurch den Faden verliert), der sei auf Abschnitt 3.2.2 verwiesen.

3.2.1 Interner Aufbau des JUNIT-3.8-Frameworks

Das Hauptproblem, das beim Entwurf eines Testframeworks wie JUNIT zu lösen ist, ist, wie man die einzelnen Testfälle hinterlegt. Als kleinste Einheiten eines Programms bieten sich hierfür Methoden an: Sie erlauben es, zunächst ein Objektgeflecht aufzubauen, auf dem der Testfall ausgeführt werden soll, dann die zu testenden Operationen darauf auszuführen und anschließend das Ergebnis auf Korrektheit zu überprüfen.

Organisation der Testfälle

Nun sind Methoden aber keine Objekte und so ist es nicht ganz offensichtlich, wie man dem Framework mitteilt, welche Methoden es sind, die die Testfälle repräsentieren. Man könnte natürlich jeden Testfall in seiner eigenen Klasse implementieren und die Methode immer gleich, zum Beispiel `test()`, nennen (eine Anwendung des *STRATEGY Pattern* [26] ; s. Abschnitt 4.4.4). Dies würde jedoch zu einer Inflation von Klassen führen, die die Handhabung erschwert. Alternativ könnte man alle Testfälle in einer Klasse implementieren und diese durch eine Methode `test()` aufrufen lassen. Dies würde aber wiederum zu einer riesigen Klasse führen, die schwer wartbar wäre und in der mögliche Ähnlichkeiten zwischen Testfällen kaum ausgenutzt werden könnten. Man beachte, dass in beiden Fällen die Objektorientierung nicht zum Zuge kommt, insbesondere weder Instanzvariablen noch Objekte benötigt werden.

In JUNIT ist man daher einen anderen Weg gegangen. Man fasst Testfälle als Instanzen der Klasse `TestCase` (bzw. einer davon abgeleiteten Klasse) auf und ordnet jeder Instanz (jedem Testfall) genau eine Testmethode zu. Die Klasse hat also genauso viele Instanzen wie in ihr Testmethoden definiert sind, und das Verhältnis zwischen diesen ist 1:1. Dabei erfolgt die Zuordnung von Instanz zu Methode über den Namen der Methode, der dazu in einer Instanzvariable (`fName`) gespeichert wird.³² Testfälle können auch auf Hilfsmethoden zurückgreifen, die ebenfalls in der Klasse definiert sind; eine Klasse kann dadurch mehr Methoden als (unterschiedliche) Testfälle haben.

Test fixture

Tatsächlich ist es in JUNIT vorgesehen, dass sich alle Testfälle einer Testklasse zwei spezielle Methoden, nämlich `setUp()` und `tearDown()`, teilen, die das Objektgeflecht, auf dem die Tests durchgeführt werden sollen, die sog. **Test fixture**, auf- bzw. abbauen. Das Objektgeflecht selbst wird dann in den Instanzvariablen der Klasse für jede Instanz (jeden Testfall) zugänglich gespeichert. Aus dieser Konvention ergibt sich, dass das Klassifikationskriterium für Testfälle die gemeinsamen Fixtures (Ausgangsobjektgeflechte) sind. Da Aus-

³² Man ahnt hier schon, dass JUNIT vom Reflection-API JAVAs Gebrauch macht.

gangsobjektgeflechte im Verlauf jedes einzelnen Testfalls verändert werden können, müssen sie vor jedem Testfall neu erstellt werden.

Eine Klasse, die eine Menge von Testfällen anbietet, muss, um vom Framework akzeptiert zu werden, zunächst das Interface **Test** implementieren:

Testklassen

```
327 public interface Test {  
328     public abstract int countTestCases();  
329     public abstract void run(TestResult result);  
330 }
```

Dieses Interface wird von zwei Klassen des JUNIT-Frameworks implementiert, nämlich **TestCase** und **TestSuite**. Die Klasse **TestSuite** spielt dabei die Rolle eines Kompositums im *COMPOSITE Pattern* [26] (Abschnitt 4.4.1), d. h., ihre Instanzen bestehen im Wesentlichen aus einer Anzahl Tests, die entweder wieder Testsuiten oder Testfälle sind.

```
331 public class TestSuite implements Test {  
332     private String fName;  
333     private Vector fTests= new Vector(10);  
...  
334     public Enumeration tests() {  
335         return fTests.elements();  
336     }  
  
337     public int countTestCases() {  
338         int count= 0;  
339         for (Enumeration e= tests(); e.hasMoreElements(); ) {  
340             Test test= (Test)e.nextElement();  
341             count= count + test.countTestCases();  
342         }  
343         return count;  
344     }  
  
345     public void run(TestResult result) {  
346         for (Enumeration e= tests(); e.hasMoreElements(); ) {  
347             if (result.shouldStop() )  
348                 break;  
349             Test test= (Test)e.nextElement();  
350             runTest(test, result);  
351         }  
352     }  
  
353     public void runTest(Test test, TestResult result) {  
354         test.run(result);
```

```
355 }
356 }
```

Man erkennt hier übrigens sehr schön, wie viel stereotyper Code in der Vergangenheit für das gemeinsam mit den Generics fehlende For-each-Konstrukt verbraten werden musste.

Die Klasse **TestCase** implementiert die beiden Methoden wie folgt:

```
357 public abstract class TestCase extends Assert implements Test {
358     private String fName;
...
359     public int countTestCases() {
360         return 1;
361     }
362     public void run(TestResult result) {
363         result.run(this);
364     }
365 }
```

TestCase ist abstrakt, weil in ihr Methoden deklariert sind, die von den Klassen, die die eigentlichen Testfälle beherbergen, implementiert werden müssen (s. u.).

Die Klasse **Assert**

Die Klasse **Assert**, von der **TestCase** erbt, stellt übrigens lediglich eine Menge von Funktionen zur Verfügung, die ein erwartetes Ergebnis mit einem tatsächlichen vergleichen und für den Fall, dass die beiden nicht übereinstimmen, einen entsprechenden Fehler (**AssertionFailedError**, von **Error** abgeleitet) werfen. Dass der Fehler von **Error** abgeleitet wird, liegt darin begründet, dass die Testerin (also z. B. Sie) selbst keinen Exception handler schreiben soll; wie wir unten sehen werden, wird das Scheitern einer Zusicherung vom Framework aufgefangen und ausgewertet.

```
366 class Assert {
...
367     static public void assertEquals(String message,
                                         String expected, String actual) {
368         if (expected == null && actual == null)
369             return;
370         if (expected != null && expected.equals(actual))
371             return;
372         throw new ComparisonFailure(message, expected, actual);
373     }
```

Testergebnisse: die Klasse **TestResult**

Wie in Zeile 363 oben zu sehen ist, leiten Testfälle die Aufgabe des Testens zunächst an das Testergebnis, eine Instanz der Klasse **TestResult**, die die

Ergebnisse eines oder mehrerer Testfälle bzw. Testsuiten akkumuliert, weiter. `TestResult` implementiert dazu ebenfalls die Methode `run`:

```
374 public class TestResult {  
375     protected Vector fFailures;  
376     protected Vector fErrors;  
377     protected Vector fListeners;  
378     protected int fRunTests;  
379     private boolean fStop;  
...  
  
380     protected void run(final TestCase test) {  
381         startTest(test);  
382         Protectable p= new Protectable() {  
383             public void protect() throws Throwable {  
384                 test.runBare();  
385             }  
386         };  
387         runProtected(test, p);  
388         endTest(test);  
389     }  
  
390     public void runProtected(final Test test, Protectable p) {  
391         try {  
392             p.protect();  
393         }  
394         catch (AssertionFailedError e) {  
395             addFailure(test, e);  
396         }  
397         catch (ThreadDeath e) { // don't catch ThreadDeath by  
            accident  
398             throw e;  
399         }  
400         catch (Throwable e) {  
401             addError(test, e);  
402         }  
403     }  
  
404     public synchronized void addError(Test test, Throwable t) {  
405         fErrors.addElement(new TestFailure(test, t));  
406         for (Enumeration e= cloneListeners().elements();  
407              e.hasMoreElements(); ) {  
408             ((TestListener)e.nextElement()).addError(test, t);  
409         }  
410     }
```

```

410 }
411 public synchronized void addFailure(Test test,
412             AssertionFailedError t) {
413     fFailures.addElement(new TestFailure(test, t));
414     for (Enumeration e= cloneListeners().elements();
415         e.hasMoreElements(); ) {
416         ((TestListener)e.nextElement()).addFailure(test, t);
417     }
418 }
```

Meldung des Testergebnisses per Exceptions

Die Methode `startTest(test)` (Aufruf in Zeile 381) benachrichtigt die sog. Listener (im Feld `fListeners` hinterlegt; Observer im *OBSERVER pattern*; s. Abschnitt 4.4.2) des Testergebnisses vom Beginn eines Tests; Entsprechendes tut `endTest(test)`. Der Methodenaufruf `runProtected(test, p)` schließlich tut etwas höchst Indirektes: Er ruft, über `p.protect()`, die Methode `runBare()` auf dem ursprünglich (in Zeile 380) übergebenen Testfall auf. Damit wird die Verantwortung für die Ausführung des Testfalls wieder an diesen zurückdelegiert. Angesichts einer gewissen Verworrenheit des Codes, die aus dem Vor- und Rückwärtsdelegieren resultiert, könnte man argumentieren, dass der Aufruf von `startTest` und `endTest` (die Bekanntmachung von Testanfang und -ende) durch den Testfall selbst hätte erfolgen können; es bleibt aber in der Verantwortung der Klasse `TestResult`, fehlerhafte Ergebnisse entgegenzunehmen und aufzuzeichnen. Da die Signalisierung von Fehlern über Exceptions erfolgt, muss auch der Aufruf des Testfalls in `TestResult` erfolgen (sonst könnten die Exceptions dort auch nicht gefangen werden). Außerdem wird durch die vorübergehende Auslagerung der Verantwortung der Testfall davon befreit, den Ausgang des Tests an das Testergebnis zu schicken; durch die Zuordnung der Methoden `addFailure` und `addError` zu `TestResult` und den Aufruf des eigentlichen Testlaufs von `TestResult` aus braucht der Testfall die Instanz von `TestResult`, an die er berichten muss, gar nicht zu kennen. Zudem spart man sich so jede Menge If-return-Paare mit expliziter Rückgabe einer entsprechenden Fehlermeldung — es wird also Schreibaufwand weg von den Testfällen, die ja der Tester schreiben muss, hin zum Framework verlagert, eine Vorgehensweise, die durch den Ansatz von JUNIT gerechtfertigt wird, Testfälle in Form von Methoden zu spezifizieren.

Failure vs. Error

Wie man aus der Implementierung von `runProtected` (Zeile 390) erkennt (und wie bereits eben erwähnt), ist die Klasse `TestResult` auch dafür verantwortlich, dass eventuelle Fehlschlagen von Testfällen zu dokumentieren. Es werden zwei Arten von Versagen unterschieden: Die Ausführung eines Testfalls liefert nicht das erwartete Ergebnis, was sich im Scheitern einer Zusicherung äußert (eine sog. *Failure*), oder die Ausführung eines Testfalls wird mit einer Exception abgebrochen (ein sog. *Error*). Wie man sieht, wird hier das Exception handling von JAVA benutzt, um unterschiedliche Ausgänge einer Methode abzufragen. Man könnte argumentieren, dass es sich hierbei um einen Missbrauch des Exception handlings handelt (siehe Abschnitt 5.2.2.4 ff.). Auf der anderen Seite ist der Kontext, das Testen, recht speziell, was man schon daran merkt, dass der Abbruch einer Methode mit einem Laufzeitfehler als erwartetes Ergebnis angesehen werden darf (Programmzeile 400 ff.). Da sich Laufzeitfehler (Exceptions, Errors im JUNIT-Jargon) nur durch Exception handling abfragen lassen,

kann man schon aus Symmetriegründen argumentieren, dass dies für Failures ebenso gehandhabt werden sollte. Wie dem auch sei, die entsprechenden Failures bzw. Errors werden, durch Aufruf der Methode `addFailure` bzw. `addError`, eingetragen und die Listener werden ebenfalls darüber informiert. Die Klasse `TestFailure` dient dabei lediglich der Paarbildung von Test und Fehler.

Die Methode `runBare()` der Klasse `TestCase` ist wiederum wie folgt implementiert:

```
419 public abstract class TestCase ...  
  
420 public void runBare() throws Throwable {  
421     Throwable exception= null;  
422     setUp();  
423     try {  
424         runTest();  
425     } catch (Throwable running) {  
426         exception= running;  
427     }  
428     finally {  
429         try {  
430             tearDown();  
431         } catch (Throwable tearingDown) {  
432             if (exception == null) exception= tearingDown;  
433         }  
434     }  
435     if (exception != null) throw exception;  
436 }
```

Die Methode `runBare()` enthält somit das allgemeine Verfahren, nach dem getestet wird. Zunächst wird das für den Test benötigte Objektgeflecht, die Fixture, durch Aufruf der Methode `setup()` aufgebaut. Danach wird der Testfall durch Aufruf der Methode `runTest()` auf dieser Datenstruktur ausgeführt und dabei das Ergebnis überprüft. Schließlich sollte die Datenstruktur durch Aufruf der Methode `tearDown()` wieder abgebaut werden, was im Falle von JAVA (ohne explizites Zerstören von Objekten) nur bedeuten kann, dass hier gegebenenfalls vom Betriebssystem in Anspruch genommene Ressourcen wie Dateien oder Datenbankverbindungen wieder freigegeben werden können.

Durchführung der Tests

Die konkrete Ausprägung dieses allgemeinen Algorithmus variiert von Testfall zu Testfall. Daher handelt es sich beim in `runBare()` festgehaltenen Algorithmus auch nur um eine Schablone (engl. template) — tatsächlich haben wir es hier mit einem Vorkommen des *TEMPLATE METHOD Pattern* ([26] ; s. Abschnitt 4.4.3) zu tun. Ein konkreter Testfall überschreibt einfach je nach Bedarf die drei Methoden; der Algorithmus selbst ist insofern generisch, als die Methode `runBare()` unverändert von allen Testfällen übernommen werden kann.

Um die drei Methoden überschreiben zu können, ist es notwendig, dass ein Testfall in einer Klasse implementiert wird, die von der Klasse `TestCase` abgeleitet ist. Nun wäre es reichlich

überzogen, wenn für jeden Testfall eine neue Klasse angelegt werden müsste, dies umso mehr, als sich verschiedene Testfälle das Objektgeflecht, auf dem sie ausgeführt werden, teilen können. Es ist also sinnvoll, mehrere Testfälle in einer Klasse zu gruppieren, wobei als Klassifikationsmerkmal die Verwendung gleicher Fixtures dient. Die verschiedenen Testfälle müssen dann in verschiedenen Methoden derselben Klasse implementiert werden. Der generische Algorithmus in der Template method muss dann so angepasst werden, dass er alle Methoden, die einen Testfall darstellen, automatisch erkennt und ausführt.

Um dies umzusetzen, gibt es mehrere Möglichkeiten:

1. Die Methoden werden beim Framework als Objekte registriert (d. h., in einer Collection gespeichert). Beim Ausführen der Tests sieht das Framework dann nach, welche Methoden registriert sind, und führt diese aus.
2. Die Methoden, die Testfälle implementieren, werden als solche gekennzeichnet. Dies kann durch *Annotationen* erfolgen, oder die Methoden folgen alle einer bestimmten Namenskonvention, wodurch das Test-Framework die Möglichkeit hat, diese Methoden per Introspektion zu identifizieren und zur Ausführung zu bringen.
3. Die Testfälle werden als anonyme innere Klassen (von **TestCase** abgeleitet) definiert, deren Instanzen dann jeweils eine eigene Implementierung der Methode **runTest()** besitzen.

Man beachte, dass in allen drei Fällen erst zur Laufzeit entschieden wird, welche Methodenimplementierungen aufgerufen werden. Sowohl das erste als auch das zweite Verfahren setzen dabei auf die Möglichkeit der Introspektion, das heißt, sie speichern irgendwo (im ersten Fall in einer speziellen Datenstruktur, im zweiten Fall im Programmtext selbst) die Namen der Methoden, die aufgerufen werden sollen. Das Framework liest diese Namen und ruft die dazu passenden Methoden auf. Das dritte Verfahren hingegen setzt auf *dynamisches Binden*: Dadurch, dass die anonymen Klassen Subklassen einer Testfallklasse sind, die sich alle (nur) in der Implementierung der einen Methode **runTest()** unterscheiden, wird von Instanz zu Instanz eine andere Methode aufgerufen.

Wie bereits zu Beginn dieses Abschnitts geschildert, sieht das Standardverfahren von JUNIT vor, dass für jeden auszuführenden Testfall, der als Methode in einer Testklasse implementiert ist, eine Instanz dieser Klasse gebildet wird, die mit dieser Methode verbunden ist. Dies geschieht durch das Feld **String fName**, dessen Inhalt der Name des Testfalls ist.³³ Der Aufruf der Methode erfolgt dann über die introspektiven Fähigkeiten JAVAs (in Kurseinheit 6 genauer behandelt), wie der Standardimplementierung von **runTest()** in der Klasse **TestCase** (kann in — möglicherweise anonymen — Subklassen überschrieben werden; s. u.) zu entnehmen ist:

³³ Man beachte, dass also nicht der Name der Klasse, in der die Methode (der Testfall) implementiert ist, den Testfall namentlich repräsentiert, sondern eben der Name der Methode. Verschiedene Instanzen der Klasse können also verschiedene Testfälle repräsentieren. Trotzdem spricht man oft (und fälschlicherweise) von der Klasse als einem Testfall.

```
437 protected void runTest() throws Throwable {  
438     assertNotNull(fName); // Some VMs crash when calling  
439             // getMethod(null, null);  
440     Method runMethod= null;  
441     try {  
442         runMethod= getClass().getMethod(fName, (Class[]) null);  
443     } catch (NoSuchMethodException e) {  
444         fail("Method "+fName+" not found");  
445     }  
446     if (!Modifier.isPublic(runMethod.getModifiers())) {  
447         fail("Method "+fName+" should be public");  
448     }  
449     try {  
450         runMethod.invoke(this, (Object[])new Class[0]);  
451     }  
452     catch (InvocationTargetException e) {  
453         e.fillInStackTrace();  
454         throw e.getTargetException();  
455     }  
456     catch (IllegalAccessException e) {  
457         e.fillInStackTrace();  
458         throw e;  
459     }  
460 }
```

Die Testmethode, die per Definition keine Parameter hat, wird also über den Inhalt des Feldes `fName` referenziert und indirekt ausgeführt; die Registrierung des Testfalls beim Framework erfolgt somit indirekt über eine Instanz vom Typ `Test` (Möglichkeit 1 aus obiger Auflistung). Man beachte, dass hiermit erreicht wurde, dass Objekte einer Klasse auf den Aufruf derselben Methode, nämlich `runTest()`, mit der Ausführung unterschiedlicher Implementierungen reagieren, eine Eigenschaft, die eigentlich den prototypenbasierten Sprachen vorbehalten ist.

Alternativ kann in JUNIT auch der Umweg über eine anonyme innere Klasse (Möglichkeit 3) gegangen werden. Eine Testmethode `xyz()` in der Klasse `ABCTest` vorausgesetzt, würde die Registrierung wie folgt aussehen:

```
461 TestCase test= new ABCTest("xyz") {  
462     protected void runTest() { xyz(); }  
463 };  
464 <Methode zur Registrierung>(test);
```

Es wird also eine neue Instanz einer anonymen Subklasse der Klasse `ABCTest` (die wiederum von `TestCase` abgeleitet ist) erzeugt, die die Methode `runTest()` wie angegeben, also mit einem Aufruf von `xyz()`, überschreibt. Dabei ist die Angabe des Strings im Konstruktoraufzug für den Registrierungsprozess nicht mehr notwendig, da die Methode `runTest()`, die auf diesen String zugreift, ja (in der anonymen Klasse) überschrieben wurde. Der Name des Testfalls wird aber für

die Berichterstattung weiterhin benötigt (und dazu nicht per Introspektion aus dem Namen der Methode abgeleitet). Möglichkeit 2 obiger Auflistung schließlich wird im Framework TESTNG, einer Alternative zu JUNIT 4 [27] [28] verwendet.

**Ausführung
der Testfälle**

Die eigentliche Registrierung ist in JUNIT stets temporärer Natur; sie erfolgt durch den Aufruf der statischen Methode `run()` eines `TestRunners` mit dem Test als Parameter, also zum Beispiel durch

```
465 junit.textui.TestRunner.run(aTestCase);
```

Diese Methode erzeugt im Wesentlichen eine Instanz der Klasse `TestResult` zur Akkumulation der Ergebnisse und ruft die Methode `run()` mit der Instanz als Parameter auf dem ihr übergebenen Testfall `aTestCase` auf. Eine dauerhafte Speicherung (Registrierung) des Testfalls findet nicht statt; stattdessen wird davon ausgegangen, dass das JAVA-Programm, das den Test angestoßen hat, nach dessen Ausführung terminiert (bzw., bei einer erneuten Ausführung, alle Testfälle neu erzeugt werden).

```
466 public class TestRunner extends BaseTestRunner {  
467     private ResultPrinter fPrinter;  
  
468     static public TestResult run(Test test) {  
469         TestRunner runner= new TestRunner();  
470         return runner.doRun(test);  
471     }  
  
472     public TestResult doRun(Test test) {  
473         return doRun(test, false);  
474     }  
  
475     public TestResult doRun(Test suite, boolean wait) {  
476         TestResult result= createTestResult();  
477         result.addListener(fPrinter);  
478         long startTime= System.currentTimeMillis();  
479         suite.run(result);  
480         long endTime= System.currentTimeMillis();  
481         long runTime= endTime-startTime;  
482         fPrinter.print(result, runTime);  
483         pause(wait);  
484         return result;  
485     }  
...  
486 }
```

Bezeichnenderweise heißt der Parameter vom Typ `Test` in Zeile 475 `suite`. Tatsächlich wäre es äußerst mühsam, wenn man jeden Testfall einzeln starten müsste. Vielmehr wäre es wünschenswert, mehrere Testfälle zu einer Testsuite zusammenfassen zu können und die enthalte-

nen Tests gebündelt ablaufen lassen zu können. Zu diesem Zweck existiert wie eingangs erwähnt die Klasse **TestSuite**, die als Composite ausgelegt ist. Einzelne Testfälle können einer Testsuite über die Methode **addTest(Test test)** zugewiesen werden:

```
487 public class TestSuite ...  
  
488 public void addTest(Test test) {  
489     fTests.addElement(test);  
490 }
```

Dabei können, aufgrund der Verwendung des Parametertyps **Test**, Testsuiten andere Testsuiten rekursiv enthalten:

```
491 public void addTestSuite(Class testClass) {  
492     addTest(new TestSuite(testClass));  
493 }
```

Der Aufruf einer ganzen Testsuite erfolgt entsprechend durch

```
494 junit.textui.TestRunner.run(aTestSuite);
```

Gleichwohl erfordert das Erstellen von Tests und Testsuites immer noch einen gewissen redundanten Aufwand, den man sich gern ersparen würde. Es muss nämlich für jeden Testfall nicht nur eine Methode geschrieben, sondern auch ein Objekt erzeugt werden, dass dann an diese Methode gebunden wird. Die Bindung erfolgt über den Namen der Methode, der ja, wie bereits erwähnt, zugleich Name des Testfalls ist und in dem betreffenden Objekt im Feld **fName** gespeichert wird. Wenn man nun wüsste, welche Methoden einer Klasse Testfälle implementieren, wäre es möglich, diese Objekte automatisch zu erzeugen. Heute würde man dieses Problem mit *Annotationen* lösen; da es dieses Konzept in JAVA bei der Entwicklung von JUNIT noch nicht gab, hat man statt dessen eine Namenskonvention eingeführt, nämlich die, dass alle Methoden einer Klasse, deren Name mit „test“ beginnt, automatisch in eine Testsuite eingefügt werden können. Ein Konstruktor, der eine solche Testsuite speziell für eine Klasse erzeugt, ist wie folgt implementiert:

```
495 public class TestSuite ...  
  
496 public TestSuite(final Class theClass) {  
497     fName= theClass.getName();  
498     try {  
499         getTestConstructor(theClass);  
            // Avoid generating multiple error messages  
500     } catch (NoSuchMethodException e) {  
501         addTest(warning("Class "+theClass.getName()+" has no  
public constructor TestCase(String name) or TestCase()"));  
    }
```

```
502     return;  
503 }  
504 if (!Modifier.isPublic(theClass.getModifiers())) {  
505     addTest(warning("Class " +theClass.getName() +  
506                     " is not public"));  
507     return;  
508 }  
509 Class superClass= theClass;  
510 Vector names= new Vector();  
511 while (Test.class.isAssignableFrom(superClass)) {  
512     Method[] methods= superClass.getDeclaredMethods();  
513     for (int i= 0; i < methods.length; i++) {  
514         addTestMethod(methods[i], names, theClass);  
515         superClass= superClass.getSuperclass();  
516     }  
517     if (fTests.size() == 0)  
518         addTest(warning("No tests found in " +  
519                         theClass.getName()));  
520 }  
521 private void addTestMethod(Method m, Vector names,  
522                             Class theClass) {  
523     String name= m.getName();  
524     if (names.contains(name))  
525         return;  
526     if (!isPublicTestMethod(m)) {  
527         if (isTestMethod(m))  
528             addTest(warning("Test method isn't public: " +  
529                         m.getName()));  
530         return;  
531     }  
532     names.addElement(name);  
533     addTest(createTest(theClass, name));  
534 }  
535 private boolean isPublicTestMethod(Method m) {  
536     return isTestMethod(m) &&  
537             Modifier.isPublic(m.getModifiers());  
538 }
```

```
539     return parameters.length == 0 && name.startsWith("test") &&
           returnType.equals(Void.TYPE);
540 }
```

Wie man sieht, ist die eigentliche Logik recht simpel; die meiste Zeit vertut man — wie so oft im Leben einer Programmiererin — damit, die vielen Ausnahmen und Spezialfälle abzuhandeln.

Zu guter Letzt werden Test runner von der Kommandozeile nicht auf einzelnen Objekten vom Typ Test, sondern auf Klassen aufgerufen (dies schon allein deswegen, weil man auf der Kommandozeile keine Objekte spezifizieren kann). Ausgehend von der Klasse wird dann deren Standardtestsuite erzeugt und mit ihr wie gehabt (Zeile 468 oben) fortgefahrene.

```
541 static public void run(Class testClass) {
542     run(new TestSuite(testClass));
543 }
```

Alternativ können die Klassen, die als Parameter einem Test runner übergeben werden, eine Methode `public static Test suite()` deklarieren, die als Ergebnis eine Instanz von `TestSuite` zurückgibt, auf der dann `run()` wie oben aufgerufen werden kann. Eine Default-Implementierung von `suite()` ist dabei die folgende:

```
544 public static Test suite() {
545     return new TestSuite(<Name der Klasse>.class);
546 }
```

Zusammenfassung

JUnit ist also ein Framework, das bestimmte Methoden zur Ausführung bringt, die — jede für sich — jeweils einen Unit-Test durchführen. Jeder solche Testfall sollte aus drei Phasen bestehen, nämlich

- dem Aufbau einer Test fixture (dem Objektgeflecht, auf dem der Test ausgeführt werden soll, durchgeführt durch die Methode `setUp()`),
- dem Aufruf der zu testenden Einheiten (Methoden) mit anschließender Überprüfung des Ergebnisses (codiert in einer entsprechenden Testmethode, angestoßen durch die Methode `runTest()`) und
- dem abschließenden Abbau des Geflechts (`tearDown()`; kann häufig entfallen).

Dabei ist der Aufruf der Testmethoden nicht fest verdrahtet; er erfolgt entweder über Introspektion (siehe Kurseinheit 6) oder über die Angabe einer anonymen inneren Klasse. Die Zuordnung von Testfällen zu den zu testenden Programmeinheiten erfolgt über Namenskonventionen.

3.2.2 Die Anwendung von JUNIT

Das Schreiben von Testfällen mit JUNIT erfolgt nach einem relativ einfachen Schema, dessen Begründung allerdings ohne Kenntnis des inneren Aufbaus des Frameworks nicht unmittelbar einsichtig ist. Wenn Sie den vorigen Abschnitt also übersprungen haben sollten, wundern Sie sich nicht, wenn das eine oder andere merkwürdig erscheint.

Ein einzelner Testfall wird in JUNIT durch eine Methode spezifiziert, deren Name traditionell (vor JUNIT 4: zwingend) mit „test“ beginnt:

```
547 public void testSimpleAdd() {
548     Money m12CHF= new Money(12, "CHF");
549     Money m14CHF= new Money(14, "CHF");
550     Money expected= new Money(26, "CHF");
551     Money result= m12CHF.add(m14CHF);
552     assertTrue(expected.equals(result));
553 }
```

Zu testen ist hier die Klasse **Money**, genauer deren Methode **add()**.³⁴ JUNIT entstammt zwar der Zeit vor JAVA 1.4 (in dem **assert** als neues Schlüsselwort aufgenommen wurde), aber die Verwendung der (zum Framework gehörenden) Methode **assertTrue()** ist trotzdem obligat. Das Beispiel habe ich übrigens direkt aus dem JUNIT Cookbook [29] übernommen.

Im obigen Beispiel enthält die Methode **testSimpleAdd()**, die einen Testfall repräsentiert, ihre eigenen Anweisungen zum Aufbau der Test fixture (des Objektgeflechts, auf dem der Test ausgeführt wird): Sie besteht aus den drei nicht weiter verbandelten Instanzen der Klasse **Money**, die in den (lokalen) Variablen **m12CHF**, **m14CHF** und **expected** gespeichert sind. Im Allgemeinen werden sich jedoch mehrere Testfälle eine Fixture teilen: Deren Erzeugung kann dann in eine separate Methode ausgelagert werden, die von allen Testfällen als erstes aufgerufen wird. Dazu kann die Fixture allerdings nicht mehr in temporären Variablen untergebracht werden, sondern muss in Felder (Instanzvariablen) ausgelagert werden (die Übergabe in Form von Parametern kommt in der Regel nicht in Frage).

Testfälle nach gemeinsamen Fixtures in Klassen aufteilen

Es ist eine der gewöhnungsbedürftigen Konventionen von JUNIT, dass Testfälle, die sich eine Fixture teilen, in einer Klasse zusammengefasst werden, die von der Klasse **TestCase** erbt. **TestCase** ist eine JUNIT-Klasse, deren Einbettung in das Framework u. a. bewirkt, dass vor bzw. nach der Ausführung eines Testfalls die Methoden **setUp()** bzw. **tearDown()** aufgerufen werden, in denen die Fixtures erzeugt bzw. abgebaut werden. Diese Methoden können vom Anwender des Frameworks, dem Tester, überschrieben werden, um eigene Fixtures in den Testprozess einzubringen. Das sieht dann z. B. so aus:

³⁴ Dass das der Fall ist, ist zunächst nur durch Hinsehen zu erkennen. Per Konvention nennt man die Klasse, die die Testfälle definiert, dann auch nach der zu testenden Klasse („MoneyTest“ im gegebenen Fall); das aber funktioniert nur in den seltensten Fällen, da in der Regel mehr als eine Testklasse pro zu testender Klasse (= Unit) zu erstellen ist. S. u.

```
554 public class MoneyTest extends TestCase {  
555     private Money f12CHF;  
556     private Money f14CHF;  
557     private Money expected;  
  
558     protected void setUp() {  
559         f12CHF= new Money(12, "CHF");  
560         f14CHF= new Money(14, "CHF");  
561         expected= new Money(26, "CHF");  
562     }  
  
563     public void testSimpleAdd() {  
564         Money result= m12CHF.add(m14CHF);  
565         assertTrue(expected.equals(result));  
566     }  
567 }
```

Ein Testfall, also eine Instanz vom Typ `TestCase` (der Klasse `MoneyTest` in diesem Fall) wird dann übrigens durch `new MoneyTest("testSimpleAdd")` erzeugt und durch `new MoneyTest("testSimpleAdd").run()` ausgeführt (die Methode `run()` wird von `TestCase` geerbt), aber das braucht Sie nicht zu kümmern, da es ebenfalls vom Framework besorgt wird.

In der Regel wird man nämlich nicht die Tests einzeln ausführen lassen wollen, sondern mehrere auf einmal. Dazu ist es in JUNIT vorgesehen, dass Testfälle (also Instanzen von Klassen, die von `TestCase` abgeleitet sind), in einer sog. Testsuite zusammengefasst werden. Dazu ist wiederum eine Frameworkklasse vorgesehen: `TestSuite`.

Eine Testsuite lässt sich manuell erstellen, z. B. durch den nachfolgenden Code:

```
568 TestSuite suite= new TestSuite();  
569 suite.addTest(new MoneyTest("testSimpleAdd"));  
...  
570 TestResult result= suite.run();
```

Dies ist aber wiederum wenig komfortabel, da man alle Testfälle einzeln nennen muss, obwohl doch eigentlich klar ist, welche es gibt, nämlich einen pro Methode, deren Name mit „test“ anfängt. JUNIT nutzt daher JAVA’s Reflection-API aus und erstellt eine Testsuite automatisch:

```
571 TestSuite suite= new TestSuite(MoneyTest.class);  
572 TestResult result= suite.run();
```

erzeugt eine Testsuite bestehend aus allen Testfällen der Klasse `MoneyTest` und führt diese aus. Der Testsuite können dann (per Methode `addTest()`) noch weitere Testfälle oder ganze Testsuiten hinzugefügt werden:

```
573 TestSuite suiteA = new TestSuite...
```

```
574 TestSuite suiteB = new TestSuite...
575 suiteA.addTest(suiteB);
```

Testsuiten können also aus Tests und Testsuiten bestehen — ein klassisches Vorkommen des *COMPOSITE Pattern*.

Um die Tests nun auszuführen und sich das Ergebnis anzeigen zu lassen, stellt das JUNIT-Framework sog. Test runner zur Verfügung. Diese nehmen als Parameter eine Testsuite (oder auch einen einzelnen Test) entgegen:

```
576 Test test = ...
577 junit.textui.TestRunner.run(test);
```

oder alternativ auch eine ganze Klasse:

```
578 junit.awtui.TestRunner(MoneyTest.class);
```

Zusammenfassung

Zusammenfassen lässt sich das Schreiben von Unit-Tests mit JUNIT wie folgt:

1. Testfälle mit Fixtures ausdenken
2. Testfälle nach gemeinsamen Fixtures in Klassen gruppieren
3. Pro Klasse eine Fixture (Methoden `setUp()` und `tearDown()`) sowie pro Testfall eine Methode (mit „test“ beginnend) schreiben
4. Mengen von Testfällen (Klassen) in Suiten zusammenfassen und einem Test runner übergeben.

Alternativ, wenn mehrere Testfälle ohne gleiche Fixtures zusammengefasst werden sollen (z. B. alle Testfälle zu einer Klasse), kann jeder Testfall (bzw. die ihn implementierende Methode) auch ihre eigene Fixture „inline“ definieren.

3.2.3 Änderungen in JUNIT 4

Mit der Version 4 wurde JUNIT an die neuen Möglichkeiten, die JAVA mit seiner Version 5 bietet, angepasst. Dazu zählt vor allem die Verwendung von *Annotationen* anstelle von Namenskonventionen zur Kennzeichnung von Testmethoden und die Nutzung von statischen Imports, um das (indirekte) Erben von der Klasse `Assert` überflüssig zu machen. Dies erlaubt es, Testmethoden (und damit Testfälle) in beliebigen Klassen, und damit auch in unmittelbarer Nachbarschaft der zu testenden Methoden (innerhalb der zu testenden Klassen!), zu definieren.

Annotationen, die in diesem Kurs erst in Abschnitt 6.2 (im Rahmen der Metaprogrammierung) eingeführt werden, erlauben, Programmelemente mit zusätzlichen Informationen zu versehen,

die zur Übersetzungs- und ggf. auch zur Laufzeit abgefragt werden können. Im Falle von JUNIT 4 ersetzt eine Annotation `@Test` vor einer Methode das Präfix `test` im Methodenamen von früheren Versionen von JUNIT. Eine weitere Annotation, `@Ignore`, weist JUNIT an, die so annotierte Testmethode zu ignorieren; gleichwohl wird sie vom Testframework als Testmethode erkannt und ihre Nichtausführung wird dokumentiert — so die Programmiererin will, mit einem entsprechenden Kommentar versehen, der als Parameter der Annotation angegeben wird. So führt

```
579 @Test  
580 @Ignore("kann noch nicht funktionieren, weil ...")  
581 public void testfall738() ...
```

dazu, dass der Test runner für den Testfall mit Namen `testfall738` anzeigt, dass er aus dem genannten Grund ignoriert wurde. Die Annotation `@Test` kann auch mit Parametern versehen werden: So führt

```
582 @Test(timeout = 500)
```

dazu, dass der Testfall spätestens nach einer halben Sekunde als fehlgeschlagen angesehen wird, wenn er nicht vorher erfolgreich beendet wurde. Ein anderer nützlicher Parameter der `@Test`-Annotation nennt eine Exception, die die getestete Methoden werfen soll:

```
583 @Test(expected = ArithmeticException.class)  
584 public void teileDurchNull() {  
585     rechner.teile(0);  
586 }
```

Man erspart sich dadurch das Schreiben eines eigenen Exception handlers und das etwas indirekte Signalisieren eines Errors durch Aufruf von `fail()` nach einem Methodenaufruf, der eigentlich eine Exception werfen sollte.

Weitere Annotations von JUNIT 4 sind `@Before` und `@After`, die die Standardmethodennamen `setUp()` und `tearDown()` ersetzen, sowie `@BeforeClass` und `@AfterClass`, deren so annotierte Methoden jeweils nur einmal pro Klasse für alle darin enthaltenen Testfälle aufgerufen werden. Soweit diese Methoden zur Aufbau der Fixture beitragen, darf kein Testfall das erzeugte Objektgeflecht verändern, da sonst keine Unabhängigkeit der Testfälle mehr gewährleistet ist. Im Fall der Vererbung, also wenn die Testfälle in einer Subklasse einer Klasse, die ebenfalls `@Before` und `@After` oder `@BeforeClass` und `@AfterClass` enthält, stehen, werden die geerbten aufbauenden und abreißenden Methoden vor bzw. nach denen der erbenden Klasse aufgerufen.

Eine weitere Neuerung von JUNIT 4 ist, dass das JAVA-1.4-Schlüsselwort `assert`, dessen Zweck Sie ja schon in Abschnitt 2.6.2 JAVA kennengelernt haben, ebenfalls zum Abgleich von tatsächlichem und erwartetem Ergebnis eines Testfalls verwendet werden darf. Da das Fehlschlagen von `assert` standardmäßig einen Fehler vom Typ `AssertionError` auslöst, wurde das Framework auch bei den selbst erzeugten Exceptions darauf umgestellt.

3.2.4 Änderungen mit JUNIT 4.4

Mit der Version 4.4 ist JUNIT noch einmal deutlich erweitert worden. Zum einen wurde die bereits in Abschnitt 2.6.2 JAVA (im Kontext von Design by contract) kurz vorgestellte HAMCREST-Bibliothek übernommen, die es gestattet, das Testergebnis betreffende Zusicherungen komfortabler auszudrücken. Zum anderen wurden die Möglichkeiten des Testens um sog. **Theorien** erweitert.

In Abschnitt 2.6.4 Grenzen der Ausdrucksstärke wurde bemängelt, dass sich beim heutigen Design by contract axiomatische Ausdrücke wie

remove(put(s, x)) = s

nicht verwenden lassen, obwohl sie eine einfache und zugleich von konkreten Ein- und Ausgaben unabhängige Spezifikation darstellen. Mittels sog. Theorien wurde nun JUNIT um die Möglichkeit erweitert, genau solche Ausdrücke als generische Testfälle zu formulieren. Dazu schreibt man einfach

```
587 @Theory
588 void removelnvertsPut(Stack s) {
589     Stack t = s.clone();
590     s.put(new Object());
591     s.remove();
592     assertEquals(s, t);
593 }
```

Die etwas dicke Bezeichnung „Theorie“ für eine solche Invariante wurde dabei aus dem Zitat „Good tests kill flawed theories; we remain alive to guess again.“ von Karl Popper abgeleitet. Tatsächlich kann man Theorien der obigen Art als implizit allquantifiziert betrachten, also als „für alle Eingaben **s** vom Typ **Stack** soll gelten: ...“ lesen. Damit stellt eine Theorie zunächst eine vollständige Spezifikation des erwarteten Verhaltens dar, im Gegensatz zu einer Menge Testfälle, die in der Regel nur eine partielle Spezifikation darstellen.

Es bleibt jedoch die Frage, wie die Einhaltung einer Theorie überprüft werden soll. Mangels Verfügbarkeit einer formalen Programmverifikation bleibt hier wieder nur ein Testen mit konkreten Werten, die per Aufruf durch das Testframework über den Parameter (hier: **Stack s**) in die Theorie eingespeist werden, übrig. Aber wo kommen diese Werte her?

JUNIT 4.4 sieht dafür die Möglichkeit vor, in einer Testklasse eine Reihe von sog. Datenpunkten anzugeben, die durch die Annotation **@DataPoint** als solche gekennzeichnet sind und die von JUNIT (genauer: einem Test runner namens „Theories“) über Reflektion erkannt und in alle Theorien der Testklasse eingespeist werden. Dazu schreibt man beispielsweise

```
594 @RunWith(Theories.class)
595 class Tests {
596     @DataPoint public static Stack EMPTY = new Stack();
597     @DataPoint public static Stack ONE_ELEMENT =
598         (new Stack()).put(new Object());
599 }
```

Dabei setzt der Test runner nur Datenpunkte des passenden Typs (im gegebenen Fall `Stack`) ein. Hat die Theoriemethode mehrere Parameter, werden auch diese, ihrem jeweiligen Typ entsprechend, mit Datenpunkten versorgt.

Nun kann eine Testklasse mehrere Theorien haben. Selbst wenn der Typ passt, sind dann nicht alle Datenpunkte der Klasse notwendigerweise für alle Theorien geeignet. Um eine entsprechende Vorauswahl treffen zu können, gibt es eine neue Methode `assumeThat(...)`, die, analog zu `assertThat(...)` (s. Abschnitt 2.6.2 JAVA), den ersten Parameter auf Einhaltung einer mittels des zweiten Parameters ausgedrückten Bedingung überprüft. Wird diese Bedingung verletzt, wird die Testmethode abgebrochen und dies entsprechend signalisiert. (Im obigen Beispiel mit dem Stack könnte diese Bedingung beispielsweise ausdrücken, dass der Stack nicht voll ist.) Bei der Anzeige der Testergebnisse wird dieser Abbruch dann separat angezeigt.

Vorbedingungen für Tests

Ein einfaches Beispiel für die Absicherung einer Theorie gegen unzulässige Eingaben ist das folgende:

```
600 @Theory void equalObjectsEqualHashes(Object a, Object b) {
601     assumeTrue(a.equals(b));
602     assertTrue(a.hashCode() == b.hashCode());
603 }
```

Die Theorie drückt eine Invariante des Inhalts „wenn *a* und *b* gleich sind, dann müssen sie einen identischen Hash code haben“ aus. Nun könnte man erwarten, dass der Test runner die Methode `equalObjectsEqualHashes` nur mit gleichen Objekten versorgt; so schlau ist er aber nicht — statt dessen werden alle Kombinationen von passenden Datenpunkten durchprobiert und die unpassenden herausgefiltert. Ineffizient, aber einfach.

Übrigens: Man hätte alternativ die Theorie auch mit einem gewöhnlichen If-Statement absichern können. Damit beraubte man JUNIT aber der Möglichkeit, mitzuschreiben, wie oft eine Theorie aufgrund von Verletzungen ihrer Vorbedingung nicht geprüft werden konnte. Ist das nämlich immer der Fall, kann JUNIT signalisieren, dass entweder an der Theorie oder an den Datenpunkten etwas nicht in Ordnung ist.

`assumeThat` kann auch außerhalb von Theorien sinnvoll eingesetzt werden. So sind z. B. manche Tests davon abhängig, dass das System in einem bestimmten Zustand ist (z. B. eine Internetverbindung besteht); ist dies nicht der Fall, kann der Test auch kein sinnvolles Ergebnis liefern. Andere Tests sind für spezielle Betriebssysteme gedacht: So liefert

```
604 @Test public void filenameIncludesUsername() {  
605     assumeThat(File.separatorChar, is('/'));  
606     assertThat(new User("optimus").configFileName(),  
607                 is("configfiles/optimus.cfg"));  
607 }
```

auf Windows-Systemen keinen Fehler.

Theorien bieten also eine komfortable Möglichkeit, die Testdichte zu erhöhen. Sie basieren auf vollständigen Spezifikationen des Verhaltens, die aber nur partiell (über Stichproben in Form von Datenpunkten) überprüft werden.

3.2.5 Probleme von JUNIT

Auch wenn JUNIT ein Paradebeispiel für eine kleine, extrem erfolgreiche Software ist, ergeben sich doch Ansätze für Verbesserungen. So ist die Formulierung und Verwaltung von Testfällen in Form äußerlich kaum von anderen zu unterscheidender JAVA-Klassen und -Methoden nicht nur unschön, sie nutzt auch den stereotypen Aufbau von Testfällen (Aufruf von Methoden und Vergleich des Ergebnisses mit den Erwartungen) nicht aus und verlangt stattdessen von der Programmiererin eine recht intime Kenntnis des Frameworks, was seiner breiten Verwendung nicht zuträglich ist. Auf der anderen Seite gibt es im Einzelfall zur Ausprogrammierung von Testfällen in JAVA wohl kaum eine wirkliche Alternative: Man wird früher oder später zur Formulierung eines Tests auf die volle Ausdrucksstärke der verwendeten Programmiersprache zurückgreifen wollen, so dass man dann fragen darf, warum man nicht gleich diese Sprache zum Formulieren der Testfälle verwendet. Trotzdem könnte man durch einen speziellen Editor die Teile des Codes, die durch das Framework bedingt werden, automatisch erzeugen und beim Editieren verborgen, so dass nur noch die eigentlichen Testfälle zu sehen sind (ein Art JUNIT-spezifische Teilsprache von JAVA), und für die Formulierung der Testfälle könnte ein Syntaxeditor oder ein Wizard so konfiguriert werden, dass der stereotype Aufbau der meisten Testfälle automatisiert wird. Doch selbst dann bleiben noch die fehlenden Querverweise zwischen Testfällen und zu testenden Einheiten — die Programmiererin muss diesen Bezug stets selbst im Auge behalten und bei Änderungen ggf. selbst die betroffenen Stellen aufsuchen. Diese Querbezüge würden nicht nur beim Pflegen des Programms und seiner Tests helfen, sie könnten auch wertvolle Hinweise bei der Fehlersuche (im Falle gescheiterter Testfälle) liefern. Mehr dazu in Abschnitt 3.6.

Ein weiteres Problem von JUNIT ist der immense Zeit- und Ressourcenverbrauch, der durch das immer wieder neue Aufbauen und Abreißen der Fixtures entsteht. Insbesondere dann, wenn eine Fixture von einem Test nicht oder nur leicht manipuliert wird, wäre es schön, sie könnte den Test überleben und bei nächster Gelegenheit wiederverwendet werden. Alternativ zur ständigen Neuerzeugung könnte man auch eine objektorientierte Datenbank einsetzen, die es erlaubt, mit Hilfe eines Rollback die Veränderungen, die ein Test an einer Fixture durchgeführt hat, rückgängig zu machen.

3.3 Vererbung von Testfällen

Bestimmte Klassen einer Anwendung stehen in Vererbungsbeziehung zueinander und so könnte man meinen, dass deren Testklassen diese Vererbungsbeziehung spiegeln, also in einer parallelen Vererbungshierarchie voneinander ableiten sollten (die Vererbung der Fixture-Methoden, die in JUNIT 4, genau wie die Testmethoden, in den getesteten Klassen selbst liegen können, legt dies ja auch nahe). Dies ist jedoch aus mehreren Gründen nicht notwendigerweise der Fall.

Zum einen ist gar nicht gesagt, dass in einer Testklasse alle und nur die Tests einer (Anwendungs-)Klasse enthalten sind. Vielmehr — und wie bereits mehrfach erwähnt — sind Testklassen nach gemeinsamen Fixtures organisiert, und diese Fixtures können durchaus Objekte mehrerer Klassen enthalten, die dadurch auch alle getestet werden (auch wenn das nicht unbedingt der Idee des Unit-Testens entspricht). Außerdem benötigt man pro zu testender Klasse u. U. mehrere Fixtures und somit mehrere Testklassen, so dass schon deswegen eine parallele Hierarchie nicht zustande kommt. Und nicht zuletzt ändert sich das Verhalten einer Klasse gegenüber dem seiner Superklasse in der Regel zumindest in Teilen so, dass die (geerbten) Tests der Superklasse scheitern müssen. Zwar könnte man diese Tests dann entsprechend überschreiben (also die betreffenden Testfälle durch andere ersetzen), aber insgesamt ist der Gewinn, der sich aus einer Vererbung von Testklassen ergeben würde, eher fraglich. Vielleicht ist auch dies der Grund, warum JUNIT keine spezielle Unterstützung für die Vererbung bietet.

Etwas anderes ist es jedoch, wenn Interfaces (also Spezifikationen) „vererbt“ werden. Diese müssen nämlich von allen implementierenden Klassen eingehalten werden, was wiederum durch die Vererbung von Tests zum Ausdruck gebracht werden kann.

3.4 Das Testen von Interfaces

Das hauptsächliche Defizit von Interfaces à la JAVA ist ja, dass es sich dabei um rein syntaktische Konstrukte handelt, dass ihnen also insbesondere die Spezifikation des erwarteten Verhaltens fehlt. Dieses fehlende Verhalten konnte ja schon durch Vor- und Nachbedingungen à la *Design by contract* spezifiziert werden (s. Kurseinheit 2); alternativ kann man aber auch Unit-Tests dafür verwenden (vgl. Abschnitte 2.7 Design by contract als Form des Testens und 3.9 zur Austauschbarkeit von Unit-Tests und Design by contract).

Ein Interface³⁵ gibt einen Vertrag vor, ohne zu sagen, wie er erfüllt wird. Das hindert eine jedoch nicht daran, Testfälle zu formulieren und zu implementieren, die die Erfüllung des Vertrags überprüfen: Der Vertrag ist ja gerade unabhängig von seiner Implementierung. Das Problem des Testens von Interfaces ist vielmehr, dass keine Instanzen von ihnen erzeugt werden können: Interfaces sind wie abstrakte Klassen *abstrakt* (im Sinne von nicht instanziierbar). Der Aufbau einer *Test fixture* für Interfaces selbst ist also unmöglich — es muss dazu erst eine konkrete Klasse vorliegen, die das Interface implemen-

implementations-unabhängige Testfälle

³⁵ Das in diesem Abschnitt zum Testen von Interfaces Gesagte gilt auch für das Testen von abstrakten Klassen, soweit nicht der konkrete Anteil der abstrakten Klasse (also das Interne vorhandener Implementation) getestet werden soll.

tiert (wodurch sie dessen Vertrag zu erfüllen verspricht) und von der Instanzen erzeugt werden können, auf denen die Tests ausgeführt werden können. Gleichwohl gehören die Tests, die das Interface betreffen, zum Interface, nicht zu der es implementierenden Klasse, dies umso mehr, als es natürlich viele Klassen geben kann, die das Interface implementieren und für die diese Tests nicht immer neu definiert werden sollen.

Interfacetestfälle in abstrakte Testklassen

Was also tun? Ein gängiger Ansatz ist, für jedes Interface eine abstrakte Testklasse vorzusehen, die eine Menge von Testmethoden implementiert, die das mit dem Interface verbundene Verhalten abprüft. Die Klasse muss abstrakt sein, weil man keine Setup-Methode vorsehen kann, die das zu testende Interface instanziert — das Interface ist nicht instanzierbar und instanzierbare Implementierungen sind nicht bekannt. Wenn man dann aber eine konkrete Klasse das Interface implementieren lässt, kann man für sie eine konkrete Testklasse von der abstrakten Testklasse ableiten; sie erbt dann deren Testfälle. Damit die geerbten Testmethoden nicht überschrieben werden können (und damit der durch das Interface vorgegebene Vertrag bzw. seine Überprüfung durch die Tests nicht geändert werden kann), sind diese `final` zu deklarieren.

Fixture per Factory-Methode

Nun benötigen die Tests der abstrakten Testklasse ein Objektgeflecht (die *Fixture*), auf dem sie ausgeführt werden können. Die Setup-Methode der abstrakten Testklasse kann jedoch keinen Konstruktor kennen, den sie aufrufen könnte — das Interface besitzt keinen und die konkreten Klassen, die es implementieren, sind der (abstrakten) Testklasse nicht bekannt. Dieses Problem lässt sich jedoch durch Angabe einer *Factory-Methode* (s. Abschnitt 4.4.6) lösen, die von den konkreten Subklassen der abstrakten Testklasse übergeschrieben wird und von da jeweils eine Instanz der betreffenden Klasse zurückgibt.

Beispiel

Das folgende einfache Beispiel veranschaulicht das Verfahren. Nehmen wir an, wir wollten einen oder mehrere Testfälle schreiben, der oder die Verhaltenskonformität mit dem Interface `java.util.Iterator` überprüfen. Eine entsprechende Testklasse könnte wie folgt aussehen:

```
608 public abstract class IteratorTest {  
  
609     @Test  
610     public final void testNoSuchElementException() {  
611         Iterator iter;  
612         for (iter = createliterator(); iter.hasNext(); iter.next());  
613         try {  
614             iter.next();  
615             fail("NoSuchElementException erwartet");  
616         }  
617         catch (NoSuchElementException ok) {}  
618     }  
}
```

```
619 ...  
620 protected abstract Iterator createliterator();  
621 }
```

Die Klasse ist abstrakt, weil die Methode `createIterator()` abstrakt ist; diese ist wiederum abstrakt, weil der Testklasse keine Klassen bekannt sind (und auch nicht bekannt sein sollen), die `Iterator` implementieren. Die Erzeugung der Instanzen wird also an konkrete Subklassen der Testklasse delegiert, die zu diesem Zweck `createIterator()` überschreiben müssen:

```
622 public class ListIteratorTest extends IteratorTest {  
  
623     @Override  
624     protected Iterator createliterator() {  
625         return Collections.emptyList().iterator();  
626     }  
627 }  
  
628 public class FlawedIteratorTest extends IteratorTest {  
  
629     @Override  
630     protected Iterator createliterator() {  
631         return new FlawedIterator();  
632     }  
633 }
```

Der Fehler im Protokoll von

```
634 public class FlawedIterator implements Iterator {  
  
635     @Override  
636     public boolean hasNext() {  
637         ...  
638     }  
  
639     @Override  
640     public Object next() {  
641         if (!hasNext())  
642             return null;  
643         ...  
644     }  
  
645     @Override  
646     public void remove() {  
647         ...
```

```
648 }  
649 }
```

wird damit erkannt, ohne dass dafür in `FlawedIteratorTest` ein spezieller Testfall hätte geschrieben werden müssen. Weitere Testfälle, die die Interna der Implementierung von `FlawedIterator` testen (von denen `IteratorTest` keine Kenntnis haben kann), könnte man wohl in `FlawedIteratorTest` unterbringen; besser wäre es jedoch, auch bei den Testfällen zwischen abstrakter Spezifikation und konkreter Umsetzung zu trennen und diese in getrennten Klassen zu belassen, zumal das Testen der internen Zusammenhänge in der Regel eine andere Fixture benötigt als das Testen des extern beobachtbaren Verhaltens. Die Anführung von `final` bei der Definition der interfacebezogenen Testfälle würde damit freilich bedeutungslos.

Interfaces mit Testfällen ausliefern

Tatsächlich wäre es sinnvoll, zu verlangen, dass jedes Interface mit einer Suite von Testfällen ausgeliefert wird, die die Einhaltung des mit dem Interface verbundenen Vertrages prüft. Jede, die ein Interface implementieren würde, könnte dann mit geringem Aufwand testen, ob sie nicht nur die syntaktischen Vorgaben des Interfaces berücksichtigt hat, sondern auch die verhaltensbezogenen (semantischen). Angesichts der Wichtigkeit der Einhaltung von Verträgen für die Korrektheit von Software ist es eigentlich unbegreiflich, warum das nicht längst der Fall ist.

Inflation an Testklassen

Das präsentierte Verfahren zum Testen von Interfaces ist einfach und effektiv. Leider führt es auch zu einer Inflation an Testklassen, denn für jede Klasse muss pro implementiertem Interface eine neue Testklasse eingeführt werden. Wenn die Testklasse zudem regelmäßig (wie in obigen Beispielen) nichts weiter als eine Factory-Methode enthält, dann darf man sich fragen, warum es für JUNIT noch keinen Test runner gibt, der zu einer Interface-Testklasse alle Implementierungen des Interfaces per Reflektion zusammensucht und für jede eine Instanz als Fixture in die abstrakte Testklasse injiziert. Vielleicht eine schöne Übungsaufgabe für Sie?

Fazit

Durch die weite Verbreitung von JUNIT und dessen eher stiefmütterliche Unterstützung des Testens von Interfaces scheint es, dass dieses in der Praxis viel zu wenig Aufmerksamkeit erfährt, obwohl gerade das Testen von Interfaces ein essentieller Beitrag zur objektorientierten und komponentenbasierten Programmierung wäre. Wenn man Interfaces ernstnimmt, dann sollte die syntaktische Spezifikation der Methodensignaturen immer über Verträge (Vor- und Nachbedingungen) *und* Tests mit einer Semantik versehen werden.

3.5 Testen mit Mock-Objekten

Idealerweise testet man beim Unit-Testen jede Klasse einzeln.³⁶ Hängt die Funktionalität der zu testenden Klasse von anderen, mit der zu testenden kollaborierenden ab und kann deren Funktionalität selbst fehlerhaft oder nicht oder nur zeitweise verfügbar sein, ersetzt man deren Objekte, soweit sie zum Testen benötigt werden, durch sog. **Mock-Objekte**. Diese Mock-Objekte müssen jedoch nicht die gesamte Funktionalität der Objekte der Originalklasse anbieten, sondern nur die, die für das Testen gebraucht wird. Auch muss die Funktionalität (das Ein-/Ausgabeverhalten) nicht echt sein, sondern kann simuliert werden. So kann eine Methode eines Mock-Objekts immer denselben Wert zurückgeben, wenn im Rahmen der Unit-Tests, die es verwenden, immer derselbe Wert erwartet wird.

Ein weiteres Problem beim Testen (nicht nur beim Unit-Testen) ist, dass Fehler nicht automatisch an der Stelle auffallen, an der sie auftreten, sondern manchmal erst sehr viel später. Wenn beispielsweise ein Objekt auf ein Dateiobjekt schreibend zugreifen will und das Öffnen der Datei nur mit Leseberechtigung durchgeführt hat, offenbart sich der Fehler beim ersten Schreibversuch und nicht an der Stelle, an der das fehlerhafte Öffnen durchgeführt wurde. Während dieser konkrete Fehler noch leicht zu finden ist, ergeben sich in der Praxis wesentlich komplexere Zusammenhänge, die die Fehlerursache verschleiern. Auch besteht ein Fehler nicht selten daraus, dass eine erforderliche Methode gar nicht aufgerufen wurde, im obigen Beispiel also wenn das Öffnen ganz vergessen wurde. Mock-Objekte können solche Fehler genauer einkreisen, indem sie selbständig überwachen, dass ihre Methoden in der richtigen Reihenfolge und mit den richtigen Parametern aufgerufen werden. Ohne Mock-Objekte kann man solche Versäumnisse selbst dann nicht feststellen, wenn die aufgerufenen Objekte über ausreichende Möglichkeiten der Statusabfrage verfügen, nämlich dann, wenn diese Objekte den Testmethoden unzugänglich sind, also innerhalb der zu testenden Objekte erzeugt werden und die entsprechenden Variablen, die die Objekte halten, für die Testmethode nicht sichtbar sind.³⁷

3.5.1 Ausprogrammierte Mock-Objekte

Das Testen einer Klasse von Objekten unter Zuhilfenahme von Mock-Objekten soll, soweit möglich, ohne Änderungen an der Klasse oder der Klasse der zu mockenden Objekte erfolgen. Andernfalls bekäme man nämlich ein Wartungsproblem, schon weil man den speziellen Testteil der Klasse immer den aus dem Programm heraus notwendigen Änderung nachführen müsste. Wir werden aber sehen, dass es nicht immer ohne Programmänderungen, die speziell für das Testen durchgeführt werden, geht.

³⁶ Beim Unit-Testen wird per Definition immer genau eine Unit getestet. JUNIT tut jedoch nichts, dies zu erzwingen. Insofern ist sein Name falsch — es müßte eigentlich JREGRESSION heißen.

³⁷ Auch insofern ist JUNIT 4 ein Fortschritt: Es erlaubt, Testfälle in derselben Klasse wie die zu testenden Methoden zu definieren und gestattet somit zumindest den Zugriff auf als privat deklarierte Instanzvariablen. Der Zugriff auf temporäre Variablen bleibt jedoch auch dann verwehrt.

Nach der langen Vorrede nun ein einfaches Beispiel³⁸. Gegeben sei die folgende zu testende Methode `run()` einer zu testenden Klasse `Application`:

```
650 class Application {  
651     ...  
652     public void run() {  
653         View v = new View();  
654         v.display();  
655     ...
```

Ein einfacher JUNIT-Testfall, der die Klasse testet, könnte wie folgt aussehen:

```
656 class ApplicationTest extends TestCase {  
  
657     public void testApplication {  
658         Application a = new Application();  
659         a.run();  
660         assert...  
661     }  
662     ...
```

Wie man sofort sieht, hängt die Klasse `Application` von der Klasse `View` ab. Deren Instanzierung oder weitere Verwendung kann fehlschlagen, ohne dass dies der zu testenden Klasse anzulasten wäre, so dass man sie zum Zweck des Testens gerne gegen eine austauschen würde, die immer funktioniert. Auch kann die Erzeugung oder Verwendung einer Instanz teuer sein, ein Aufwand, den man sich, wenn der Unit-Test häufig aufgerufen wird, gern sparen würde. Die Einführung eines Mock-Objekts scheint also gerechtfertigt. Es bleibt allerdings die Frage, wie man es gegen das „echte“ Objekt austauschen kann — der obige Code bietet dafür keinen Ansatzpunkt.

Das Beispiel offenbart aber noch ein zweites, bereits oben erwähntes Problem: Der Testfall könnte prüfen wollen, ob eine Instanz von `View` erzeugt und angezeigt wird. Das aber ist schwierig, da die Instanz in einer lokalen Variable hinterlegt wird und sie somit für den Testfall nicht zugänglich ist. Man müsste also die Klasse so umschreiben, dass die `View` in einer Instanzvariable gespeichert wird, die dann auch noch für den Testfall sichtbar sein muss. Dies ist natürlich nicht so schön. Auf der anderen Seite gibt es so, wie der Code derzeit gestrickt ist, keine Möglichkeit, an die `View` heranzukommen, so dass eine gewisse Änderung unvermeidbar ist.

Eine Möglichkeit, die vergleichsweise wenig Änderung verlangt, ist die, die neue `View` nicht in der zu testenden Methode `run()` zu erzeugen, sondern dies in eine Factory-Methode (wie

³⁸ übernommen aus <http://www.ibm.com/developerworks/library/j-mocktest.html>

schon beim Testen von Interfaces in Abschnitt 3.4; s. Abschnitt 4.4.6) auszulagern. Das würde dann wie folgt aussehen:

```
663 class Application {  
664     ...  
665     public void run() {  
666         View v = createView();  
667         v.display();  
668     ...  
669     protected View createView() {  
670         return new View();  
671     }  
672     ...  
673 }
```

Man ahnt vielleicht schon, was passieren wird: Die Factory-Methode `createView()`, protected deklariert, kann in einer Subklasse überschrieben werden und dort ein geeignetes Mock-Objekt zurückgeben. JAVA bietet dazu die Möglichkeit, `createView()` in einer inneren anonymen Subklasse zu überschreiben, und zwar mittels folgender Syntax:

```
674     public void testApplication {  
675         Application a = new Application() {  
676             protected View createView() {  
677                 return mockView;  
678             }  
679         };  
680         a.run();  
681         assert...  
682     }
```

Es muss also keine neue Subklasse angelegt werden, sondern es wird einfach die Factory-Methode für die erzeugte Instanz (und nur für die) überschrieben. Diese Factory-Methode liefert das Mock-Objekt zurück, das zu diesem Zweck in der Variable `mockview` gespeichert wurde. Bleibt die Frage, wo das Mock-Objekt herkommt und Instanz welcher Klasse es ist. Dazu der folgende Code:

```
683     class ApplicationTest extends TestCase {  
684  
685         MockView mockView = new MockView();  
686  
687         public void testApplication {  
688             Application a = new Application() {  
689                 protected View createView() {  
690                     return mockView;  
691                 }  
692             };
```

```

693     a.run();
694     assert...
695 }
696
697 private class MockView extends View {
698     ...
699 }
700 }
```

Die Klasse **Mockview** ist also eine innere Subklasse der Klasse **View** und damit lokal zur Testklasse **ApplicationTest**; dies ist auch ganz richtig so, denn nur da wird sie gebraucht.

Die Klasse **Mockview** kann nun, als Subklasse von **View**, das benötigte Verhalten spezifizieren. (Man hätte zunächst auch **Mockview** als anonyme innere Subklasse definieren können, aber wie wir gleich sehen werden, braucht **Mockview** Methoden, die **View** nicht hat.) Dazu gehört sicher eine Implementierung von **display()**, die nicht fehlschlagen kann, die also z. B. gar nichts tut. Wenn man aber, wie oben angedeutet, durch den Testfall überprüfen lassen möchte, ob die **View** auch angezeigt, also die Methode **display()** aufgerufen wurde (man beachte, dass dieser Aufruf nicht durch den Testfall selbst erfolgt), dann muss man auch den *Zustand* des Mock-Objekts abfragen können, ja es muss überhaupt einen solchen haben. Die Implementierung von **Mockview** könnte also wie folgt aussehen:

```

701 private class MockView extends View {
702     boolean isDisplayed = false;
703
704     public void display() {
705         isDisplayed = true;
706     }
707
708     public void validate() {
709         assertTrue(isDisplayed);
710     }
711 }
```

Dabei übernimmt die Methode **validate()** die Funktion, die normalerweise mittels **assert...** im Testfall ausgedrückt wird, nämlich die Überprüfung des erwarteten Ergebnisses. Zeilen 681 und 694 oben können also durch

```
712 mockView.validate();
```

ersetzt werden. Alternativ könnte man natürlich auch den Status des Mock-Objekts von außen zugänglich machen und die Überprüfung im Testfall (mittels **assert...**) durchführen, aber spätestens, wenn mehrere Mock-Objekte in einem Testfall vorkommen, droht das Ganze unübersichtlich zu werden. Außerdem ist denkbar, dass das gleiche Mock-Objekt in mehreren Testfällen zu gleichen Bedingungen zum Einsatz kommen soll, so dass man in den Assert-Methoden jedes

Mal die gleichen Bedingungen ausdrücken müsste. Dies erspart man sich durch die Kapselung im Mock-Objekt selbst.

Eigentlich wären Mock-Objekte keine große Sache, wenn da nicht das *Typsystem* wäre. In typlosen Sprachen wie SMALLTALK beispielsweise reicht es aus, die im Rahmen des Testfalls vom Mock-Objekt benötigten Methoden in einer speziellen Klasse zu implementieren und dem zu testenden Objekt eine Instanz davon, das Mock-Objekt, als Kollaborationspartner unterzuschieben. Da nur genau die implementierten Methoden aufgerufen werden (und die in jedem Testfall mit den gleichen Eingaben und erwarteten Ausgaben, so dass die Methoden auch keine weiteren aufrufen müssen), kann es keine Probleme geben; insbesondere spielt der Typ des Mock-Objekts beim Test keine Rolle. In Sprachen mit *starker Typprüfung* hingegen muss das Mock-Objekt zuweisungskompatibel, d. h., entweder vom gleichen oder von einem Subtyp sein. Wenn es vom gleichen Typ ist, ist es Instanz der gleichen Klasse und damit kein Gewinn (oder man müsste die Klasse ersetzen); wenn es von einem Subtyp ist, müssen sich die zu mockenden Methoden auf jeden Fall überschreiben lassen (dürfen also nicht `final` deklariert sein). Da man das nicht von vornherein ausschließen kann, sind der Ausprogrammierung von Mock-Objekten Grenzen gesetzt.

Hindernis Typsystem

3.5.2 Mock-Frameworks

Das obige Vorgehen zeigt, wie man ein Objekt, von dem das zu testende Objekt abhängt, durch ein Mock-Objekt ersetzen kann. Die dazu notwendigen Änderungen am zu testenden Code und auch am Testfall waren vergleichsweise gering. Höher ist da schon der Aufwand für das Erstellen der neuen Subklasse, die das Mock-Objekt liefert. Da die Erfahrung gezeigt hat, dass diese Klassendefinition stets einem bestimmten Muster folgt, erzeugen viele der heute gebräuchlichen Mock-Frameworks für JAVA „künstlich“ Objekte, die den geforderten Typ und alle benötigten Methoden enthalten, die aber nicht durch Instanziierung einer kompilierten Klasse entstehen. Die dazu verwendeten Bibliotheken sind nicht Teil der JAVA-Sprachdefinition oder der Standard-Klassenbibliothek; sie erzeugen vielmehr direkt (also unter Umgehung des Compilers) den benötigten Bytecode und müssen sich damit nur an die Spezifikation der JAVA Virtual Machine (JVM) und nicht an die JAVA-Sprachdefinition halten (deswegen von mir auch „künstliche“ Objekte genannt). Diese Bibliotheken bedienen sich der Möglichkeit der *Metaprogrammierung* (Kurseinheit 6), auf die an dieser Stelle aber nicht weiter eingegangen werden soll.

Mit der Erzeugung eines Objekts des benötigten Typs ist es aber nicht getan — wie bereits das obige Beispiel andeutete, muss das Objekt auch noch die im Rahmen des Testfalls benötigten Methoden besitzen, die obendrein die erwarteten Ergebnisse zurückliefern müssen. Dazu genügt es in der Regel nicht, dass ein Mock-Objekt auf den Aufruf einer bestimmten Methode immer denselben Wert zurückgibt (eine sog. *konstante Methode*; vgl. Kurs 01814): Es kann nämlich sein, dass dieselbe Methode im Rahmen eines Testfalls mehrfach aufgerufen wird und dabei jeweils unterschiedliche Ergebnisse liefern muss. Statt eines Rückgabewertes muss also eine Sequenz von Rückgabewerten spezifizierbar sein. Um den Code, der das realisiert, nicht jedes Mal von Hand (in Form entsprechender Methodenimplementierungen) erstellen zu müssen, erlauben gebräuchliche Mock-Frameworks eine parametrische Definition (durch Aufruf entsprechender API-Methoden des Frameworks).

Spezifikation der benötigten Methoden

Es bleibt das Problem, wie man dem zu testenden Objekt die Mock-Objekte unterschiebt. Im obigen Beispiel wurde das ja durch Einführung einer dynamisch gebundenen und in einer für den Testfall speziell erzeugten Subklasse überschriebenen Factory-Methode erreicht. Es findet hier also eine gewisse Form der Dependency injection (s. Abschnitt 1.6) statt (auch wenn das zu testende Objekt jetzt einen anderen Typ hat, der allerdings anonym bleibt). Bevor ich weiter darauf eingehere, sollen eben schnell die Alternativen exploriert werden.

Austausch der Klassen

Wenn das zu testende Objekt seine Kollaborationspartner selbst erzeugt und dazu den Konstruktor der betreffenden Klasse(n) aufruft, geht dies nur, indem man deren Klassen austauscht. Ein solcher Austausch müsste aber in der Regel global erfolgen, da nicht ausgeschlossen werden kann, dass ein Mock-Objekt im Rahmen des Tests an andere Objekte (als Parameter eines Methodenaufrufs) weitergeleitet werden muss, so dass dieser Parameter denselben Typ haben muss, usw. Außerdem kann nicht ausgeschlossen werden, dass andere Tests (so z. B. die der Klasse des zu mockenden Objekts) Instanzen der Klasse benötigen, die nicht gemockt sind, so dass die originale und die gemockte Klasse parallel existieren und u. U. sogar dieselben Programmelemente typisieren müssen. Da das nicht geht, muss dafür testfallweise kompiliert werden, was (wie leicht einzusehen) keine echte Option ist.

Metaprogrammierung

Eine Alternative wäre, dem zu testenden Objekt die Mock-Objekte mit den Mitteln der *Metaprogrammierung* (Kurseinheit 6), genauer der *Interzession* (Abschnitt 6.1.3), unterzujubeln. Dazu könnte z. B. der Konstruktorauftrag im Rahmen des Testfalls abgefangen und anstelle einer Instanz der Klasse ein Mockobjekt zurückgegeben werden. Für solche Manipulationen ist die aspektorientierte Programmierung (Abschnitt 6.3) à la ASPECTJ (Abschnitt 6.3.6) besonders gut geeignet (auch wenn es sich konzeptuell bei der Einbringung von Mock-Objekten zu Testzwecken nicht wirklich um einen Aspekt des Programms handelt). Diese erfreut sich jedoch nicht uneingeschränkter Beliebtheit und ist schon von daher keine Patentlösung; auch ist die Werkzeugunterstützung und Integration in gängige IDEs heute noch nicht so, dass der Nutzen die Nachteile in jedem Fall aufwiegen würde.

Dependency injection

Was aber bleibt, ist die Möglichkeit, den Code so zu refaktorieren, dass das zu testende Objekt die zu mockenden Objekte nicht selbst erzeugt, sondern sie ihm von außen eingeflößt, oder injiziert, werden. Da die *Dependency injection* allgemein nützlich ist³⁹, ist eine Änderung eines zu testenden Programms hin zur Verwendung derselben in der Regel kein Schaden und das oben angedeutete Wartungsproblem besteht zumindest dann nicht, wenn auch die reguläre Verwendung von Objekten der Klasse auf die Dependency injection umgestellt wird. Dabei ist es gleich, welche Form der Dependency injection verwendet wird, solange sie nur auch für den Testfall zugänglich ist. Und so verlangen die meisten der heutigen gebräuchlichen Mock-Frameworks die Verwendung von Dependency injection als Voraussetzung.

³⁹ Dies sollte aus Abschnitt 1.6 hinreichend deutlich geworden sein. Manche behaupten allerdings, die Dependency injection diene einzig und allein dem Zweck des Testens mit Mock-Objekten.

3.5.3 Grenzen der Einsetzbarkeit von Mock-Objekten

Das Testen mit Mock-Objekten, die den zu testenden Objekten als Instanzen anderer Klassen untergeschoben werden, basiert auf dynamischem Binden: Anstelle der Methoden des Originalobjekts werden die des Mock-Objekts aufgerufen. Dies funktioniert freilich nicht mehr, sobald die aufgerufenen Methoden statische sind, also nicht dynamisch gebunden werden. Auch dürfen die Methoden natürlich nicht final (also als nicht überschreibbar) deklariert sein.

Mock-Objekte der oben geschilderten Art können ebenfalls nicht eingesetzt werden, wenn die Methodenaufrufe auf den Originalobjekten Nebeneffekte haben, die sich in anderen als gemockten Objekten (einschließlich dem zu testenden Objekt selbst) manifestieren und die für die erfolgreiche Durchführung des Tests notwendig sind. Diese Nebeneffekte fallen nämlich (aufgrund der Elimination der sie auslösenden Methodenaufrufe oder Feldzugriffe) aus und können nicht anders als durch weitere Mock-Objekte simuliert werden. Die Verwendung von Mock-Objekten, deren Gebrauch durch das zu testende Objekt vom Prinzip her ein *anbietender* (vgl. Abschnitt 1.5.2) ist, ist damit in der Regel auf isolierte Client-/Server-Konstellationen (also solche, in denen der Server autonom agiert) beschränkt.

3.5.4 Mock-Objekte bei ermöglichenen Interfaces

Spielen die Mock-Objekte die Rolle der Nutznießerin (bei *ermöglichen*dem Gebrauch; s. Abschnitt 1.5.8), ergeben sich schon aus der Natur der Sache heraus in der Regel keine Einschränkungen der Einsetzbarkeit von Mock-Objekten: Das Mock-Objekt gibt sich einfach mit dem ihm gewährten Dienst zufrieden. Wenn hingegen das Testergebnis gerade aus dem Aufruf von bestimmten Methoden des Mock-Objekts, vielleicht sogar in einer bestimmten Reihenfolge, besteht, dann muss das Mock-Objekt Protokoll über die Aufrufe führen oder wird sogar selbst zum *Testorakel* (in dem Sinne, dass es Abweichungen von der erwarteten Reihenfolge dem Testframework als Fehler signalisiert). Da trifft es sich günstig, wenn Mock-Objekte schon zum Zweck der Fehlereingrenzung (s. Beginn des Abschnitts 3.5) solche Protokolle führen; der Test der richtigen Dienstleistung (im Sinne eines ermöglichenen Interfaces) durch das zu testende Objekt fällt dabei quasi gratis ab.

3.6 Auswertung von Unit-Tests zur Fehlerlokalisierung

Einen Fehler machen und sich nicht bessern: Das erst heißt fehlen.

Konfuzius

Idealerweise würde ein Testframework nicht nur den Test anzeigen, der fehlgeschlagen ist, sondern auch die Stelle im Quellcode, die für den Fehler verantwortlich ist. Dies würde das Testen auf die Ebene der syntaktischen und semantischen (d. h. die Typkorrektheit betreffenden; vgl. Abschnitt 3.9) Prüfungen heben, die vom Compiler durchgeführt werden und deren Ergebnis ja ebenfalls nicht nur in der Angabe der verletzten Regel, sondern auch in der Stelle, an der sie

verletzt wurde, besteht [32]. Es stellt sich allerdings das Problem, wie der Zusammenhang von gescheiterten Tests und verursachenden Programmteilen hergestellt werden kann. Da der kausale Zusammenhang vom fehlerhaften Programmteil zum gescheiterten Testfall führt und dieser Zusammenhang im allgemeinen nicht bijektiv, also nicht eindeutig umkehrbar, ist, handelt es sich hierbei um ein klassisches Diagnoseproblem: Die Symptome (die gescheiterten Testfälle) können durch mehrere Diagnosen (mögliche Fehler im Programm) erklärt werden, wovon jedoch die meisten in der Regel nicht zutreffend sind. Daraus resultiert eine oftmals mühsame Suche.

Für die Zuordnung von (gescheiterten) Testfällen zu (möglicherweise verursachenden) Programmteilen ergeben sich mehrere Möglichkeiten:

1. Testfälle können über *Annotationen* (Metadaten) manuell an die zu testenden Einheiten gebunden werden.
2. Die Menge der von einem Testfall (einer Testmethode) aufgerufenen Programmteile (Methoden) kann durch eine statische Programmanalyse bestimmt werden. Hierzu wird anhand der im Quelltext vorkommenden Methodenaufrufe ein Aufrufgraph erzeugt.
3. Alternativ kann die Menge der von einem Testfall (einer Testmethode) aufgerufenen Programmteile (Methoden) auch durch Tracen (also das Verfolgen eines konkreten Programmablaufs) bestimmt werden.

Dabei verlangt die erste Möglichkeit die manuelle Wartung der Annotationen und ist somit sehr fehleranfällig. Die zweite Möglichkeit führt in der Regel zu zahlreichen Methoden, die niemals aufgerufen werden, und zwar wegen expliziter und impliziter (per *dynamischem Binden*) Verzweigungen, von denen — aufgrund des immer gleichen Ablaufs von Testfällen — ja immer nur ein Zweig durchlaufen wird. Gleichzeitig kann sie aber auch zu wenige Methoden enthalten, nämlich dann, wenn der Methodenaufruf reflektiv erfolgt (s. Abschnitt 6.1). So bleibt eigentlich nur die dritte Möglichkeit, nämlich die Ausführung von Testfällen mitzuschneiden (zu tracen). Dafür bedarf es allerdings einer speziellen Instrumentierung der Testausführung, die in der Regel nicht mitgeliefert wird.

3.6.1 Abdeckungsbasierte Fehlerlokalisierung

Wenn man erst einmal weiß, welche Programmeinheiten⁴⁰ durch welche Testfälle aufgerufen werden (die sog. *Testabdeckung*), lässt sich eine Tabelle wie die nachfolgende aufstellen:

GETESTETE PROGRAMMEINHEIT	TESTFÄLLE						FEHLERHAFTIGKEIT	
	fehlgeschlagen			erfolgreich			tatsächlich	errechnet
	t_1	...	t_m	t_{m+1}	...	t_a		
e_1	$x_{1,1}$...	$x_{1,m}$	$x_{1,m+1}$...	$x_{1,a}$	f_1	$L(e_1)$
...
e_n	$x_{n,1}$...	$x_{n,m}$	$x_{n,m+1}$...	$x_{n,a}$	f_n	$L(e_n)$

⁴⁰ Die sog. *Granularität der Fehlerlokalisierung* ist variabel: sinnvolle Auflösungen sind Klassen, Methoden und einzelne Anweisungen

In dieser Tabelle, in der die Testfälle in Spalten angeordnet und nach fehlgeschlagenen und erfolgreichen sortiert sind, steht ein $x_{i,j}$ für eine 1, wenn die Einheit e_i von Testfall t_j ausgeführt wurde, andernfalls für eine 0. Die Kunst ist nun, aus dieser Tabelle (bzw. der dazugehörigen binären Matrix) diejenigen Programmeinheiten e_i zu bestimmen, die tatsächlich fehlerhaft sind (in der Tabelle durch eine 1 anstelle des f_i gekennzeichnet). Dies macht man mithilfe sog. **Fehlerlokatoren** L , die für jede Einheit e_i einen Wert berechnen, der aussagt, ob der Lokator die Einheit für fehlerhaft hält. Lokatoren können binär (also für jede Einheit 0 oder 1 liefern) oder graduell (kontinuierlich) sein; in letzterem Fall normiert man ihr Ergebnis in der Regel auf das Intervall $[0; 1]$.⁴¹ Für einen perfekten Fehlerlokator L ist $L(e_i) = f_i$ für alle $1 \leq i \leq n$. [33]

Den Möglichkeiten der Definition von Fehlerlokatoren auf Basis obiger Tabelle (die aufgrund der Tatsache, dass sie auf Testabdeckung basieren, auch **abdeckungsbasierte Fehlerlokatoren** genannt werden) sind kaum Grenzen gesetzt. Ein erster naiver Ansatz berechnet für jede Einheit e_i das Verhältnis der Anzahl Ausführungen durch gescheiterte Testfälle zu der Gesamtzahl der Ausführungen (gescheiterte plus erfolgreiche Testfälle):

$$L_1(e_i) = \frac{\sum_{j=1}^m x_{i,j}}{\sum_{j=1}^a x_{i,j}}$$

Danach erhält eine Programmeinheit e_i einen Wert von 1, wenn jeder der sie ausführenden Testfälle scheitert, einen von 0,5, wenn genau die Hälfte, und einen von 0, wenn keiner scheitert. Dieser Ansatz hat allerdings den Nachteil, dass er einer Einheit, die von nur einem Testfall ausgeführt wird und dieser scheitert, genauso bewertet wie eine, die von allen gescheiterten Testfällen ausgeführt wird. Man würde nämlich annehmen, dass letztere mit einer höheren Wahrscheinlichkeit falsch ist, weil das alle gescheiterten Testfälle mit einem Schlag erklärt. Dies wird durch folgende Definition berücksichtigt:

$$L_2(e_i) = \frac{\sum_{j=1}^m x_{i,j}}{m} \cdot L_1(e_i) = \frac{\left(\sum_{j=1}^m x_{i,j} \right)^2}{m \sum_{j=1}^a x_{i,j}}$$

$L_2(e_i)$ ist das Quadrat des Ochiai-Ähnlichkeitskoeffizienten zwischen dem Zeilenvektor $\vec{e}_i = (x_{i,1} \dots x_{i,a})^T$ und dem a -elementigen Vektor, dessen erste m Elemente 1 sind und der Rest 0 (entsprechend einem Programmelement, das von allen scheiternden Testfällen und nur von diesen aufgerufen wird, so dass es sämtliches Scheitern vollständig erklärt). Dass L_2 besser ist als L_1 basiert allerdings auf der Annahme, dass es wahrscheinlicher ist, weniger Fehler zu haben als mehr.

⁴¹ Das Einheitsintervall legt die Interpretation des Ergebnisses als Wahrscheinlichkeit nahe. Man kann dabei jedoch nur dann von einer Wahrscheinlichkeit sprechen, wenn die Kolmogorov-Axiome erfüllt sind, wenn sich also insbesondere die Wahrscheinlichkeiten von sich gegenseitig ausschließenden Aussagen zu 1 addieren. Dies ist für obige Tabelle nur dann möglich, wenn man es sicher nur mit einer fehlerhaften unter den zu testenden Einheiten zu tun hat (die sog. *Einzelfehlerannahme*, engl. *single-fault assumption*). Das ist aber in der Regel nicht der Fall.

3.6.2 Modellbasierte Fehlerlokalisierung

Den obigen Fehlerlokatoren L_1 und L_2 ist gemeinsam, dass sie für ein e_i jeweils nur Tabelleneinträge aus der Zeile i heranziehen. Sie betrachten also alle Testfälle für eine Programmeinheit, aber eben nur für die. Dabei bleibt die Information, die aus der Betrachtung aller Programmeinheiten, die von einem Testfall ausgeführt werden, hervorgeht, gänzlich unberücksichtigt [33].

Diese Information kann aber auch sehr aufschlussreich sein. Wenn beispielsweise ein scheiternder Testfall nur eine Programmeinheit ausführt, dann muss diese fehlerhaft sein, egal wie viele Testfälle sonst noch scheitern und welche Einheiten diese ausführen. Wenn weiterhin zwei Testfälle scheitern, deren Testabdeckungen disjunkt sind, dann muss man davon ausgehen, dass man es mit mindestens zwei fehlerhaften Programmelementen zu tun hat. Tatsächlich gilt für jeden scheiternden Testfall t_j , dass mindestens eine der von ihm ausgeführten Programmeinheiten fehlerhaft sein muss, d. h.,

$$\forall 1 \leq j \leq m: \bigvee_{i: x_{i,j}=1} f_i$$

Man kann also aus einer Tabelle wie der obigen eine Klauselmenge⁴² ableiten, die ein SAT-Problem darstellt, dessen Lösungen das Scheitern aller Testfälle erklären, also mögliche Diagnosen darstellen. Man nennt ein solches Verfahren zur Fehlerlokalisierung auch *modellbasierte Diagnose*, weil es auf einem Erklärungsmodell für das Scheitern („ein Testfall scheitert genau dann, wenn ...“) basiert.

3.6.3 Andere Arten von Fehlerlokatoren

Neben abdeckungsbasierten (einschließlich der obigen modellbasierten) Fehlerlokatoren gibt es noch zahlreiche andere Arten. Einige beziehen sich auf historische Daten (wann lief ein scheiternder Testfall das letzte Mal durch und welche Änderungen wurden seither gemacht?), andere, sog. *A-priori-Fehlerlokatoren* [32] auf Komplexitäts- oder andere Maße, die einen Fehler in einem Programmelement *a priori* wahrscheinlicher erscheinen lassen als in anderen (wie viele Fallunterscheidungen werden in einer Methode gemacht?). Ein perfekter Fehlerlokator für alle Fälle ist nicht bekannt (und wird wahrscheinlich auch nie gefunden werden), aber wenn der Preis für die Fehlerlokalisierung niedrig ist und sie in Treffsicherheit die der Erfahrung und Intuition von Entwicklerinnen erreicht oder vielleicht sogar übertrifft, kann sie einen nützlichen Beitrag zur rascheren Fehlerbehebung liefern.

3.6.4 Kombination von Fehlerlokatoren

Wie so häufig, wenn ein einzelner Ansatz zu keinem befriedigenden Ergebnis führt, kann man versuchen, verschiedene Ansätze miteinander zu kombinieren. Tatsächlich gleicht die Fehlersuche in einem Programm nicht selten einem kriminalistischen Indizienprozess: Jedes Beweisstück für sich genommen lässt viele Verdächtige zu, aber wenn man die Evidenz kombiniert, häufen

⁴² Eine Klausel ist eine Disjunktion von Literalen. Ein Literal ist ein negiertes oder ein nicht negiertes Atom. Eine Klauselmenge steht für aussagenlogische Ausdrücke der Form $(L_1 \vee \dots \vee L_n) \wedge \dots$, wobei die L_i Literale sind.

sich meistens die Hinweise auf einige wenige Kandidaten. Ein Framework, das die Kombination beliebiger Fehlerlokatoren erlaubt und das Ergebnis in Analogie zu vom Compiler gemeldeten Fehlern in der JAVA-Entwicklungsumgebung ECLIPSE präsentiert (so dass die Programmiererin direkt zu den diagnostizierten Fehlerstellen navigieren kann), wird derzeit unter dem Namen EZUNIT entwickelt [32] [33]; der aktuelle Stand kann unter <http://www.fernuni-hagen.de/ps/prjs/EzUnit/> abgerufen werden.

3.7 Kontinuierliches Testen

Eine weitere Verbesserung des Testprozesses, die den Testvorgang auf die Ebene des Kompilationsvorgangs anhebt, ist das sog. **kontinuierliche Testen** (engl. continuous testing, <http://groups.csail.mit.edu/pag/continoustesting/>). Dabei wird ausgenutzt, dass nach einem erfolgreichen Durchlauf aller Tests (wie nach einem vollständigen Build des Programms) bei einer Änderung immer nur die Teile getestet werden müssen, die von der Änderung betroffen waren (wie beim inkrementellen Build). Da diese Tests (aufgrund des niedrigeren Testbedarfs) dann schneller fertig sind, können sie auch automatisch angestoßen werden und dabei die freie CPU-Zeit, die beim weiteren Editieren des Codes abfällt, nutzen.

3.8 Wer testet die Tests?

Wie schon der eingangs bemühte Merksatz von Dijkstra nahelegte, gibt der Umstand, dass ein Programm alle Tests passiert hat, keine Garantie dafür, dass es fehlerfrei ist: Die Abdeckung der Spezifikation durch Tests kann nur in seltenen Fällen vollständig sein. Es gibt aber noch eine andere, viel gemeinere Ursache für Fehler in als einwandfrei getesteten Programmen: Ein Test kann nämlich selbst fehlerhaft sein. Wenn man annimmt, dass ein Test nur zwei mögliche Ergebnisse hat (rot soll einen gefundenen Fehler in der getesteten Funktion anzeigen, grün soll Fehlerfreiheit gemäß Tests signalisieren), dann ergibt sich die in Tabelle 3.1 gezeigte Aufstellung von Möglichkeiten.

Man möchte zunächst vermuten, dass ein grünes Testergebnis⁴³ ein gutes Testergebnis ist. Bei genauerer Betrachtung von Tabelle 3.1 ist dies jedoch trügerisch. So kann ein Test trivialerweise immer grün liefern (und damit eigentlich falsch sein), ohne dass dies auffällt. Man muss dann schon sehr wachsam sein und vielleicht zufällig selbst einen Fehler in der zu testenden Funktion entdecken, um sich dann zu fragen, warum der Test denn keinen Fehler gemeldet hat. Während immergrüne Tests bei fehlerfreien Funktionen zumindest zunächst kein Problem darstellen (s. Zeile 3 in Tabelle 3.1), können falsch negative Ergebnisse die ganze Testerei in Frage stellen.

trügerische Sicherheit

⁴³ Die Sprechweise „grünes“ oder „rotes Testergebnis“ mag albern erscheinen, verhindert aber eine gefährliche Zweideutigkeit: Spräche man stattdessen von einem „falschen“ Testergebnis, so wäre nicht klar, ob ein Fehler gefunden wurde (die getestete Funktion also falsch ist) oder ob der Test selbst das falsche Ergebnis hat (also fehlerhaft ist).

Ursachen von falsch negativen Testergebnissen

Ursache von falsch negativen Testergebnissen kann der Umstand sein, dass sich getestete Funktion und Test auf dieselbe (Teil-) Funktion stützen und deren Richtigkeit voraussetzen, obwohl sie tatsächlich falsch ist. So könnte beispielsweise die Funktion `isEmpty` eines Stacks immer `true` zurückgeben. Beim Testen der Funktion `pop`, bei dem von einem einelementigen Stack ein Element entfernt und dann `isEmpty` geprüft wird, fällt z. B. dann nicht auf, dass `pop` gar kein Element vom Stack entfernt (der Fehler), wenn es intern erst auf `not isEmpty` prüft und der fehlerhafte Teil von `pop` somit nie ausgeführt wird. Diese Situation kann wirksam dadurch verhindert werden, dass auch `isEmpty` (in einem getrennten Test) getestet wird, und allgemein dadurch, dass man beim Schreiben von Testfällen *bottom up* vorgeht, also die primitiven Funktionen zuerst testet.

Tabelle 3.1: Aussage von positiven und negativen Testergebnissen. Ein grünes Licht bedeutet nicht automatisch, dass alles in Ordnung ist.

GETESTETE FUNKTION	TEST	TESTERGEBNIS	SCHWEREGRAD DES PROBLEMS	KOMMENTAR
korrekt	korrekt	grün	kein Problem	Im Prinzip ist alles OK, aber woher weiß man, dass der Test einen Fehler finden würde, wenn einer drin wäre?
korrekt	korrekt	rot	kommt nicht vor	Eines von beiden muss falsch sein!
korrekt	falsch	grün	erst dann ein Problem, wenn die Funktion geändert wird und sich dabei ein Fehler einschleicht, der durch den Test nicht aufgedeckt wird	Der Fehler im Test bleibt unbemerkt.
korrekt	falsch	rot	dann ein Problem, wenn man nicht daran denkt, dass der Fehler beim Test liegen könnte	In der Statistik spricht man auch von einem <i>falsch positiven</i> Ergebnis (weil der Test etwas aufgedeckt hat, das gar nicht da ist).
falsch	korrekt	grün	kommt nicht vor	Der Test prüft nicht auf den Fehler und ist somit falsch.
falsch	korrekt	rot	kein Problem	Dies ist der Standardfall — der Test ist sein Geld wert!
falsch	falsch	grün	großes Problem	Beide Fehler (der im Test und der in der getesteten Funktion) passieren unbemerkt den Unit-Test (ein sog. <i>falsch negatives</i> Ergebnis).
falsch	falsch	rot	mittleres Problem	Man wird zunächst einen Programmfehler suchen. Hat man ihn gefunden und repariert und zeigt der (fehlerhafte) Test immer noch Fehler an, dann wird man erst nach weiteren Fehlern suchen, bevor man an einen fehlerhaften Test (mit falsch positivem Ergebnis) denkt.

Weit schwieriger zu verhindern ist der Fall, dass sich die Fehler in der getesteten Funktion und im Programm gegenseitig aufheben — hier bleibt nur zu hoffen, dass der Programmfehler Auswirkungen hat, die indirekt Tests einer anderen Funktion scheitern lassen (weil hier die Richtigkeit der unerkannt fehlerhaft gebliebenen Funktion vorausgesetzt wird).

So gesehen ist ein rotes Ergebnis zunächst einmal ein gutes Ergebnis: Es zeigt immerhin an, dass der Fehlersuchmechanismus gegriffen hat (egal, ob der Fehler in der getesteten Funktion oder im Test selbst lag), und erlaubt somit, Gegenmaßnahmen zu ergreifen (und das selbst dann, wenn der — fehlerhafte — Test den tatsächlichen Fehler in der zu testenden Funktion gar nicht bemerkt hat). Wie aber findet man die Fehler in falsch negativen Tests?

Unit-Tests eignen sich aufgrund ihres Aufbaus nicht selbst für Unit-Tests à la JUNIT. Statt dessen ist, wie sich bereits oben andeutete, die getestete Einheit selbst eine Art Testsuite für die Tests, ist sie doch immerhin in der Lage, falsch positive Tests zu entlarven: Wenn sich, nach eingehender Untersuchung, herausstellt, dass die Einheit für gegebene Eingaben die richtige Ausgabe geliefert hat und ein Test trotzdem rot anzeigt, muss der Test falsch gewesen sein. Falsch negative Tests kann man nun auf einfache Weise entdecken, indem man in eine korrekte Funktion wissentlich Fehler einbaut und die Unit-Tests daraufhin überprüft, ob sie die Fehler finden. Der Fehlereinbau kann manuell oder automatisch (durch einen Programmtransformator) erfolgen; wichtig ist nur, dass man nicht vergisst, die Fehler hinterher wieder zu entfernen (was einem bei falsch negativen Tests schon mal passieren kann und doppelt tragisch wäre)!

Man unterscheidet verschiedene Verfahren zum Einbau von Fehlern in Programme zum Testen der Tests, so u. a. das sog. *Error seeding* und das *Mutation testing*. Ersteres bringt gezielt Fehler in ein Programm ein (was einen gewissen Sachverstand erfordert und deswegen nur relativ schwer zu automatisieren ist), während letzteres ein Programm zufällig abwandelt (eben mutiert), ein Prozess, der auch in der genetischen Programmierung zur Anwendung kommt. Auch wenn die Frage nach der Qualität der Tests eine ganz wesentliche ist, scheinen Error seeding und Mutation testing als Techniken in der JAVA-Gemeinde keine besondere Rolle zu spielen, was vermutlich damit zusammenhängt, dass die Werkzeugunterstützung noch in den Anfängen steckt. Das Programm JESTER (<http://jester.sourceforge.net>) bildet hier eine erwähnenswerte Ausnahme.

Verfahren zum Testen von Tests

3.9 Unit-Testen, Design by contract, Typprüfung – drei Wege, ein Ziel

Genau wie das Design by contract dient auch das Unit-Testen dem Ziel, die Einhaltung der Spezifikation zu überprüfen.⁴⁴ Ähnlich wie beim Design by contract wird dabei die Spezifikation auf die Funktion kleiner Einheiten, Methoden und Klassen, heruntergebrochen und besteht im Wesentlichen aus der Angabe von Paaren aus Ein- und zugehöriger Ausgabe (beim Design by

⁴⁴ Von einer Gewährleistung der Einhaltung der Spezifikation kann man nach heutigem Stand der Technik ja wie gesagt nicht sprechen, weil diese immer nur fallweise und somit stichprobenartig überprüft wird.

contract jeweils generisch, in Form von allgemeinen Vor- und Nachbedingungen spezifiziert). Die Korrektheit des gesamten Systems ergibt sich dann idealerweise aus der Korrektheit seiner Komponenten — da sich aber auch bei der Komposition Fehler einschleichen können, ist dies in der Regel nicht ausreichend. Vielmehr sind noch sog. *Integrationstests* durchzuführen, die hier aber nicht Gegenstand der Betrachtung sind.

Typprüfung als eingeschränkte Form des Design by contract

Nun gibt es aber noch eine dritte Möglichkeit, die Korrektheit der Abbildung von Eingabe- auf Ausgabewerte zumindest grob zu prüfen: die Verwendung eines *Typsystems*. Wie bereits zu Beginn von Abschnitt 2.2 Ein paar einfache Beispiele erwähnt, werden durch die Angabe von Typen zu den Ein- und Ausgabeparametern einer Methode Vor- und Nachbedingungen spezifiziert, die von den Aufruferrinnen der Methode bzw. von der Methode selbst einzuhalten sind. Im Falle der *statischen Typprüfung* handelt es sich dabei um eine Form des *Verified design by contract*, also um einen formalen, von der tatsächlichen Ausführung des Programms unabhängigen Beweis, dass bestimmte Fehler (nämlich die, die zu einer Typverletzung führen würden; vgl. Kurs 01814) nicht vorkommen. Diese Fehlerklasse ist zugegebenermaßen nicht besonders groß, aber die Einfachheit und Schnelligkeit der statischen Typprüfung sind ein starkes Argument für deren Verwendung zur Fehlervermeidung.

Die in letzter Zeit vielgepriesene dynamische Typprüfung (sog. *dynamischer* oder *dynamisch typisierter Sprachen* wie SMALLTALK oder RUBY; vgl. aber Kurs 01814, warum diese Bezeichnungen unsinnig sind) gibt es entweder gar nicht (in SMALLTALK beispielsweise gibt es keine Typen, so dass Variablen nicht typisiert und somit Zuweisungen auch nicht auf Typkorrektheit geprüft werden können — die oft als Typprüfung missverstandene Fehlermeldung „message not understood“ erfolgt erst nach einer Zuweisung und ist im Prinzip nicht mehr als die Meldung einer Speicherschutzverletzung) oder wird zur Laufzeit durchgeführt. Dann ist sie aber nur eine ausdrucksschwache Form des Non-verified design by contract — ausdrucksschwach deswegen, weil die Menge der ausdrückbaren Vor- und Nachbedingungen, wie in Abschnitt 2.2 Ein paar einfache Beispiele beispielhaft ausgeführt, denkbar klein ist, und non-verified, weil die Überprüfung nur fallweise und damit stichprobenartig stattfindet.

Zusammenhang von Unit-Testen, Design by contract und Typprüfung

Es ergibt sich also der folgende Zusammenhang von Unit-Testen, Design by contract und Typprüfung. Unit-Tests prüfen die korrekte Abbildung von Eingaben auf Ausgaben für genau vorherbestimmte Fälle. Die zu testenden Programmeinheiten werden zu diesem Zweck mit den Eingabewerten versorgt und ausgeführt. Beim Design by contract sind die Unit-Tests gewissermaßen in die Programmeinheiten integriert: Bei jeder Ausführung, ob während des Testens oder während der normalen Anwendung, werden die Eingaben in die Programmeinheiten auf Zulässigkeit und die Ausgaben auf Korrektheit im Sinne der durch die Vor- und Nachbedingungen ausgedrückten Spezifikation geprüft. Das Unit-Testen ist in der Regel genauer, weil es 1:1-Paarungen von Ein- und Ausgaben vorgeben kann; das Design by contract ist hingegen umfassender, weil es jede, noch so unvorhergesehene Eingabe in die Prüfung aufnimmt. Beiden ist gemeinsam, dass sie nur fallweise Prüfungen durchführen und die Überprüfung damit lückenhaft ist. Diese Lückenhaftigkeit ist dem Testen immanent (vgl. den Leitspruch eingangs dieser Kurseinheit), dem Design by contract jedoch nur technisch bedingt.

Das *Verified design by contract* hebt die Lückenhaftigkeit der heute üblichen, zur Laufzeit überprüfenden Instrumentierung des Design by contract auf, indem es einen formalen Beweis führt, dass die Einhaltung der Nachbedingungen aus der Einhaltung der Vorbedingungen stets folgt. Es ist damit unabhängig von der konkreten Ausführung eines Programmteils mit bestimmten oder zufälligen Eingaben. Während sich das Verified design by contract bis heute jedoch der praktischen Umsetzung weitgehend entzieht, bietet die statische Typprüfung von Sprachen wie JAVA eine kleine, aber hocheffizient realisierbare Teilmenge des Verified design by contract, mit der die Einhaltung der Wertebereiche von Ein- und Ausgaben statisch, also zur Übersetzungzeit, überprüft werden kann. Die Alternative zur statischen Typprüfung, die dynamische Typprüfung, stellt im Grunde keine wirkliche Alternative dar, da sie von allen Ansätzen die wenigsten Möglichkeiten bietet: Sie vereint die Ausdrucksschwäche der Typprüfung allgemein (es können keine Zusammenhänge zwischen den Werten verschiedener Variablen oder den Werten vorher und nachher ausgedrückt werden; s. Abschnitt 2.2 Ein paar einfache Beispiele) mit der mangelnden Allgemeingültigkeit von Tests und von nicht verifiziertem Design by contract. Wer also auf eine dynamische Prüfung setzt, sollte sich wenigstens von dem engen Korsett der Typprüfung lösen und ein Design by contract mit beliebigen Vor- und Nachbedingungen anstreben.

Zusammenfassung

Unit-Tests sind ein wirksames Mittel, Fehler in Programmen zu finden, aber Fehlerfreiheit garantieren sie nicht. Wie so häufig, wenn keine einzelne Methode optimal ist, lassen sich die besten Ergebnisse durch eine geschickte Kombination mehrerer erzielen. So ergänzen sich Vor- und Nachbedingungen à la Design by contract mit den Unit-Tests à la JUNIT wirkungsvoll. Genauso, wie es den Programmiererinnen lieb gewordene Pflicht sein sollte, für jede Methode Vor- und Nachbedingungen zu formulieren, sollte man für jede Methode eine Anzahl von Testfällen schreiben, anhand derer sich zu erwartende Programmierfehler aufdecken lassen.

3.10 Weiterführende Literatur

Es gibt eine ganze Reihe von Büchern zu JUNIT, von denen ich hier keins ausdrücklich empfehlen möchte. Die klassischen Einführungen in die Verwendung sind [29] und [30]. Eine konkrete Alternative zu JUNIT ist beispielsweise TESTNG (<http://testng.org>); es ist jedoch fraglich, ob ein anderes Open-Source-Werkzeug in der Lage ist, die Dominanz von JUNIT zu durchbrechen.

Ganz interessant liest sich übrigens [31]; der dort vorgeschlagene Weg, ein Design zu entwickeln bzw. zu beschreiben, hätte wohl nach Vorstellung der Autoren Schule machen sollen, ist vermutlich aber für das Alltagsgeschäft der meisten Programmiererinnen, dessen Anforderungen kaum Kreativität erlauben, eher wenig hilfreich.

Das Thema Fehlerlokalisierung (Abschnitt 3.6) war eine Zeit lang recht populär, was nicht verwundern sollte, darf man doch davon ausgehen, dass das Aufspüren von Fehlern gemessen an der Routinehaftigkeit eine der am wenigsten automatisierten Tätigkeiten der Softwareentwicklung ist und heute immer noch sehr viel Zeit damit verbracht wird. Interessierte finden in [32] einen Einstieg in die abdeckungsbasierte Fehlerlokalisierung (und das Framework EZUNIT); [33] erweitert den Ansatz hin zur modellbasierten Diagnose. Leider scheinen die entwickelten Verfahren bisher wenig Eingang in die Praxis gefunden zu haben.

- [26] E Gamma, R Helm, R Johnson, J Vlissides Design Patterns — Elements of Reusable Software (Addison-Wesley, 1995). Dublette zu [1].
- [27] E Harold An early look at JUnit 4
 - (<http://www-128.ibm.com/developerworks/java/library/j-junit4.html>)
 - (<http://www.frankwestphal.de/JUnit4.0.html>)
 - (<http://junit.sourceforge.net/README.html>)
 - (<https://github.com/junit-team/junit4/blob/master/doc/ReleaseNotes4.8.md>)
 - (<https://github.com/junit-team/junit4/blob/master/doc/ReleaseNotes4.9.md>)
 - (<https://github.com/junit-team/junit4/blob/master/doc/ReleaseNotes4.10.md>)
 - (<https://github.com/junit-team/junit4/blob/master/doc/ReleaseNotes4.11.md>)
 - (<http://junit.org/junit5/docs/current/user-guide/>).
- [28] F Westphal JUnit 4 (<http://www.frankwestphal.de/JUnit4.0.html>).
- [29] K Beck, E Gamma JUnit Cookbook
 - (<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>).
- [30] K Beck, E Gamma Test Infected: Programmers Love Writing Tests
 - (<http://junit.sourceforge.net/doc/testinfected/>).
- [31] JUnit A Cook's Tour (<http://junit.sourceforge.net/doc/cookstour/cookstour.htm>).
- [32] F Steimann, T Eichstädt-Engelen, M Schaaf „Towards raising the failure of unit tests to the level of compiler errors“ in: Proc. of TOOLS (2008) 60–79.
- [33] F Steimann, M Bertschler „A simple coverage-based locator for multiple faults“, in: Proc. of IEEE International Conference on Software Testing Verification and Validation (2009) im Druck.
- [34] M Frenkel “Partitionierung von Fehlerlokalisierungsproblemen mit Algorithmen aus der ganzzahligen linearen Optimierung”
 - (<http://www.fernuni-hagen.de/ps/veroeffentlichungen/>).

3.11 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 3.1 (Seite 85)

Sinnvolle Ein-/Ausgabepaare z. B. sind $(0, 0)$, $(1, 1)$, $(2, \sqrt{2})$ sowie $(-2, \perp)$. Wenn die Wurzel von 2 richtig berechnet wird, ist zu vermuten, dass die Wurzel für höhere Zahlen ebenfalls stimmt. Evtl. kann man auch noch versuchen, die größtmögliche Zahl einzugeben.

Für die Überprüfung der Implementierung von `stack` reicht eine einfache Angabe von Ein- und Ausgabepaaren nicht immer aus. Auf diese Weise kann man nämlich beispielsweise schlecht ausdrücken, dass ein Stack nach einer Push-Operation nicht (mehr) leer sein darf. Manche regelmäßige Testfälle lassen sich zwar aus der axiomatischen Spezifikation eines abstrakten Datentyps `Stack` ableiten; so lassen sich leicht Tests wie $\text{pop}(\text{push}(s, e)) = s$ und $\text{top}(\text{push}(s, e)) = e$ formulieren (wobei s und e durch konkrete Stack-Objekte ersetzt werden), aber schon hierbei handelt es sich nicht mehr um den Standardtestfall, da jeweils nicht eine Funktion, sondern zwei geprüft werden (die entsprechend verkettet aufgerufen werden müssen). Für die Formulierung von $\text{push}(s, e) \rightarrow \neg \text{empty}(s)$ bietet sich ein programmatischer Ausdruck wie

```
713 s.push(e);  
714 assert ! s.empty();
```

an.

4 Entwurfsmuster

Der einen Sprache Muster ist der anderen Sprache Konstrukt.

Anonym.

Trotz ständig neuer Sprachkonstrukte in den höheren Programmiersprachen, die zu immer kompakteren Programmen führen, müssen für bestimmte Probleme immer wieder ähnliche Lösungen ausprogrammiert werden. Dabei sind die Lösungen niemals so gleich, dass sie sich in einer Bibliotheksroutine einfangen ließen, die sich, ggf. nach entsprechender Parametrisierung, immer wieder verwenden ließe. Sie sind aber so ähnlich, dass man als Programmiererin irgendwann den Eindruck bekommt, man müsse das Rad ständig neu erfinden, und sich entsprechend darüber ärgert. Letzteres ist insbesondere dann der Fall, wenn die Lösung dieser Probleme mehr Zeit in Anspruch nimmt, als dies aufgrund ihres routinehaften Charakters angemessen erscheint oder, schlimmer noch, wenn bei der Umsetzung immer wieder aufs Neue Fehler gemacht werden. Dies zu verhindern ist Ziel der Analyse-, Entwurfs- und Implementationsmuster.

4.1 Historisches

Der Begriff des Musters im Zusammenhang mit Entwürfen wurde angeblich zuerst vom Architekten Christopher Alexander benutzt [35] :

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Mit dem Erscheinen des Entwurfsmusterbuchs „Design Patterns — Elements of Reusable Software“ [35] im Jahr 1995 hat ein regelrechter Pattern hype in der Informatik eingesetzt. Tatsächlich dürfte das besagte Buch eines der am besten rezipierten Bücher zur objektorientierten Programmierung sein.⁴⁵ Inzwischen ist es, zumindest was die Zahl der Publikationen zu dem Thema angeht, wieder etwas ruhiger um Patterns geworden. Dennoch wird man in unserer Branche das

⁴⁵ Das Buch ist aus meiner Sicht praktisch ohne Konkurrenz, nicht nur wegen des darin enthaltenen Katalogs von Patterns, sondern auch und gerade wegen seiner grundsätzlichen Anmerkungen zur objektorientierten Programmierung. Das hauptsächliche Manko aus heutiger Sicht ist, dass es JAVA nicht kennt, insbesondere das mit JAVA verbreitete Interface-als-Typ-Konzept mittels abstrakter Klassen umsetzt und somit nicht immer klar ist, ob es sich bei einer solchen konzeptuell um eine Generalisierung oder um eine Rolle handelt (vgl. die Abschnitte 1.2.3 und 1.8). Einige meiner ehemaligen Studenten empfehlen stattdessen „Head First Design Patterns“ (erschienen by O'Reilly), dessen Prologkapitel ich allerdings in seinem Bemühen, die Nintendo-Generation zu erreichen, recht nervig fand. Die beiden Bände „Patterns in Java“ von Mark Grand (beide erschienen bei Wiley) sind m. E. aber auch keine Alternative zum Klassiker von Gamma et al.

Gefühl wohl so schnell nicht los, etwas Wichtiges verpasst zu haben, wenn man beim Thema Patterns nicht mitreden kann.

Die Verwendung des Begriffs „Design“ im Titel des besagten Buchs legt nahe, dass das Konzept der Muster zunächst für den Softwareentwurf gedacht war, also für die Phase der Softwareentwicklung, in der man sich über die Implementierung Gedanken macht, ohne jedoch schon damit zu beginnen. Der Begriff lässt sich jedoch auf natürliche Weise auch auf die Analysephase (Analysemuster, engl. analysis patterns), auf die Softwarearchitektur (Architekturmuster, engl. architectural patterns) sowie noch so manch anderes ausdehnen, was in der Folge auch systematisch getan wurde. Wir beschäftigen uns hier jedoch nur mit Entwurfsmustern.

4.2 Übergeordnete objektorientierte Programmierprinzipien

Unabhängig von konkreten Entwurfsmustern lassen sich zwei Grundprinzipien der objektorientierten Programmierung ausmachen, deren Einhaltung für wiederverwendbare Entwürfe sorgt. Das eine haben Sie schon kennengelernt: Es lautet „program to an interface, not an implementation“ und war Motiv der gesamten Kurseinheit 1. Im Zusammenhang mit Entwurfsmustern bekommt dieses Motto noch eine zusätzliche Bedeutung: Wie Sie noch sehen werden, stellen viele in Entwurfsmustern vorkommende Klassen keine Klassen eines tatsächlichen Programms dar, sondern vielmehr *Rollen* des Entwurfsmusters, die von den Klassen des Programms bei der Realisierung eines Entwurfsmusters ausgefüllt werden. Nicht selten spielt eine Anwendungsklasse sogar mehrere solche Rollen, nämlich wenn mehrere Entwurfsmuster zur Anwendung kommen (oder ein Entwurfsmuster mehrfach) und ein und dieselbe Anwendungsklasse mehrfach beteiligt ist. Wenn eine Rolle eines Entwurfsmusters als Interface umgesetzt werden kann, werden durch die Verwendung eines Entwurfsmusters keine zusätzlichen, entwurfsmusterspezifischen Klassen benötigt.

Das andere übergeordnete objektorientierte Programmierprinzip ist das folgende:

Favor object composition over class inheritance.

Zweites Prinzip wiederverwendbaren objektorientierten Designs, aus [35].

Das verlangt nach einer Erklärung. Zunächst einmal mag man erstaunt darüber sein, dass hier eines der charakteristischen Merkmale der objektorientierten Programmierung, die *Vererbung*, infrage gestellt wird. Dafür gibt es jedoch gute Gründe: Wie schon in Kurs 01814 ausführlich erläutert wurde, entsteht durch das *Subclassing* eine enge Kopplung zwischen Superklasse und Subklasse. Das sog. **Vererbungsinterface**, das die Kopplung festhalten sollte, bleibt jedoch zumindest teilweise implizit, d. h., man kann den abhängigen Teilen der Implementierung nicht ansehen, wovon sie abhängig sind oder wer von ihnen abhängt. Dies soll hier noch einmal näher ausgeführt werden.

4.2.1 Offene Rekursion und das Vererbungsinterface

Wenn ein Objekt ein anderes verwendet, dann muss es dafür eine Referenz auf das andere haben. Diese Referenz, meistens eine Variable (aber grundsätzlich jeder Ausdruck, der zu einem Objekt auswertet), hat einen Typ und dieser Typ bestimmt, welche Elemente des referenzierten Objekts zugreifbar sind. Wenn das Objekt ein Interface implementiert und der Typ der Referenz dieses Interface ist, ergibt sich die in Abbildung 1.2 dargestellte Situation. Die entstandene Abhängigkeit ist die des Aufrufs oder der Benutzung.

Aufrufabhängigkeit von Subklasse zu Superklasse: Vererbung

Bei der Vererbung ergibt sich eine andere Situation. Zunächst ist klar, dass ein Objekt neben den Elementen seiner eigenen Klasse auch Zugriff auf die seiner Superklasse hat. Da es diese Elemente wie seine eigenen verwenden und damit auch als seine eigenen betrachten kann, spricht man hier eben von **Vererbung**. Rein technisch gesehen bleiben die geerbten Elemente jedoch „im Besitz“ der Superklasse; ihre Verwendung durch die Subklasse hat damit eine gewisse Ähnlichkeit mit der durch Benutzung oder Aufruf entstehenden Abhängigkeit zwischen zwei verschiedenen Objekten. Der entscheidende Unterschied ist hier jedoch, dass es keine zwei Objekte gibt, sondern nur eines; dessen Definition ist aber auf zwei Klassen aufgeteilt.

umgekehrte Aufrufabhängigkeit: offene Rekursion

Damit aber noch nicht genug. Es kann sich nämlich, durch ein anderes Konstrukt der objektorientierten Programmierung, ergeben, dass die Superklasse Methoden der Subklasse au ruft: Wenn aus einer Methode einer Superklasse eine andere Methode mit `this` als Empfänger aufgerufen wird, die in derselben Klasse oder einer ihrer Superklassen definiert oder zumindest (abstrakt) deklariert und von einer ihrer Subklassen überschrieben wird, dann ruft die Methode der Superklasse u. U. eine Methode einer ihrer Subklassen auf, nämlich genau dann, wenn `this` auf eine Instanz einer solchen Subklasse verweist. Dieses Phänomen nennt man auch **offene Rekursion**, und zwar Rekursion, weil die Methode auf demselben Empfängerobjekt aufgerufen wird, und offen, weil offen ist, wo (in welcher Klasse) sich die Implementierung findet.

Deklaration des Vererbungs-interfaces

Es ergibt sich also der etwas komplexe Begriff eines zweiseitigen Vererbungsinterfaces: Zum einen legt es fest, welche Elemente einer Superklasse von ihren Subklassen aus zugreifbar sind (Vererbung), zum anderen legt es fest, welche Methoden der Subklassen von der Superklasse aus aufrufbar sind (offene Rekursion). Je nach Programmiersprache werden zur Kennzeichnung des Vererbungsinterfaces Schlüsselwörter zur Verfügung gestellt: In Java beispielsweise kennzeichnet `protected` die Vererbung (kann aber durch `public` ersetzt oder innerhalb eines Pakets auch weggelassen werden), in C++ und C# kennzeichnen zusätzlich `virtual` und `override` die offene Rekursion.

Selbsttestaufgabe 4.1

Folgende Konstellation zweier Klassen enthält einen Aufruf einer geerbten Methode und eine offene Rekursion. Identifizieren Sie beide. Überlegen Sie auch, wie Sie das Vererbungsinterface mit Ihnen bekannten programmiersprachlichen Mitteln kennzeichnen würden.

```
715 class Super {  
  
716     void n() {  
717         o();  
718     }  
  
719     void o() {  
720         System.out.println("Super");  
721     }  
722 }  
  
723 class Sub extends Super {  
  
724     void m() {  
725         n();  
726     }  
  
727     void o() {  
728         System.out.println("Sub");  
729     }  
730 }
```

Um sich das Vererbungsinterface zu verdeutlichen, ist es hilfreich, die Vererbung durch Komposition zu ersetzen. Man entfernt dazu die Klausel `extends Super` aus der Definition von `Sub` und führt stattdessen eine Instanzvariable (Feld) `_super` vom Typ `Super` ein, der eine Instanz der Klasse `Super` zugewiesen wird. Umgekehrt führt man in `Super` eine Instanzvariable `_this` vom Typ `Sub` ein, der eine Instanz der Klasse `Sub` zugewiesen wird. Der Code könnte wie folgt aussehen:⁴⁶

```
731 class Super {  
732     Sub _this = null;  
  
733     Super(Sub sub) {  
734         _this = sub;  
735     }  
  
736     void n() {  
737         if (_this != null)  
738             _this.o();  
739         else  
740             o();
```

⁴⁶ Man beachte, dass die beiden Programme keineswegs äquivalent sind; das zweite soll lediglich der experimentellen Feststellung der Vererbungsschnittstelle dienen.

```

741 }
742 void o() {
743     System.out.println("Super");
744 }
745 }

746 class Sub {
747     Super _super;

748     Sub() {
749         _super = new Super(this);
750     }

751     void m() {
752         _super.n();
753     }

754     void o() {
755         System.out.println("Sub");
756     }
757 }
```

Wenn man nun noch die Typen in den Deklarationen von `_this` und `_super` durch minimale, kontextspezifische Interfaces ersetzt, hat man genau das Vererbungsinterface:

```

758 interface Sub2Super {
759     void n();
760 }

761 interface Super2Sub {
762     void o();
763 }
```

Man beachte die Asymmetrie des Vererbungsinterfaces, die sich darin ausdrückt, dass man nicht ein, sondern zwei Interfaces bekommt.

Selbsttestaufgabe 4.2

Klassifizieren Sie die beiden Interfaces gemäß den Kriterien von Abschnitt 1.5.

Offene Rekursion ist das Prinzip der *Umkehrung der Ausführungskontrolle* (engl. *inversion of control*) vieler sog. *White-Box-Frameworks*. Die Benutzerin des Frameworks muss dazu eine Subklasse einer (abstrakten) Frameworkklasse bilden, diese instanziieren und die Instanz dem Framework übergeben (z. B. indem sie eine geerbte Methode der Instanz aufruft). Ein Beispiel hierfür hatten Sie in Abschnitt 1.5.9 schon kennengelernt: Um in den Genuss des Multi-

threading-Frameworks von JAVA zu kommen, genügt es, eine Subklasse von der Klasse `Thread` zu bilden und auf einer Instanz dieser Subklasse die geerbte Methode `start()` aufzurufen. Als Reaktion darauf wird, nach allerlei Mimik, die in der Subklasse überschriebene Methode `run()` in einem neuen Thread aufgerufen.

4.2.2 Vererbung vs. Komposition

Die vorangegangenen Ausführungen zum Vererbungsinterface sollten klar gemacht haben, dass die Vererbung nicht zum Nulltarif zu haben ist: Insbesondere die enge Kopplung bei gleichzeitig fehlendem (oder implizitem) Vererbungsinterface machen die Wartung von Super- oder Subklasse ohne gleichzeitige Betrachtung der jeweils anderen zu einem Glücksspiel. Dazu kommt, dass die Superklasse einer Klasse immer schon zur Übersetzungszeit angegeben wird und daher nicht zur Laufzeit ausgewechselt werden kann, was immer dann einen Mangel an Flexibilität darstellt, wenn sich das Verhalten eines Objekts aufgrund von Ereignissen während der Programmalaufzeit grundlegend ändern soll. Alle diese Gründe haben dazu geführt, dass man die Vererbung nicht mehr für alleinseligmachend hält.

Die Alternative zur Vererbung hatten Sie eben bereits kennengelernt, wenn auch nur zu Behelfszwecken: die Objektkomposition. Anstatt eine Klasse von einer anderen erben zu lassen, kann man auch ein Objekt der erbenden Klasse aus zwei Objekten zusammensetzen (komponieren), nämlich aus einem der ehemals erbenden Klasse und einem der ehemals beerbten. Je nachdem, ob das Verhältnis der beiden Objekte Vererbung oder offene Rekursion erfordert, braucht man dazu allerdings eine explizite Verknüpfung der beiden Objekte und eine Weiterleitung der Methoden.

4.2.3 Forwarding vs. Delegation

In diesem Zusammenhang ist eine recht subtile, aber dennoch bedeutsame Unterscheidung von Interesse: die zwischen Delegation und Forwarding. Der Unterschied lässt sich am besten am Beispiel der oben vorgestellten, behelfsweisen Ersetzung von Vererbung durch Komposition erläutern.

Wenn man in einer Objektkomposition eine Methode in einer Klasse (im obigen Beispiel die Klasse `Sub`) nicht selbst implementieren, sondern auf die Implementierung der anderen Klasse (`Super` im Beispiel) zurückgreifen möchte, dann hält sich die komponierte Klasse (`Sub`) eine Instanzvariable (`_super` im Beispiel) mit dem Typ der anderen Klasse (`Super`) und leitet den Methodenaufruf über die Instanzvariable an diese andere Klasse weiter. Wenn die Methode, an die weitergeleitet wird, über `this` (in JAVA meistens implizit) wieder andere aufruft, die dann in derselben Klasse gesucht werden, spricht man von einer einfachen Weiterleitung, dem **Forwarding**.

Was beim einfachen Forwarding nicht geschieht, ist, dass die Methode, an die weitergeleitet wird, auf Methoden der Klasse, von der weitergeleitet wurde, zurückgreift. Dazu müsste sie einen Verweis auf die weiterleitende Instanz haben (`_this` im obigen Beispiel). Ist ein solcher Verweis vorhanden und ist insbesondere dazu nicht einmal eine explizit vereinbarte Instanzvariable not-

wendig, sondern ist es das eingebaute `this` (in anderen Sprachen auch `self` genannt), das auf die weiterleitende Instanz verweist, spricht man von **Delegation**. Delegation muss in Sprachen wie JAVA über eine zusätzliche Instanzvariable (eben `_this` im Beispiel) simuliert werden; in anderen Sprachen, insbesondere solchen, die ohne Klassen auskommen und Delegation zum Zwecke der *Vererbung unter Objekten* einsetzen (wie z. B. die prototypenbasierte Sprache SELF oder die Sprache Go), ist sie Standard.

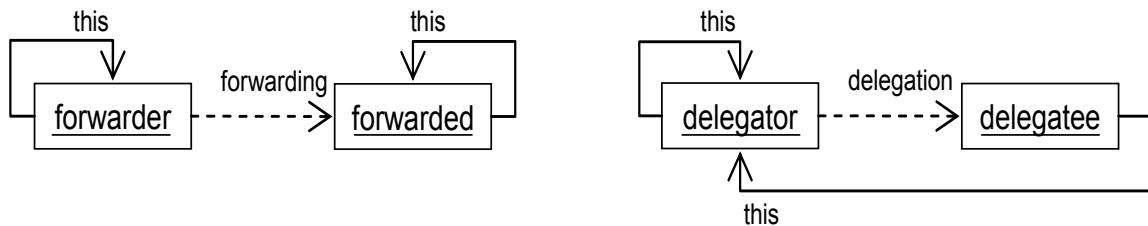


Abbildung 4.1: Gegenüberstellung von Forwarding und Delegation. Der Unterschied liegt in der Bindung von `this`: Beim Forwarding bezeichnet `this` das Objekt, an das weitergeleitet wurde, bei der Delegation das, das delegiert hat. Die Unterscheidung ist sprachspezifisch — die meisten objektorientierten Programmiersprachen bieten nur Forwarding.

Abbildung 4.1 zeigt den Unterschied zwischen Delegation und Forwarding kurz zusammengefasst. Die Begriffswahl ist m. E. nicht sonderlich gelungen, da man von deren umgangssprachlicher Bedeutung nicht auf die konkreten Unterschiede schließen kann. Und so wird denn im Zuge der klassenbasierten objektorientierten Programmierung fast immer von Delegation gesprochen, wenn Forwarding gemeint ist. Wir werden im Verlauf dieses Kurstextes (Abschnitte 5.1.5 und 5.2.4.8) wieder darauf zu sprechen kommen.

4.3 Definition

Unter einem **Entwurfsmuster** (engl. design pattern) versteht man eine Vorlage, anhand derer ein neues Exemplar gebaut werden kann. Dabei muss das neue Exemplar keine exakte Kopie des Musters sein — vielmehr weist das Muster bestimmte Freiheitsgrade auf, die man nutzen kann, um das Exemplar an eine konkrete Problemstellung anzupassen.

A design pattern is a written document that describes a general solution to a design problem that recurs repeatedly in many projects.

Quelle: <http://www.whatis.com/>

Ein Entwurfsmuster beschreibt schematisch eine Lösung für ein Standardproblem. Damit ein Problem als Standardproblem betrachtet werden darf, sollten sich nach einer nicht weiter begründeten Regel mindestens drei konkrete Beispiele dafür finden lassen. Die Lösung wird dann

häufig in Form eines oder mehrerer UML-Diagramme skizziert,⁴⁷ wobei die beteiligten Klassen in der Regel als Stellvertreter für die Klassen des Problems stehen. Diese Platzhalterklassen bezeichnen eigentlich eher die *Rollen*, die die tatsächlichen Klassen einer konkreten Lösung im Rahmen des Musters spielen.⁴⁸ Sie könnten also hier und da durch Interfaces à la JAVA ersetzt werden (vgl. Abschnitt 1.8). Die zum Zeitpunkt der Entstehung des Entwurfsmusterkonzepts vorherrschenden objektorientierten Programmiersprachen, SMALLTALK und C++, kannten jedoch beide kein explizites Interfacekonstrukt.

4.4 Wichtige Entwurfsmuster

Entwurfsmuster stellen den kondensierten Erfahrungsschatz von Programmiererinnen dar. Programmiererinnen machen jedoch unterschiedliche Erfahrungen, die durch ihre Herkunft und ihr Einsatzgebiet geprägt sind. Es geht ihnen damit genau wie Ihnen, mit der Konsequenz, dass das, was die eine als wichtig empfindet, für die andere kaum von Bedeutung ist.

Wenn hier also einige Muster als besonders wichtig herausgehoben werden, dann, weil sie vergleichsweise anwendungsgebietsunabhängig sind oder weil sie ein Prinzip oder eine Technik herauskehren, die ich gern vermitteln möchte. Wer das eine oder andere Muster überflüssig vorkommt, die möge bedenken, dass es mit dem Lesen von Mustern ist wie mit dem Lesen von Programmen: Man kann nur dazulernen.

4.4.1 COMPOSITE Pattern

Häufig besteht ein Ganzes aus Teilen, die selbst wieder Ganze (also aus Teilen zusammengesetzt) sein können und so weiter, bis hinunter zu atomaren (also nicht weiter zusammengesetzten) Teilen. Konkrete Beispiele hierfür sind die Baugruppen eines technischen Geräts, die Verzeichniseinträge eines Dateisystems oder die Elemente einer Zeichnung, die rekursiv in Gruppen zusammengefasst werden können. Charakteristisch für diese Situationen ist, dass — in bestimmten Kontexten — Ganze und Teile gleichbehandelt werden können, weil sie dieselben Funktionen aufweisen. So kann man Baugruppen und Einzelteilen gleichermaßen Teilenummern, Lieferzeiten und Preise zuordnen, Verzeichnisse und Dateien haben einen Namen, ein Erstellungsdatum und können aufgelistet werden, primitive Elemente und Gruppen von Grafiken können auf dem Bildschirm dargestellt, gedreht, gestreckt werden usw. Abbildung 4.2 zeigt beispielhaft eine solche (Baum-)Struktur.

⁴⁷ Zum Zeitpunkt der Drucklegung des Design-pattern-Buchs war UML noch nicht geboren — die im Buch verwendeten Diagramme basieren auf einer Vorläufernotation.

Die Übersetzungen der Klassendiagramme nach UML, die in diesem Text verwendet wird, basieren auf <http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/>.

⁴⁸ Diese Rollen werden in der Standardbeschreibung von Entwurfsmustern, die selbst einem Muster folgt, üblicherweise unter der Überschrift „Participants“ gelistet. Wenn in den in diesem Kapitel vorkommenden UML-Diagrammen Klassen mit dem Stereotyp «**stereotype**» versehen sind, dann um darauf hinzuweisen, dass die tatsächliche Klasse in der Anwendung eines Musters der angegebenen folgt (nach der Art der angegebenen aufgebaut ist).

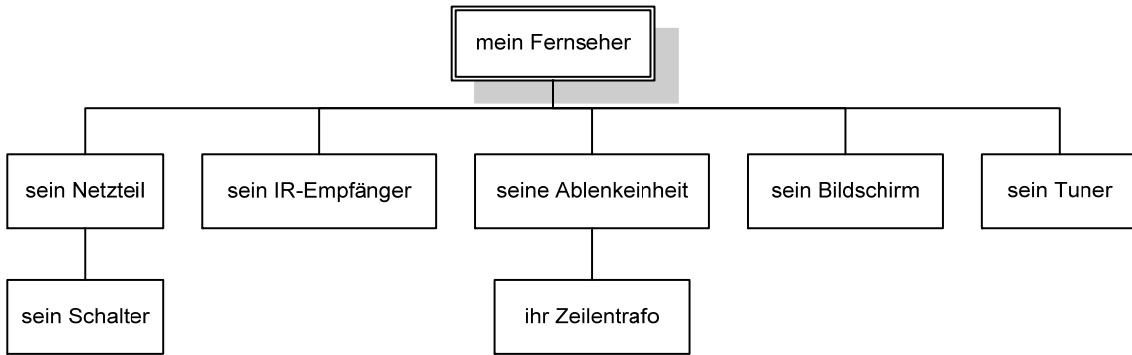


Abbildung 4.2: Die Bauelemente meines Fernsehers (unvollständig). Man erkennt, dass der Fernseher ein Ganzen ist, das aus Teilen besteht, die selbst wieder aus Teilen bestehen können, also auch Ganze sind.

Neben den ebengenannten Gemeinsamkeiten von Teilen und Ganzen, die spezifisch für die jeweiligen Anwendungsklassen sind, gibt es auch Eigenschaften, die allen Teil-Ganzen-Strukturen (unabhängig vom jeweiligen Beispiel) gemeinsam sind. Dazu zählt etwa das Hinzufügen und Entfernen von Teilen oder das Iterieren über alle Elemente eines Ganzen. Im Sinne einer Musterbildung wird man also versuchen, diese Gemeinsamkeiten zu isolieren und in eine Form zu bringen, in der man sie leicht wiederverwenden kann.

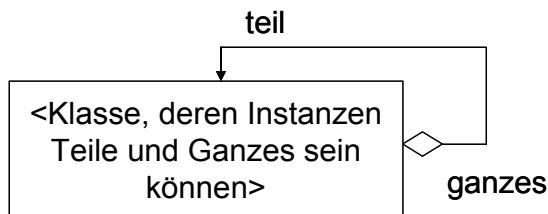


Abbildung 4.3: Die Aggregation verbindet ein Ganzes mit seinen Teilen, die selbst wieder Ganze sein können. Ein Ende der Rekursion ist nicht in Sicht. (Notation: UML)

Einen ersten, naiven Ansatz, dieses immer wiederkehrende Muster zu fassen, stellt das Diagramm aus Abbildung 4.3 dar. Es zeigt eine nicht näher spezifizierte Klasse, deren Instanzen mit Instanzen derselben Klasse eine Teil-Ganzen-Beziehung (Aggregation) eingehen können. In Diagrammen zu Entwurfsmustern wird solchen Klassen meistens ein prägnanter, für das Muster charakteristischer Name gegeben; wie bereits erwähnt, spiegelt dieser Name häufig die Rolle der Klasse in dem jeweiligen Entwurfsmuster wider — in einer tatsächlichen Anwendung dieses Musters haben die Klassen meistens andere, anwendungsspezifische Namen, die für die eigentlichen Funktionen der Klassen stehen.

Abbildung 4.3 deutet bereits an, dass Instanzen der besagten Klasse zwei Rollen spielen können, nämlich die des Ganzen und die des Teils. Nun liegt es aber meistens in der Natur der Sache, dass es auch Teile gibt, die keine Ganze sein können (die obenerwähnten atomaren Teile), und dass es Ganze gibt, die selbst keine Teile sind (die Wurzel eines Baums, spielt für die Praxis jedoch kaum eine Rolle). Das vorliegende Diagramm suggeriert hingegen, dass die Zerlegung nach unten unbegrenzt ist, es also keine Teile gibt, die von Natur aus (aufgrund ihrer Beschaffenheit) nicht weiter geteilt werden können. Das entspricht jedoch nur selten der Realität.

Selbsttestaufgabe 4.3

Versuchen Sie, bevor Sie weiterlesen, sich zu überlegen, wie Sie das „Muster“ reparieren können und schreiben Sie es auf!

Das Diagramm muss also erweitert werden, um die unterschiedliche Natur unterschiedlicher Instanzen wiedergeben zu können. Dazu liegt es nahe, die eine, allgemeine Klasse in zwei aufzuteilen, deren Instanzen entweder atomare Teile sind oder Teile, die auch Ganze sein können. Das Klassendiagramm in Abbildung 4.4 gibt genau dies wieder: Komponenten, also Atome oder Komposita, können Teil sein, Komposita nur Ganzes.

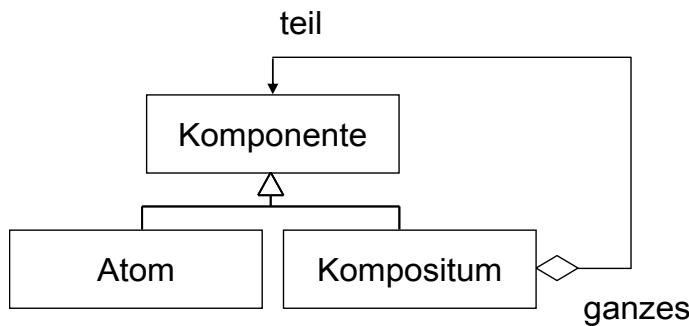


Abbildung 4.4: Unterscheidung der Komponenten einer Teil-Ganzes-Struktur nach Atomen und Komposita. Während das Ganze immer ein Kompositum sein muß, können Teile beides sein. Für diejenigen, die UML nicht kennen, nur kurz das folgende: Rechtecke stehen für Klassen, die Pfeile mit dem geschlossenen hohlen Dreieck als Spitze stehen für die Generalisierung (die Klasse an der Spitze des Pfeils ist ein Supertyp der Klasse(n) am Fuß des Pfeils), die Linie mit der Raute an einem Ende steht für die Aggregation, wobei das Ganze durch das Ende mit der Raute gekennzeichnet ist.

Was diesem Diagramm noch fehlt, ist die Verteilung der obenerwähnten Operationen, die man üblicherweise mit Teil-Ganzes-Strukturen verbindet und die einen wesentlichen Teil des Patterns ausmachen. Damit atomare und zusammengesetzte Teile von der Klientin gleichbehandelt werden können, müssen sie demselben Protokoll gehorchen. Dazu wird man versuchen, so viel wie möglich von beiden Klassen in **Komponente** zu deklarieren. Dies gilt auch für die patternspezifischen Funktionen, die ja die Behandlung der Aggregation betreffen, also z. B. Funktionen zum Hinzufügen, Entfernen und Holen von Teilen. Man kann diese sogar schon in **Komponente** mit einer Implementierung (und sei es nur ein Default) versehen, was immer dann sinnvoll ist, wenn man — in Erweiterung des Patterns — verschiedene Blattklassen vorsehen möchte, die sich nur darin gleichen, dass sie alle Teile sind.

Diese Verteilung ist dem Diagramm in Abbildung 4.5 zu entnehmen, dass die klassische Form des COMPOSITE Pattern aus [35] darstellt. Man beachte, dass die Superklasse (**Component**) abstrakt ist, was so viel heißt wie dass von ihr keine Instanzen gebildet werden können — diese müssen entweder vom Typ **Composite** oder vom Typ **Leaf** sein. Auch hier stehen die Klassensymbole des Diagramms nicht für die konkreten Klassen einer gegebenen Problemstellung, sondern dienen lediglich als Platzhalter. Dies wird durch den Zusatz «**stereotype**» symbolisiert. Wenn Sie also etwa ein Dateisystem unter Verwendung des obigen Patterns programmieren wollten, würden Sie die Klassen entsprechend umbenennen, also etwa in **Eintrag**, **Verzeichnis** und **Datei**. Diese würden dann, aus Sicht des Musters, die Rollen **Component**, **Composite** bzw. **Leaf** spielen. Die Methode **Operation** steht nur als Platzhalter für beliebige Methoden des Anwendungsgebiets, die sinnvoll gleichermaßen auf Ganze und atomare Teile angewendet werden können, wie z. B. Öffnen bei Dateisystemen.

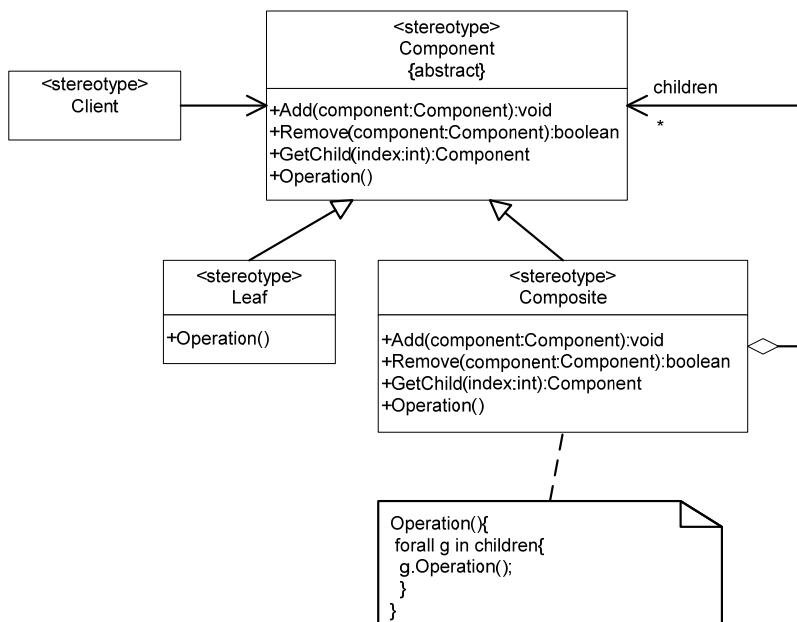


Abbildung 4.5: Das COMPOSITE Pattern in seiner klassischen Form (angelehnt an [35]). Sowohl Blätter als auch Komposita folgen demselben Protokoll. Dies ist jedoch nicht ohne Probleme (s. Text).

```

764 abstract class Component {
    ...
765     void forAllDo() {
766         // tue irgend etwas Sinnvolles
767         for (Component child : this.GetChildren())
768             child.forAllDo();
769     }
770
771     abstract List<Component> GetChildren();
772     abstract void Add(Component c);
773     abstract void Remove(Component c);
774     abstract Component GetChild(int i);
  
```

```
774 }

775 class Leaf extends Component {

776     List<Component> GetChildren() {
777         return new ArrayList<Component>();
778     }

779     void Add(Component c) {
780         throw new Error ("not implemented");
781     }

782     void Remove(Component c) {
783         throw new Error ("not implemented");
784     }

785     Component GetChild(int i) {
786         throw new Error ("not implemented");
787     }
788 }

789 class Composite extends Component {
790     List<Component> children;
791     ...
792     List<Component> getChildren() {
793         return children;
794     }

795     void Add(Component c) {
796         children.add(c);
797     }

798     void Remove(Component c) {
799         children.remove(c);
800     }

801     Component GetChild(int i) {
802         return children.get(i);
803     }
804 }

805 class Client {
806     Component component;
807     ...
```

```

807     component.forAllDo();
...
808     component.Add(new Leaf()); // u. U. Error zur Laufzeit
...
809     if (component instanceof Composite)
810         component.Add(new Leaf()); // OK
811     else
812         // meistens ist es nicht egal, ob Add geklappt hat!
...
813 }
```

Leider ist diese Form des COMPOSITE Pattern, die Sie sich gut einprägen sollten (da sie Ihnen vermutlich häufig begegnen wird), nicht ganz ohne Probleme. Wie oben bereits erwähnt und aus dem Codebeispiel zu ersehen, können manche der in **Component** deklarierten Operationen sinnvoll nur auf Ganzen ausgeführt werden (**Add**, **Remove**, **GetChild**), andere hingegen auch auf Teilen (**GetChildren** sowie ggf. weitere anwendungsspezifische Operationen). Das Pattern in seiner hier gezeigten Form sieht jedoch vor, dass aus Sicht der Klientin alle Elemente vom Typ **Component** sind — atomare und zusammengesetzte Teile sehen damit zunächst gleich aus, obwohl sie es nicht sind. In bestimmten Situationen — und das ist ein Beitrag des Patterns — kann die Klientin aber die Unterschiede ignorieren und beide gleich behandeln. Leider eben nur in bestimmten Situationen.

Was also tun? Die Gleichbehandlung von Ganzen und Teilen bedingt, dass bestimmte Operationen nicht ausgeführt werden können, obwohl die Deklaration dies suggeriert. Das Werfen einer Fehlermeldung (Error) ist hier nicht angezeigt, da die Deklaration verspricht, dass alle Operationen (ohne Ausnahmen) auf alle Instanzen angewendet werden können. Da dies regelmäßig nicht der Fall ist, kann auch nicht von einer Ausnahme (Exception) im eigentlichen Sinne gesprochen werden (vgl. dazu auch Abschnitt 5.2.2.4 ff.). Vielmehr muss man einsehen, dass das gemachte Versprechen, **Component** könne Teile und Ganze ersetzen, falsch ist.

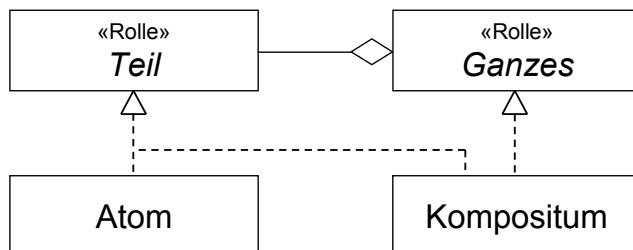


Abbildung 4.6: Teil-Ganzes-Beziehung als die zwischen zwei Rollen. Die Rolle des Teils kann dabei von Atomen und von Komposita gespielt werden, die des Ganzen hingegen nur von Komposita. Technisch drückt sich das Spielen der Rollen durch Implementierung der entsprechenden Interfaces aus.

Wir kehren also noch mal zur Ausgangslage (Abbildung 4.3) zurück und analysieren den Sachverhalt neu. Wir hatten ja festgestellt, dass die Komposition zwei Rollen hat, nämlich die des Ganzen und die des Teils (die Rollennamen an den Aggregationsenden). Weiterhin ergab sich,

dass es zwei Arten von komponierbaren Elementen gibt: Atome und Komposita. Dabei können Atome von Natur aus nur die Rolle des Teils, Komposita hingegen beide Rollen, die des Teils und die des Ganzen, spielen. In UML übertragen sieht das wie in Abbildung 4.6 dargestellt aus. Die Rolle des Ganzen gibt dann die Methoden vor, die sich auf dessen Teile beziehen (`Add`, `Remove`, `GetChild`), die Rolle des Teils hingegen hat solche Methoden nicht (und deklariert sie entsprechend auch nicht). In Code sieht das dann wie folgt aus:

```
814 interface Ganzes {  
815     void forAllDo();  
816     void Add(Teil teil);  
817     void Remove(Teil teil);  
818     Teil GetChild(int i);  
819 }  
  
820 interface Teil {  
821     void forAllDo();  
822 }  
  
823 class Atom implements Teil {  
824     ...  
825     void forAllDo() {  
826         // irgend etwas Sinnvolles  
827     }  
828 }  
  
829 class Kompositum implements Teil, Ganzes {  
830     ArrayList<Teil> teile;  
831     ...  
832     void forAllDo() {  
833         // irgend etwas Sinnvolles  
834         for (Teil teil : teile)  
835             teil.forAllDo();  
836     }  
  
837     void Add(Teil teil) {  
838         teile.add(teil);  
839     }  
  
840     void Remove(Teil teil) {  
841         teile.remove(teil);  
842     }  
  
843     Teil GetChild(int i) {  
844         return teile.get(i);  
845     }  
846 }
```

```

845 }

846 class Client {
847   Ganzes ganzes;
848   Teil teil;
849   ...
850   ganzes.forAllDo();
851   teil.forAllDo();
852   ...
853   teil = ganzes.GetChild(1); //OK
854   teil.GetChild(1); // Typfehler zur Übersetzungszeit
855   ((Ganzes) teil).GetChild(1); // möglicher Laufzeitfehler
856   if (teil instanceof Ganzes)
857     ((Ganzes) teil).GetChild(1); // OK
858   ...
859 }

```

**Rollenwechsel:
Cross casts**

Diese Form des Musters trennt die beiden Rollen des Teils und des Ganzen, die in der ersten Form noch vermischt waren (und zwar in Gestalt des Supertyps **Component**). Die Trennung macht die Rollen der Elemente klarer — sie befreit einen jedoch nicht von der Notwendigkeit, den Typ eines Elements zur Laufzeit prüfen zu müssen: Wenn ein Teil als ein Ganzes auftreten soll, muss auch hier zuvor festgestellt werden, ob das konkrete Objekt (in der Rolle des Teils) auch die Rolle des Ganzen spielen kann (Zeilen 852 ff.). Ist dies der Fall, lässt sich der Rollenwechsel durch einen sog. *Cross cast* erledigen. Man beachte, dass der erfolgreiche Cross cast bedingt, dass das Element vom Typ **Kompositum** ist, weil nur diese Klasse beide Interfaces implementiert.

Ob man sich für die erste oder die zweite Version des Entwurfsmusters entscheidet, hängt auch davon ab, was einem wichtiger ist: Transparenz, also hier das gleiche Protokoll von Teilen und Ganzen (das aber nicht in allen Fällen auch deren Gleichbehandlung garantiert) oder Sicherheit im Sinne einer Vermeidung von Programmierfehlern (Gleichbehandlung an Stellen, wo diese nicht zulässig ist) durch das Type checking.

Selbsttestaufgabe 4.4

Versuchen Sie, beide Varianten des Musters so abzuwandeln, dass neben atomaren Teilen auch Wurzeln (Ganze, die keine Teile sind) ihrer Bedeutung entsprechend dargestellt werden können. Was stellen Sie fest?

**Das Problem
der Koordination**

Die zweite Alternative (mit den expliziten Rollen) legt nahe, die Interfaces **Teil** und **Ganzes** innerhalb einer Anwendung mehrfach wiederzuverwenden, um so das mehrfache Vorkommen des Musters auszudrücken. Dabei ergibt sich dann aber das Problem der mangelnden Koordination: Sobald verschiedene Klassen, die nur paarweise in der Teil-Ganze-Beziehung stehen, die gegebenen Interfaces implementieren, können sich zumindest theoretisch unerlaubte Vermischungen ergeben. Wenn nämlich zum Beispiel in einem gegebenen System **Baugruppe** und **Verzeichnis** das Interface **Ganzes** implementieren sowie

parallel dazu **Bauelement** und **Datei** das Interface **Teil**, dann können ohne weitere Maßnahmen Baumstrukturen entstehen, in denen Baugruppen/Bauelemente und Verzeichnisse/Dateien gemischt auftreten. Solche Probleme können aber durch geeignete programmiersprachliche Konzepte gelöst werden.

Selbsttestaufgabe 4.5

Versuchen Sie, das Problem mit Hilfe von generischen Typen für JAVA zu lösen. Falls es Ihnen nicht gelingt: Woran hat es gelegen?

4.4.2 OBSERVER Pattern

Ein Pattern sollten alle, die sich schon einmal in der GUI-Programmierung in JAVA versucht haben, kennen: das **OBSERVER** Pattern. Beim **OBSERVER** Pattern wird ein Objekt, das sog. Subjekt, von einem oder mehreren anderen, den Observern, beobachtet. Wenn sich der Zustand des Subjekts ändert, bekommen die Observer davon Nachricht, so dass sie reagieren können. In der Regel besteht die Reaktion eines Observers dann daraus, den eigenen Zustand anzupassen; falls dies dazu notwendig sein sollte, fragt der Observer beim Subjekt benötigte Informationen nach.

Ein klassisches Einsatzgebiet des **OBSERVER** Patterns ist die GUI-Programmierung. Wenn sich ein Datenelement ändert, dann sollte sich das in den Anzeigeelementen widerspiegeln, die das Datenelement auf dem Bildschirm repräsentieren. Aber auch GUI-Elemente selbst können Observer haben und somit Subjekt sein: So kann beispielsweise die Titelleiste eines Fensters auf ein Texteingabefeld reagieren, indem sie dessen Inhalt während der Eingabe, Buchstabe für Buchstabe, übernimmt. Wie man sich leicht vorstellen kann, wird die Klasse, die das Texteingabefeld zur Verfügung stellt, keine feste Verknüpfung mit Titelleisten von Fenstern vorsehen.

Das **OBSERVER** Pattern ist tatsächlich immer dann interessant, wenn Art und Zahl von Observern eines Objekts nicht feststehen. Ansonsten könnte man nämlich den Benachrichtigungsmechanismus zwischen dem sich ändernden Objekt und denen, die von den Änderungen abhängig sind, auch fest verdrahten. Wesentlicher Bestandteil des **OBSERVER** Patterns ist hingegen, dass sich die Observer nach Belieben (zur Laufzeit) beim Subjekt registrieren und auch wieder abmelden können. Aufgrund des offenen Verhältnisses von Subjekt und Observern, das auf Anmeldung beruht, spricht man auch von einem *Publish-subscribe-Verfahren*; bei der Art der Verbreitung der Nachricht spricht man gelegentlich von einem *Multicast*.

Abbildung 4.7 zeigt das Klassendiagramm des **OBSERVER** Patterns. Konkrete Subjekte haben neben den Methoden zur Registrierung und Notifikation, die sie erben, auch noch einen Zustand, der von konkreten Observern im Rahmen des Update-Prozesses abgefragt werden kann. Dass der Observer eine feste Verbindung zum Subjekt haben muss, scheint übertrieben — häufig übergibt sich das Subjekt als Teil eines Ereignisses selbst an seine Observer, wodurch jeweils eine temporäre Verbindung hergestellt wird (vgl. dazu das JAVA-Beispiel unten). Gleichwohl besteht dadurch eine Abhängigkeit im Sinne einer starken Kopplung zwischen dem konkreten Observer und dem konkreten Subjekt, denn damit der Observer den Zustand des Subjekts abfragen kann, muss er dessen Typ kennen (weil er sonst die passenden Methoden nicht aufrufen kann). Eine umgekehrte Kopplung besteht jedoch nicht, denn das konkrete Subjekt muss den genauen Typ

des Observers nicht kennen — es reicht, zu wissen, dass es sich um ein Objekt mit dem Interface **Observer** handelt.

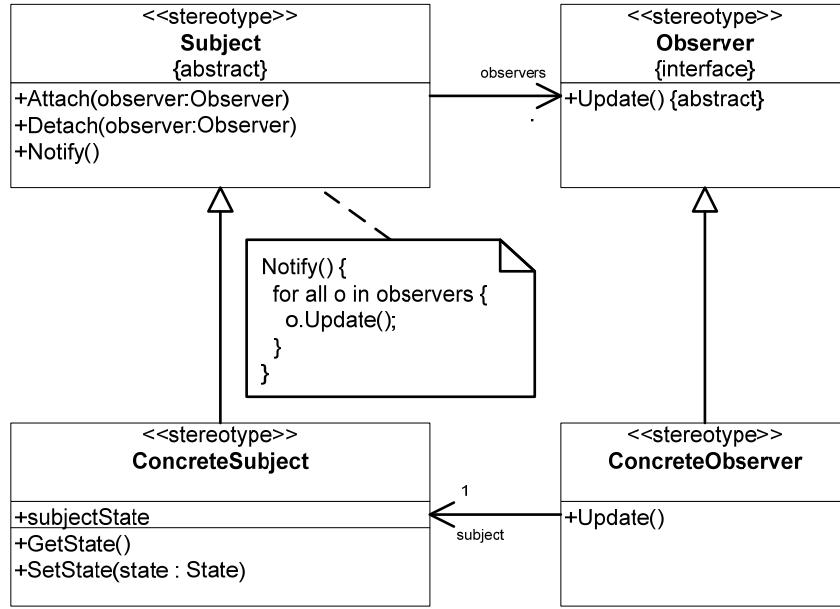


Abbildung 4.7: Das Klassendiagramm des **OBSERVER** Pattern. Es zeigt neben den Rollen Subject und Observer (mit den dazugehörigen Methoden zur Registrierung und Benachrichtigung) auch zwei konkrete Implementierungen.

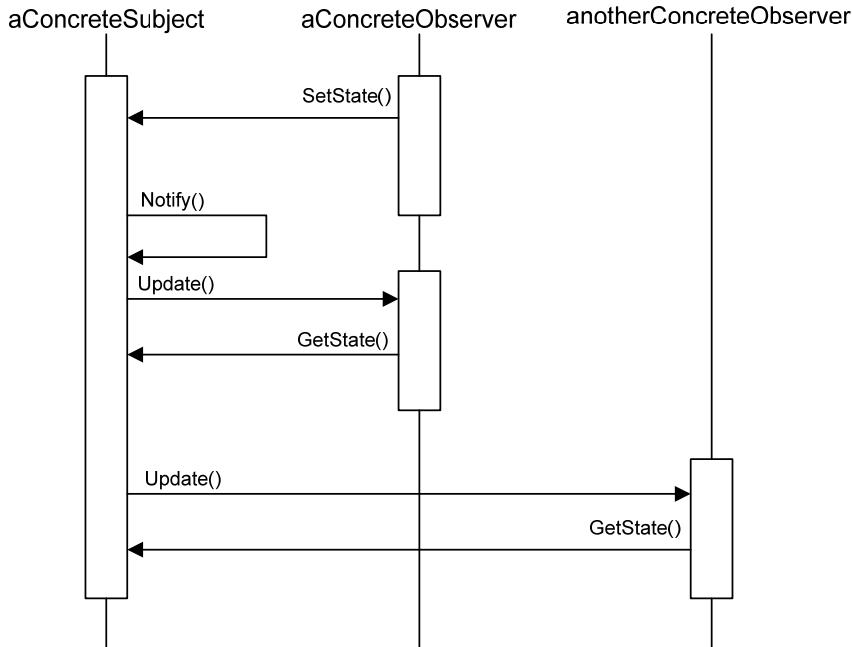


Abbildung 4.8: Interaktion zwischen Subjekt und seinen Observern. In diesem speziellen Beispiel löst ein Observer selbst die Änderung des Zustands des Subjekts aus. Er wartet aber, genau wie der andere registrierte Observer, auf die Notifikation durch das Subjekt, um auf die Änderung zu reagieren.

Der konkrete Ablauf einer Notifikation, beginnend mit der auslösenden Zustandsänderung, ist dem Interaktionsdiagramm aus Abbildung 4.8 zu entnehmen. Dass das die Änderung verursachende Objekt selbst ein Observer ist, spricht dafür, dass das Beispiel einem GUI entnommen wurde, bei dem sich verschiedene Elemente koordiniert verhalten müssen.

Die Verwendung des OBSERVER Pattern wird in JAVA durch die Klasse **observable** (für die Subjekte, im Paket **java.util** definiert) und das dazu passende Interface **Observer** (für die Observer, was sonst) unterstützt. **Observable** implementiert Methoden zum Registrieren und Notifizieren von Observern sowie die Verwaltung eines **Changed** flag, mit dem festgehalten werden kann, ob sich am Subjekt etwas geändert hat. Allerdings offenbart sich hier die Schwäche von JAVA, keine sog. *Mix ins* zu unterstützen: Da die konkreten Subjekte von der Klasse **observable** abgeleitet werden müssen, um in den Genuss der Implementierung des Patterns zu kommen, können sie nicht mehr anderweitig erben. Das ist schlecht, denn die meisten Anwendungsklassen sind nicht von Natur aus Subjekte, sondern etwas ganz anderes. „Subjekt“ ist eben nur eine Rolle, wenn auch mit bestimmtem Verhalten verbunden (das in JAVA aber nicht von einem entsprechenden Interface geerbt werden kann). An diesem Beispiel zeigt sich übrigens sehr schön das Wesen von Entwurfsmustern: Sie sind Muster, die immer wieder neu in Code umgesetzt werden müssen. Würden sie sich durch Sprachkonstrukte oder Bibliotheksfunktionen vollständig automatisieren lassen, wären sie keine Muster im ursprünglichen Sinne mehr!

Das OBSERVER Pattern ist in JAVA auch als Event-listener-Mechanismus bekannt.⁴⁹ Observer heißen dann Listener und müssen entsprechend ein sog. Listener-Interface, das in der Regel von **EventListener** abgeleitet ist, implementieren. Für Subjekte gibt es keine vorgegebenen Klassen oder Interfaces: Sie müssen jeweils selbst dafür sorgen, dass die Methoden zur Registrierung und Notifikation vorhanden sind (s. Klasse **Subject** im nachfolgenden Beispiel). Das Erben von einer Superklasse wie **observable** wäre auch — aufgrund der fehlenden kovarianten Spezialisierung der Variablen vom Typ **EventListener** in JAVA — gar nicht gut möglich (s. Selbsttestaufgabe 4.6). Statt dessen wird in der Regel eine zum Listener-Interface passende, von **EventObject** abgeleitete Klasse angelegt, die neben dem Auslöser des Ereignisses (standardmäßig so vorgesehen) auch noch einen Zustand haben kann, durch den z. B. auch der Zustand des Subjekts kodiert werden kann, so dass der Listener (Observer) nicht beim Subjekt nachfragen muss.

Beispiel

```
857 class AnswerEvent extends java.util.EventObject {  
858     protected int id;  
859     public static final int YES = 0, NO = 1, CANCEL = 2;  
  
860     public AnswerEvent(Object source, int id) {  
861         super(source);  
862         this.id = id;
```

⁴⁹ Genau genommen ist der Begriff des Listener auch besser gewählt, da er, anders als Observer, auf die eher passive Rolle hinweist: der Listener/Observer wird vom Subjekt aufgerufen.

```

863 }
864 public int getID() { return id; }
865 }

866 interface AnswerListener extends java.util.EventListener {
867   public void yes(AnswerEvent e);
868   public void no(AnswerEvent e);
869   public void cancel(AnswerEvent e);
870 }

871 class Subject {
872   ...
873   protected Vector listeners = new Vector();
874
875   public void addAnswerListener(AnswerListener l) {
876     listeners.addElement(l);
877   }
878
879   public void removeAnswerListener(AnswerListener l) {
880     listeners.removeElement(l);
881   }
882
883   public void fireEvent(AnswerEvent e) {
884     Vector list = (Vector) listeners.clone();
885     for (int i = 0; i < list.size(); i++) {
886       AnswerListener listener = (AnswerListener)
887         list.elementAt(i);
888       switch(e.getID()) {
889         case AnswerEvent.YES:
890           listener.yes(e); break;
891         case AnswerEvent.NO:
892           listener.no(e); break;
893         case AnswerEvent.CANCEL:
894           listener.cancel(e); break;
895         }
896       }
897     }
898   }
899 }
```

Man beachte, dass hier das Subjekt, das das Event verschickt, Bestandteil des Events ist. Allerdings hat es im gegebenen Beispiel nur den Typ **Object**, so dass der Listener bei ihm nichts Konkretes nachfragen kann. Die benötigte Information ist dafür schon im Event verpackt — allerdings ist sie im gegebenen Beispiel redundant, da die Art des Events zuvor schon, per Switch-Statement, in einen bestimmten Methodenaufruf aufgelöst wurde.

Das Interface **AnswerListener** des Beispiels ist übrigens ein *ermöglichendes Interface*: Es ermöglicht den Objekten der implementierenden Klassen, auf die Änderungen des Subjekts zu reagieren. Da sie selbst die Nutznießer sind, ist es genauer ein *Server/Client-Interface*, wobei das Subjekt die Rolle des Servers spielt und der Observer die des Clients (vgl. Abschnitt 1.5.9). Man beachte, dass der Server seine Clients registriert (hier durch die Variable **listeners**). Es handelt sich also bei der obigen Implementierung des Event-listener-Mechanismus um ein Beispiel eines *Black-Box-Frameworks* mit *Umkehrung der Ausführungskontrolle*.

Selbsttestaufgabe 4.6

Versuchen Sie, die Klasse **Subject** aus obigem Codebeispiel in eine abstrakte Klasse zu überführen, von der, wie in Abbildung 4.7 angedeutet, konkrete Subjektklassen abgeleitet werden können. Worin liegt das Problem?

Die Programmierung des Event-Listener-Mechanismus in JAVA ist recht wortreich und daher mühselig. In C# ist das **OBSERVER** Pattern dagegen fest eingebaut (ein Beleg für den eingangs dieser Kurseinheit zitierten Merksatz). Das nachfolgende Beispiel dient zwar nicht unbedingt dem Verständnis, es zeigt jedoch, dass der Event-listener-Mechanismus in C# deutlich kompakter umgesetzt werden kann (wenn man denn erst einmal die wenig aussagekräftige, da total überladene Syntax gefressen hat). Zur Erklärung sei nur kurz erwähnt, dass die Registrierung und Benachrichtigung der Observer durch *Delegates* erfolgt, also durch typisierte Methodenpointer. Sie ersetzen zugleich das in JAVA benötigte Interface, das das Vorhandensein einer Methode mit bestimmtem Namen garantiert (vom Observer implementiert), und das Registrieren des Objekts, auf dem die Methode bei der Notifikation aufgerufen wird (der Observer).⁵⁰ Das Delegate steht also gewissermaßen für die Methode des konkreten Objekts, nicht seiner Klasse — es ist Methode und Empfänger in einem. Da man sich so obendrein das dynamische Binden spart, ist die Lösung von C# in der Ausführung schneller.

```
894 public delegate void SomeEvent();  
  
895 public class Subject {  
896     public event SomeEvent Observers;  
897 }  
  
898 public class Observer {  
899     public void OnSomeEvent() {  
900         //  
901     }  
902 }  
  
903 ...  
904 Subject subject = new Subject();  
905 Observer observer1 = new Observer();
```

⁵⁰ Delegates, also Methodenpointer, lassen sich in JAVA durch anonyme innere Klassen emulieren.

```

906 Observer observer2 = new Observer();
907 subject.Observers += new SomeEvent(observer1.OnSomeEvent);
908 subject.Observers += new SomeEvent(observer2.OnSomeEvent);
909 subject.Observers();
910 subject.Observers -= new SomeEvent(observer2.OnSomeEvent);
911 ...

```

4.4.3 TEMPLATE METHOD Pattern

Wie der Name schon nahelegt, besteht das TEMPLATE METHOD Pattern im Wesentlichen aus einer Methode, die für eine bestimmte Funktionalität, den Zweck der Methode, ein Schema (engl. template) vorgibt. Die Template-Methode besteht dazu aus einer durch die Verwendung von Methodenaufrufen allgemein gehaltenen Ablaufbeschreibung. Die konkreten Handlungen müssen von Subklassen der Klasse, die die Template-Methode beherbergt, geliefert werden. Sie werden von der Template-Methode per *offener Rekursion* (s. Abschnitt 4.2.1) aufgerufen. In der Regel ist die Klasse, die die Template-Methode beherbergt, abstrakt und ihre Subklassen sind konkret.

primitive Methoden

Von den Methoden, deren Aufrufe das allgemeine Schema (die Template) vorgibt, gibt es beim TEMPLATE METHOD Pattern zwei Varianten: Die einen, auch *primitive Methoden* genannt, sind in der Klasse der Template-Methode gar nicht definiert; sie werden hier lediglich durch eine abstrakte Methodendeklaration vertreten (die in statisch typgeprüften Programmiersprachen notwendig ist, damit der Code kompiliert). Die Bezeichnung primitive Methoden leitet sich wohl daraus ab, dass sie die Grundbausteine des in der Template-Methode spezifizierten Algorithmus sind; sie müssen notwendigerweise von den konkreten Subklassen geliefert werden.

Hook-Methoden

Die anderen, auch *Hook-Methoden* genannt, geben ein Default-Verhalten vor, das von Subklassen überschrieben oder erweitert werden kann. Der Name röhrt vermutlich daher, dass sich die Subklassen mit ihren Methoden an den durch die Hook-Methoden gekennzeichneten Stellen in die Template-Methode einhängen; dies gilt jedoch auch für primitive Methoden. Die Erweiterung einer Hook-Methode erfolgt übrigens durch Überschreiben und Aufrufen der überschriebenen Methode mittels `super`; wenn dieser Aufruf nicht vergessen werden darf, kann man auch die Hook-Methode durch eine weitere Template-Methode ersetzen, die zunächst das tut, was in jedem Fall zu tun ist, und die die ggf. einzuhängende Methode über eine Hook-Methode aufruft, die dann in der Superklasse leer ist.

Ein typisches Vorkommen des TEMPLATE METHOD Patterns ist Ihnen in Abschnitt 3.2.1 bereits begegnet, und zwar in Form der Methode `runBare()` (ab Programmzeile 420): Die Methoden `setUp()`, `runTest()` und `tearDown()` sind Hook-Methoden, die in Subklassen von `TestCase` überschrieben werden können. Das Default-Verhalten von `setUp()` und `tearDown()` ist jeweils, nichts zu tun; das von `runTest()` kann in der Regel unverändert übernommen werden und wird nur selten überschrieben. Man beachte, dass die Anwendungsklassen, also die, die man als Programmiererin schreibt (in diesem Fall die Testfälle), keine Ablaufsteuerung enthalten, die zu ihrer Ausführung führen — vielmehr wird `runBare()` vom Framework aufgerufen. Es ist dies

ein Fall von *Umkehrung der Ausführungskontrolle*; da zudem Vererbung zum Einsatz kommt, handelt es sich bei JUNIT um ein *White-Box-Framework*.

4.4.4 STRATEGY Pattern

Das STRATEGY Pattern verfolgt ein ähnliches Ziel wie das TEMPLATE METHOD Pattern: einen (Teil eines) Algorithmus in eine andere Klasse auszulagern und damit flexibel zu gestalten. Allerdings setzt das STRATEGY Pattern dazu nicht auf Vererbung, sondern auf Komposition.

Beim STRATEGY Pattern wird ein Algorithmus aus der Klasse, die seinen Anwendungskontext darstellt (und die die Daten liefert, auf denen der Algorithmus operiert) herausgezogen und in eine separate, sog. Strategiekasse ausgelagert. Wenn mehrere alternative Implementierungen des Algorithmus zur Auswahl gestellt werden sollen, dann werden diese in jeweils separate Klassen gesteckt und durch ein gemeinsames Interface gegenüber dem Kontext repräsentiert. Die Kontextklasse erhält dann eine Instanzvariable vom Typ der Strategiekasse oder des Interfaces der Strategieklassen, wenn es mehrere gibt; Aufrufe des Algorithmus erfolgen über diese Instanzvariable. Welcher Algorithmus verwendet wird, bestimmt (über das dynamische Binden) der jeweilige Inhalt der Instanzvariable.

Man kann das STRATEGY Pattern aus dem TEMPLATE METHOD Pattern herleiten, indem man die Kontextklasse mit der Klasse der Template-Methode sowie die Strategiekasse(n) mit deren Subklasse(n) gleichsetzt und die behelfsweise Simulation der Vererbung aus Abschnitt 4.2.1 heranzieht: Die Aufrufe von primitiven und *Hook-Methoden* aus der Template-Methode heraus erfolgen dazu einfach nicht mehr auf (dem impliziten) **this**, sondern auf einer Instanzvariable (Feld), die die austauschbaren Teile des Algorithmus beherbergt. Je nachdem, ob der Algorithmus zu seiner Erledigung auf Methoden des Kontextes zugreifen muss, kommt dabei Forwarding oder Delegation zum Einsatz.

Der Hauptnachteil des STRATEGY Pattern gegenüber dem TEMPLATE METHOD Pattern ist, dass der Strategiekasse der für die Abarbeitung des Algorithmus' notwendige Zugriff auf die Daten des Kontexts gewährt werden muss. Beim TEMPLATE METHOD Pattern war dies ja noch über Vererbung möglich (die konkrete Subklasse hat Zugriff auf die Daten der abstrakten Superklasse) — beim STRATEGY Pattern müssen die Daten jedoch entweder als Parameter übergeben werden (ggf. auch durch ein *Parameterobjekt*; s. Abschnitt 5.2.2.2) oder es muss ein Interface vorgesehen werden, über das die Strategiekasse (bzw. deren Instanzen) die benötigten Daten vom Kontext erfragen können. Wegen der mangelnden Möglichkeit der meisten gebräuchlichen objektorientierten Programmiersprachen, dedizierte Schnittstellen vorzusehen, wären diese Daten dann aber auch anderen zugänglich, was nicht immer beabsichtigt ist.

Vergleich mit dem TEMPLATE METHOD Pattern

Der Hauptvorteil des STRATEGY Pattern gegenüber dem TEMPLATE METHOD Pattern ist, dass der Algorithmus, den der Kontext benutzt, einfach ausgetauscht werden kann, und zwar auch zur Laufzeit. Dies war beim TEMPLATE METHOD Pattern nicht möglich, da die Instanz, die ja (solange die Superklasse abstrakt ist; s. o.) eine Instanz einer Subklasse sein muss, fest an ihre Superklasse gebunden ist.

4.4.5 ROLE OBJECT Pattern

Wie bereits mehrfach angedeutet, spezifizieren die meisten Entwurfsmuster die Beziehungen und das Zusammenspiel zwischen *Rollen*, die in der Anwendung des Musters von Klassen eingenommen werden können. Angesichts dessen mag es etwas verwundern, dass für die Realisierung von Rollen selbst ein Entwurfsmuster, das ROLE OBJECT Pattern [36], vorgeschlagen wurde, das sich zudem großer Beliebtheit erfreut.

Beispiel

Das Prinzip dieses Entwurfsmusters ist anhand eines Beispiels in Abbildung 4.9 ersichtlich: Einem Subjekt, hier konkret einer Person oder einer Firma, können verschiedene Rollen, hier konkret die der Kundin oder die der Lieferantin, zugeordnet werden, und zwar beliebig viele von jeder Sorte (durch den Stern auf der Seite der Rolle angedeutet). Es kann also beispielsweise eine bestimmte Person (gleichzeitig oder nacheinander) beliebig oft Kundin und/oder Lieferantin sein, wobei jede einzelne der Rollen für das Vorkommen der Person in einer Beziehung (zu einem anderen Objekt, hier einer anderen Person oder Firma) steht. Obwohl das Subjekt und seine Rollen durch verschiedene Instanzen repräsentiert werden, bilden sie logisch eine Einheit.

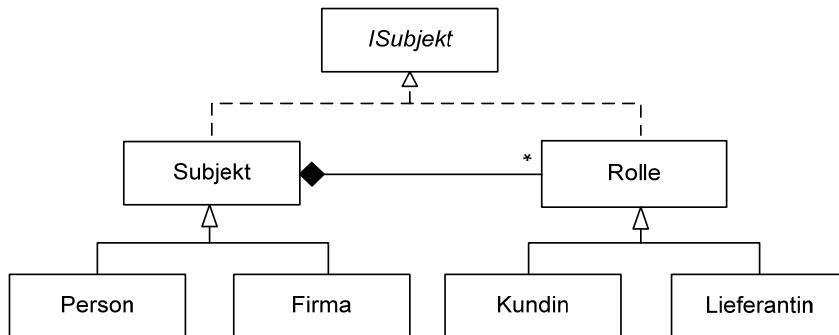


Abbildung 4.9: Das ROLE OBJECT Pattern am Beispiel von Personen und Firmen, die die Rollen der Kundin und/oder der Lieferantin einnehmen können. Entscheidend ist, dass ein Objekt und seine Rollen als separate Instanzen dargestellt werden, die alle einen eigenen Zustand und eine eigene Identität haben. Ersteres ist in der Regel von Vorteil, letzteres von Nachteil (s. Text).

Allgemein drückt das Entwurfsmuster aus, dass ein Objekt — häufig „Subjekt“ genannt (ein Objekt in der Rolle des Subjekts) — eine oder mehrere Rollen (Objekte in der Rolle der Rolle!) gleichzeitig spielen kann und in diesen Rollen jeweils unterschiedliche, rollenspezifische Eigenschaften und Zustände aufweist. Rollen können nach diesem Muster dynamisch angenommen und abgelegt werden, was sich darin ausdrückt, dass dem Subjekt neue Rollenobjekte zugeordnet bzw. bestehende weggenommen werden. Ein Subjekt in einer bestimmten Rolle wird dann durch das jeweilige, dazu gehörende Rollenobjekt vertreten. Dabei müssen Rollenobjekte nicht alle Eigenschaften des Subjekts wiederholen — sie können Anfragen, im Falle einer Person zum Beispiel nach dem Namen oder dem Alter des Subjekts, an das Subjekt delegieren bzw. forwarden (vgl. Abschnitt 4.2.3). Wenn die Rollen (wie in Abbildung 4.9) zudem das gleiche Interface implementieren wie das Subjekt, hat man es sogar mit einem Fall von *Ersetzung der Vererbung durch Delegation* (s. Abschnitt 5.2.4.8) zu tun.

Abbildung 4.10 zeigt das ROLE OBJECT Pattern im Original, mit anderen Namen. Sowohl Subjekt (**ComponentCore**) als auch Rollen (**ComponentRole**) verfügen über Methoden zur Verwaltung von Rollen; die Organisation und Koordination von Rollen bedienen sich der gegenseitigen Verknüpfungen. Die konkreten Rollen fügen dann den für sie spezifischen Zustand und das Verhalten hinzu.

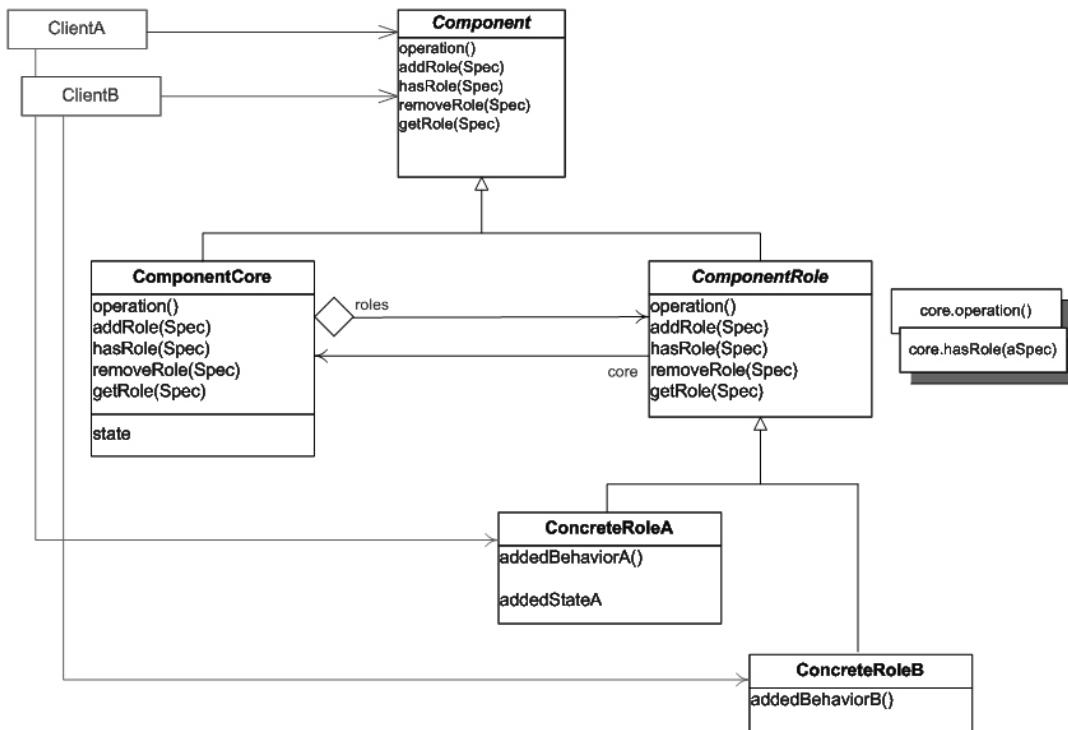


Abbildung 4.10: Das ROLE OBJECT Pattern im Original (aus [36]).

Eine häufige, an das Subjekt delegierte Anfrage wird die nach der Identität sein: Wenn zwei verschiedene Objekte dasselbe Subjekt repräsentieren, dann müssen Anfragen bezüglich der Identität dieser Objekte in bestimmten Fällen positiv ausfallen, obwohl diese Objekte eigentlich verschieden sind. Wenn z. B. eine Person zugleich Kundin und Lieferantin ist (und nach dem ROLE OBJECT Pattern in diesen Rollen durch verschiedene Instanzen der entsprechenden Klassen repräsentiert wird), dann sollte die Frage, ob es sich bei Kundin und Lieferantin um dieselbe Person handelt, bejaht werden. Dieses Problem, nämlich dass eine (logische) Instanz plötzlich mehrere (realisierungstechnische) Identitäten hat, ist auch unter dem Namen *Objektschizophrenie* bekannt. Es wird technisch in der Regel durch einen zusätzlich eingeführten Identitätstest gelöst: In JAVA könnte man beispielsweise zusätzlich zum Test `==` und zur Methode `equals` noch eine weitere Methode einführen, die die Identität von Rollenspielern prüft. Diese Methode (in Abbildung 4.10 nicht enthalten) wäre dann Bestandteil des ROLE OBJECT Pattern.

Problem und Vermeidung der Objektschizophrenie

Wie die meisten Entwurfsmuster, so wird auch das ROLE OBJECT Pattern durch ein Zusammenspiel von Rollen beschrieben: Die Kollaborationen von Objekten, die darin ausgedrückt werden (in Klassendiagrammen durch Assoziationen, also durch Linien repräsentiert), sind Kollaborationen von Rollen, nicht von bestimmten Objekten oder gar Klassen. Man findet also in beinahe allen Entwurfsmustern Rollen, auch wenn es in den Mustern

Rekursivität des ROLE OBJECT Pattern

selbst nicht um das Konzept der Rolle geht. In UML-Diagrammen werden diese Rollen meistens durch Klassensymbole, lediglich mit einem Rollen-Stereotypen «Role» versehen, dargestellt; den gängigen objektorientierten Programmiersprachen fehlt das Rollenkonzept jedoch leider ganz. In der Darstellung von Entwurfsmustern entfällt übrigens selbst der Rollen-Stereotyp häufig (so auch in den Beispielen dieser Kurseinheit), da sowieso klar ist, dass es sich bei den beteiligten Klassen lediglich um Repräsentanten, genauer um die Rollen richtiger Klassen in einer konkreten Anwendung des Musters handelt. Das Besondere am ROLE OBJECT Pattern ist nun, dass damit eben dieses (in den Programmiersprachen fehlende) Rollenkonzept nachgebildet (emuliert) werden soll. Diese Nachbildung unterscheidet sich aber in fundamentaler Weise von dem Rollenkonzept, das es dazu verwendet, denn auch im ROLE OBJECT Pattern kommen Rollen vor, nämlich die des Subjekts und die der Rolle! Man versucht nur einmal, die Rollen des ROLE OBJECT Pattern mit dem ROLE OBJECT Pattern zu modellieren!

4.4.6 FACTORY METHOD Pattern

Die Einführung von Interfaces oder abstrakten Klassen in ein Programm kann der Entkopplung dienen und damit einem hohen Ziel. Doch erst wenn in einem Teilprogramm (oder einer Komponente) ein Klassename gar nicht mehr erwähnt wird, ist es von der Klasse wirklich unabhängig, also entkoppelt. Leider reicht die Einführung von Interfaces dazu nicht aus — solange Instanzen von dieser Klasse gebraucht werden, muss die Klasse — und sei es nur für den Konstruktoraufruf — auch erwähnt werden (s. a. Abschnitte 1.4 und 1.6). Wie lässt sich das vermeiden?

Wenn eine Klientin eine Klasse nicht kennt, aber trotzdem eine Instanz von ihr benötigt (eigentlich: eine Instanz, die dem Interface genügt, das die Klientin kennt — die genaue Klasse kann ihr egal sein), dann muss diese Instanz entweder schon vorher existieren und ihr übergeben werden oder eine andere, der Klientin bekannte Klasse muss sie für sie erzeugen. Wir betrachten hier nur den zweiten Fall; der erste, auch als *Dependency injection* bekannt, wurde in Kurseinheit 1 bereits ausführlich behandelt, führt aber nicht zum FACTORY METHOD Pattern und ist daher hier nicht von Belang.

Factory und Factory-Methode

Um eine Instanz zu erzeugen, die das gewünschte Interface implementiert, wendet sich die Klientin also an eine ihr bekannte Instanz oder Klasse — im folgenden *Factory* genannt — und bittet diese, die Instanz für sie zu erzeugen. Die Factory bietet dazu eine Methode, die sog. **Factory-Methode** (im Falle einer Klasse eine Klassenmethode, in JAVA als **static** deklariert) an, die als Ergebnis die gewünschte Instanz liefert. Für die Factory ist das kein Problem, denn diese darf die Klasse der erzeugten Instanz kennen. Statt

```
912 class Client {
913     IServer x = new Server();
914 }
```

schreibt man dann

```
915 class Client {
916     IServer x = Factory.createServer();
```

```
917 }  
  
918 class Factory {  
  
919     static Server createServer() {  
920         return new Server();  
921     }  
922 }
```

Allerdings, und das sollte Ihnen sofort aufgefallen sein, tauscht man hier nur die eine Abhängigkeit, nämlich die des **Client** von **Server**, gegen eine zweite: die des **Client** von **Factory**. Viel scheint damit nicht gewonnen.

Der Vorteil einer Factory wird erst dann ersichtlich, wenn sie Instanzen verschiedener Klassen erzeugen kann. Der nachfolgende Code erlaubt der Klientin die Auswahl zwischen zwei verschiedenen Arten von Objekten, obwohl sie nur eine Klasse (nämlich **Factory**) kennt:

Erzeugung von Instanzen verschiedener Klassen

```
923 class Factory {  
  
924     static Server1 createServer1() {  
925         return new Server1();  
926     }  
  
927     static Server2 createServer2() {  
928         return new Server2();  
929     }  
930 }
```

Doch auch hier scheint der Gewinn fraglich, muss doch die Klientin jetzt verschiedene Methodennamen kennen, die jeweils die Klasse der geforderten Instanz durchscheinen lassen. Tatsächlich rentiert sich eine Factory-Methode erst dann, wenn alle Objekte von einer Methode erzeugt werden und der Typ des erzeugten Objekts von den Parametern dieser Methode (oder irgendeiner anderen Bedingung) abhängt, wie in folgendem Codeausschnitt zu sehen:

```
931 class Factory {  
  
932     static IServer createServer(...) {  
933         if (...)  
934             return new Server1(...);  
935         if (...)  
936             return new Server2(...);  
937         ...  
938     }  
939 }
```

Man beachte, dass dazu alle konkreten Serverklassen Subtypen von **Iserver** sein müssen, ein Umstand, der aber sowieso schon durch die Klientin vorgegeben war (die Zuweisung in Programmzeile 913 oben). Dies ist insbesondere dann kein Problem, wenn Factories im Zusammenhang mit Klassenhierarchien auftreten, was sie in der Regel tun.

Factory und Familieninterface in einem

Es ergibt sich bei der Verwendung von Factories zur Erzeugung von Instanzen einer Klassenhierarchie eine attraktive Verschmelzungsmöglichkeit, die eine separate Factory-Klasse überflüssig macht: Was spricht dagegen, die Klasse **Factory** durch eine abstrakte Klasse **Server** zu ersetzen, die zugleich Factory und Wurzel der Klassenhierarchie ist, die, da sie ja abstrakt ist, keinen Konstruktor hat, statt dessen aber eine Factory-Methode, die in Abhängigkeit von übergebenen Parametern etc. Instanzen ihrer konkreten Subklassen zurück gibt? Nichts!

```

940 abstract class Server implements Iserver {
941     static Server createServer(...) {
942         ... weiter wie in Zeile 933
943     }
944 }

945 class Server1 extends Server {
946     ...
947 }

948 class Server2 extends Server {
949     ...
950 }
...

```

Ein konkretes **Beispiel** dafür ist das folgende:

```

951 public abstract class Number {
952     static Number fromString(String aString) {
953         if (... aString ...)
954             return new Integer(aString);
955         if (... aString ...)
956             return new Float(aString);
957     }

958     abstract Number plus(Number aNumber);
959     abstract Number minus(Number aNumber);
960     ...
961 }

```

```
962 class Integer extends Number {  
963     ...  
964 }
```

```
965 class Float extends Number {  
966     ...  
967 }
```

Man beachte, dass `Number` eigentlich ein *Familieninterface* (s. Abschnitt 1.5.5) der (dahinter verborgenen, nicht `public` deklarierten) Klassen `Integer` und `Float` darstellt, lediglich erweitert um eine Factory-Methode, die die Instanzen erzeugt. Da in JAVA Interfaces keine Factory-Methoden anbieten können, greift das Beispiel auf eine abstrakte Klasse zurück. Es ist hier aber auch konzeptuell gerechtfertigt, das Familieninterface durch eine abstrakte Klasse zu repräsentieren, da eine starke inhaltliche Verwandtschaft (genaugenommen sind die ganzen Zahlen ja eine Teilmenge aller Zahlen, so dass `Number` eine echte *Generalisierung* ist) besteht.

Wenn man trotzdem gern beim Interface bleiben möchte, kann man (in JAVA) die Factory-Methode auch in eine innere Klasse des Interfaces verbannen:

Verwendung einer inneren Factory-Methode

```
968 interface Number {  
  
969     class Factory {  
970         static Number fromString(String aString) {  
971             if (... aString ...) return new Integer(aString);  
972             if (... aString ...) return new Float(aString);  
973         }  
974     }  
975 }
```

Die Factory-Methode wird dann über `Number.Factory.fromString(...)` qualifiziert aufgerufen.

Solche Factory-Methoden findet man relativ häufig. Sie sind jedoch nur eine Vorstufe zum FACTORY METHOD Pattern, das nachfolgend näher erläutert wird.

Im FACTORY METHOD Pattern wird in einer Superklasse eine abstrakte Methode deklariert, die eine Instanz eines bestimmten Typs zurückgeben soll. Die Methode kann das natürlich nicht selbst (denn sie ist ja abstrakt); deswegen wird sie in den konkreten Subklassen der abstrakten Klasse überschrieben. Die überschreibenden Methoden geben dann eine Instanz zurück, deren Typ von der jeweils implementierenden Klasse vorgeschrieben wird. Anders als im obigen Beispiel zu den Factory-Methoden ist hier das erzeugte Objekt aber keine Instanz der Subklasse selbst, sondern einer anderen, in der Regel mit der Subklasse per *Subtyping* nicht verwandten. Tatsächlich müsste ja schon eine Instanz der Subklasse existieren, damit die überschriebene Methode, eben auf dieser Instanz, aufgerufen werden kann (und es

Factory METHOD Pattern

stellt sich dann die Frage, wozu man noch die Factory braucht, wenn Instanzen offensichtlich auch anders hergestellt werden können). Abbildung 4.11 zeigt das zum Muster gehörige Klassendiagramm.

Da der Factory-Methodaufruf dynamisch gebunden wird, heißt dieses Muster auch *virtueller Konstruktor* (engl. virtual constructor). Es wird vor allem im Kontext von Frameworks benutzt, in denen die Programmiererin eine Kollaboration zwischen abstrakten Klassen vorfindet, für die sie konkrete Subklassen liefern möchte, und wo die Instanzerzeugung bestimmter benutzerdefinierter Klassen (im gegebenen Fall Subklassen von **Product**) vom Framework selbst vorgenommen werden soll, also unter Verwendung von Code der abstrakten Klassen (hier: **creator**).

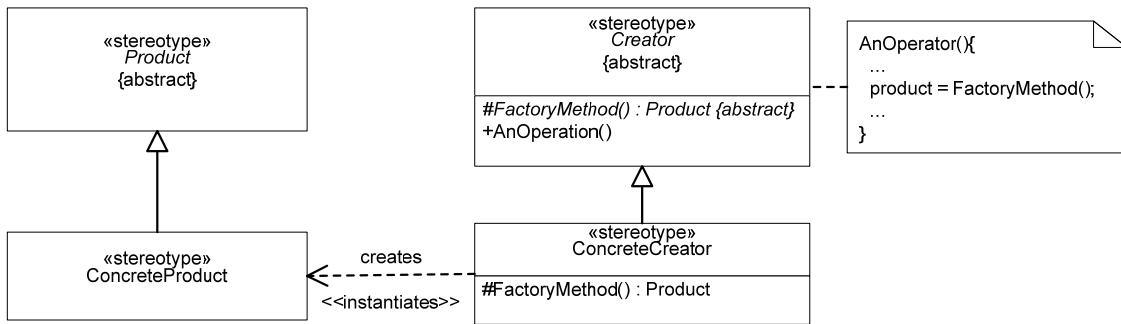


Abbildung 4.11: Das FACTORY METHOD Pattern sieht für jede konkrete Subklasse einer Produktfamilie (angeführt durch die abstrakte Superklasse **Product**) eine konkrete Subklasse einer abstrakten Factory-Klasse (**Creator**) vor, die eine Factory-Methode (**FactoryMethod**) überschreibt. Der dynamisch gebundene Aufruf dieser Methode gibt dann die Instanz einer Klasse zurück, die vom jeweiligen Empfängerobjekt abhängt. Allerdings muß für dieses Verfahren zunächst die konkrete Factory-Klasse instanziert worden sein.

Zusammenhang zwischen Factory-Methoden und dem FACTORY METHOD Pattern

Zunächst scheint es, als ob das FACTORY METHOD Pattern mit den einfachen Factory-Methoden aus den obigen Beispielen nur wenig gemeinsam hätte. Wenn man jedoch den *impliziten Parameter* des Methodenaufrufs, das Empfängerobjekt (eine Instanz einer Subklasse von **Creator**), zum expliziten Parameter macht (und den Empfänger durch eine Klasse oder andere Instanz ersetzt), dann erhält man eine Factory-Methode wie ab Programmzeile 932 oben (wobei die Bedingung aus einer Auswertung des Parameterobjekts besteht). Tatsächlich wird in [35] eine Variante des FACTORY METHOD Patterns, *Parameterized factory method*, definiert, bei der der Typ der erzeugten Instanzen auch von den Parametern der Factory-Methode abhängt. Wenn diese zusätzlich noch eine statische (also nicht dynamisch gebundene) Klassenmethode wäre, dann wären wir genau bei der obigen Variante, nämlich der Factory-Methode, angelangt. Umgekehrt lässt sich eine Factory-Methode, in deren Rumpf die zu instanzierende Klasse per Fallunterscheidung ausgewählt wird (wie in Zeile 933 ff. oben), durch das Refactoring „Bedingung durch Polymorphismus ersetzen (REPLACE CONDITIONAL WITH POLYMORPHISM)“ aus Abschnitt 5.2.1.3 in das FACTORY METHOD Pattern überführen.

Persönlich bin ich noch nie einem echten (im Sinne der ursprünglichen Definition aus [35]) FACTORY METHOD Pattern begegnet — es waren immer Factory-Methoden. Auch finde ich das FAC-

TORY METHOD Pattern einigermaßen schwer vom *ABSTRACT FACTORY Pattern* zu unterscheiden (was wir uns hier dann auch ersparen wollen).

4.4.7 ADAPTER Pattern

Manchmal hat man es mit Klassen zu tun, die einander brauchen, die aber nicht zueinander passen. Etwas weniger beziehungstechnisch ausgedrückt: Wenn eine Klasse (per *Required interface*) bestimmte Funktionen aufrufen will, die eine andere Klasse (per *Provided interface*) zwar im Prinzip anbietet, doch leider anders benennt, und wenn dann — aus welchen Gründen auch immer — weder das eine noch das andere Interface angepasst werden kann, dann braucht man einen Adapter.⁵¹

Der Adapter vertritt die aufzurufende, aber nicht passende Klasse gegenüber der Aufruferin. Anders ausgedrückt: Der Adapter stellt genau das Interface zur Verfügung, das die Aufruferin braucht — das *angebotene Interface* des Adapters ist also gleich dem *benötigten Interface* der Aufruferin. Wohlgemerkt: Er stellt das Interface zur Verfügung, nicht die Implementierung! Für diese greift der Adapter auf die eigentlich aufzurufende Klasse zurück.

Ganz in der Tradition der objektorientierten Programmierung gibt es zwei Möglichkeiten, wie der Adapter sich die benötigte Funktionalität besorgen kann:

1. Er erbt sie.
2. Er hält sich eine Sklavin.

Im ersten Fall bekommt der Adapter doch noch selbst die Funktionalität, die die Aufruferin möchte, allerdings unter anderen (den geerbten) Methodennamen. Die Methoden, die er der Aufruferin anbietet, müssen dann lediglich die entsprechenden geerbten Methoden aufrufen.

Beispiel

Angenommen, ein Client will die Methode **x** des Servers aufrufen, der die benötigte Funktion zwar anbietet, sie aber leider **y** nennt:

```
976 class Server {  
977     ...  
978     void y() {...}  
979 }  
  
980 class Client {  
981     Server server;  
982     ...
```

⁵¹ Die unterschiedliche Benennung ist noch nicht einmal entscheidend: Es reicht, dass die Klientin eine Anbieterin eines bestimmten Typs erwartet (ausgedrückt durch den Typ einer Variable der Klientin), den die Anbieterin nicht erweitert oder implementiert (so dass Instanzen der Anbieterklasse nicht zuweisungskompatibel mit der Variable der Klientin sind); vgl. hierzu auch „Nominale vs. strukturelle Typkonformität“ in Abschnitt 1.2.2.

```

983     server.x(); // geht nicht, weil Server x nicht anbietet
984     ...
985 }
```

Dann führt man eine von Server abgeleitete Klasse ein, die den Methodenruf umleitet und die den Server beim Client ersetzt:

```

986 class Adapter extends Server {
987     ...
988     void x() {
989         y();
990     }
991 }
```

```

992 class Client {
993     Adapter adapter;
994     ...
995     adapter.x(); // jau!
996     ...
997 }
```

Man spricht dann auch von einer **Adapterklasse**. Wenn man zusätzlich noch die Methoden, die er nicht braucht oder kennt (z. B. **void y()**), vor dem Client verbergen will, kann man noch ein kontextspezifisches *Client/Server-Interface* für **client** und **Adapter** einführen (im Original des Patterns Target genannt):

```

998 interface Target {
999     void x();
1000 }
```

```

1001 class Client {
1002     Target target;
1003     ...
1004     target.x();
1005     ...
1006 }
```

```

1007 class Adapter extends Server implements Target {
1008     ...
1009     void x() {
1010         super.y();
1011     }
1012 }
```

```

1013 class Server {
1014     ...
1015     void y() {...}
```

```
1016 }
```

Im zweiten Fall hält der Adapter einfach eine Instanz der unpassenden Anbieterin und delegiert die Methodenaufrufe der Klientin an diese weiter:

```
1017 class Adapter implements Target {
1018     Server server; // die Sklavin ...
1019     ...
1020     void x() {
1021         server.y(); // ... macht die Arbeit!
1022     }
1023 }
```

Man spricht dann auch von einem **Adapterobjekt**. Man beachte, dass das Interface **Target** hier nur noch zum Entkoppeln vom Adapter und nicht mehr zum Verbergen von **Server** geerbter Methoden vor **Client** benötigt wird.

Die zweite Variante, die in Abbildung 4.12 dargestellt ist, kommt ohne *Vererbung* aus und ist zudem flexibler: Falls dies notwendig werden sollte, kann das Objekt, das die Funktion bereitstellt, zur Laufzeit ausgewechselt werden. Diese Variante (mit dem Adapterobjekt) geht übrigens aus der ersten (mit der Adapterklasse) durch das *Refactoring „Vererbung durch Delegation ersetzen (REPLACE INHERITANCE WITH DELEGATION)“* hervor, das in Abschnitt 5.2.4.8 beschrieben wird.

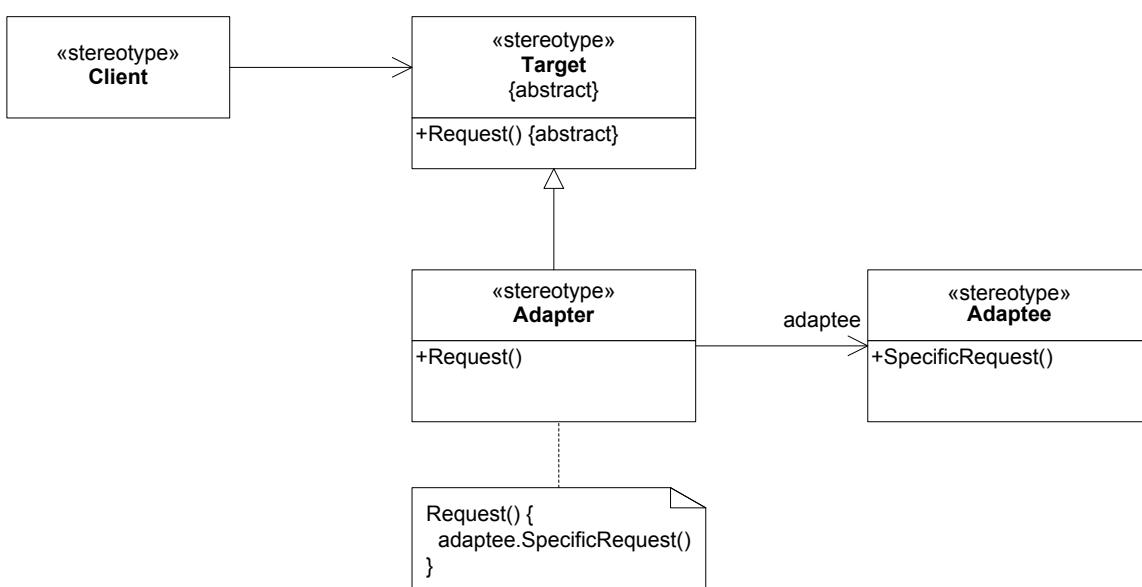


Abbildung 4.12: Beim Adapterobjekt wird im Gegensatz zur Adapterklasse nicht mit Vererbung gearbeitet, sondern mit Delegation (Forwarding). Das Interface Target dient der Entkopplung des Clients vom Adapter selbst (der wiederum von Adaptee entkoppelt), eine Funktion, die sich nur lohnt, wenn man den Adapter ebenfalls austauschbar halten möchte.

4.4.8 FACADE Pattern

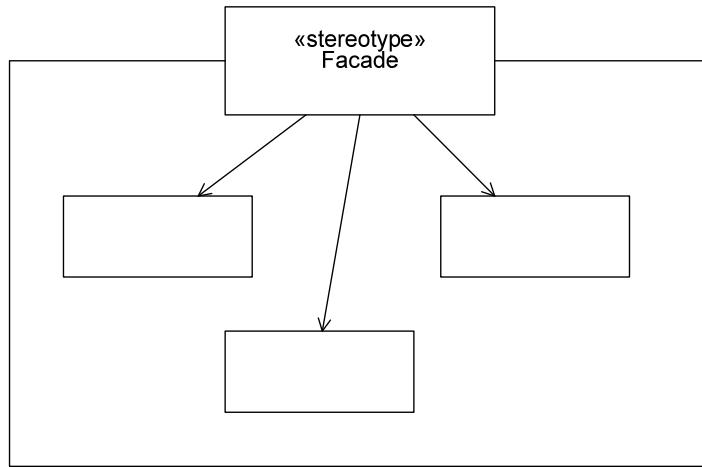


Abbildung 4.13: Beim Fassademuster steht ein Objekt bzw. Klasse für ein ganzes Teilsystem. Der Zugriff auf Funktionen dieses Teilsystems läuft beim Fassademuster stets über die Fassade, die diese entsprechend weiterleitet.

Große Subsysteme bestehen leicht aus mehreren hundert Klassen. Das erschwert ihre Verwendung insofern, als man, wenn man eine bestimmte Funktionalität benötigt, nicht nur wissen muss, wie die dazugehörige Methode heißt, sondern auch, welche Klasse diese anbietet. Dies gilt nicht nur für irgendwelche Spezialfunktionen, sondern häufig auch für grundlegende Funktionen des Subsystems, die — je nach Dünkel der Designerin — von der einen oder der anderen Klasse angeboten werden. Bildlich gesprochen entspräche die Verwendung eines solchen Systems der Aufgabe, bei einem integrierten Schaltkreis, von dem man nur eine bestimmte Teilfunktion benötigt, die Zellen ausfindig zu machen, die diese Funktionalität bereitstellen, und diese dann direkt (also nicht über die Pins des Gehäuses des Schaltkreises) zu verdrahten. Wenig komfortabel.

Eine Fassadenklasse ist eine Klasse, die ein Subsystem kapselt (s. Abbildung 4.13). Sie bietet all die Funktionen des Subsystem gebündelt an, auf die Klientinnen ohne gesondertes Spezialwissen zugreifen können sollen. Einzige Funktion der Fassade ist es, Methodenaufrufe an die richtigen Klassen (bzw. deren Objekte) weiterzuleiten. Von außen gesehen steht die Fassadenklasse (bzw. ein Objekt derselben) für das gesamte Subsystem; sie verbirgt die Teile hinter dem Ganzen.

Ein angenehmer Nebeneffekt der Verwendung einer Fassadenklasse ist die bessere Entkopplung der Klientin vom Subsystem: Anstatt sich an eine Vielzahl von einzelnen Klassen zu binden, muss die Klientin nur noch die Fassade kennen. Da die Fassade zudem von Natur aus eine Umsetzerfunktion (*Adapter*; siehe Abschnitt 4.4.7) wahrnimmt, ist es ohne Probleme möglich, die gesamten Interna des Subsystems (einschließlich der inneren Struktur) auszutauschen, ohne dass eine Klientin davon betroffen sein muss. Dies setzt jedoch die Verwendung geeigneter Interfaces als Parametertypen der Fassadenmethoden voraus.

Das Fassademuster gehört zu der Klasse der Muster, die, wie auch schon das *ROLE OBJECT Pattern*, eigentlich nur eine Schwäche einer Programmiersprache ausmerzen, in diesem Fall der fehlende sinnvolle Komponentenbegriff: Wie man sich leicht überzeugen kann, ist das Konzept des Packages nicht dazu geeignet, die Funktion einer Fassade zu übernehmen. Und so muss — wie so oft — das inzwischen etwas überstrapazierte Konzept der Klasse herhalten, um ein fehlendes Element der objektorientierten Programmierung zu simulieren.

4.4.9 VISITOR Pattern

Es gibt Situationen, in denen in einem Teilbaum der Klassenhierarchie alle Klassen die gleichen Methoden implementieren, die Implementierungen sich aber derart voneinander unterscheiden, dass die *Vererbung* innerhalb der Klassenhierarchie nicht zu Zwecken der Wiederverwendung genutzt werden kann.⁵² Wenn dann noch hinzukommt, dass die Methoden jeder einzelnen Klasse (die ja — mit unterschiedlichen Implementierungen — in jeder Klasse vorkommen) wenig miteinander verwandt sind und womöglich auch noch, im Zuge der Weiterentwicklung, häufig um weitere ergänzt werden müssen, dann hat man es mit einem Phänomen zu tun, das in der aspektorientierten Gemeinde gern als *Crosscutting concern* umschrieben wird: Zwei Ordnungen, in diesem Fall die bestehende Klassenhierarchie und die Menge der zusammengehörigen Methoden, überschneiden sich, ohne dass sich die eine Ordnung sinnvoll der anderen unterwerfen ließe.

Tabelle 4.1: Das Problem: Verschiedene Klassen implementieren dieselben Methoden, wobei die Implementierungen alle so unterschiedlich sind, dass sie nicht geerbt werden können, und zwar selbst dann nicht, wenn man die Funktionen zu den Klassen machen würde, also

	Klasse 1	Klasse2	Klasse3	...
Funktion 1	Implementierung 1/1	Implementierung 1/2	Implementierung 1/3	...
Funktion 2	Implementierung 2/1	Implementierung 2/2	Implementierung 2/3	...
Funktion 3	Implementierung 3/1	Implementierung 3/2	Implementierung 3/3	...
...

gewissermaßen Zeilen und Spalten vertauschen würde.

Man kann sich die Situation gut anhand einer Funktionsmatrix veranschaulichen. Wenn man in der Waagerechten die Klassen aufträgt und in der Senkrechten die Funktionen, dann ergibt sich das Bild, das in Tabelle 4.1 dargestellt ist. Wie man leicht sieht, müssen für eine neue Klasse (eine neue Spalte in der Tabelle) alle Funktionen neu implementiert werden. Das ist jedoch kein

⁵² Die Wurzel des Teilbaums könnte also durchaus ein Familieninterface sein.

Problem, da die Ergänzungen lokal beschränkt sind — sie betreffen ja nur die Definition der neuen Klasse. Etwas anderes ist es, wenn man eine neue Funktion (eine neue Zeile) hinzufügt: Dann müssen alle Klassen angefasst (erweitert) werden. Dies ist insbesondere dann ein Problem, wenn man die Klassen selbst gar nicht unter Kontrolle hat, z. B. weil sie eine andere implementiert oder sie Teil eines Frameworks sind, über dessen Quellcode man nicht verfügt.

Das Standardbeispiel für diesen Fall ist eine integrierte Entwicklungsumgebung, die ein vorliegendes Programm intern durch einen *abstrakten Syntaxbaum* repräsentiert. Der Syntaxbaum hat Knoten, die, je nachdem, was sie repräsentieren, Instanzen unterschiedlicher Klassen sind. Allen Knoten gemeinsam sind bestimmte Operationen, die auf dem gesamten Programm ausgeführt werden müssen. Das sind zum Beispiel Funktionen zur Syntaxprüfung, zur Codegenerierung oder auch zum Refaktorieren (s. Kurseinheit 5). Jede dieser Funktionen ist knotenspezifisch: Sie muss auf die interne Repräsentation des jeweiligen Knotens zurückgreifen und wird schon deswegen für jede einzelne Klasse (= Knotentyp) separat implementiert werden. Die Situation ist im Diagramm von Abbildung 4.14 (wieder in einer Vorläufernotation von UML) dargestellt.

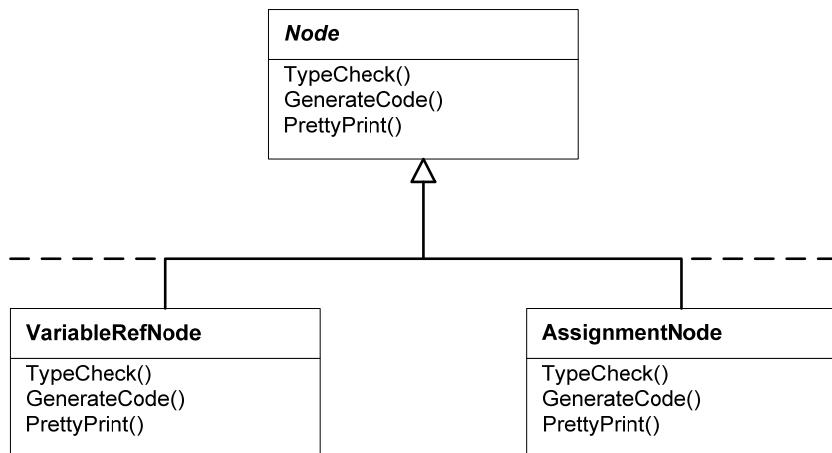


Abbildung 4.14: Die Knoten eines abstrakten Syntaxbaums eines Programms repräsentieren höchst unterschiedliche Dinge, auf denen jeweils gleiche Operationen durchgeführt werden müssen. Aufgrund der Unterschiedlichkeit, insbesondere der internen Repräsentation, unterscheidet sich aber auch die Implementierung der Operationen erheblich — Vererbung von der Superklasse Node kommt deswegen nicht in Frage (die Methoden in Node sind alle abstrakt).

Aus Gründen der Wartbarkeit und besseren zukünftigen Erweiterbarkeit (Ergänzung um Plugins) würde man nun gerne die Methoden aus der Klassenhierarchie herausfaktorisieren. Da diese Methoden aber (und wie schon gesagt) auf den Zustand der einzelnen Knoten zugreifen können müssen (und nicht zuletzt deswegen auch genau dort implementiert wurden), ist dies nicht ganz so einfach, wie man vielleicht meinen möchte.

Eine Lösung besteht nun darin, für jede Funktion (Zeile der Tabelle 4.1) eine eigene Klasse einzuführen, die die unterschiedlichen Implementierungen der Funktion für die einzelnen Klassen (Spalten der Tabelle) enthält. Es werden dann Spalten *und* Zeilen durch Klassen repräsentiert und die Auswahl einer Implementierung hängt von zwei Klassen oder, genauer, von den Typen zweier Objekte ab (s. u.). Da die Implementierung der Funktionen nun bei den neuen Klassen liegt, erhalten Instanzen dieser neuen Klasse eine Art Besuchsrecht bei den Instanzen der ur-

sprünglichen Klassenhierarchie eingeräumt, dass es ihnen erlaubt, auf die Interna der Instanzen zuzugreifen. Dieses Besuchsrecht (oder besser die Ausübung dessen, das Besuchen) gibt dem resultierenden Muster seinen Namen: *VISITOR Pattern*.

Offensichtlicher Nachteil dieses Patterns ist, dass bei Hinzufügen einer neuen Knotenklasse (Spalte in Tabelle 4.1) alle Funktionsklassen (Zeilen) angefasst werden müssen. Das VISITOR Pattern lohnt sich also nur, wenn die Zahl der Klassen gegenüber der Zahl der Funktionen relativ konstant ist.

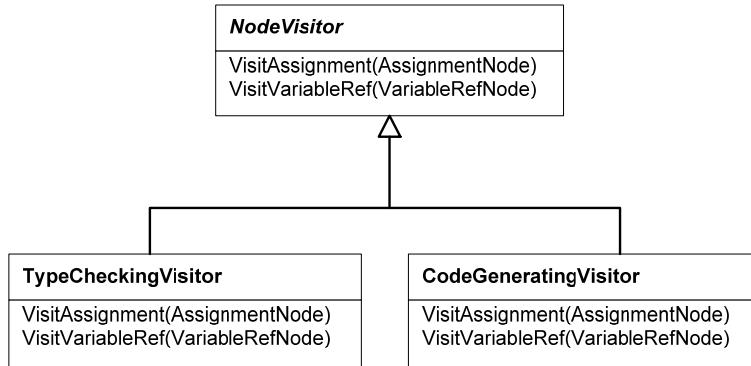


Abbildung 4.15: Die Klassen mit den herausfaktorisierten Methoden. Jede Subklasse entspricht einer Zeile einer Tabelle wie Tabelle 4.1

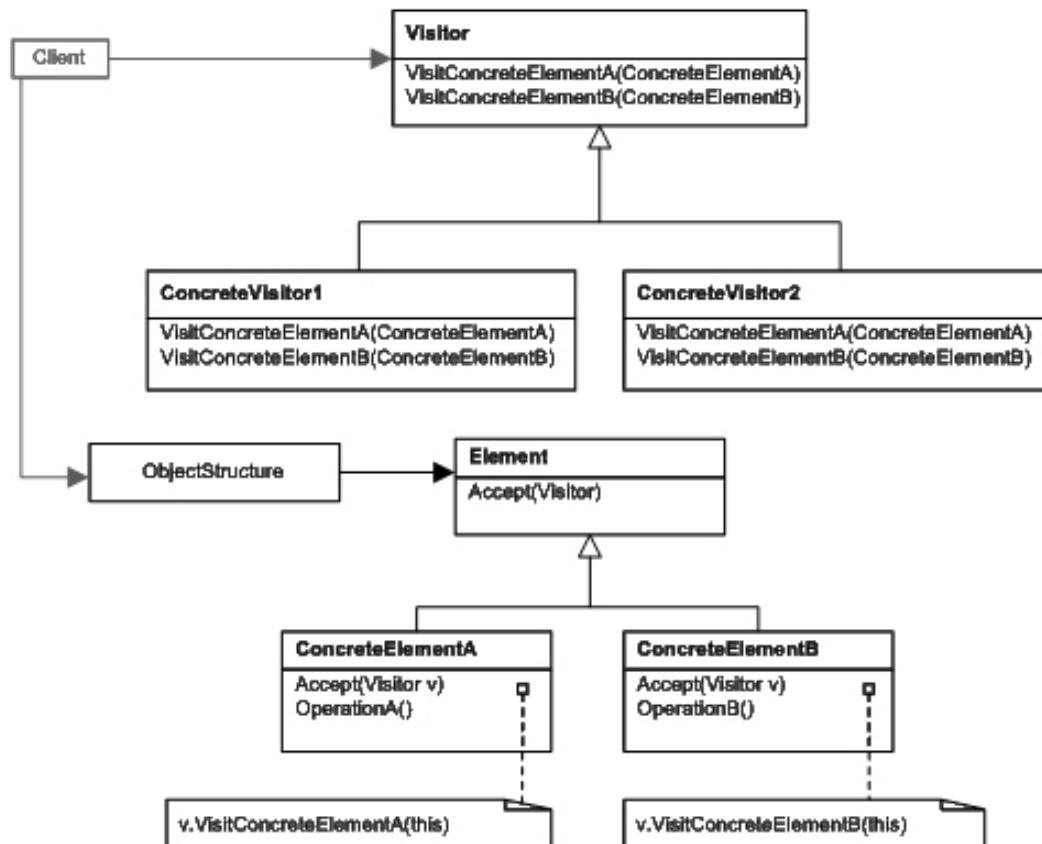


Abbildung 4.16: Das VISITOR Pattern als Ganzes, aus [35]. Leider ist das Klassendiagramm wenig ausschussreich — das Codebeispiel unten (Programmzeile 1024 ff.) mag da weiterhelfen.

Abbildung 4.15 zeigt zunächst zwei der für das Beispiel aus Abbildung 4.14 resultierenden Klassen beispielhaft. Man beachte, dass die Namen der herausfaktorisierten Methoden zu Klassennamen geworden sind: `TypeCheck()` zu `Typecheckingvisitor` etc. Die einzelnen Methoden einer solchen Klasse erfüllen alle dieselbe Funktion (nämlich Type checking usw.), aber für jeweils eine andere besuchte Klasse. Dies spiegelt sich in der Namensgebung der Methoden wider: Sie beginnen alle mit „Visit“, gefolgt vom Namen der Klasse, deren Instanzen sie besuchen. Der eigentliche Methodenname, der einen Hinweis auf die Funktion der Methode enthält, wird nicht wiederholt — er steckt ja schon im Namen der Klasse. Man beachte, dass bei Verwendung einer Programmiersprache, die die Überladung von Methoden erlaubt, alle Methoden „Visit“ heißen könnten; sie werden allein aufgrund der Parametertypen unterschieden und korrekt gebunden.

Das Klassendiagramm, das das Muster zusammenfasst, ist, in verallgemeinerter Form, in Abbildung 4.16 zu sehen. Man erkennt zwei Hierarchien, die obere für die Zeilen aus Tabelle 4.1, die untere für die Spalten. Wenn Sie nicht auf Anhieb durchblicken, wundern Sie sich nicht: Das VISITOR Pattern ist — zumindest meiner Ansicht nach — eines der am schwersten nachzuvollziehenden. Auch dessen Umsetzung in Code verursacht regelmäßig Kopfzerbrechen, da sie reichlich Gelegenheit bietet, Fehler zu machen, und das Debugging aufgrund des häufigen Hin- und Herspringens einen leicht den Faden verlieren lässt (s. nachfolgendes Codebeispiel). Auf der anderen Seite werden Sie auch nicht so häufig in die Verlegenheit kommen, es verwenden zu müssen — wenn doch, nehmen Sie sich am besten ein funktionierendes Beispiel und wandeln Sie es nach Ihren Bedürfnissen ab.

VISITOR Pattern bedingt Öffnung der besuchten Klassen

Bei der oben erwähnten Einräumung des Besuchsrechts wurde so getan, als ob die für die Durchführung der Methoden in den Visitor-Klassen notwendigen Interna nur den Visitoren zugänglich gemacht werden könnten. Eine solche dedizierte Veröffentlichung von Eigenschaften ist in den meisten Programmiersprachen jedoch nicht möglich. (Die besuchte Klasse kann zwar ein kontextspezifisches Interface implementieren, das nur für den Visitor bestimmt ist, aber dieses Interface wäre kaum effektiv vor anderen Klassen zu verbergen.) Als unmittelbare Folge bedeutet die Verwendung des VISITOR Pattern, dass die besuchten Klassen Eigenschaften öffentlich machen müssen, die eigentlich ihr Implementationsgeheimnis bleiben sollten. Es ist also genau abzuwägen, ob dies hinnehmbar ist.

Die nachfolgende Codestrecke zeigt die Umsetzung des VISITOR Patterns, allerdings anhand eines anderen Beispiels.

Beispiel

```

1024 interface Visitor {
1025     void visit(Wheel wheel);
1026     void visit(Engine engine);
1027     void visit(Body body);
1028     void visit(Car car);
1029 }

1030 class Wheel {
```

```
1031  private String name;  
  
1032  Wheel(String name) {  
1033      this.name = name;  
1034  }  
  
1035  String getName() {  
1036      return this.name;  
1037  }  
  
1038  void accept(Visitor visitor) {  
1039      visitor.visit(this);  
1040  }  
1041 }  
  
1042 class Engine {  
  
1043     void accept(Visitor visitor) {  
1044         visitor.visit(this);  
1045     }  
1046 }  
  
1047 class Body {  
  
1048     void accept(Visitor visitor) {  
1049         visitor.visit(this);  
1050     }  
1051 }  
  
1052 class Car {  
1053     private Engine engine = new Engine();  
1054     private Body body = new Body();  
1055     private Wheel[] wheels  
1056     = { new Wheel("front left"), new Wheel("front right"),  
1057           new Wheel("back left") , new Wheel("back right") };  
  
1058     void accept(Visitor visitor) {  
1059         visitor.visit(this);  
1060         engine.accept(visitor);  
1061         body.accept(visitor);  
1062         for (Wheel wheel : wheels)  
1063             wheel.accept(visitor);  
1064     }  
1065 }  
1066 }
```

```

1067 class PrintVisitor implements Visitor {
1068     public void visit(Wheel wheel) {
1069         System.out.println("Visiting " +wheel.getName()
1070                 +" wheel");
1071     }
1072     public void visit(Engine engine) {
1073         System.out.println("Visiting engine");
1074     }
1075     public void visit(Body body) {
1076         System.out.println("Visiting body");
1077     }
1078     public void visit(Car car) {
1079         System.out.println("Visiting car");
1080     }
1081 }
1082 public class VisitorDemo {
1083     public static void main(String[] args){
1084         Car car = new Car();
1085         Visitor visitor = new PrintVisitor();
1086         car.accept(visitor);
1087     }
1088 }
```

Das Codebeispiel zeigt sehr schön, was an der Problemstellung crosscutting ist: Das Interface **visitor**, das von der Menge der Visitoren abstrahiert, listet gleichzeitig alle Klassen der besuchten Knoten auf, und zwar in Form formaler Parametertypen. Die Struktur der Knoten wiederholt sich also in der Struktur der Visitoren, nur eben leider orthogonal dazu (in Form einer Menge von Methodenüberladungen).

Selbsttestaufgabe 4.7

Bei Betrachtung des obigen Beispielcodes mag man sich fragen, warum die Methode **accept**, die ja in jeder Klasse gleich implementiert wird (oder zumindest gleich implementierte Anteile hat), nicht einfach in eine gemeinsame Superklasse ausgelagert wird. Warum funktioniert das in JAVA nicht?

Double dispatch und das VISITOR Pattern

Das VISITOR Pattern offenbart übrigens ein interessantes programmiersprachliches Problem. Während es in der objektorientierten Programmierung üblich ist, dass die zu einem Methodenaufruf gehörende Implementierung der Methode vom Typ des Empfängers abhängt, muss im Fall des VISITOR Patterns die zu einem Aufruf

von `visit` gehörende Implementierung in Abhängigkeit von zwei Typen ermittelt werden: dem des Empfängers und dem des Parameters. Im gegebenen Beispiel äußert sich das darin, dass zunächst die Methode `accept` mit dem Visitor als Parameter aufgerufen und (in Abhängigkeit vom Empfänger) dynamisch gebunden wird und dann die Methode `visit` mit `this` (also dem ursprünglichen Empfänger) als Parameter und dem Visitor (dem ursprünglichen Parameter) als Empfänger. Im Ergebnis wird also genau die Methode ausgewählt, die zum dynamischen Typ des ursprünglichen Empfängers und des ursprünglichen Parameters passt. Dies ist insbesondere dann interessant, wenn an der ursprünglichen Aufrufstelle weder der konkrete (dynamische) Empfänger- noch der konkrete Parametertyp bekannt sind (wie in Programmzeile 1086 der Fall). Man nennt die in diesem Beispiel verwendete Technik, die Simulation des gleichzeitigen Dispatching auf Empfänger- und Parametertyp durch zwei aufeinanderfolgende Dispatches (wobei das zweite den Parameter des ersten zum Empfänger macht), **Double dispatch**.

In der Literatur wird Double dispatch übrigens häufig mit dem VISITOR Pattern gleichgesetzt, was aber so nicht ganz richtig ist, denn das VISITOR Pattern löst zunächst das Problem, eine Matrix von $m \times n$ Operationen in einer Klassenstruktur unterzubringen — das Double dispatch hilft dann lediglich bei der Auswahl (beim Aufruf) der Methoden. Double dispatch kann aber auch ohne Anwendung des VISITOR Patterns zum Einsatz kommen [38].

VISITOR Pattern ≠ Double dispatch

Wenn die verwendete Programmiersprache bei der dynamischen Bindung auch die tatsächlichen Parametertypen berücksichtigen würde (sog. *Multi dispatch*), könnte man sich den Aufruf von `accept` sparen und direkt `visit` aufrufen, allerdings nur, wenn `accept` selbst nicht noch weitere Funktionen (neben der, `visit` aufzurufen) erfüllt. Man beachte jedoch, dass auch bei einer Lösung mit Multi dispatch die zweite Klassenhierarchie benötigt wird und das Prinzip des Visitors somit erhalten bleibt (nicht zuletzt sollen die Methoden ja nicht frei im Raum herumschwirren, sondern in Klassen organisiert werden).

VISITOR Pattern mit Multi dispatch

Alternativ wäre auch denkbar, dass jede Zelle der obigen Tabelle durch genau eine Klasse abgedeckt wird. Auch wenn dadurch das Prinzip der Klasse (oder sogar der Objektorientierung insgesamt) in Frage gestellt wird (man könnte ja stattdessen freie, also nicht an eine Klasse gebundene Funktionen haben), so wird auch diese Lösung von einem Muster nahegelegt: dem *EXTENSION OBJECT Pattern* [35] (hier nicht behandelt). Eine andere Möglichkeit, die insbesondere für nicht vollständig besetzte Matrizen (also Tabellen, in denen einige Kombinationen nicht vorkommen) geeignet ist, ist das *ACYCLIC VISITOR Pattern* [37] (hier ebenfalls nicht behandelt).

Alternativen und Varianten

4.5 Bewertung

Auch wenn sich die durch die Entwurfsmuster ausgelöste Euphorie inzwischen als übertrieben herausgestellt hat, so hat sich deren Nutzen doch als vielfältiger erwiesen, als man vermuten könnte. Folgende Vorteile sollte man nicht übersehen:

1. Mit der Definition und Katalogisierung von Entwurfsmustern wurde ein Vokabular geschaffen, mit dem sich Softwareentwickler kompakt austauschen können. Wenn etwa jemand von einem Composite oder einem Visitor redet, hat jeder gleich eine recht konkrete Vorstellung davon, was gemeint und wie es zu implementieren ist. Über Details wird man zwar noch reden müssen, aber das grobe Vorgehen ist klar.
2. Programmiererfahrung, die sonst jede mühevoll selbst erwerben muss, kann auf kompakte Art und Weise weitergegeben werden. Viele Entwurfsmuster enthalten die Essenz Jahrzehntelanger Programmierpraxis und bieten damit Lösungen an, auf die so manch eine selbst erst nach Jahren des Herumprobierens (wenn überhaupt) gekommen wäre. Und so bietet jede Definition eines Entwurfsmusters immer auch eine hübsche kleine Einheit für den Unterricht in (guter) objektorientierter Programmierung.
3. Durch die Vereinheitlichung bestimmter Mechanismen wird auch klar, um welche Sprachkonstrukte es sich lohnt, Programmiersprachen zu erweitern. So hat z. B. das OBSERVER Pattern in der Programmiersprache C# seinen Niederschlag in Form der eingebauten Sprachkonstrukte `event` und `delegate` gefunden. Allgemein gilt: Der einen Sprache Muster ist der anderen Sprache Konstrukt.

4.6 Ausblick

Derzeit sind Muster zum Großteil noch Wissen ohne andere Materialisierungen als die auf Papier und in Köpfen. Natürlich wünscht man sich Möglichkeiten, Muster in Bibliotheken vorrätig zu halten und, analog zu den heutigen Bibliotheksklassen, auf einfache Weise in ein Projekt einbinden („wiederverwenden“) zu können.

Anders als einzelne Klassen leben Muster aber vom Zusammenspiel mehrerer, aufeinander abgestimmter Klassen. Dabei sind die Klassen von Anwendungsfall zu Anwendungsfall verschieden — der Teil der Funktionalität, der musterspezifisch ist, ist in den *Rollen* des Musters festgeschrieben. Der *Vererbungsmechanismus* der objektorientierten Programmierung lässt sich nur bedingt dafür einsetzen, ein Muster zu verallgemeinern bzw. von der standardisierten Implementierung eines Musters abzuleiten, und auch ein Komponenten- oder Framework-Ansatz helfen wenig. Die Parametrisierung von Typen (Generics) mag es erlauben, dass eine oder andere Patterns in abstrakter Form zu spezifizieren, doch noch sind mir keine speziellen Sprachkonstrukte bekannt, die die Definition und Implementierung von Patterns erleichtern. Mit einer Ausnahme.

Ein Teil des Problems scheint zu sein, dass Muster stets nur einen Aspekt des Zusammenspiels zwischen Klassen beschreiben: Der primäre Zweck der konkreten, in Art eines Musters zusammen spielenden Anwendungsklassen ist ein anderer als der, der ihnen innerhalb des Musters zukommt. Es liegt also nahe, das Vorkommen von Mustern als *Aspekte* (im Sinne der *aspektorientierten Programmierung*, s. Abschnitt 6.3) eines Programms zu begreifen.

4.7 Weiterführende Literatur

Das Entwurfsmusterbuch [35] ist sicherlich ein Klassiker der objektorientierten Programmierung. Leider ist es inzwischen etwas veraltet, insbesondere was die Umsetzung der Muster in Code angeht. Neuere Bücher zum Thema sind häufig Machwerke, denen man das Bemühen anmerkt, die ersten zu sein — sie wirken hastig verfasst und sind voller Fehler. Buchempfehlungen möchte ich also keine aussprechen. Wer jedoch gern im Internet stöbert, kann ja mal mit <http://c2.com/cgi/wiki?WikiPagesAboutWhatArePatterns> anfangen.

- [35] E Gamma, R Helm, R Johnson, J Vlissides Design Patterns — Elements of Reusable Software (Addison-Wesley, 1995). Dublette zu [1].
- [36] D Bäumer, D Riehle, W Siberski, M Wulf „Role object pattern“ in: In Proceedings of PLoP '97 Technical Report WUCS-97-34 (Washington University, Dept. of Computer Science 1997).
- [37] RC Martin „Acyclic visitor“ in: Pattern Languages of Program Design 3 (1997) 93–103.
- [38] DHH Ingalls „A simple technique for handling multiple polymorphism“ in: NK Meyrowitz (Ed.) Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86) (1986) 347–349.

4.8 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 4.1 (Seite 130)

Der Aufruf von `n()` aus `m()` in `Sub` ist der Aufruf einer geerbten Methode, der Aufruf von `o()` aus `n()` in `Super` ist eine offene Rekursion, bei der, je nach Klasse des Empfängers des ursprünglichen Aufrufs `n()`, die Implementierung von `o()` in `Super` oder `Sub` aufgerufen wird. In JAVA könnte man das Vererbungsinterface durch die Verwendung des Access modifiers `protected` für `n()` und `o()` in `Super` kenntlich machen, wobei dann aber Aufrufe dieser Methoden von Klienten von `Super` unmöglich würden; außerdem könnte man `o()` in `Sub` mit der Annotation `@override` kenntlich machen. Letzteres wäre aber in Super, der Aufrufsstelle, nicht zu sehen. In C# und C++ würde man `o()` in `Super` als `virtual` kennzeichnen, in C# zusätzlich `o()` in `Sub` als `override`. Die (Möglichkeit zur) offene(n) Rekursion würde dadurch klar.

Selbsttestaufgabe 4.2 (Seite 132)

`Sub2Super` ist ein Client/Server-Interface, `Super2Sub` ein Server/Client-Interface.

Selbsttestaufgabe 4.3 (Seite 137)

s. Abbildung 4.4

Selbsttestaufgabe 4.4 (Seite 142)

Die vererbungsbasierte Variante aus Abbildung 4.5 erlaubt es nicht, die (neue) Klasse **Root** sinnvoll einzusortieren. Die interfacebasierte Variante aus Abbildung 4.6 hingegen würde **wurzel** einfach als weitere Klasse aufnehmen, die lediglich das Interface Ganzes implementiert.

Selbsttestaufgabe 4.5 (Seite 143)

Es scheitert unseres Wissens daran, dass es in JAVA 1.5 keine Möglichkeit gibt, Beziehungen zwischen zwei Typparametern auszudrücken, also z. B. dass wenn ein Typparameter mit dem Typ X instanziert wird, ein anderer mit dem Typ Y instanziert werden muss. Wenn Sie dennoch eine Lösung finden, immer her damit!

Selbsttestaufgabe 4.6 (Seite 147)

Die resultierende abstrakte Klasse **Subject** könnte der Klasse **Observable** entsprechen. Um sicherzustellen, dass ein Subjekt eines bestimmten Typs nur zu ihm passende Observer registrieren kann, muss der Typ der Observer entsprechend eingeschränkt werden. Dazu wäre es notwendig, die Methode **addEventListener(.)** bzw. **addobserver(.)** kovariant zu überschreiben, also den Parametertyp, der ja zunächst mit **AnswerListener** bzw. **Observer** angegeben ist, zu einem entsprechenden Subtyp zu verfeinern. Das ist jedoch in JAVA nicht erlaubt. (Vgl. Selbsttestaufgabe 2.3.)

Selbsttestaufgabe 4.7 (Seite 166)

Weil der (statische) Typ von **this** dann der der Superklasse ist. Die Methode **visit(.)** müsste dann auch noch mit der Superklasse als Parametertyp deklariert werden, damit das Programm überhaupt kompiliert. Da aber in JAVA beim dynamischen Binden nur der Typ des Empfängerobjekts berücksichtigt wird, nicht der der Parameterobjekte, würden alle Versuche des Double dispatch an dieser neuen Methode hängenbleiben; die anderen Implementierungen von **visit(.)** würden niemals aufgerufen.

5 Refactoring

So wie sich in der Software bestimmte Muster ständig wiederholen, so gibt es auch im Prozess der Programmierung selbst bestimmte stereotype Tätigkeiten, die immer wieder und dabei mit nur leichten Variationen ausgeführt werden müssen. Dies betrifft insbesondere die Änderung von Code, bei der selbst so einfache Dinge wie das Umbenennen einer Klasse eine Masse von manuellen Änderungen nach sich ziehen, die eigentlich vollkommen schematisch und damit automatisch erfolgen können sollten.

Selbsttestaufgabe 5.1

Was muss alles geändert werden, wenn sich der Name einer Klasse ändert?

Der zum Teil erhebliche Aufwand, den selbst solch kleine Änderungen nach sich ziehen, führt allzu häufig dazu, dass nur die unvermeidlichen, weil die Funktionalität betreffenden Änderungen im Code durchgeführt werden, während die Modifikationen, die lediglich dem Erhalt von Struktur, Wartbarkeit und Lesbarkeit des Codes dienen würden, ausbleiben. Der Code verkommt damit in dem Maße, in dem er geändert wird; man spricht hier auch von *Softwarerefäulnis*. Das sog. Refaktorieren wirkt der Softwarerefäulnis entgegen.

Refaktorisieren und Softwarerefäulnis

Refaktorisierung: eine Änderung der internen Softwarestruktur mit dem Ziel, sie verständlicher und einfacher änderbar zu machen, ohne das beobachtbare Verhalten zu ändern

übersetzt aus [40]

Da die mit der Refaktorisierung verbundenen Änderungen auf ein Umbauen des Programms abzielen, ohne dessen Bedeutung zu verändern, heißen sie englisch Refactorings. Dabei ist das Wort „Refactoring“ eine Neuschöpfung, die durch die Analogie erklärbar ist, dass ein Programm ein aus einer Menge von Faktoren bestehendes Produkt, eben faktorisiert ist. Diese Faktorisierung offenbart eine innere Struktur, die ohne sie (die Faktorisierung) nicht so leicht sichtbar wäre. Eine Änderung der Faktorisierung nennt man dann folgerichtig Refaktorisierung.⁵³

zum Begriff der Refaktorisierung

Mit der systematischen Befassung und der Katalogisierung möglicher Refaktorisierungen hat der Begriff der Refaktorisierung eine Mehrdeutigkeit erfahren: Er bezeichnet nicht nur den Vorgang oder das Ergebnis einer entsprechenden Tätigkeit (an sich schon eine Mehrdeutigkeit), sondern auch das Muster, nach dem die Tätigkeit erfolgt bzw. dem das Ergebnis gleicht. Während man im Deutschen noch eine sprachliche Unterscheidung treffen kann (das „Refaktorieren“ für die Tätigkeit vs. „die Refaktorisierung“ für das Ergebnis oder das Muster), gelingt dies im Englischen

⁵³ Bei der im Deutschen ebenfalls üblichen Übersetzung „Refaktorierung“ handelt es sich um eine sinnberaubende Verstümmelung, die hier bewußt nicht verwendet wird.

nicht: Dort heißt es in allen Fällen schlicht „refactoring“. Dazu kommt, dass mit Refaktorisierung bzw. Refactoring in zunehmendem Maße auch die Werkzeuge, die eine solche automatisch durchführen, benannt werden. Diese sollten aber zur besseren Unterscheidung Refaktorisierungswerkzeuge bzw. Refactoring tools genannt werden.

5.1 Einordnung

Allgemein dient das Refaktorisieren der Verbesserung des Designs einer Software, *nachdem* diese geschrieben wurde. Dieses Verhalten ist (war) eigentlich verpönt (das Design hat schließlich nach gängigen Vorstellungen von einem geregelten Softwareentwicklungsprozess vor der Implementierung zu erfolgen), die Notwendigkeit ergibt sich aber faktisch aus der Praxis — sie ist schlichtweg Realität. Inzwischen ist man dazu übergegangen, dies anzuerkennen; nicht zuletzt führen die Änderungen der Anforderungen während der Entwicklungsphase (sog. *Requirements drift*, im Mittel ca. 1% der Anforderungen pro Monat) dazu, dass man sich mit dem Thema auseinandersetzen muss. Die Methode des *Extreme Programming* (Kurseinheit 7) verzichtet sogar vollständig auf ein A-priori-Design und setzt vollständig auf das Refaktorisieren.

Refactorings wurden zuerst in der SMALLTALK-Programmierung eingesetzt. Die erste literaturgeschichtliche Erwähnung ist angeblich von 1990, die erste größere wissenschaftliche Abhandlung ist die Dissertation von Opdyke [39]. Das erste bekannte Werkzeug war der sog. *Refactoring-Browser* für SMALLTALK. Aufgrund seines vorwiegend praktischen Anteils ist das Thema Refactoring in der akademischen Welt bislang nicht besonders vertreten. Gleichwohl gibt es eine ganze Menge offener, nichttrivialer Probleme, die anzugehen sich auch für Wissenschaftlerinnen lohnen kann.

5.1.1 Katalogisierung

Ähnlich wie die Entwurfsmuster aus Kurseinheit 4 lassen sich Refaktorisierungen standardisieren, indem man Schemen in Form strukturierter Beschreibungen vorgibt und diese mit Namen versieht. Diese Namen werden dann zu Synonymen häufig recht komplexer Tätigkeiten und damit Bestandteil des Vokabulars von Programmiererinnen.

Die Beschreibung eines Refactorings geht in der Regel von einem Design aus, das (aus welchem Grund auch immer) verbesserungswürdig erscheint. Es stellt diesem Design ein alternatives gegenüber, das das Ziel des Refactorings darstellt. Die notwendige Transformation des Quellcodes vom alten in das neue Design erfolgt dann mittels einer Reihe von kleinen Schritten, die in der Summe die erwünschte Änderung bewirken.

Die wohl auch heute noch umfassendste Katalogisierung von Refactorings ist die von Martin Fowler in seinem gleichnamigen Buch [40]. Eine Erweiterung dieses Katalogs wird (einigermaßen lieblos) von ihm selbst im Internet gepflegt⁵⁴; eine Neuauflage des Buchs mit den entsprechenden Aktualisierungen ist jedoch nicht geplant. Dies steht m. E. im Gegensatz zu der Wich-

⁵⁴ www.refactoring.com/catalog/

tigkeit des Themas für die Programmierpraxis: Während von der Weiterentwicklung von Programmiersprachen in den letzten Jahren kaum noch Produktivitätssteigerungen ausgingen, so können zuverlässige Programmierwerkzeuge die Arbeit erheblich vereinfachen, indem sie stereotype Änderungen wie eben das Refaktorisieren von Code automatisieren. Damit diese Werkzeuge jedoch auch eingesetzt werden, ist es notwendig, dass sie über die jeweiligen Produktgrenzen hinaus bekannt sind und in den Softwareentwicklungsprozess auch gedanklich einbezogen werden (so wie das bei den Entwurfsmustern aus Kurseinheit 4 längst der Fall ist).

Die katalogisierte Beschreibung von Refactorings erfasst fast ausschließlich kleine Refaktorisierungen, also solche, die sich als eine überschaubare Menge

große Refactorings

von Einzelschritten durchführen lassen und deren Erfolg sich leicht überprüfen lässt. Für die Praxis genauso relevant ist aber die Definition und Umsetzung von sog. **großen Refactorings**, also Refactorings, die nicht nur einzelne Programmelemente betreffen, sondern größere Zusammenhänge bis hin zur gesamten Architektur eines Systems. Von einer formalen Vorgehensbeschreibung (oder gar von einer Werkzeugunterstützung) ist man hier aber noch weit entfernt; stattdessen verlegt man sich darauf, große Refactorings als eine Aneinanderreihung kleiner Refactorings zu betrachten. Dagegen spricht jedoch die hohe Fehlerquote, die entsteht, wenn eine große Transformation jedes Mal wieder neu in viele kleine zerlegt werden soll, die dann jeweils einzeln durchgeführt werden müssen. Das wird insbesondere dann schwierig, wenn dabei Zwischenprodukte entstehen, die das große Ziel aus den Augen verlieren lassen.

5.1.2 Refaktorisierungen als Algorithmen

Wenn man sich die Beschreibungen von Refaktorisierungen anschaut, fällt auf, dass sie einem (verbal spezifizierten) Algorithmus gleichen: Ausgehend von einem vorgefundenen Design, der Eingabe, wird durch eine Reihe von Schritten ein Zieldesign, die Ausgabe, erzeugt. Dabei kann das Refactoring während seiner Durchführung weitere Eingaben (von der Benutzerin) erwarten oder bereits vor seinem Start mit Parametern versorgt werden. Das besondere an Refactorings ist lediglich, dass es sich bei Ein- und Ausgabe um Programme handelt. Ein Refaktorisierungswerkzeug ist damit ein Programm, das Programme verarbeitet, nämlich ein *Metaprogramm* (s. Kurseinheit 6).

Refaktorisierungen können damit genau wie andere Programme spezifiziert werden. Insbesondere hat jedes Refactoring eine Menge von *Vorbedingungen*, die erfüllt sein müssen, damit das Refactoring anwendbar ist, und eine Menge von *Nachbedingungen*, die erfüllt sein müssen, wenn die Refaktorisierung abgeschlossen ist. Vor- und Nachbedingungen sind insbesondere auch dann interessant, wenn man mehrere Refactorings hintereinander ausführen will, wobei jedes folgende Refactoring eine oder mehrere Nachbedingungen des Vorgängers als Vorbedingungen zur Voraussetzung hat. Allerdings ist der Beweis der Korrektheit von Refactorings, genau wie der Beweis der Korrektheit von Algorithmen, alles andere als einfach.

Vor- und Nachbedingungen von Refactorings

Eine besondere Nachbedingung jedes Refactorings ist definitionsgemäß, dass es die Bedeutung des Programms nicht ändert. Neben der trivialen Bedingung, dass sich das Programm nach dem Refactoring weiter problemlos übersetzen

automatisches Testen der Korrektheit

lassen muss (zumindest wenn dies zuvor der Fall war), gehört auch dazu, dass verfügbare Unit-Tests (Kurseinheit 3) äquivalente Ergebnisse liefern, in der Regel also weiter erfolgreich durchlaufen. Unabhängig von der Verfügbarkeit von Unit-Tests eröffnet die Anwendung von Refactorings zusätzlich die Möglichkeit des sog. *Back-to-back-Testens*: Man kann einfach die Ausgaben des Programms vor und nach der Refaktorisierung gegeneinander vergleichen. Voraussetzung hierfür ist allerdings, dass man über ein Testframework verfügt, das solche Tests (inkl. der automatischen Generierung der dafür notwendigen Eingaben) automatisch durchführt. Der Grundsatz des Testens, dass man damit nur Fehler finden kann und keine Fehlerfreiheit nachweisen, behält aber auch hier Gültigkeit.

5.1.3 Refactoring to patterns

Zweck eines Refactorings ist es, ein vorgefundenes Design in ein angestrebtes zu überführen. Angestrebtes objektorientiertes Design ist Ihnen ja schon begegnet, und zwar in Form von Entwurfsmustern (Kurseinheit 4). Was läge also näher, als spezielle Refactorings vorzusehen, die die Verwendung eines Entwurfsmusters zum Ziel haben?

Einer katalogartigen Fassung solcher Refaktorisierungen steht vor allem im Wege, dass die Ausgangslage, also ein Design, das kein Entwurfsmuster verwendet, nahezu beliebig ist. Wenn aber die Form der Eingabe unbekannt ist, ist es schwer, ein allgemeines Verfahren anzugeben, wie sie in die Ausgabe überführt werden kann. Stattdessen kann man nur von der gewünschten Ausgabe (die ja feststeht) rückwärts ausgehend angeben, welche der katalogisierten Refactorings dorthin führen können, und die Auswahl der für die jeweils benötigten Transformationen geeigneten Refactorings der Entwicklerin überlassen. Die dabei auftretenden Probleme entsprechen doch im Wesentlichen denen der *großen Refactorings*: Das Refactoring muss jedes Mal neu als eine Folge von kleineren Einzel-Refactorings geplant werden und damit hängt der Erfolg ganz wesentlich vom Geschick der Programmiererin ab.

Etwas anderes ist es jedoch, wenn ein Entwurfsmuster (genauer: seine Anwendung) in ein anderes überführt werden soll: Hier sind sowohl Ausgangs- als auch Zielsituation bekannt. Ein Beispiel für ein solches Refactoring werden wir in Abschnitt 5.1.5 ausführlicher durchgehen. Im allgemeinen dienen jedoch verschiedene Entwurfsmuster verschiedenen Zwecken und zwei Entwurfsmuster, die auf dieselbe Problemstellung angewendet dasselbe Programmverhalten bewirken, sind relativ selten, so dass man auch eher selten von einem Entwurfsmuster in ein anderes refaktorisieren wird.

5.1.4 Werkzeugunterstützung

Die oben erwähnte algorithmische Fassung von Refactorings legt nahe, dass sich Refactorings automatisieren lassen. Allerdings ist es bei fast allen Refactorings (wie schon beim eingangs erwähnten einfachen Beispiel des Umbenennens einer Klasse) nicht mit einem globalen, textuellen Suchen und Ersetzen getan — praktisch immer erfordern die Änderungen eine Interpretation des Programmtextes. Sie benötigen in der Regel den *abstrakten Syntaxbaum* (engl. abstract syntax tree, AST) des Programms.

Nun erlauben moderne IDEs mit ihren APIs in der Regel den Zugriff auf den abstrakten Syntaxbaum eines Programms, so dass die erforderlichen Programmanalysen und -transformationen von einem Refaktorisierungswerkzeug, das auf diesen APIs aufsetzt, direkt auf den benötigten Datenstrukturen durchgeführt werden können. Allerdings sind derartige APIs zum einen nicht sonderlich stabil, zum anderen unterliegen auch die Programmiersprachen, auf deren ASTs die Refactorings ausgeführt werden sollen, ständigen Änderungen, die nicht nur eine Anpassung von Editor und Compiler, sondern auch der Refactorings erfordern. So führte beispielsweise der Übergang von JAVA 2 zu JAVA 5 (mit seinen generischen Typen) dazu, dass viele Refactorings nicht mehr funktionieren und deshalb komplett überarbeitet werden müssten (was aber aufgrund des erheblichen Aufwands nicht immer passiert). Dazu kommt, dass gleiche Refactorings selbst für syntaktisch stark ähnliche Programmiersprachen (wie etwa JAVA und C#) heute nicht einfach übernommen werden können, da die Implementierungen zu sehr von den konkreten ASTs der jeweiligen Sprache abhängen. Höhere Abstraktionsstufen, die eine allgemeine Formulierung von Refactorings erlauben, sind jedoch noch nicht gefunden (oder zumindest noch nicht etabliert) — eine allgemeine, einheitliche Refactoring-Schnittstelle von IDEs ist (noch) nicht in Sicht.

So kommt es, dass sich heutige IDEs auch darin stark unterscheiden, ob und welche Refactorings sie anbieten. Besonders hervor tut sich auf diesem Gebiet die IDE INTELLIJ IDEA der Firma JET-BRAINS, die mittlerweile eine beachtliche Menge an Refactorings anbietet. In der akademischen Gemeinde am sichtbarsten ist aber zurzeit immer noch ECLIPSE, dessen Refactorings teilweise einen beträchtlichen Funktionsumfang erreicht haben. Entwicklungsumgebungen wie VISUAL STUDIO ziehen hier erst langsam nach. Alleinstehenden Refaktorisierungswerkzeugen scheint, eben aufgrund ihrer mangelnden Integration, in der Praxis keine besondere Bedeutung mehr zuzukommen, selbst wenn ihnen noch am ehesten die Lösung der obengenannten technischen Probleme zuzutrauen wäre.

Zum Schluss noch eine Bemerkung zur Qualität: Trotz des beträchtlichen Potentials zur Automatisierung des Tests von Refaktorisierungswerkzeugen (s. o.) weisen die heute verfügbaren Implementierungen zum Teil erhebliche Mängel auf. So führen selbst einfache Refaktorisierungen in der Praxis schon zu Syntax- oder Typfehlern, so dass das wichtigste Merkmal heutiger Refaktorisierungswerkzeuge ist, ob die Undo-Funktion zuverlässig funktioniert. Daraus lassen sich zwei Dinge ableiten: Die korrekte Fassung eines Refactorings, einschließlich seiner Vorbedingungen, für alle möglichen Verwendungen ist viel komplexer, als es die oft informellen Beschreibungen glauben machen wollen, und die Verfügbarkeit von zuverlässigen, automatisierten Refaktorisierungen ist weit weniger entscheidend für den Erfolg einer IDE, als man vielleicht erwarten würde. Tatsächlich ergeben Umfragen unter Entwicklerinnen und die Rückmeldungen aus Fehlerdatenbanken, dass verfügbare Refaktorisierungswerkzeuge längst nicht im erwarteten Umfang genutzt werden.

5.1.5 Ein Beispiel

Nach den eher theoretischen Betrachtungen der vorigen Abschnitte wollen wir uns nun etwas ausführlicher mit einem konkreten Beispiel befassen, und zwar einem, das Ihnen aus Abschnitt 4.2 bereits bekannt ist: Vererbung durch Delegation ersetzen. Das dazugehörige Refactoring nennt sich denn auch genau so, auf Englisch **REPLACE INHERITANCE WITH DELEGATION**.

In [40] wird dieses Refactoring kurz und knapp wie folgt charakterisiert:

A subclass uses only part of a superclasses interface or does not want to inherit data.

Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.

Es folgt eine etwas ausführlichere Motivation und Beschreibung der Schritte, die jedoch nicht über den Inhalt des obigen hinausgeht. Die zur Durchführung des Refactorings konkret notwendigen Schritte werden in einem Abschnitt „Mechanics“ wie folgt ausgeführt:

- *Create a field in the subclass that refers to an instance of the superclass. Initialize this field to this.*
- *Change each method defined in the subclass to use the delegate field. Compile and test after changing each method.*
- *Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.*
- *For each superclass method used by a client, add a simple delegating method.*
- *Compile and test.*

Vorbedingungen

Wir erkennen im ersten Satz die *Vorbedingung* (wenn auch nicht unbedingt als solche formuliert): Eine Klasse ist Subklasse einer anderen und erbt von ihr Dinge, die sie selbst nicht braucht. Der erste Teil der Vorbedingung ist ein hartes Ausschlusskriterium: Wenn eine Klasse keine Subklasse ist, lässt sich das Refactoring nicht anwenden. Der zweite Teil ist etwas weicher: Wenn sie keine Dinge erbt, die sie nicht braucht, ist das Refactoring zwar immer noch anwendbar, bringt aber nicht den beschriebenen Nutzen.

Schritte und Nachbedingungen

Der zweite Satz stellt eine Kombination von grob zusammengefassten Handlungsanweisungen und *Nachbedingung* dar: Nachdem man das Erforderliche getan hat, kann die Vererbungsbeziehung gelöst werden, die damit nicht mehr besteht. Dass das Programm weiter funktioniert wie bisher, ist im Begriff des Refactorings implizit (wenn sich die Bedeutung ändern würde, wäre es kein Refactoring) und braucht daher nicht extra erwähnt zu werden.

Der aufmerksamen Leserin von Kurseinheit 4 wird nicht entgangen sein, dass das Refactoring *Delegation* einzuführen vorgibt, aber nur *Forwarding* einführt: Für eine echte Delegation fehlt ja der Verweis (das Feld) von der ehemaligen Superklasse zurück zur Klasse. Das bedeutet, dass wenn es vorher offen rekursive Aufrufe in der Superklasse gab, dass dann die refaktorierte

Klasse nach dem Refactoring davon nicht mehr erreicht wird. Es folgt, dass entweder die Vorbedingungen oder die Handlungsanweisungen unvollständig sind.⁵⁵ Eine genauere Untersuchung des Refactorings ist also angebracht.

5.1.5.1 Vorbedingungen

Eine triviale Vorbedingung des Refactorings, die in [40] unerwähnt bleibt, ist die, dass die Superklasse nicht abstrakt sein darf, da sonst keine Instanz gebildet werden kann, an die „delegiert“ (eigentlich: geforwardet) wird. Zudem müssen alle Konstruktoren der Superklasse, die benötigt werden, um eine brauchbare Instanz zu erhalten, an die delegiert werden kann, zugreifbar sein. Sie dürfen also in JAVA insbesondere nicht **private** oder **protected** deklariert sein. Welche Konstruktoren (neben dem Default-Konstruktor, der in JAVA bei Vererbung automatisch mit aufgerufen wird) man braucht, lässt sich aus den in den Konstruktoren der Klasse vorhandenen Super-Aufrufen ablesen.

Eine dritte, schon weniger offensichtliche Vorbedingung ist die, dass das Programm keine Zuweisungen von Instanzen der zu refaktorisierenden Klasse an Variablen vom Typ der Superklasse oder dessen Supertypen enthalten darf, denn mit der Elimination der Vererbungsbeziehung geht auch die Aufgabe der Subtypenbeziehung einher. Diese ist aber zumindest in Sprachen mit *nominalem Typsystem* (worunter fast alle heute gebräuchlichen Sprachen mit statischer Typprüfung fallen; vgl. Kurs 01814) Voraussetzung für die Zulässigkeit einer Zuweisung. Diese Vorbedingung lässt sich etwas abschwächen, wenn man bereit ist, Subtypenbeziehungen zu Supertypen der Superklasse durch neue Extends- bzw. Implements-Klauseln wiederherzustellen, wobei man sich bei ersteren natürlich wieder Vererbung einkauft, die man ja gerade eliminieren wollte. Wenn aber Zuweisungen an den Typ der Superklasse im Programm vorhanden sind, lässt sich das Refactoring einfach nicht anwenden.

Die vierte Vorbedingung hatten wir bereits erwähnt: Da es sich bei der durch das Refactoring eingeführten „Delegation“ lediglich um schnödes Forwarding handelt, dürfen Instanzen der zu refaktorisierenden Klasse nicht von ihrer Superklasse aus offen rekursiv, also über **this**, aufgerufen werden. Eine Quelle solch offen rekursiver Aufrufe wurde dabei jedoch schon ausgeschlossen: Da die Zuweisungskompatibilität der Klasse mit ihrer ehemaligen Superklasse aufgehoben wird, können Instanzen der Klasse die ihrer Superklasse nicht mehr ersetzen. Das Vorkommen entsprechender Zuweisungen im Programm schließt die dritte Vorbedingung bereits aus. Es bleiben jedoch geforwardete Aufrufe der Klasse selbst an die ehemalige Superklasse, die nun, wie in Abbildung 4.1 zu sehen, nicht mehr die Methoden der Klasse aufrufen können, weil **this** nun eine Instanz der ehemaligen Superklasse bezeichnet. Solche Aufrufe müssen also ebenfalls ausgeschlossen werden, wenn keine Änderung des Programmverhaltens provoziert werden soll.

⁵⁵ Wie schon im Kontext von *Design by contract* (in Kurseinheit 2) erwähnt, kann man ein fehlerhaftes Programm dadurch korrigieren, das man seine Vorbedingungen verschärft. Wenn die Vorbedingungen nicht eingehalten werden, braucht das Programm auch nichts (Sinnvolles) zu tun.

Eine fünfte Vorbedingung ergibt sich aus dem Umstand, dass in JAVA nur Methodenaufrufe weitergeleitet werden können: Greifen Klientinnen der Klasse auf von der Superklasse geerbte Felder zu, so würden diese Zugriffe nach dem Refactoring ins Leere gehen. Das ließe sich allerdings vermeiden, wenn Feldzugriffe konsequent über Zugriffsmethoden (Accessoren) erfolgen würden, die dann weitergeleitet werden könnten. In EIFFEL ist das übrigens immer der Fall, da dort Feldzugriffe transparent, also für die Zugreiferinnen nicht sichtbar, immer über Accessoren erfolgen; in C# lässt es sich über sog. *Properties* ebenfalls einrichten, ohne die Klientinnen anfassen zu müssen. Man beachte, dass es nicht ausreicht, die betreffenden Felder in der Klasse einfach zu wiederholen (neu zu deklarieren): Die ehemals geerbten Methoden, an die jetzt weitergeleitet wird, hätten nur Zugriff auf ihre eigenen Versionen dieser Felder und eine Synchronisation der Inhalte ist aus technischen Gründen nicht möglich.

Eine sechste Vorbedingung ergibt sich aus den Sichtbarkeitsregeln der jeweils verwendeten Programmiersprache: Wenn beispielsweise in JAVA eine Methode mit dem Access modifier **protected** geerbt wird und vererbende und erbende Klasse nicht im selben Paket sind, dann ist diese Methode nach dem Refactoring nicht mehr zugreifbar und kann auch nicht per Forwarding aufgerufen werden.

Eine siebte Vorbedingung ergibt sich daraus, dass bestimmte Subtypbeziehungen aufrecht erhalten werden müssen, selbst wenn keine sie verlangenden Zuweisungen in einem Programm vorhanden sind: So dürfen beispielsweise keine entsprechenden expliziten Typtests (in JAVA per **instanceof** oder **getClass()**) vorkommen und die Ableitung von sog. Unchecked exceptions wie **Error** und **RuntimeException** muss aufrecht erhalten bleiben, damit der Compiler keine fehlenden Exception handler bzw. Throws-Klauseln moniert.

Eine letzte Vorbedingung schließlich ist so unoffensichtlich, dass vermutlich nur die allerwenigsten durch bloßes Überlegen auf sie kommen. Wenn auf den Instanzen einer Subklasse synchronisierte Methoden aufgerufen werden, die teilweise von einer Superklasse geerbt werden, und sich diese Aufrufe nach dem Refactoring auf zwei verschiedene Instanzen verteilen, klappt die Synchronisation u. U. nicht mehr (da jetzt zwei anstelle eines Monitors herangezogen werden).

Wie man sieht, sind die Voraussetzungen für die Anwendung selbst eines scheinbar so simplen Refactorings wie des hier beschriebenen alles andere als banal. Dabei ist man noch gut bedient, wenn der Compiler die Verletzung einer Vorbedingung entdeckt, weil sich nämlich das trotzdem refaktorierte Programm nicht mehr übersetzen lässt; weit schlimmer ist es, wenn die Verletzung — oder gar die Existenz! — einer Vorbedingung unentdeckt bleibt. Da aber der formale Beweis, dass die für ein Refactoring genannten Vorbedingungen vollständig sind und ausreichen, eine korrekte, bedeutungserhaltende Refaktorisierung durchzuführen, alles andere als auf der Straße liegt, ist man bei allen Refaktorisierungen, und seien sie noch so gut dokumentiert, darauf angewiesen, ihren Erfolg per Testen zu überprüfen. Nicht gerade schmeichelhaft für eine Disziplin, die der Softwareentwicklung eine bedeutende Produktivitätssteigerung bescheren will.

5.1.5.2 Durchführung

Sind die Vorbedingungen erfüllt, kann das Refactoring durchgeführt werden. Doch auch dabei ergibt sich ein nichttriviales Problem: Es ist nämlich gar nicht automatisch klar, welche geerbten Felder und Methoden einer Klasse nicht gebraucht werden. Der Wunsch allein, bestimmte Elemente loszuwerden, reicht dazu nicht aus — es muss auch sichergestellt werden, dass diese Elemente nicht von Klientinnen oder Subklassen der Klasse benötigt werden.

Nun kann man sich auf den Standpunkt stellen, der Compiler wird einem schon sagen, welche Elemente erhalten bleiben müssen — Zugriffe auf nicht mehr vorhandene Programmelemente führen schließlich zu einem Übersetzungsfehler. Aber dieser Ansatz ist der des Trial and error: Man entfernt einfach die Elemente, die man gern loswerden würde, und wenn darunter eines zuviel war, bekommt man dies rückgemeldet. Für die Praxis ist das jedoch wenig befriedigend: Zum einen werden in den typischen Anwendungsfällen dieses Refactorings sehr viele Elemente geerbt (50 und mehr bei Ableitung von Frameworkklassen wie denen des AWT oder SWING) und zum anderen kann sich dabei herausstellen, dass das Refactoring insgesamt keine gute Idee war (nicht den gewünschten Effekt hat), man es also gern komplett wieder rückgängig machen möchte. Dann aber ist schon viel geändert und die Wiederherstellung des Ausgangszustands entsprechend aufwendig. Nicht zuletzt ist es, wenn kein entsprechendes Refaktorisierungswerkzeug vorhanden ist, kaum zumutbar, erst alle Methodenweiterleitungen einzuführen, um diese dann einzeln wieder zu entfernen. Macht man es aber andersherum, beginnt man also ohne eine Methodenweiterleitung und führt diese wie vom Compiler gefordert ein, kann es sein, dass man zunächst so viele Fehlermeldungen bekommt, dass der Überblick verlorengeht.

Was man statt dessen für eine zielgerichtete Anwendung des Refactorings bräuchte, ist eine Programmanalyse, die vorab feststellt, welche Elemente von der zu refaktorisierenden Klasse verlangt werden. Eine solche Analyse wird im Prinzip vom Compiler durchgeführt, während er die Referenzen auflöst — ein entsprechendes Refaktorisierungswerkzeug kann sich diese zunutze machen, wenn es Zugriff auf den AST des Programms hat. Tatsächlich verfügt die Implementierung des REPLACE INHERITANCE WITH DELEGATION Refactorings in INTELLIJ IDEA über eine solche Analyse; leider ist sie nicht vollständig, so dass das Programm nach der Durchführung des Refactorings so manches Mal nicht mehr kompiliert. Das zeigt einmal mehr, wie wenig entwickelt das Thema Refaktorisierungswerzeuge heute noch ist.

5.1.5.3 Nachbedingungen

Nach einer erfolgreichen Durchführung des Refactorings sollten neben der allgemeinen Nachbedingung für Refactorings, dass das Programm hinterher immer noch dasselbe tut, auch die speziellen Ziele erreicht sein. Das heißt konkret, dass die refaktorierte Klasse nicht mehr von ihrer ehemaligen Superklasse erbt, sondern statt dessen ein Feld besitzt, mittels dessen ihre Instanzen auf jeweils eine Instanz der ehemaligen Superklasse verweisen, an die sie weiterdelegieren können. Dieses Feld muss bei jeder möglichen Form der Instanziierung der refaktorisierten Klasse automatisch mit einer Instanz der ehemaligen Superklasse versorgt werden. Weiterhin enthält die Klasse für mindestens all die Methoden, die ehemals geerbt wurden und die vom Programm gebraucht werden, entsprechende Weiterleitungsmethoden an die ehemalige Superklasse. Kon-

strukturen der Superklasse, die zuvor per `super` aufgerufen wurden, werden jetzt zur Erzeugung der Instanz, an die delegiert wird, aufgerufen. Zuletzt ist die Klasse weiterhin Subklasse von Superklassen ihrer ehemaligen Superklasse, nämlich dann, wenn Zuweisungen im Programm eine entsprechende Zuweisungskompatibilität verlangen, und sie implementiert alle Interfaces, die das Programm (wiederum per Existenz entsprechender Zuweisungen) verlangt. Man beachte, dass durch die verlangten Interfaceimplementierungen u. U. auch Methoden in das *Klasseninterface* aufgenommen und weitergeleitet werden müssen, die von keiner Klientin jemals aufgerufen werden; in diesem Fall ist jedoch das entsprechende Interface zu hinterfragen (und ggf. zu refaktorieren).

Eine allgemeine, in den Kontext anderer Refactorings eingebundene Beschreibung von REPLACE INHERITANCE WITH DELEGATION finden Sie zusammen mit einem Anwendungsbeispiel in Abschnitt 5.2.4.8. Eine Implementierung des Refactorings für JAVA und das ECLIPSE JDT finden Sie unter <http://www.fernuni-hagen.de/ps/prjs/RIWD>.

5.2 Eine Auswahl von Refactorings

Leider gibt es für längst nicht alle bekannten Refactorings heute schon Implementierungen, die den Ablauf automatisieren. Wie das vorangegangene ausführliche Beispiel klargemacht haben sollte, liegt das nicht am mangelnden Interesse der Programmiererinnen an diesen Refactorings — vielmehr zeigt erst der Versuch, ein Refactoring in einem Werkzeug umzusetzen, auf, was alles berücksichtigt werden muss und worin die eigentlichen Schwierigkeiten dabei liegen. Die meiste Änderungsarbeit muss daher immer noch von Hand gemacht werden. Dennoch haben auch Refactorings ohne Implementierung einen Wert für sich: Sie geben nämlich — in strukturierter Form — praktische Beispiele dafür, wie schlechter Code aussieht, wie guter Code aussieht und wie man vom einen zum anderen gelangt. In diesem Sinne trifft die nachfolgende Vorstellung einzelner Refactorings auch eine Auswahl auf Basis dessen, was eine „gute“ objektorientierte Idiomatik (also eine Art, sich in einem Programm objektorientiert auszudrücken) darstellt.

Ähnlich wie Entwurfsmuster lassen sich auch Refactorings nach ihrem Inhalt klassifizieren. Im Folgenden werden Beispiele aus den folgenden Kategorien besprochen:

1. Bedingungen vereinfachen
2. Lesbarkeit verbessern
3. Daten organisieren
4. Generalisierung einsetzen
5. Methoden organisieren

Die Darstellung orientiert sich dabei an [40].

5.2.1 Bedingungen vereinfachen

Die Kontrolllogik eines Programms verursacht häufig einen großen Teil seiner Komplexität. Dabei muss nach Fred Brooks zwischen der natürlichen Komplexität, die in der Natur des Problems begründet ist, und der künstlichen Komplexität, die durch eine nichtideale Umsetzung entsteht,

unterschieden werden.⁵⁶ Letztere kann reduziert werden, indem man die Bedingungen, die die Kontrolllogik eines Programms ausmachen, vereinfacht.

5.2.1.1 Verschachtelte Bedingungen durch Wächter ersetzen (REPLACE NESTED CONDITIONAL WITH GUARD CLAUSES)

Wer hat das Problem noch nicht selbst erlebt: Eine zunächst einfache Fallunterscheidung muss immer mehr Spezialfälle berücksichtigen, so dass immer neue Else-if-Zweige hinzugefügt und bestehende Bedingungen mit Und-, Oder, und/oder Nicht-Ausdrücken verfeinert werden müssen. Am Ende steht man vor einem unüberschaubaren Wust aus Zweigen, der sich trotz bester Einrückungspraxis nicht mehr nachvollziehen lässt und in dem Fehler zu beheben wie eine Sisyphusarbeit anmutet: Kaum passt die eine Bedingung, stimmt es an einer anderen Stelle nicht mehr. Das folgende Beispiel mag in dieser Hinsicht als harmlos angesehen werden:

```
1089 double getPayAmount() {  
1090     double result;  
1091     if (_isDead) result = deadAmount();  
1092     else {  
1093         if (_isSeparated) result = separatedAmount();  
1094         else {  
1095             if (_isRetired) result = retiredAmount();  
1096             else result = normalPayAmount();  
1097         };  
1098     }  
1099     return result;  
1100 }
```

Eine einfache, aber recht effektive Art, dieses Problem zu lösen, ist, die Else- und Else-if-Teile aufzulösen und stattdessen nur noch Ifs zu verwenden. Deren Bedingungen müssen natürlich komplexer sein, da sie die beim Else implizite Negation des vorangegangenen Ifs wiederholen müssen. Auf der anderen Seite ist so für jeden Zweig die Bedingung, die für dessen Ausführung erfüllt sein muss, unmittelbar dem Zweig zugeordnet (und muss nicht umständlich und fehleranfällig aus dem gesamten Verzweigungskomplex zusammengesucht werden). Da die Bedingung die Anweisungen gewissermaßen (vor der Ausführung) schützt, spricht man auch von einem Guard bzw. von *Guarded commands* (so genannt von Dijkstra). Untereinander hingeschrieben entsprechen die Guarded commands den Einträgen in einer Wahrheitstabelle, wobei im Ergebnisteil natürlich beliebige Anweisungen (Commands) (anstelle von Wahrheitswerten) eingetragen sein können und all die Zeilen, in denen keine Anweisung steht, weggelassen werden. Eine solche Wahrheitstabelle (durch Einfügung von Don't cares weiter vereinfacht) ist die nachfolgende:

⁵⁶ FP Brooks „No silver bullet: essence and accidents of software engineering“ IEEE Computer 20:4 (1987) 10–19.

<code>_isdead</code>	<code>_isseparated</code>	<code>_isretired</code>	Anweisungen
true	don't care	don't care	<code>result = deadAmount()</code>
false	true	don't care	<code>result = separatedAmount()</code>
false	false	true	<code>result = retiredAmount()</code>
false	false	false	<code>result = normalPayAmount()</code>

Diese Tabelle lässt sich in folgenden Code übersetzen:

```

1101 if (_isDead) result = deadAmount();
1102 if (! _isDead && _isSeparated) result = separatedAmount();
1103 if (! _isDead && !_isSeparated && _isRetired) result =
    retiredAmount();
1104 if (! _isDead && !_isSeparated && !_isRetired) result =
    normalPayAmount();

```

Im gegebenen Beispiel lässt sich zusätzlich ausnutzen, dass die Ausführung einer Methode jederzeit durch Rückgabe (Return) abgebrochen werden kann. Zwar lässt sich diskutieren, ob ein Return inmitten einer Methode (oder an mehreren Stellen einer Methode) noch der strukturierten Programmierung entspricht (nach der jedes Konstrukt genau einen Eingang und einen Ausgang haben sollte), man wird aber wohl zustimmen, dass die Lesbarkeit hier keinen Schaden nimmt.

```

1105 double getPayAmount() {
1106     if (_isDead) return deadAmount();
1107     if (_isSeparated) return separatedAmount();
1108     if (_isRetired) return retiredAmount();
1109     return normalPayAmount();
1110 };

```

5.2.1.2 Bedingung zerlegen (DECOMPOSE CONDITIONAL)

Manchmal ist schon *eine* Bedingung so kompliziert, dass die Lesbarkeit darunter leidet. Wenn dann noch dazukommt, dass die bedingten Aktionen nicht selbsterklärend sind, kann dieses Refactoring helfen. Aus

```

1111 if (date.before(SUMMER_START) || date.after(SUMMER_END))
1112     charge = quantity * _winterRate + _winterServiceCharge;
1113 else
1114     charge = quantity * _summerRate;

```

wird dann

```

1115 if (notSummer(date))
1116     charge = winterCharge(quantity);
1117 else

```

```
1118 charge = summerCharge (quantity);

1119 private boolean notSummer(Date date) {
1120     return date.before(SUMMER_START) || date.after(SUMMER_END);
1121 }

1122 private double summerCharge(int quantity) {
1123     return quantity * summerRate;
1124 }

1125 private double winterCharge(int quantity) {
1126     return quantity * _winterRate + _winterServiceCharge;
1127 }
```

Die Namen der eingefügten Methoden haben den Charakter ausführbarer Kommentare. Das Refactoring löst damit auf einfache Weise das Problem, dass man bei eingefügten Kommentaren oft nicht so genau weiß, auf welchen Teil des Quellcodes sie sich beziehen, ein Problem, das man nur durch längliche Formulierungen oder Wiederholung von Teilen des Quellcodes im Kommentar lösen kann.

Man könnte argumentieren, dass die bessere Lesbarkeit teuer, nämlich um den Preis der verlangsamten Ausführung erkauft wird. Ein geschickter Compiler wird jedoch versuchen, die Methodenaufrufe zu inlinen (also an Ort und Stelle — in der Zeile — einzufügen), so dass keine Verschlechterung des Laufzeitverhaltens entsteht. Außerdem hilft die durchgeführte Fragmentierung, doppelten Code — der von manchen als die Wurzel allen Übels angesehen wird — zu vermeiden: Wenn an anderer Stelle dieselbe Bedingung oder Aktion noch einmal benötigt werden sollte, kann man direkt darauf zurückgreifen. Ist das jedoch nicht der Fall, wird die Klasse allerdings mit Methoden überfrachtet, deren Nutzen außerhalb des Kontextes der Bedingung nicht ersichtlich ist, was die Lesbarkeit in gewisser Weise verschlechtert. Eine Lösung könnte hier die Möglichkeit der Definition von lokalen Methoden (Methoden, die innerhalb von Methoden deklariert sind) bieten; diese besteht in JAVA aber leider nicht.

Dieses Refactoring ist übrigens eine konkrete Anwendung des Methode-extrahieren-Refactorings, das in Abschnitt 5.2.5.1 behandelt wird.

5.2.1.3 Bedingung durch Polymorphismus ersetzen (REPLACE CONDITIONAL WITH POLYMORPHISM)

Nicht selten hängt eine Fallunterscheidung am Typ eines Objekts. Im folgenden (nicht ganz ernstgemeinten) Beispiel (übernommen aus [40]) ist das der Fall:

```
1128 double getSpeed() {
1129     switch (_type) {
1130         case EUROPEAN:
1131             return getBaseSpeed();
1132         case AFRICAN:
```

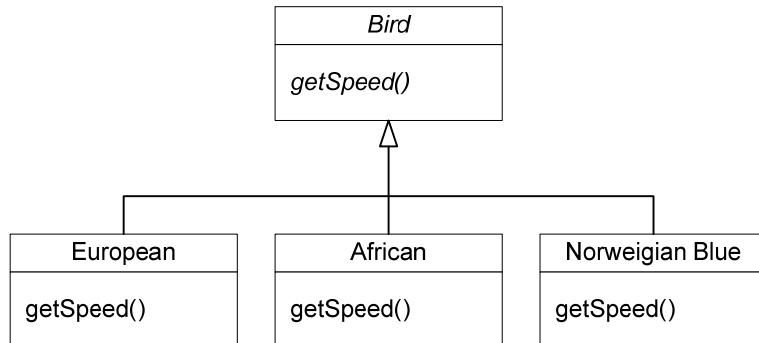
```

1133     return getBaseSpeed() - getLoadFactor() *
1134         _numberOfCoconuts;
1135     case NORWEGIAN_BLUE:
1136         return (_isNailed) ? 0 : getBaseSpeed(_voltage);
1137     }
1138     throw new RuntimeException ("Should be unreachable");
1139 }

```

Die Variable `_type` soll hier offensichtlich nur einen von drei verschiedenen Werten annehmen können: `EUROPEAN`, `AFRICAN` oder `NORWEIGIAN_BLUE`. In Abhängigkeit vom jeweiligen Wert, der wohl eine Vogelart repräsentiert, wird der Wert für eine Variable `speed` berechnet und zurückgegeben. Da es sich bei der Variable `_type` um ein Attribut (Feld) des Objektes handelt, das die Methode `getSpeed()` implementiert (dessen Speed also berechnet werden soll), ist davon auszugehen, dass es sich bei dem Objekt um einen Vogel handelt, der eben von einem der drei genannten Arten (Typen) sein kann.

In der objektorientierten Programmierung würde man genau diesen Umstand, nämlich dass es drei verschiedene Arten von Vögeln gibt, dadurch ausdrücken, dass man eine abstrakte Klasse `Bird` vorsieht und von ihr drei konkrete Klassen, `European`, `African` und `NorweigianBlue`, ableitet, wie im nachfolgenden UML-Diagramm dargestellt. Die abstrakte Methode `getSpeed()`, die in der gemeinsamen Superklasse deklariert ist, wird dann in den drei Unterklassen konkret überschrieben. Die explizite Verzweigung der Switch-Anweisung wird damit durch die implizite Verzweigung des *dynamischen Bindens* (der technischen Umsetzung der Polymorphie) ersetzt.



Refaktoriert sieht die obige Methode dann so aus:

```

1140 abstract class Bird {
1141     abstract double getSpeed();
1142 }
1143 class European extends Bird {
1144     double getSpeed() {
1145         return getBaseSpeed();
1146     }

```

```
1147 }

1148 class African extends Bird {
1149     double getSpeed() {
1150         return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
1151     }
1152 }

1153 class NorwegianBlue extends Bird {
1154     double getSpeed() {
1155         return (_isNailed) ? 0 : getBaseSpeed(_voltage);
1156     }
1157 }
```

Die refaktorierte Version hat gleich mehrere Vorteile:

1. Mit der Anzahl der zu unterscheidenden Fälle (Typen) steigt die Länge der Switch-Anweisungen. Einzelne Methoden können dadurch sehr lang werden, was in der objektorientierten Programmierung einigermaßen verpönt ist (vgl. Kurs 01814). Bei der polymorphismusbasierten Lösung wird hingegen jeder Fall einzeln (in einer anderen Klasse) abgehandelt und die Methoden bleiben entsprechend kurz.
2. Häufig bleibt es nicht bei *einer* Fallunterscheidung nach dem Typ: Switch-Anweisungen der obigen Art, die große Redundanzen enthalten, durchziehen dann den Code. Dies ist insbesondere dann ein Problem, wenn sich die Zahl der zu unterscheidenden Arten (Typen) irgendwann einmal ändert: Dann müssen alle Switch-Anweisungen nachgepflegt werden — wehe der, die eine vergisst! Bei der polymorphismusbasierten Lösung hingegen wird beim Einfügen eines neuen Typs (repräsentiert durch eine neue konkrete Klasse) vom Compiler erzwungen, dass alle in der Superklasse abstrakt deklarierten Methoden auch implementiert werden und die Fallunterscheidung damit stets vollständig getroffen wird.
3. Nicht zuletzt ergibt sich aus dem Polymorphismusansatz eine bessere Erweiterbarkeit: Während bei einer Erweiterung des Aufzählungstyps und entsprechend der Switch-Anweisungen bereits bestehender Quellcode (auf den man unter Umständen gar keinen oder nur eingeschränkten Zugriff hat) geändert werden muss, kann man im anderen Fall einfach eine neue Klasse hinzufügen — im günstigsten Fall muss an bereits bestehendem Code überhaupt nichts geändert werden.

Selbsttestaufgabe 5.2

Unter welchen Voraussetzungen muss am Code nichts geändert werden?

Dem gegenüber steht aber auch ein durchaus gravierender Nachteil:

- Da die einzelnen Fälle und damit die Fallunterscheidung als Ganzes auf mehrere Klassen verteilt ist, ist es schwierig, sich einen Überblick über alle Alternativen (die ja gegebenenfalls noch nicht einmal vollständig bekannt sind; siehe Punkt 3 oben) zu verschaffen: Man muss schon alle Klassen nebeneinander legen, um zu sehen, worin sich die verschiedenen

Fälle unterscheiden. Damit einhergehend (und nicht selten als Hauptnachteil der objekt-orientierten Programmierung bezeichnet) ist der Umstand, dass das Programm bei der schrittweisen Ausführung (beim Debuggen) wild hin und her springt und man schnell den Überblick darüber verliert, wo man gerade ist (und wie man dort hingekommen ist; vgl. Kurs 01814).

Zusammenfassung

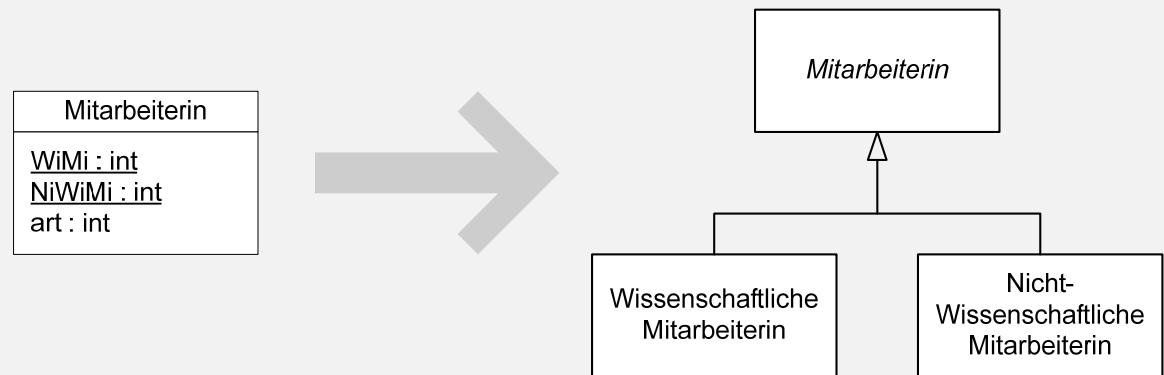
Es ergibt sich also, dass die Fallunterscheidungen jeweils anders gruppiert werden: Bei einer (expliziten) Verzweigung stehen alle Fälle an einer Stelle, aber mehrere Fallunterscheidungen mit den gleichen Alternativen über den Code verstreut; bei der Polymorphie stehen die Fälle an verschiedenen Stellen (Klassen), aber die sonst verstreuten Fallunterscheidungen stehen (nach Alternativen geordnet) zusammen (in Klassen, die die Alternativen repräsentieren).

konsequente Anwendung in SMALLTALK

Das Prinzip, den Polymorphismus an die Stelle der Bedingung bzw. der bedingten Verzweigung zu setzen, wurde übrigens in der Programmiersprache SMALLTALK auf die Spitze getrieben: Da dort alles, also auch die Wahrheitswerte `true` und `false`, ein Objekt ist, kann man auf die If-Anweisung ganz verzichten, einfach indem man eine abstrakte Klasse `Boolean` mit zwei konkreten Subklassen `True` und `False` vor sieht, die jeweils verschiedene Implementierungen für die (in `Boolean` abstrakt deklarierten) Methoden `ifTrue` und `ifFalse` vorsehen (wobei der vom Ergebnis abhängig auszuführende Code als Parameter an die jeweilige Methode übergeben werden muss).

Beispiel

Anmerkung: Stark mit diesem Refactoring verwandt ist „Typcode durch Subtypen ersetzen“ (REPLACE TYPE CODE WITH SUBCLASSES):



5.2.1.4 Einführung eines Nullobjekts (INTRODUCE NULL OBJECT)

Gewissermaßen ein Spezialfall des vorgenannten Refactorings stellt die Einführung eines Nullobjekts dar. Häufig ist es zulässig (entspricht es der Realität), dass ein Objekt als Attributwert anstelle eines anderen eben auch kein Objekt haben kann. So kann es beispielsweise sein, dass jemand nicht verheiratet ist (und damit eben keine Ehepartnerin hat), dass eine Kundin kein Konto hat etc. In solchen Fällen muss vor einem Zugriff auf das Attributobjekt geprüft werden,

ob es überhaupt vorhanden ist oder ob stattdessen ein Nullwert vorliegt. Der Nullwert wäre dann entsprechend anders zu behandeln.

Das nachfolgende Codefragment zeigt ein Beispiel aus einem gedachten Kursbetreuungssystem:

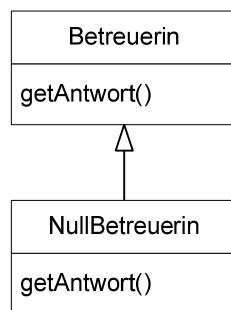
```

1158 Betreuerin betreuerin = kurs.betreuerin;
1159 if (betreuerin == null)
1160     antwort = "leider keine da, die eine Antwort wüßte";
1161 else
1162     antwort = betreuerin.getAntwort();

```

Wesentlich an dem Beispiel ist, dass für den Fall des Vorliegens eines Nullwertes eine wirkliche Alternative vorgesehen ist und nicht nur ein für den Normalfall gedachter Codeblock übersprungen wird.

Diese Fallunterscheidung lässt sich nun eliminieren (korrekter: durch Polymorphismus ersetzen), indem man eine spezielle Klasse **NullBetreuerin** einführt, die angibt, was zu tun ist, wenn zu einem Kurs keine Betreuerin (die Nullbetreuerin) vorliegt. Das dazugehörige Klassendiagramm kann so aussehen:



Anmerkung: Es könnte aber auch eine abstrakte Klasse **Betreuerin** mit zwei oder mehr konkreten Subklassen, darunter **EchteBetreuerin** und **NullBetreuerin** vorgesehen werden. Da im gegebenen Fall die Klasse **NullBetreuerin** ein Singleton ist (nur genau eine Instanz hat; s. Hinweis unten), ist dies vielleicht etwas übertrieben.

Das obige Codefragment würde damit auf das folgende zusammenschrumpfen:

```

1163 Betreuerin betreuerin = kurs.betreuerin;
1164 antwort = betreuerin.getAntwort();

```

Dafür wäre dann noch das folgende zu implementieren:

```

1165 class NullBetreuerin extends Betreuerin {
1166     String getAntwort() {
1167         return "leider keine da, die eine Antwort wüßte";
1168     }
1169 }

```

Diese Klasse würde dann ggf. noch um zusätzliche Methoden zur Behandlung von weiteren Spezialfällen bei nicht vorhandenen Betreuerinnen ergänzt. Im Gegensatz zu den möglichen Problemen des allgemeineren Refactorings oben (Bedingung durch Polymorphismus ersetzen) hat dies nicht den Nachteil, dass der Code weit verstreut wird: es gibt nur den einen Sonderfall (den Nullwert) und die Behandlung dessen wird an einer Stelle konzentriert.

Hinweis: Da es im Allgemeinen nicht sinnvoll sein wird, mehrere Instanzen einer solchen Nullklasse zuzulassen, wird man diese in Form des *SINGLETON Pattern* realisieren (siehe [42]). Da **NullBetreuerin** zudem eng an **Betreuerin** gebunden ist (und außerhalb des Kontexts von **Betreuerin** wohl kaum Verwendung finden wird), kann man sich überlegen, ob man erstere nicht als innere Klasse implementieren will. Die äußere Klasse (im gegebenen Fall **Betreuerin**) würde dann nur noch eine Konstante (das Nullobject) öffentlich machen.

5.2.1.5 Zusicherung einfügen (INTRODUCE ASSERTION)

Das Design by contract verlangt, dass jede Methode die Voraussetzungen für ihr Gelingen mittels Vorbedingung explizit macht. Das sog. *defensive Programmieren* verlangt dagegen, dass mögliche Zustände (Eingaben etc.), mit denen eine Methode nicht zurechtkommen würde, in dieser Methode geprüft werden und bei Vorliegen entsprechend (fehlertolerant) reagiert wird⁵⁷. In der Praxis wird jedoch häufig weder das eine noch das andere gemacht.

Wann immer man findet, dass eine bestimmte, im Code implizite Bedingung Voraussetzung für die fehlerfreie Ausführung des Codes (einer Codesequenz) ist, sollte man diese Bedingung explizit machen, wenn schon nicht durch Vorbedingungen à la Design by contract oder Guards, dann durch das Einstreuen von allgemeinen Zusicherungen (Asserts). Der nachfolgende Code, der nur funktioniert, wenn ein Angestellter eine Spesenbeschränkung oder das Projekt, dem er primär zugeordnet ist, eine solche hat, enthält eine solche implizite Bedingung (im Kommentar ausgedrückt):

```

1170 class Employee {
1171     private static final double NULL_EXPENSE = -1.0;
1172     private double _expenseLimit = NULL_EXPENSE;
1173     private Project _primaryProject;
1174
1175     double getExpenseLimit() {
1176         // should have either expense limit or primary project
1177         return (_expenseLimit != NULL_EXPENSE) ?
1178             _expenseLimit :
1179             _primaryProject.getMemberExpenseLimit();
1180     }

```

⁵⁷ was jedoch nicht immer möglich ist; s. Abschnitt 2.4 Design by contract in der Programmierung

Er wird zu

```
1181 double getExpenseLimit() {  
1182     assert (_expenseLimit != NULL_EXPENSE  
1183             || _primaryProject != null);  
1184     return (_expenseLimit != NULL_EXPENSE) ?  
1185         _expenseLimit :  
1186         _primaryProject.getMemberExpenseLimit();  
1187 }
```

5.2.2 Lesbarkeit verbessern

5.2.2.1 Methode oder Variable umbenennen (RENAME)

Es ist leider immer noch ein weit verbreitetes Übel, dass Programmiererinnen für die Bezeichner in ihren Programmen Abkürzungen wählen, die außer ihnen selbst kaum jemandem (und manchmal nach gewisser Zeit selbst ihnen nicht mehr) zugänglich sind. Dies mag hier und da gute Gründe haben (ausführliche Namen werden unter Umständen zu lang und führen deswegen zu unerwünschten Zeilenumbrüchen, bestimmte Abkürzungen haben sich eingebürgert), aber die Lesbarkeit des Programms sollte eigentlich für gar nichts geopfert werden. Überhaupt nicht hinzunehmen sind Argumente wie „Ja glauben Sie denn, ich tippe mir die Finger wund?“: Heutige Entwicklungsumgebungen nehmen einem durch die Funktion der automatischen Vervollständigung (sog. Code completion) die meiste Schreibarbeit sowieso schon ab, und bei der Einführung neuer Bezeichner wird man einem gebildeten Mitmenschen noch abverlangen können, sich allgemeinverständlich auszudrücken. Bei der Verwendung kryptischer Abkürzungen hingegen kann niemand mehr durch bloßes Hinsehen entscheiden, ob sich die Programmiererin vielleicht im Bezeichner geirrt hat und aus Versehen die falsche Methode aufruft oder das falsche Feld referenziert. Wann immer also jemand einen unverständlichen oder irreführenden Bezeichner in einem Programm findet, sollte sie ihn schleunigst ändern.

Eine Methode sollte immer danach benannt werden, *was* sie tut, nicht danach, *wie* sie es tut. Wie sie es tut ist nur von Interesse, wenn man etwas daran ändern will. Dazu müssen Sie aber an die Definitionsstelle der Methode gehen, sich die Implementierung ansehen und diese verstehen. Wenn Sie die Implementierung verstanden haben, wissen Sie auch, wie die Methode etwas tut, und zwar viel genauer, als das irgendein Name sagen könnte. An der Aufrufstelle einer Methode ist hingegen nur von Interesse, *was* sie tut; so bleibt der Fokus in der Umgebung der Aufrufstelle, die die Implementierung der *aufrufenden* Methode darstellt, auf der Erklärung von deren *Wie*.

Leider ist das mit dem Ändern von Bezeichnern nicht ganz so einfach. Ändert man nämlich den Namen einer Methode nur lokal, so können die Aufrufe der Methode an anderen Stellen nicht mehr gebunden werden. Außerdem muss geprüft werden, ob diese Methode eine andere, geerbte überschreibt, und ob sie ggf. (in Unterklassen) überschrieben wird. Schließlich muss überprüft werden, ob es nicht schon eine Methode mit gleichem Namen gibt; wenn dies überhaupt

erlaubt ist, kann die so neu entstehende Überladung zu Problemen führen, die vorher nicht da waren⁵⁸. All diese Feinheiten sind auch der Grund dafür, warum man tunlichst davon Abstand nehmen sollte, den Namen einer Methode per globalem Suchen und Ersetzen zu ändern. Das würde nämlich schon daran scheitern, dass einfaches Suchen und Ersetzen zufällige von beabsichtigter Namensgleichheit nicht unterscheiden kann.

Glücklicherweise besitzen moderne Entwicklungsumgebungen bereits ein Refactoring, das die Umbenennung automatisch vornimmt.

5.2.2.2 Parameterklasse einführen (INTRODUCE PARAMETER OBJECT)

Wenn eine Methode mit vielen Parametern aufgerufen werden muss, die inhaltlich stark zusammenhängen, oder wenn mehrere Methoden mit genau den gleichen Parametern aufgerufen werden, kann es sinnvoll sein, diese Parameter in einem Record zusammenzuführen. In der objektorientierten Programmierung wird man dafür eine Klasse nehmen, die keine eigenen Methoden hat, da ihr einziger Zweck ist, die Parameter zu gruppieren.

5.2.2.3 Konstruktor durch eine Factory-Methode ersetzen (REPLACE CONSTRUCTOR WITH FACTORY METHOD)

In SMALLTALK sind Konstruktoren Klassenmethoden (also Methoden, die in JAVA oder C# als `static` deklariert würden), die eine neue Instanz der Klasse zurückgeben (letztendlich realisiert von der Klassenmethode `new` in der Klasse `Object`). In JAVA und C# hingegen sind Konstruktoren besondere Methoden, deren Name und Rückgabetyp durch die Klasse, in der sie definiert werden, festgelegt ist. Nicht selten kommt es jedoch vor, dass der Typ einer zurückgegebenen Instanz von anderen Faktoren abhängt, etwa den Parametern des Konstruktoraufrufs, oder dass vielleicht gar kein neues Objekt zurückgegeben werden soll, sondern vielmehr eines, das bereits existiert (das *SINGLETON Pattern* [42]). Konstruktoren sind dafür nicht flexibel genug.

Allgemein wird es gelegentlich als guter Stil erachtet, anstelle von Konstruktoren statische Methoden zu verwenden, die eine neue Instanz zurückgeben. Da diese Methoden wesentlich flexibler sind, was das zurückgegebene Objekt (insbesondere seinen Typ) angeht, sie insbesondere das Objekt nach den Vorgaben der Parameter zusammenbauen können, nennt man sie auch Factory-Methoden (vgl. Abschnitt 4.4.6). Natürlich können diese Factory-Methoden ein neues Objekt nicht selbst erzeugen; sie müssen dazu einen Konstruktor der Klasse aufrufen. Es reicht dafür jedoch der Standardkonstruktor. So wird beispielsweise aus dem Konstruktor

```
1188 Employee (int type) {
1189     _type = type;
1190 }
```

⁵⁸ z. B. einen Method-ambiguous-Error

die Factory-Methode

```
1191 static Employee create(int type) {  
1192     return new Employee(type);  
1193 }
```

die den ursprünglichen Konstruktor oder, besser noch,

```
1194 static Employee create(int type) {  
1195     Employee employee = new Employee();  
1196     employee._type = type;  
1197     return employee;  
1198 }
```

die den Standardkonstruktor aufruft. Der Standardkonstruktor kann dann **protected** deklariert werden, um die Umleitung aller Instanzerzeugungen auf die Factory-Methode zu erzwingen.

Richtig interessant wird die Einführung einer Factory-Methode allerdings erst dann, wenn die Parameter ausgewertet und davon abhängig Objekte verschiedener Typen zurückgegeben werden, wie etwa in

```
1199 static Employee create(int type) {  
1200     switch (type) {  
1201         case 1: return new UnskilledEmployee();  
1202         case 2: return new Worker();  
1203         case 3: return new Manager();  
1204     }  
1205 }
```

Dieser Code lässt sich noch weiter verkürzen, indem man die *Reflection*-Fähigkeit von JAVA ausnutzt (vgl. Abschnitt 6.1):

```
1206 static Employee create (String name) {  
1207     try {  
1208         return (Employee) Class.forName(name).newInstance();  
1209     }  
1210     catch (Exception e) {  
1211         throw new IllegalArgumentException("Unable to instantiate");  
1212     }  
1213 }
```

5.2.2.4 Fehlercode durch Ausnahme ersetzen (REPLACE ERROR CODE WITH EXCEPTION)

Viele gängige Bibliotheks Routinen (wie auch Betriebssystemfunktionen) verwenden sog. Fehlercodes, um anzugeben, dass mit der Ausführung etwas nicht geklappt hat. Diese Fehlercodes

können gemischt mit den richtigen Rückgabewerten einer Methode auftreten (sie stellen also quasi eine Erweiterung des Wertebereichs einer Funktion dar), oder sie verbannen den eigentlichen Rückgabewert einer Methode in die Liste der Parameter, wobei dann ein spezieller Rückgabewert (oder Fehlercode) anzeigt, dass keine Fehler aufgetreten sind.

Im Beispiel eines Kontos (**Account**) und der Methode Abheben (**withdraw**) sieht das so aus:

```

1214 class Account {
1215     private int _balance;
1216
1217     int withdraw(int amount) {
1218         if (amount > _balance)
1219             return -1;
1220         else {
1221             _balance -= amount;
1222             return 0;
1223         }
1224     ...

```

An der Aufrufstelle findet man dann solchen Code:

```

1224     if (account.withdraw(amount) == -1)
1225         handleOverdrawn();
1226     else
1227         doTheUsualThing();

```

Diese Praxis, so etabliert sie auch sein mag, hat verheerende Auswirkungen auf die Lesbarkeit von Programmen. Das größte Problem dabei ist, dass der Code, der den regulären Ablauf einer Routine beschreibt, mit Code zur Fehlerbehandlung durchsetzt wird, ohne dass das eine vom anderen klar getrennt wäre. Das zweite Problem ist, dass Fehler, die in einer geschachtelten Struktur irgendwo tief unten auftreten, unter Umständen zum Abbruch vieler äußerer Blöcke führen, so dass große Codestrecken, die unmittelbar gar nichts mit dem auftretenden Fehler zu tun haben, mit Bedingungen versehen (geschützt; *Guarded commands!*) werden müssen, um eine Ausführung nach Auftreten des Fehlers zu verhindern bzw. um alternative Maßnahmen vorzusehen.

Um genau dies zu verhindern, also um den Code zur regulären Ausführung von dem zur Fehlerbehandlung syntaktisch klar zu trennen und um sich die Durchsetzung von Code mit Verzweigungen zur Berücksichtigung von Fehlercodes (in der Regel zum Abbruch eines Blocks) zu ersparen, wurde das Konzept der Exceptions und des Exception handlings eingeführt.

Im einfachsten Fall wird aus obiger Methode **withdraw** mit einem Fehlercode als Rückgabe eine Methode ohne Rückgabewert, dafür mit der Möglichkeit, eine Exception zu werfen:

```

1228     void withdraw(int amount) throws BalanceException {

```

```
1229 if (amount > _balance)
1230     throw new BalanceException();
1231     _balance -= amount;
1232 }
```

Verzweigungen auf der Basis von If-Anweisungen, die den Fehlercode auswerten, entfallen damit vollständig, sie werden durch Catch-Klauseln oder durch weitere Throws-Deklarationen im Kopf der aufrufenden Methoden ersetzt. Der resultierende Code an der Aufrufstelle ist also entweder

```
1233 try {
1234     account.withdraw(amount);
1235     doTheUsualThing();
1236 } catch (BalanceException e) {
1237     handleOverdrawn();
1238 }
```

oder

```
1239 void someMethod() throws BalanceException {
1240     account.withdraw(amount);
1241     doTheUsualThing();
1242 }
```

In gewisser Weise erweitert die Deklaration, (möglicherweise) Exceptions zu werfen, die Menge der möglichen Rückgabewerte einer Methode. Folgerichtig müsste die Menge der deklarierten Exceptions einer Methode zu ihrer Signatur zählen, wenn in JAVA nur der Rückgabetyp auch zur Signatur gehörte.

Das Exception handling stellt einen der größten Fortschritte in der Programmierung seit der Einführung der strukturierten Programmierung dar. Auch wenn es manchmal lästig erscheint (psychologisch schon allein deswegen, weil es ja nur die Fälle behandelt, die man eigentlich gar nicht haben möchte), sollte man es sich zur Gewohnheit machen, beim Schreiben einer Methode auch an die möglichen Ausnahmen zu denken und diese entsprechend zu deklarieren. Im schlimmsten Fall, also wenn man sich wirklich nicht um die Ausnahme kümmern möchte oder keine Ahnung hat, wie man dies sinnvoll tun könnte, kann man im Exception handler dann immer noch eine Fehlermeldung im Klartext ausgeben und somit nachfolgenden Programmiererinnen Hinweise geben, wo der Fehler aufgetreten ist und dass man ihn vielleicht noch abstellen müsste. (Siehe in diesem Zusammenhang auch das nachfolgende Refactoring „Ausnahme durch Vorbedingung ersetzen“).

Man sieht gelegentlich, dass das Exception handling missbraucht wird, um Blöcke abzubrechen, obwohl das verwendete Abbruchkriterium ein regelmäßiges ist, also nichts mit einer Ausnahme zu tun hat. So kommt es beispielsweise vor, dass Programmiererinnen über die Elemente eines Arrays iterieren, ohne ein Abbruchkriterium für das Erreichen der oberen Array-Grenze vorzuse-

hen, und statt dessen die Index-out-of-bounds-Exception ausnutzen, um die Schleife zu beenden:

```
1243 try {
1244     for (i=0;; i++)
1245         System.out.println(line[i]);
1246 } catch (IndexOutOfBoundsException e) {}
```

Dies ist natürlich ganz schlechter Stil und sollte sich jeder von selbst verbieten. Man trifft jedoch auch auf weniger offensichtliche Fälle; hierfür ist dann das übernächste Refactoring „Ausnahme durch Test ersetzen“ gedacht.

5.2.2.5 Ausnahme durch Vorbedingung ersetzen (REPLACE EXCEPTION WITH PRECONDITION)

Nicht immer ist die Signalisierung einer Ausnahme gerechtfertigt. So kann man beispielsweise argumentieren, dass der Versuch, sein Bankkonto zu überziehen, regulären Charakter hat und das Werfen einer Ausnahme einen Missbrauch des Konzepts darstellt (wenn auch nicht ganz so krass wie in den Programmzeilen 1243 –1246 in Abschnitt 5.2.2.4). Ähnlich zum obigen Beispiel ersetze man dann in

```
1247 class Account {
1248     private int _balance;
1249
1250     int withdraw(int amount) {
1251         if (amount > _balance)
1252             return -1;
1253         else {
1254             _balance -= amount;
1255             return 0;
1256         }
1257     ...
1258 }
```

die Methode `withdraw()` durch

```
1257 void withdraw(int amount) {
1258     assert canWithdraw(amount) : "Amount too large";
1259     _balance -= amount;
1260 }
1261
1262 boolean canWithdraw(amount) {
1263     return amount > _balance;
1264 }
```

An der Aufrufstelle muss dann die Einhaltung der Vorbedingung sichergestellt werden:

```
1264 if (!account.canWithdraw(amount))  
1265     handleOverdrawn();  
1266 else {  
1267     account.withdraw(amount);  
1268     doTheUsualThing();  
1269 }
```

Die Aufrufstelle unterscheidet sich damit nicht übermäßig von der mit Fehlercode (Programmzeilen 1224 – 1227 in Abschnitt 5.2.2.4); zwar ist die Überprüfung auf Fehler (`!account.canwithdraw(amount)`) von der Ausführung der eigentlichen Aktion getrennt (`account.withdraw(amount)`), aber die Fehlerbehandlung steht direkt neben der Behandlung des Regelfalls, ist insbesondere nicht syntaktisch (durch einen Catch-Block) hervorgehoben. Dies entspricht im gegebenen Fall aber gerade der Absicht (es sollte sich bei der Überziehung ja auch um einen regulären Fall handeln).

5.2.2.6 Ausnahme durch Test ersetzen (REPLACE EXCEPTION WITH TEST)

Im nachfolgenden Fall ist die Provokation einer Exception und die Verwendung eines Catch-Blocks missbräuchlich:

```
1270 class ResourcePool {  
1271     Stack _available;  
1272     Stack _allocated;  
  
1273     Resource getResource() {  
1274         Resource result;  
1275         try {  
1276             result = (Resource) _available.pop();  
1277             _allocated.push(result);  
1278             return result;`  
1279         } catch (EmptyStackException e) {  
1280             result = new Resource();  
1281             _allocated.push(result);  
1282             return result;  
1283         }  
1284     }
```

Dass die Ressourcen irgendwann erschöpft sind (der Stack `_available` irgendwann leer ist), ist ganz klar zu erwarten. Die refaktorierte Variante der Methode sieht so aus:

```
1285     Resource getResource() {  
1286         Resource result;  
1287         if (_available.isEmpty())  
1288             result = new Resource();  
1289         else
```

```

1290     result = (Resource) _available.pop();
1291     _allocated.push(result);
1292     return result;
1293 }
```

Man beachte, dass `_allocated.push(result)` aus den beiden alternativen Pfaden herausfaktorisiert wurde (ein weiteres Refactoring).

5.2.3 Daten organisieren

5.2.3.1 Feld kapseln (ENCAPSULATE FIELD)

Während in SMALLTALK Instanzvariablen (Felder) grundsätzlich nur über Methoden (also nicht direkt) gelesen und geschrieben werden können, ist es in vielen anderen Sprachen möglich, direkt darauf zuzugreifen. Dies erspart der Programmiererin einiges an Schreibarbeit, da man keine Zugriffsmethoden (Getter und Setter, sog. Accessoren) braucht. Auf der anderen Seite wird so auch eine wichtige Entwurfsentscheidung, nämlich ein (logisches) Attribut als (physische) Instanzvariable zu repräsentieren, nach außen getragen: Wenn man sich später entschließen sollte, den Attributwert nicht direkt zu speichern, sondern zu berechnen, muss man nachträglich solche Zugriffsmethoden einführen, was potentiell Änderungen im gesamten Code nach sich zieht. Das Argument, direkter Feldzugriff sei effizienter, weil man sich den Methodenaufruf erspare, sollte vernachlässigt werden; die Optimierung sollte Aufgabe des Compilers sein (wobei bei standardmäßig dynamischem Binden von Methodenaufrufen hierzu Laufzeittechniken wie Just-in-time-Compilierung notwendig sind).

Ein weiteres Problem des direkten Feldzugriffs bereitet der Umstand, dass bei den Zugriffsrechten nicht zwischen lesendem und schreibendem Zugriff unterschieden werden kann: Ein Feld ist immer entweder für beides zugreifbar (`public` deklariert) oder eben nicht zugreifbar (`private` deklariert). Bei der Verwendung von Accessoren hingegen kann, durch die Bereitstellung entsprechender Interface-Typen, zwischen nur lesenden und nur schreibenden Zugriffen auf ein Feld unterschieden werden (s. Beispiel in Abschnitt 1.5.6).

So wird denn, durch ein entsprechendes Refactoring, aus

```
1294 public String _name;
```

einfach

```

1295 private String _name;
1296 public String getName() {
1297     return _name;
1298 }
```

```
1299 public void setName(String arg) {  
1300     _name = arg;  
1301 }
```

Anmerkung: Spätestens, wenn man den Namen oder den Typ des Feldes ändern möchte, wünscht man sich hier das nächste Refactoring herbei.

Wie man leicht sieht, ist es nun möglich, den Attributzugriff mit Nebeneffekten zu versehen oder auch das Attribut ganz einzusparen, indem man den Attributwert berechnet bzw., beim schreibenden Zugriff, die Berechnung umkehrt und den Ausgangswert entsprechend ändert. Ein typisches Beispiel hierfür ist die Berechnung von radialen Koordinaten aus kartesischen, wie im nachfolgenden Beispiel aufgeführt.

```
1302 public class Coordinate {  
1303     private double x, y;  
  
1304     public void setCartesian(double x, double y) {  
1305         this.x = x;  
1306         this.y = y;  
1307     }  
  
1308     public double getX () {  
1309         return x;  
1310     }  
  
1311     public double getY () {  
1312         return y;  
1313     }  
  
1314     public void setPolar(double radius, double angle) {  
1315         x = Math.cos(angle) * radius;  
1316         y = Math.sin(angle) * radius;  
1317     }  
  
1318     public double getRadius() {  
1319         return Math.sqrt(x*x + y*y);  
1320     }  
  
1321     public double getAngle () {  
1322         return Math.acos(x/getRadius());  
1323     }  
1324 }
```

Nachtrag: Was den Schreibaufwand angeht, so bieten die meisten modernen Entwicklungsumgebungen Funktionen, die einem für ein **privat** deklariertes Feld automatisch die Getter und Setter erzeugen. Was bleibt ist massenhaft immer gleicher Quellcode, den man nicht unbedingt

sehen müssen will; doch auch hier bietet beispielsweise ECLIPSE die Möglichkeit zur Ausblendung. Es bleibt dann allerdings immer noch der vermehrte Schreibaufwand beim Feldzugriff selbst, wo jetzt anstelle des Feldes ein Methodenaufruf stehen muss. Programmiersprachen wie EIFFEL, DELPHI oder C# haben daher die Feld-Accessoren-Kombination durch sog. Properties ersetzt, deren Zugriff zwar durch Methoden ähnlich den Accessoren erfolgt, sich syntaktisch aber nicht vom Zugriff auf ein Feld unterscheidet. Diese Transparenz kann jedoch auch von Nachteil sein, nämlich wenn dem Benutzer nicht klar ist, dass sich hinter einem scheinbaren Feldzugriff ein Methodenaufruf (möglicherweise mit weiteren Nebeneffekten) verbirgt.

Anmerkung: Accessoren zur Kapselung von Feldern können selbst dann sinnvoll sein, wenn die Felder nur aus der eigenen Klasse heraus aufgerufen werden, die Accessoren selbst also privat deklariert sind. Das entsprechende Refactoring wird SELF ENCAPSULATE FIELD genannt (und hier nicht behandelt).

5.2.3.2 Collections kapseln (ENCAPSULATE COLLECTION)

Zwischen UML-Klassendiagrammen und objektorientiertem Code gibt es einen entscheidenden Unterschied: Während es im UML-Diagramm keinen großen Unterschied macht, ob ein Objekt mit *einem* oder mit *mehreren* anderen (gleichen Typs) über ein Attribut oder eine Assoziation in Verbindung steht, ist diese Unterscheidung im objektorientierten Programm substantieller Natur. Das Feld, das das Attribut oder die Assoziation repräsentiert, enthält im ersten Fall nämlich das Objekt direkt, im zweiten Fall jedoch nur ein sog. Collection-Objekt. Dieses Collection-Objekt enthält dann erst die eigentlichen Werte des Attributs bzw. der Assoziation. Dieser übliche Workaround enthält einen subtilen logischen Fehler: Anstatt eine 1:n-Relation abzubilden, stellt er in Wirklichkeit eine 1:1-Beziehung dar, wobei das assozierte Objekt (die rechte 1) eine Menge ist, also ein Objekt, das unter Umständen mehrere andere enthält. Diese Unterscheidung, die sich auf UML-Seite lediglich in der Angabe einer 1 bzw. eines Sterns (*) niederschlägt, ist ein Ärgernis für die gesamte objektorientierte Programmierung, und man kann den Reifegrad einer objektorientierten Programmiersprache daran erkennen, wie gut sie 1:n-Beziehungen unterstützt.

Wenn man das Prinzip der Zugriffsmethoden auf mengenwertige Felder direkt anwendet, erhält man entsprechend Zugriff auf die Collection-Objekte. Diese Objekte sollten konzeptuell (und damit idealerweise auch für die Programmiererin) jedoch gar nicht in Erscheinung treten. Man sollte also auch diese vor dem direkten Zugriff kapseln. Im folgenden Beispiel sollte also etwa

```

1325 class Person {
1326     private List kinder;
1327
1328     List getKinder() {
1329         return kinder;
1330     }
1330     void setKinder(List kinder) {
```

```
1331     this.kinder = kinder;  
1332 }  
1333 }
```

durch

```
1334 class Person {  
1335     private List kinder = new ArrayList();  
  
1336     void addKind(Person kind) {  
1337         kinder.add(kind);  
1338     }  
  
1339     void removeKind(Person kind) {  
1340         kinder.remove(kind);  
1341     }  
1342 }
```

ersetzt werden. Das Hinzufügen und Löschen von Kindern kann dann nicht mehr von extern direkt auf der Collection durchgeführt werden. Wenn trotzdem immer noch Zugriff auf die Menge der Kinder erforderlich sein sollte, kann man diese wie folgt als unveränderliches Objekt zurückgeben:

```
1343 class Person {  
1344     ...  
1345     List getKinder() {  
1346         return Collections.unmodifiableList(kinder);  
1347     }  
1348 }
```

Besser ist es jedoch, Berechnungen (Anfragen und Änderungen), die die ganze Liste der Kinder betreffen, in Methoden der Klasse Person zu kapseln. So sollte z. B.

```
1349 int kinderzahl = person.getKinder().size();
```

durch

```
1350 int kinderzahl = person.getKinderzahl();
```

mit

```
1351 class Person {  
1352     ...  
1353     int getKinderzahl() {  
1354         return kinder.size();  
1355     }  
1356 }
```

ersetzt werden. S. dazu auch das *Law of Demeter* in Abschnitt 5.2.5.4.

5.2.3.3 Attributwert durch Objekt ersetzen (REPLACE DATA VALUE WITH OBJECT)

Während man so vor sich hin programmiert, ohne groß darüber nachzudenken, führt man manchmal neue Attribute (Felder) ein, deren Werte primitiven Typs (einschließlich String) sind. Später kann sich dann herausstellen, dass diese Werte für Objekte stehen, die noch mehr Eigenschaften oder vielleicht sogar ihre eigenen Operationen haben. In diesen Fällen wird man den primitiven Typ durch eine speziell für den Attributwert eingeführte Klasse ersetzen.

Beispiel

Einem Kurs (repräsentiert durch die Klasse **Kurs**) wird immer eine Betreuerin zugeordnet, der Kurs erhält dafür ein Attribut mit Namen **betreuerin** vom Typ **String**. Später stellt sich heraus, dass man von der Betreuerin noch ihre Telefonnummer braucht, vielleicht noch ihre Anschrift etc. Anstatt nun immer neue Betreuerinnenattribute für den Kurs einzuführen, würde man doch eher feststellen, dass es sich bei einer Betreuerin um ein eigenständiges Objekt (oder vielleicht, besser noch, um eine eigenständige *Rolle* eines Objektes) handelt und eine entsprechende Klasse bzw. ein Interface einführen. Die vorausschauende Programmiererin wird nicht erst abwarten, bis diese Situation eintritt, sondern von Anfang an, aufgrund konzeptueller Überlegungen, einen eigenen Betreuerinnentyp (Klasse oder Interface) einführen.

Die Ausgangssituation ist also die folgende:

```

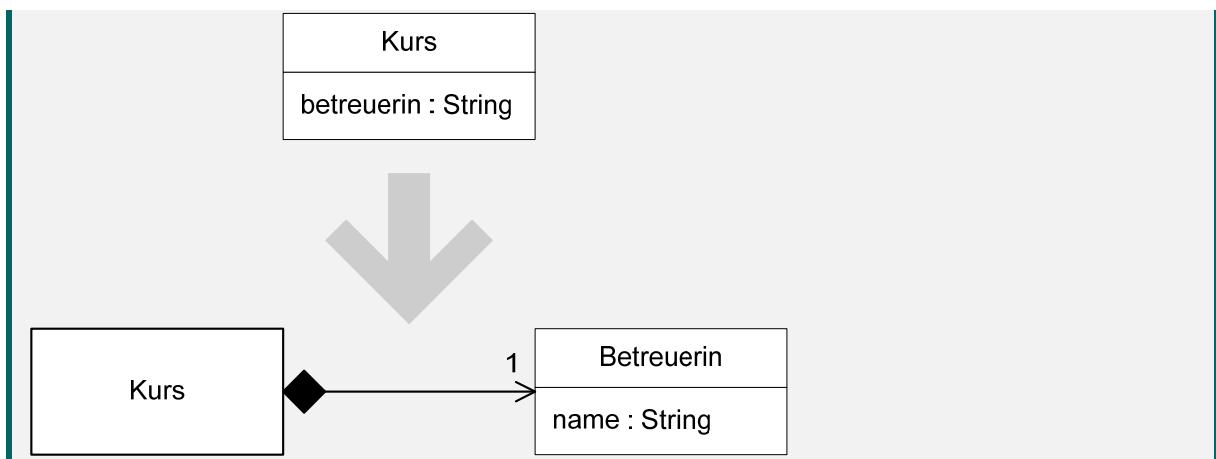
1357 class Kurs {
1358     private String _betreuerin;
1359
1360     public Kurs(String betreuerin) {
1361         _betreuerin = betreuerin;
1362     }
1363
1364     public String getBetreuerin () {
1365         return _betreuerin;
1366     }
1367
1368     public void setBetreuerin (String betreuerin) {
1369         _betreuerin = betreuerin;
1370     }
1371 }
```

Es fällt sofort auf, dass das Attribut **betreuerin**, das die Betreuerin hält, „**betreuerin**“ und nicht „**name**“ oder zumindest „**betreuerinnenname**“ heißt, obwohl der Typ **String** und nicht etwa **Betreuerin** ist. Dies ist ein Indiz dafür, dass hier ein primitiver Wert (hier: ein

String) eingesetzt wird, der ein Objekt (hier: eine Betreuerin) repräsentieren soll. Der Wert vom Typ **String** kann nur wie folgt durch einen Wert vom Typ **Betreuerin** ersetzt werden:

```
1369 class Kurs {  
1370     private Betreuerin _betreuerin;  
  
1371     public Kurs(String betreuerinnenname) {  
1372         _betreuerin = new Betreuerin(betreuerinnenname);  
1373     }  
  
1374     public String getBetreuerin() {  
1375         return _betreuerin.getName();  
1376     }  
  
1377     public void setBetreuerin(String betreuerinnenname) {  
1378         _betreuerin = new Betreuerin(betreuerinnenname);  
1379     }  
1380 }  
  
1381 class Betreuerin {  
1382     private final String _name; // final macht Betreuerin  
                                immutable  
  
1383     public Betreuerin(String name) {  
1384         _name = name;  
1385     }  
  
1386     public String getName() {  
1387         return _name;  
1388     }  
1389 }
```

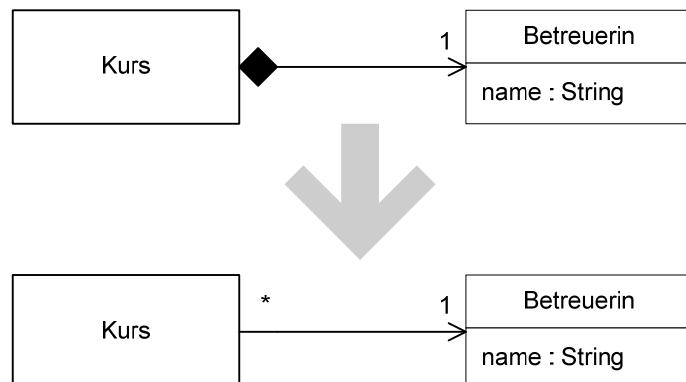
Dabei zeichnen sich die sog. *Wertobjekte* dadurch aus, dass sie unveränderlich (immutable) sind. Man beachte, dass auf diese Weise immer wieder neue Betreuerinnen erzeugt werden, auch wenn sie dieselben Namen haben.



5.2.3.4 Wert durch Referenz ersetzen (CHANGE VALUE TO REFERENCE)

Für JAVA-Programmiererinnen etwas subtil ist die folgende Unterscheidung. Ein Objekt als Attributwert kann dem attribuierten Objekt gehören oder ein allgemeines, von überall zugreifbares Objekt sein. Diese Unterscheidung entspricht in C++ (oder auch Pascal) der Unterscheidung zwischen *Wertobjekten* (für die der benötigte Speicherplatz bereits während der Übersetzung reserviert wird) und *Referenzobjekten* (für die der benötigte Speicherplatz dynamisch, bei Ausführung von `new`, allokiert wird). In JAVA kann diese Unterscheidung auf Sprachebene nicht getroffen werden, da alle Objekte Referenzobjekte sind (und, ohne weitere Vorkehrungen, von überall her referenziert werden können). In UML wird das Besitzen eines Objektes (mit dem auch das Löschen des besessenen Objektes gemeinsam mit dem besitzenden verbunden ist) durch die Komposition (schwarz gefüllte Raute) ausgedrückt.

Die Löschsemantik des Besitzens verlangt im Umkehrschluss, dass wenn ein Objekt Attribut mehrerer Objekte gleichzeitig sein können soll, dass es dann kein Wertobjekt sein darf. Das ist insbesondere (aber nicht nur) bei *m:n*-Beziehungen der Fall. Die entsprechende Änderung, vom besessenen zum frei zugänglichen Objekt, sieht in einem Klassendiagramm wie folgt aus:



Der Stern auf der Seite des Kurses zeigt an, dass eine Betreuerin zu mehreren Kursen gehören kann. Eine Änderung weg vom Wertattribut hin zum Referenzattribut, ohne das Interface von **Kurs** zu verändern, sieht dann z. B. so aus:

1390 **class** Betreuerin {

```
1391 private static Dictionary _instances = new Hashtable();
1392 private final String _name;
1393
1394     private Betreuerin (String name) {// nur noch intern gebraucht
1395         _name = name;
1396     }
1397
1398     public String getName() {
1399         return _name;
1400     }
1401
1402     public static Betreuerin create(String name) {
1403         Betreuerin betreuerin;
1404         if (_instances.contains(name))
1405             betreuerin = (Betreuerin) _instances.get(name);
1406         else {
1407             betreuerin = new Betreuerin(name);
1408             _instances.put(name, betreuerin);
1409         }
1410         return betreuerin;
1411     }
1412
1413 }
1414
1415     class Kurs {
1416
1417     ...
1418
1419     public Kurs(String betreuerinnenname) {
1420         _betreuerin = Betreuerin.create(betreuerinnenname);
1421     }
1422
1423
1424     public void setBetreuerin(String betreuerinnenname) {
1425         _betreuerin = Betreuerin.create(betreuerinnenname);
1426     }
1427
1428     public String getBetreuerinnenname() {
1429         return _betreuerin.getName();
1430     }
1431 }
```

5.2.3.5 Klasse extrahieren (EXTRACT CLASS)

Die beiden vorgenannten Refactorings, Attributwert durch Objekt ersetzen (REPLACE DATA VALUE WITH OBJECT) und Wert durch Referenz ersetzen (CHANGE VALUE TO REFERENCE), können auch in einem Rutsch durchgeführt werden. Dies ist insbesondere dann sinnvoll, wenn sich in einer Klasse abzeichnet, dass Teile der Eigenschaften eigent-

lich Objekte eines anderen Typs beschreiben, wie z. B. in einer Klasse **Person** mit den folgenden Eigenschaften:

```

1421 class Person {
1422     private String _name;
1423     private String _bankleitzahl;
1424     private String _kontonummer;
1425
1426     public String getName() {
1427         return _name;
1428     }
1429
1430     public String getBankverbindung() {
1431         return "Kto: " + _kontonummer + " BLZ: " + _bankleitzahl;
1432     }
1433
1434     void setBankleitzahl(String arg) {
1435         _bankleitzahl = arg;
1436     }
1437
1438     String getKontonummer() {
1439         return _kontonummer;
1440     }
1441
1442     void setKontonummer(String arg) {
1443         _kontonummer = arg;
1444     }

```

Ein nicht unerheblicher Teil der Klassendefinition befasst sich nun mit der Bankverbindung der Person und nicht der Person selbst, was man schon allein an den Namen der Methoden erkennt. Die natürliche Maßnahme ist also, den Bankverbindungsanteil aus der Personendefinition herauszuziehen, wie im Folgenden geschehen:

```

1444 class Person {
1445     private String _name;
1446     private Bankverbindung _bankverbindung = new Bankverbindung();
1447
1448     public String getName() {
1449         return _name;
1450     }

```

```

1449    }

1450  public String getBankverbindung() {
1451      return _bankverbindung.getBankverbindung();
1452  }
1453 }

1454 class Bankverbindung {
1455     private String _bankleitzahl;
1456     private String _kontonummer;

1457     public String getBankverbindung() {
1458         return "Kto: " + _kontonummer + " BLZ: " + _bankleitzahl;
1459     }

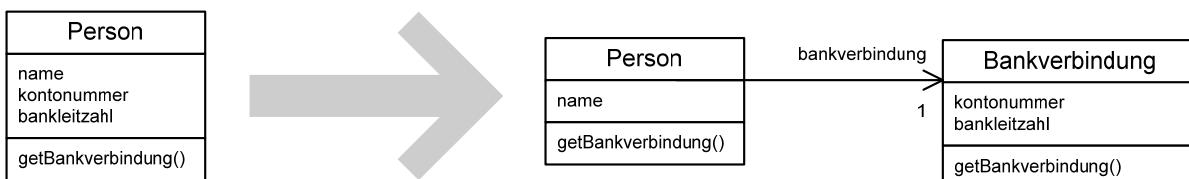
1460     String getBankleitzahl() {
1461         return _bankleitzahl;
1462     }

1463     void setBankleitzahl(String arg) {
1464         _bankleitzahl = arg;
1465     }

1466     String getKontonummer() {
1467         return _kontonummer;
1468     }

1469     void setKontonummer(String arg) {
1470         _kontonummer = arg;
1471     }
1472 }
```

Das Refactoring ist im nachfolgenden UML-Diagramm zusammengefasst:



Man beachte, dass im Beispiel die Beziehung zwischen Person und Bankverbindung eine 1:1-Beziehung ist. Es können aber auch sog. Wiederholungsgruppen (also sich wiederholende Gruppen von Feldern, wie sie etwa bei mehreren Bankverbindungen einer Person vorkommen würden; ein Begriff aus der Datenbankwelt) vorkommen, die natürlich umso mehr in eine eigene Klasse ausgelagert werden sollten (entsprechend einer Normalisierung).

5.2.3.6 Unidirektionale in bidirektionale Verknüpfung ändern (CHANGE UNIDIRECTIONAL ASSOCIATION TO BIDIRECTIONAL)

Die Verbindung, die von einem Objekt zu einem anderen durch ein Attribut entsteht, ist gerichtet in dem Sinne, dass das attributierte Objekt auf sein Attributobjekt zugreifen kann, umgekehrt aber nicht: Das Attributobjekt weiß nicht, welches Objekt es attribuiert (von welchem Objekt es referenziert wird). Während dies bei Attributen meistens auch gar nicht vonnöten ist, so sind Assoziationen doch häufig in beide Richtungen navigierbar: Die assoziierten Objekte haben gegenseitig voneinander Kenntnis.⁵⁹

Das nachfolgende **Beispiel** ist typisch für eine unidirektionale Verknüpfung:

```
1473 class Order {
1474     Customer _customer;
1475
1476     Customer getCustomer() {
1477         return _customer;
1478     }
1479
1480     void setCustomer (Customer arg) {
1481         _customer = arg;
1482     }
1483 }
```

Die gegenseitige (bidirektionale) Verknüpfung von Objekten ist technisch um einiges vielschichtiger, als man zunächst vermuten würde. Selbst wenn die Verknüpfung eine 1:1-Beziehung darstellt, ist es nicht mit der gegenseitigen Einrichtung von zwei Attributen getan: Wenn 1:1 auch noch bedeuten soll, dass die Objekte ein Paar sind, also einander wechselseitig referenzieren (dies ist zunächst gar nicht gesagt!), dann muss auch noch sichergestellt werden, dass jede Änderung eines Attributwerts automatisch die Änderung des anderen nach sich zieht. Dies würde beispielsweise durch den folgenden, vergleichsweise umfangreichen Quellcode sichergestellt:

```
1482 class A {
1483     private B b;
1484
1485     void setB(B b) {
1486         if (b == this.b)
1487             return; // vermeidet Rekursion
1488         if (this.b != null)
1489             b.getA().setB(null);
1490         this.b = b;
1491         b.setA(this);
```

⁵⁹ Wie bereits erwähnt, existiert in JAVA keine Unterscheidung zwischen Attribut und Assoziation — beide werden durch Referenzen (Pointer) realisiert. Konzeptuell, und damit auch in UML, existiert diese Unterscheidung aber sehr wohl.

```
1491     }
1492 }

1493 class B {
1494     private A a;

1495     void setA(A a) {
1496         if (a == this.a)
1497             return; // vermeidet Rekursion
1498         if (this.a != null)
1499             a.getB().setA(null);
1500         this.a = a;
1501         a.setB(this);
1502     }
1503 }
```

oder so ähnlich. Wenn es sich hingegen um eine $1:n$ -Beziehung handelt (mit der Nebenbedingung, dass jedes der Objekte auf der n -Seite mit dem Objekt auf der 1-Seite, durch das es referenziert wird, verknüpft sein soll), dann wäre in etwa der folgende Code notwendig:

```
1504 class Customer {
1505     Set _orders = new HashSet(); // kann nicht private sein!

1506     void addOrder(Order order) {
1507         order.setCustomer(this); // Weiterleitung an die Bestellung
1508     }
1509 }

1510 class Order {
1511     ...
1512     void setCustomer(Customer customer) {
1513         if (_customer != null)
1514             _customer._orders.remove(this);
1515         _customer = customer;
1516         if (_customer != null)
1517             _customer._orders.add(this);
1518     }
1519 }
```

Selbsttestaufgabe 5.3

Die beiden Alternativen benötigen jeweils einen Zugriff auf ein fremdes Feld, das dadurch öffentlich zugänglich gemacht werden muss, so dass man eigentlich das Feld- (oder Methode-) verlagern-Refactoring (s. 5.2.5.3) verwenden möchte. Was ist hierbei das Problem?

Versuchen Sie, eine Lösung zu finden, die ohne Über-Kreuz-Zugriffe auskommt.

Wollte man hingegen eine $m:n$ -Beziehung realisieren, so wäre der folgende Code eine Lösung:

```

1520 class Order {
...
1521     void addCustomer (Customer customer) {
1522         customer._orders.add(this);
1523         _customers.add(customer);
1524     }
1525     void removeCustomer (Customer customer) {
1526         customer._orders.remove(this);
1527         _customers.remove(customer);
1528     }
1529 }
1530
1531 class Customer {
...
1532     void addOrder(Order order) {
1533         order.addCustomer(this);
1534     }
1535     void removeOrder(Order order) {
1536         order.removeCustomer(this);
1537     }
1538 }
```

Anmerkung: Wie bereits erwähnt, erkennt man den Reifegrad einer Programmiersprache daran, inwieweit sie solche Wendungen (Idiome) oder auch Muster (Programming/Implementation patterns), die in der Praxis extrem häufig vorkommen, unterstützt. In JAVA, C++ und C# ist diese Unterstützung praktisch nicht vorhanden⁶⁰; in SMALLTALK lässt sie sich durch die dort vorhandenen Möglichkeiten der *Metaprogrammierung* zumindest ziemlich einfach ergänzen.

5.2.4 Generalisierung einsetzen

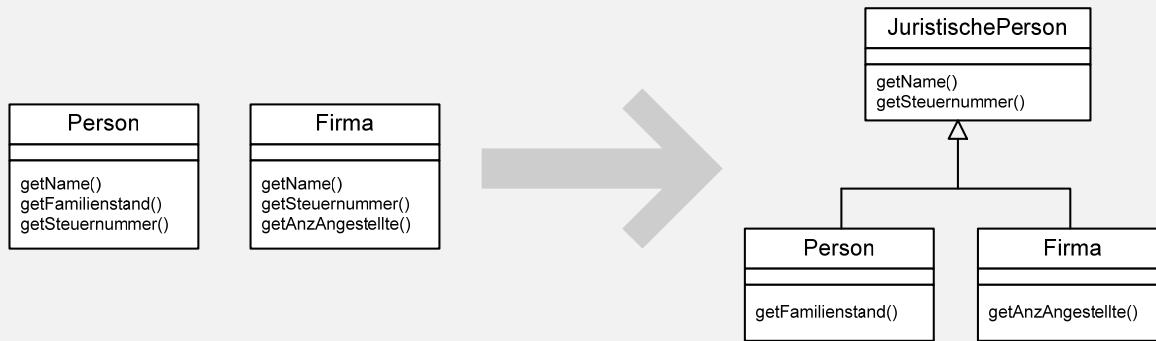
Eine Vielzahl vorhandener Refactorings, einige davon mit Werkzeugunterstützung, befasst sich mit der *Generalisierung*, also der Einführung oder Verwendung von Supertypen. Diese Refactorings erscheinen auf den ersten Blick recht einfach, induzieren jedoch zum Teil recht subtile Probleme.

⁶⁰ Immerhin bereitet zumindest das Foreach-Konstrukt im Zusammenspiel mit Generic collections ab JAVA 5 eine gewisse Erleichterung.

5.2.4.1 Superklasse extrahieren (EXTRACT SUPERCLASS)

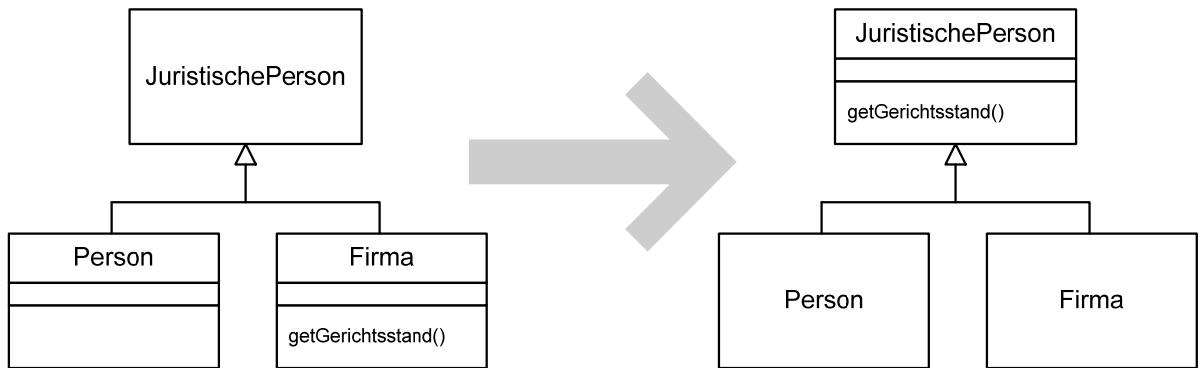
Wenn zwei Klassen inhaltlich eng miteinander verwandt sind, ohne dass die eine eine (logische) Superklasse der anderen wäre, dann kann es sinnvoll sein, diese Verwandtschaft dadurch auszudrücken, dass man den beiden eine gemeinsame Superklasse gibt, die die Gemeinsamkeiten herausfaktorisiert. Man beachte, dass sich die Verwandtschaft nicht unbedingt von vorneherein in der syntaktischen Gleichheit von Eigenschaften (Feldern und Methoden) ausdrücken muss. Wichtig ist vielmehr eine inhaltliche Übereinstimmung, die eine prinzipielle Austauschbarkeit (*Substituierbarkeit*) der Instanzen gegeneinander gewährleistet. Ist diese gegeben, dann sollte auch eine syntaktische Angleichung der Eigenschaften kein Problem sein. Schließlich stehen dafür genügend andere Refactorings zur Verfügung.

Beispiel



5.2.4.2 Feld oder Methode nach oben verschieben (PULL UP FIELD/METHOD)

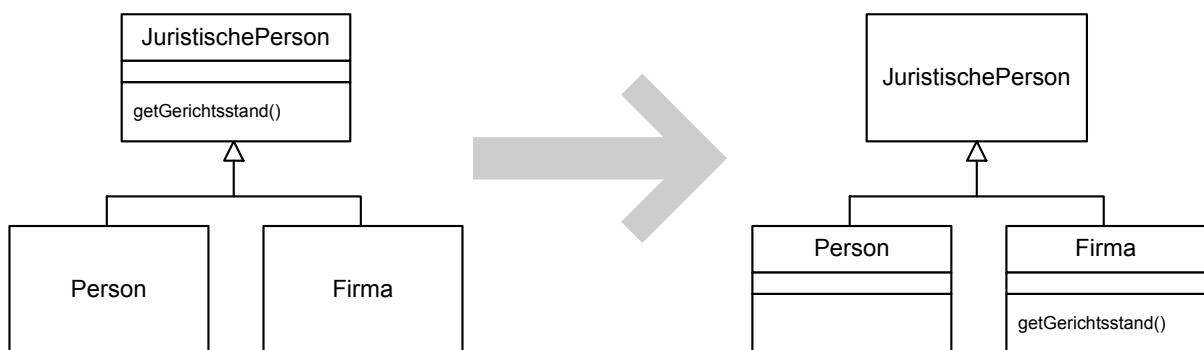
Wenn zwei Klassen, die durch eine gemeinsame Superklasse miteinander in Verbindung stehen, dasselbe Feld oder dieselbe Methode deklarieren, so muss man fragen, warum diese Deklaration nicht in der Superklasse stattfindet. Dies ist dann wohl begründet, wenn das Feld oder die Methode in den beiden Klassen nur zufällig gleich ist, also die syntaktische Gleichheit keine semantische Austauschbarkeit impliziert. Die gemeinsame Abstraktion in der Superklasse wäre dann irreführend und würde eine Austauschbarkeit der Objekte der verschiedenen Klassen ermöglichen, die gar nicht gegeben ist. Sind die beiden jedoch auch semantisch gleich (oder zumindest äquivalent im Sinne einer *Substituierbarkeit*), dann kann das Hochziehen des Feldes/der Methode durchaus günstig sein. Man muss dabei jedoch stets prüfen, ob dadurch nicht dritte Klassen, die zuvor nicht über das Feld/die Methode verfügt haben, diese nun erben, und wenn das der Fall ist, ob es auch inhaltlich korrekt ist. In der Praxis wird man dieses Refactoring also vor allem dann verwenden, wenn man zu einer bestehenden Klasse eine zweite, ähnliche definieren will, und zu diesem Zweck eine neue Superklasse einführt, die alle Eigenschaften der ersten Klasse übernimmt, die die zweite ebenfalls haben soll.



Vorsicht: Wenn man in JAVA eine Methode nach oben verschiebt, ändert sich der statische Typ von `this`. Dies hat u. U. Auswirkungen auf das Binden von Methoden, nämlich immer dann, wenn `this` als Parameter eines Methodenaufrufs vorkommt (vgl. **Selbsttestaufgabe 4.7**).

5.2.4.3 Feld oder Methode nach unten verschieben (PUSH DOWN FIELD/METHOD)

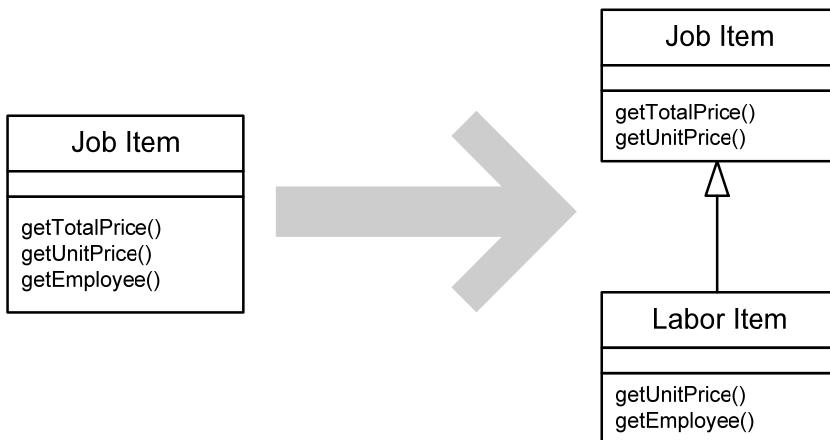
Gelegentlich ist bereits das der Fall, was durch das vorgenannte Refactoring nicht passieren soll: Eine Eigenschaft (Feld oder Methode), die in einer Superklasse definiert ist, gilt nicht für alle ihre Subklassen. Ein gutes Beispiel dafür ist die Originalversion des COMPOSITE Pattern, bei der ja in der gemeinsamen Abstraktion von Ganzen und Teilen Eigenschaften definiert werden, die für Teile gar nicht zutreffend sind (siehe Abschnitt 4.4.1). In einem solchen Fall sollte die unzulässige *Generalisierung* aufgehoben werden, indem die Eigenschaft in die Klasse(n) verschoben wird, für die sie auch zutrifft. Die logische Konsequenz dessen ist, dass die Klassen, was die verschobene Eigenschaft angeht, nicht mehr gegeneinander austauschbar sind.



5.2.4.4 Subklasse extrahieren (EXTRACT SUBCLASS)

Manchmal enthält eine Klassendefinition Eigenschaften, die gar nicht für alle Instanzen der Klasse zutreffend sind, ohne dass es schon eine entsprechende Unterklasse gäbe, die diese Instanzen aufnehmen könnte. Die Klasse vereinigt dann Instanzen unterschiedlichen Typs, was in der Regel dazu führt, dass von Fall zu Fall (mittels Fallunterscheidung) zwischen diesen verschiedenen Typen unterschieden werden muss (siehe „Bedingung durch Polymorphismus ersetzen“ in Abschnitt 5.2.1.3). Fallunterscheidungen, die auf den Typ von Instanzen zurückzuführen sind, sollten in der objektorientierten Programmierung aber immer durch die Klassenhierarchie (in

Verbindung mit *Polymorphismus*) ausgedrückt werden. Es empfiehlt sich also in einem solchen Fall, eine Subklasse zu bilden, die die Eigenschaften enthält, die nur für manche Instanzen zutreffend sind, wie in der folgenden Abbildung dargestellt.



Der nachfolgende Code zeigt das dazugehörige Vorher

```
1539 class JobItem {  
1540     private int _unitPrice;  
1541     private int _quantity;  
1542     private boolean _isLabor;  
1543     private Employee _employee;  
  
1544     public JobItem (int unitPrice, int quantity, boolean isLabor,  
                      Employee employee) {  
1545         _unitPrice = unitPrice;  
1546         _quantity = quantity;  
1547         _isLabor = isLabor;  
1548         _employee = employee;  
1549     }  
  
1550     public int getTotalPrice() {  
1551         return getUnitPrice() * _quantity;  
1552     }  
  
1553     public int getUnitPrice() {  
1554         return (_isLabor) ? _employee.getRate() : _unitPrice;  
1555     }  
  
1556     public int getQuantity(){  
1557         return _quantity;  
1558     }  
  
1559     public Employee getEmployee() {  
1560         return _employee;
```

```
1561     }
1562 }

1563 class Employee {
1564     private int _rate;

1565     public Employee (int rate) {
1566         _rate = rate;
1567     }

1568     public int getRate() {
1569         return _rate;
1570     }
1571 }
```

und Nachher

```
1572 class JobItem {
1573     private int _unitPrice;
1574     private int _quantity;
1575     private boolean _isLabor;

1576     protected JobItem (int unitPrice, int quantity, boolean
1577                         isLabor) {
1577         _unitPrice = unitPrice;
1578         _quantity = quantity;
1579         _isLabor = isLabor;
1580     }

1581     public int getUnitPrice(){
1582         return _unitPrice;
1583     }
1584 }

1585 class LaborItem extends JobItem {
1586     private Employee _employee;

1587     public LaborItem (int quantity, Employee employee) {
1588         super(0, quantity, true);
1589         _employee = employee;
1590     }

1591     public Employee getEmployee() {
1592         return _employee;
1593     }
```

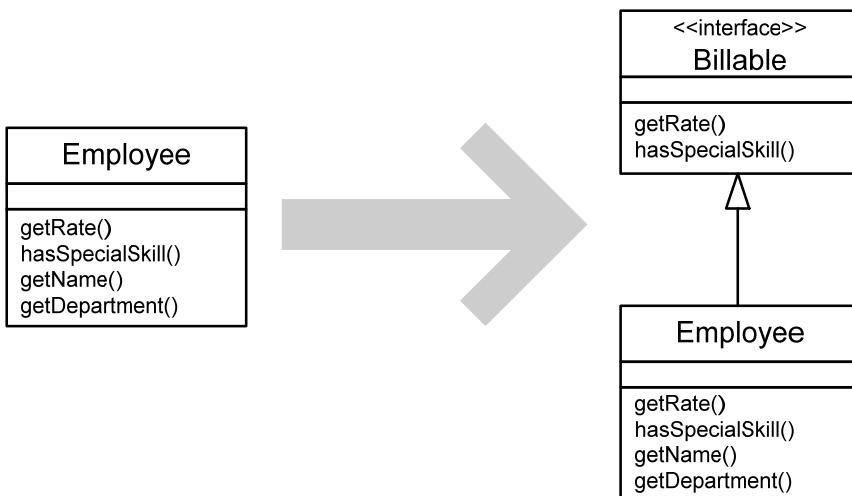
```

1594
1595     public int getUnitPrice() {
1596         return _employee.getRate();
1597     }
1598 }
```

Dieses Refactoring kann noch einmal wiederholt werden, um die Subklasse **PartItem** aus **JobItem** zu extrahieren. **JobItem** würde damit abstrakt.

5.2.4.5 Interface extrahieren (EXTRACT INTERFACE)

Recht häufig wird innerhalb eines bestimmten Kontexts (z. B. in einer Klasse) von einem Objekt nur ein kleiner Teil seines Protokolls benötigt. Es kann dann sinnvoll sein, vom konkreten Typ des Objekts (seiner Klasse) zu abstrahieren und statt dessen ein Interface einzuführen, das genau die Methoden enthält, die in dem Kontext gebraucht werden (ein *kontextspezifisches Interface*; siehe Abschnitt 1.5.6). Dieser Vorgang wird durch das folgende Diagramm dargestellt.



So verwendet beispielsweise der nachfolgende zugehörige Code

```

1599 double charge(Employee emp, int days) {
1600     int base = emp.getRate() * days;
1601     if (emp.hasSpecialSkill())
1602         return base * 1.05;
1603     else
1604         return base;
1605 }
```

von **Employee** lediglich die beiden Methoden **getRate()** und **hasSpecialSkill()**. Extrahiert man beide in ein Interface **Billable**, lässt **Employee** dieses implementieren und verwendet es anschließend in der Deklaration von **emp**, so erhält man

```

1606 interface Billable {
1607     public int getRate();
1608     public boolean hasSpecialSkill();
1609 }
1610 class Employee implements Billable ...
1611 double charge(Billable emp, int days) {
1612     int base = emp.getRate() * days;
1613     if (emp.hasSpecialSkill())
1614         return base * 1.05;
1615     else
1616         return base;
1617 }

```

Tatsächlich ist dieses Refactoring in der Praxis sehr viel komplizierter, als man meinen könnte. Dies liegt vor allem daran, dass das Objekt, für das das Interface extrahiert werden soll (`emp` im obigen Beispiel) unter Umständen aus dem gegebenen Kontext heraus in andere übergeben werden muss (z. B. durch geschachtelte Methodenaufrufe), in denen weitere Eigenschaften von ihm gefordert werden. Diese zusätzlichen Eigenschaften sind nicht unmittelbar ersichtlich; man muss dazu die Zuweisungen eines Programms genau untersuchen. Und so überlassen die gängigen Implementierungen der Programmiererin auch, selbst die Eigenschaften zu bestimmen, die in das Interface sollen. Ob diese dann auch ausreichen, zeigt letztlich nur der Versuch, das Programm zu kompilieren. Abhilfe bietet hier das nächste Refactoring.

5.2.4.6 Interface berechnen (INFERTYPE**)**

Die Information, welche Methoden ein zu extrahierendes Interface haben muss, damit es als Typ einer Variable verwendbar ist, steckt im Programm selbst. Um sich das zu verdeutlichen, stelle man sich vor, man habe genügend Zeit und lösche einfach probeweise eine Methode aus einem Interface, das anfänglich alle (öffentlichen) Methoden einer Klasse enthält. Kompiliert das Programm danach nicht mehr, wird die Methode benötigt und muss wieder aufgenommen werden, sonst kann sie wegbleiben. Dieses Verfahren ist jedoch kaum praktikabel.

Um den Typ einer Variable konstruktiv (ohne Herumprobieren) zu bestimmen, lässt sich die sog. *Typinferenz* einsetzen. Dabei werden die Zuweisungen in einem Programm verfolgt und ein entsprechendes Netz von sog. *Typconstraints* erstellt, die die benötigte Typkonformität (*Subtyping*) abbilden. Das gesuchte Interface ist das kleinste, das alle *Typconstraints* erfüllt; es kann durch Traversierung des Constraintnetzes konstruktiv ermittelt werden. Ein Refactoring, das dieses Verfahren umsetzt und das so anhand einer Variable das von dieser Variable benötigte Interface automatisch extrahiert (und somit das obige Problem löst), finden Sie unter <http://www.fernuni-hagen.de/ps/docs/InferType/>.

5.2.4.7 Subklassen durch Felder ersetzen (REPLACE SUBCLASS WITH FIELDS)

Manchmal hat man es auf konzeptueller Ebene mit einer großen Zahl von Subklassen zu tun, die sich in der Implementation nur marginal unterscheiden. Zum Beispiel kann man in der Veranstaltungsorganisation Klassen (eigentlich *Rollen*) wie Beleuchter, Souffleur, Tontechniker, Platzanweiser, Eisverkäufer etc. identifizieren, die sich zunächst einmal konzeptuell klar unterscheiden, die im zu bauenden System dennoch alle mehr oder weniger dieselbe Rolle spielen: zu bestimmten Zeiten irgendwo erscheinen, Dienst verrichten, abkassieren. In solchen Fällen lohnt sich die Definition entsprechender Klassen nicht unbedingt.

Folgendes Beispiel mag als Verdeutlichung dienen.

```
1618 abstract class Person {  
1619     abstract boolean isMale();  
1620     abstract char getCode();  
1621 }  
  
1622 class Male extends Person {  
  
1623     boolean isMale() {  
1624         return true;  
1625     }  
  
1626     char getCode() {  
1627         return 'M';  
1628     }  
1629 }  
  
1630 class Female extends Person {  
  
1631     boolean isMale() {  
1632         return false;  
1633     }  
  
1634     char getCode() {  
1635         return 'F';  
1636     }  
1637 }
```

Anmerkung: Methoden wie `getCode()` nennt man übrigens auch *polymorphe Konstantenmethoden* (engl. polymorphic constant methods, [40] S. 334), da sie Konstanten repräsentieren, deren Wert vom Typ des besitzenden Objekts abhängt. Dieser Begriff kommt offenbar aus der SMALLTALK-Programmierung, da dort Konstanten immer als Methoden implementiert werden müssen.

polymorphe Konstanten-methoden

Auch wenn die konsequente Verwendung von Subklassen objektorientierter Stil ist (und viele Refactorings auf eine solche Art der Programmierung hinwirken), kann es doch Fälle geben, in denen sich der Aufwand der vielen Subklassen nicht lohnt. Dies ist insbesondere dann der Fall, wenn die Zahl der Subklassen groß ist, sich diese aber kaum unterscheiden und die Unterschiede gut parametrisierbar sind. Dies ist im gegebenen Beispiel der Fall, und somit könnte man auch schreiben:

```

1638 class Person {
1639     private final boolean _isFemale;
1640     private final char _code;
1641
1642     private Person (boolean isFemale) { // private deklariert!
1643         _isFemale = isFemale;
1644         _code = _isFemale ? 'F' : 'M';
1645     }
1646
1647     static Person createMale() { // Factory
1648         return new Person(false);
1649     }
1650
1651     static Person createFemale() { // Factory
1652         return new Person(true);
1653     }
1654
1655     boolean isMale() {
1656         return !_isFemale;
1657     }
1658     boolean isFemale() {
1659         return _isFemale;
1660     }
1661
1662     char getCode() {
1663         return _code;
1664     }
1665 }
```

Man erkennt jedoch schon an diesem kurzen Beispiel, dass sich der Aufwand kaum lohnt: Das refaktorierte Programm ist weder kürzer noch besser lesbar. Etwas anderes ist es jedoch, wenn eine zunächst fixe Anzahl von Subklassen zur Laufzeit des Programms erweiterbar werden soll: Dann müsste das Programm schon selbst Klassen generieren können (eine Form der *Metaprogrammierung*; s. Kurseinheit 6), um mit der Variante mit den Subklassen arbeiten zu können. In der refaktorisierten, parametrisierten Variante wären dazu lediglich die Methoden, deren Namen direkt auf die Subklassen Bezug nehmen, so abzuändern, dass auch die Klassennamen parametrisch sind.

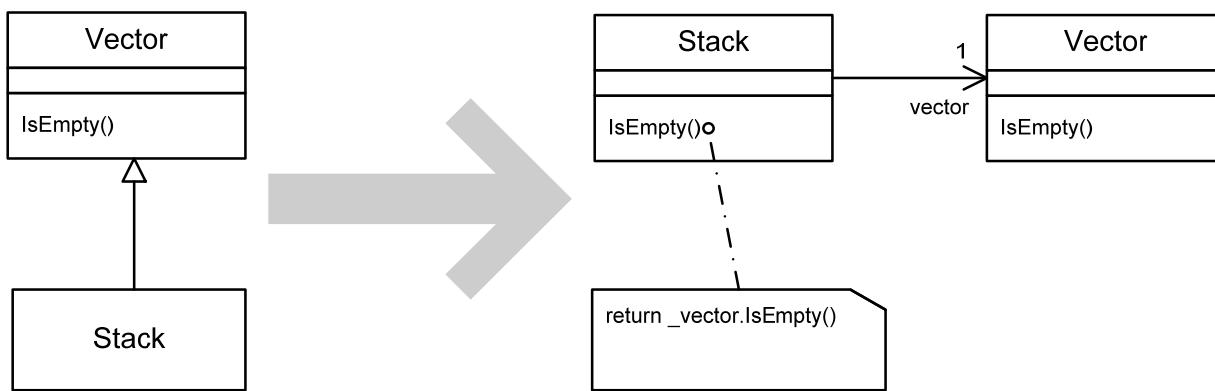
Selbsttestaufgabe 5.4

Modifizieren Sie die obige refaktorierte Variante so weiter, dass Sie damit auch noch beliebige weitere Kategorien (Zwitter, Transvestiten, etc.), die der Benutzer zur Laufzeit eingeben kann, abdecken können.

5.2.4.8 Vererbung durch Delegation ersetzen (REPLACE INHERITANCE WITH DELEGATION)

Aus der Anfangszeit der objektorientierten Programmierung stammt eine gewisse Euphorie, was die Möglichkeiten der Vererbung angeht. Man hatte geglaubt, man könne durch Vererbung die massenhafte Wiederverwendung von Code betreiben. Tatsächlich, so hat sich herausgestellt, muss aber die Vererbung von Code zwischen vererbender und erbender Klasse sorgfältig abgestimmt werden; das Erben von Klassen, die man nicht selbst programmiert (mehr noch: deren Programmierung man nicht selbst unter Kontrolle) hat, erweist sich als ausgesprochen fehleranfällig.

Erschwerend kommt hinzu, dass in den heute gängigsten Sprachen, JAVA, C++ und C#, Vererbung an *Subtyping* gekoppelt ist. Während wenn Subtyping im Sinne einer *Substituierbarkeit* vorliegt die gleichzeitige Vererbung von Code meistens kein Problem darstellt, so führt das Subtyping aus dem Bestreben, etwas zu erben, mitunter zu absurdem Verhältnissen. Das beste Beispiel dafür ist die Ableitung der Klasse **Stack** aus der Klasse **vector** im JDK: Kann ein Stack überall da auftreten, wo ein Vector verlangt wird? Wohl kaum. In solchen Fällen ist es unbedingt angezeigt, die Vererbung über *Forwarding* bzw. *Delegation* (s. Abschnitt 4.2.3 sowie die beispielhafte Behandlung des Refactorings in Abschnitt 5.1.5 für die Unterscheidung der beiden), also wie im folgenden Diagramm gezeigt, zu realisieren.



Aus

```

1661 class Stack extends Vector {
1662     public void push(Object element) {
1663         insertElementAt(element, 0);
1664     }
1665     public Object pop() {
  
```

```

1666     return remove(0);
1667 }
1668 }
```

wird

```

1669 class Stack {
1670     private Vector _vector = new Vector();

1671     public void push(Object element) {
1672         _vector.insertElementAt(element, 0);
1673     }

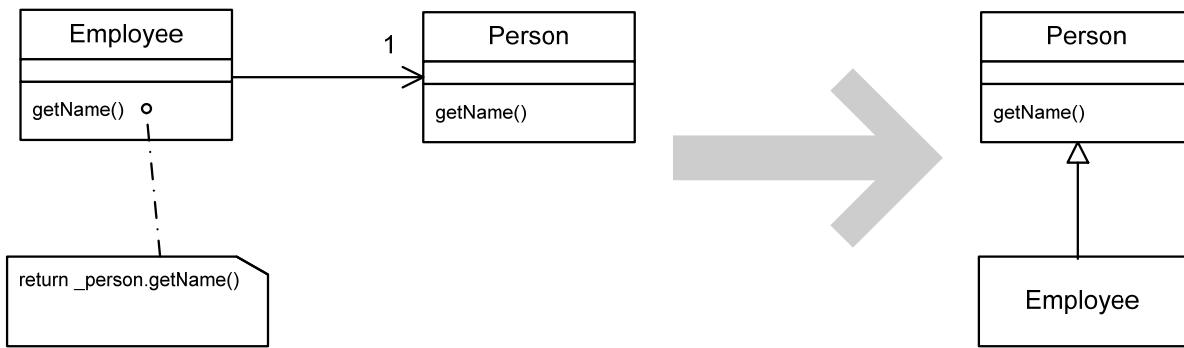
1674     public Object pop() {
1675         return _vector.remove(0);
1676     }

1677     public int size() {
1678         return _vector.size();
1679     }

1680     public boolean isEmpty() {
1681         return _vector.isEmpty();
1682     }
1683 }
```

Der Stack hält sich also quasi einen Vector als Sklaven und lässt diesen die Arbeit tun, ohne dass äußerlich irgendein erkennbarer Zusammenhang zwischen den beiden Klassen bestünde, insbesondere eine Substituierbarkeit von Instanzen der einen für Instanzen der anderen gegeben wäre. Darüber hinaus, und anders als beim *Subclassing* möglich, kann **Stack** auch noch die von **Vector** genutzten Methoden selbst benennen; da **Stack** nicht von **Vector** ableitet, erbt es auch keine Methodendeklarationen und kann seine Methoden nennen, wie es will (Programmzeilen 1671 und 1672 geben ein Beispiel dafür). Der Preis hierfür ist allerdings jeweils ein weiterer Methodenaufruf, der jedoch von einem geschickten Compiler wegoptimiert werden kann (allerdings nur, wenn der Vector-Sklave **final** deklariert und eine *dynamische Bindung* damit ausgeschlossen ist).

Anmerkung: Fowler bietet in seinem Buch auch das inverse Refactoring, „Delegation durch Vererbung ersetzen“, an. Er gibt dafür das folgende Beispiel:



Schlechter hätte das Beispiel kaum gewählt sein können: **Employee** (Angestellter) ist eine *Rolle* von **Person** und kann daher nicht durch einen Subtyp von **Person** repräsentiert werden: Was wäre, wenn eine Person zugleich auch noch andere Rollen, z. B. **Employer**, spielen können soll? Abgesehen davon, dass das Beispiel nichts taugt, sind nur wenige Fälle vorstellbar, in denen man Delegation tatsächlich durch Vererbung ersetzen wollte: Allzu groß sind die Anforderungen, die die resultierende Substituierbarkeit mit sich bringt. Die gesparte Schreibarbeit, die im Buch vorgegeben wird, kann kaum als Grund herhalten; die beste Erklärung ist die Vermutung, dass das Refactoring vor allem aus Symmetriegründen in das Buch hineingeraten ist.

5.2.5 Methoden organisieren

5.2.5.1 Methode extrahieren (EXTRACT METHOD)

Ein für die objektorientierte Programmierung typisches Merkmal ist, dass Methoden nicht besonders lang sind. Tatsächlich haben in SMALLTALK viele Methoden nicht mehr als fünf Zeilen; nicht wenige bestehen sogar nur aus einer. Für JAVA gilt das nicht ganz, da es schon allein aufgrund mangelnder Unterstützung von 1:n-Beziehungen (s. Abschnitt 5.2.3.2; vgl. a. Kurs 01814) fürchterlich wortreich ist — doch auch hier gilt, dass lange Methoden verpönt sind.

Die Kürze der Methoden röhrt zum großen Teil daher, dass die meisten komplexen Funktionen in ein Zusammenspiel mehrerer Objekte aufgelöst werden. Da eine Methode unmittelbar nur Zugriff auf die Attribute (Instanzvariablen, den *Zustand*) des eigenen Objekts hat (in SMALLTALK ist das zwangsläufig so, in anderen Sprachen sollte es zumindest so sein — siehe Refactoring „Feld kapseln“ in Abschnitt 5.2.3.1), enthalten einzelne Methoden oft nur die Anweisungen, die den Zustand direkt manipulieren — die Manipulation des Zustands anderer Objekte wird an diese delegiert. Zu implementierende Funktionen werden demzufolge mal mehr, mal weniger gleichmäßig auf die Methoden der verschiedenen Objekte aufgeteilt.

Der positive Nebeneffekt kurzer Methoden, von denen es dann auch entsprechend viele gibt, ist die erhöhte Lesbarkeit: Wenn die Methoden sinnvoll benannt sind (s. Abschnitt 5.2.2.1), dann liest sich der Rumpf einer Methode, der nicht selten im Wesentlichen nur noch aus einer Sequenz von Methodenaufrufen besteht, wie ein verbaler Algorithmus. Nachteilig ist hingegen, wenn die aufgerufenen Methoden (trotz sprechender Namen) zurate gezogen werden müssen (z. B. zum Debuggen; s. u.) — dann erscheint einem der Quelltext unnötig verteilt.

Mit dem Methode-extrahieren-Refactoring ist es möglich, aus einer langen Methode einen Teil zu entfernen, in einer eigenen Methode zur Verfügung zu stellen und an seiner ursprünglichen Stelle durch einen entsprechenden Methodenaufruf zu ersetzen. Die Kunst dabei ist, zu erkennen, welche Variablen im extrahierten Teil seiner ursprünglichen Umgebung entstammen (also dort deklariert und initialisiert wurden), und diese in formale Parameter für die neue Methode zu transformieren. Da die Originalmethode direkten Zugriff nur auf das aktuelle Objekt hat, wird die extrahierte Methode standardmäßig auch derselben Klasse zugeordnet.

Beispiel

aus

```
1684 void printOwing() {  
1685     printBanner();  
1686     //print details  
1687     System.out.println ("name: " + _name);  
1688     System.out.println ("amount: " + getOutstanding());  
1689 }
```

wird

```
1690 void printOwing() {  
1691     printBanner();  
1692     printDetails(getOutstanding());  
1693 }  
  
1694 void printDetails (double outstanding) {  
1695     System.out.println ("name: " + _name);  
1696     System.out.println ("amount: " + outstanding);  
1697 }
```

Man sieht, wie in der Originalmethode ein Kommentar eingeführt wurde, um zu erklären, was im folgenden Abschnitt geschieht; diese Funktion übernimmt nun der Name der neuen Methode.

Anmerkung: Kent Beck antwortet auf die Frage, wie viele Kommentarzeilen ein Programm haben sollte, mit „gar keine“. Wenn die Methoden kurz und gut benannt sind, dann bedarf das Programm keiner weiteren Erklärungen. Das stimmt zwar nicht immer, enthält aber viel Wahrheit.

Manchmal kann es sogar sinnvoll sein, einen einzigen Methodenaufruf aus einer Methode zu extrahieren, nämlich dann, wenn dadurch der Zweck der Methode im gegebenen Kontext klarer wird. Zum Beispiel könnte man

```
1698 void edit(Text text) {  
...  
1699     invert(range);  
...  
1700 }
```

wobei **invert(Range)** eine Methode ist, die einen Textbereich invers (also beispielsweise weiß auf schwarz) darstellt, durch

```
1701 void edit(Text text) {  
...  
1702     highlight(range);  
...  
1703 }  
  
1704 void highlight(Range range) {  
1705     invert(range);  
1706 }
```

ersetzen, nur um klar zu machen, dass eine Hervorhebung gewünscht ist (das Was), die im gegebenen Fall durch eine Invertierung realisiert wird (das Wie; siehe auch das Methodennameändern-Refactoring, Abschnitt 5.2.2.1). Man beachte, dass **highlight** in diesem Fall eine Art Funktionskonstante ist, also eine Festlegung an zentraler Stelle, für was der Name „highlight“ stehen soll, d. h., durch was er im Programm ersetzt werden soll (ähnlich einer Wertkonstante wie **Pi**).

Anmerkung: So attraktiv kurze Methoden auch erscheinen mögen, sie haben trotzdem einen gravierenden Nachteil: Beim Source-level-Debugging springt die Ausführung ständig zwischen verschiedenen Methoden hin und her, ein Umstand, der einen schnell den Überblick verlieren lässt. Auch das Lesen von Quellcode wird erschwert, wenn man ständig an anderer Stelle nachsehen muss, wie es weitergeht, und wenn man vor lauter verschachtelten Methodenaufrufen irgendwann vergisst, wohin man zurücksspringen muss.

Anmerkung: ECLIPSE verfügt über eine erstaunlich umfassende Implementierung von EXTRACT METHOD. Probieren Sie diese aus und machen Sie es sich zur Gewohnheit, sie immer zu verwenden, sobald Sie eine Methode aufteilen wollen.

Anmerkung: Die Umkehrung des Methode-extrahieren-Refactorings ist natürlich auch definiert: Sie nennt sich Methode inlinen (INLINE METHOD). Inlining verschlechtert jedoch im Allgemeinen die Lesbarkeit eines Programms und sollte daher der Optimierung durch den Compiler vorbehalten bleiben. Im folgenden Beispiel für Method inlining ist dies jedoch nicht der Fall:

```

1707 int getRating() {
1708     return (moreThanFiveLateDeliveries()) ? 2 : 1;
1709 }
1710 boolean moreThanFiveLateDeliveries() {
1711     return _numberOfLateDeliveries > 5;
1712 }

```

ist nicht besser lesbar als

```

1713 int getRating() {
1714     return (_numberOfLateDeliveries > 5) ? 2 : 1;
1715 }

```

Hier ist der Name der Methode nicht aussagekräftiger als deren Implementierung. Allerdings hätte man nach obiger Argumentation die Methode auch besser `tooManyLateDeliveries()` anstelle von `moreThanFiveLateDeliveries()` genannt.

5.2.5.2 Methode in Methodenobjekt auslagern (REPLACE METHOD WITH METHOD OBJECT)

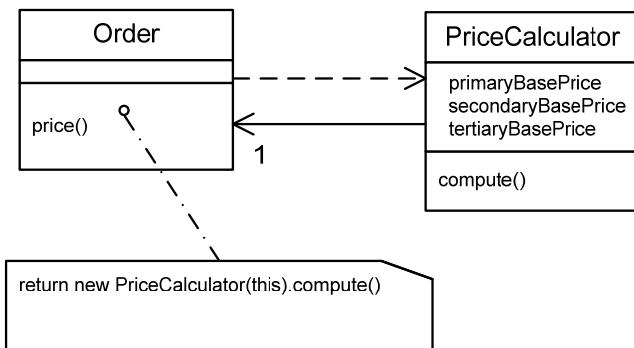
Wenn eine Methode sehr lang ist und dabei vielleicht auch noch so komplex, dass sie den Charakter einer eigenständigen Einheit hat (also insbesondere keine oder nur wenige andere Methoden aufruft), dann kann es sinnvoll sein, sie in eine eigene Klasse auszulagern. Auch wenn diese neue Klasse von keiner anderen als der, aus der sie erzeugt wurde, benutzt wird, ergibt sich daraus eine gewisse Erhöhung der Les- und Wartbarkeit. Aus

```

1716 class Order...
1717     double price() {
1718         double primaryBasePrice;
1719         double secondaryBasePrice;
1720         double tertiaryBasePrice;
1721         // long computation;
1722         ...
1723     }

```

wird beispielsweise



Ein anderer Grund, eine Methodenklasse zu erzeugen, liegt darin, dass die lokalen (temporären) Variablen einer Methode es nicht erlauben, diese per „Methode extrahieren“ in kleinere zu zerlegen. Die Methodenklasse deklariert dann alle temporären Variablen der Methode als Felder, die Variablen sind damit global zu allen Methoden der Klasse und müssen nicht als Parameter an diese übergeben werden. Initialisiert werden die Variablen durch den Konstruktor der Klasse, der mit den Parametern der Originalmethode aufgerufen wird. Für Zugriffe auf Methoden und Felder des Objektes, dem die Methode ursprünglich gehörte, muss noch ein weiteres Feld in der neuen Klasse eingerichtet werden, das dieses Objekt (beim Konstruktorauftrag ebenfalls übergeben) enthält.

So wird beispielsweise aus

```

1724 class Account {
1725     ...
1726     int gamma (int inputVal, int quantity, int yearToDate) {
1727         int importantValue1 = (inputVal * quantity) + delta();
1728         int importantValue2 = (inputVal * yearToDate) + 100;
1729         if ((yearToDate - importantValue1) > 100)
1730             importantValue2 -= 20;
1731         int importantValue3 = importantValue2 * 7;
1732         return importantValue3 - 2 * importantValue1;
1733     }
1734 }
```

die Methodenklasse

```

1735 class Gamma {
1736     private final Account account;
1737     private int inputVal;
1738     private int quantity;
1739     private int yearToDate;
1740     private int importantValue1;
1741     private int importantValue2;
1742     private int importantValue3;
1743
1744     Gamma(Account source, int inputValArg,
1745           int quantityArg, int yearToDateArg) {
```

```

1745     account = source;
1746     inputVal = inputValArg;
1747     quantity = quantityArg;
1748     yearToDate = yearToDateArg;
1749 }

1750 int compute() {
1751     importantValue1 = (inputVal * quantity) + account.delta();
1752     importantValue2 = (inputVal * yearToDate) + 100;
1753     if ((yearToDate - importantValue1) > 100)
1754         importantValue2 -= 20;
1755     int importantValue3 = importantValue2 * 7;
1756     return importantValue3 - 2 * importantValue1;
1757 }
1758 }
```

Die Originalmethode kann damit zu

```

1759 int gamma (int inputVal, int quantity, int yearToDate) {
1760     return new Gamma(this, inputVal, quantity,
1761                     yearToDate).compute();
1761 }
```

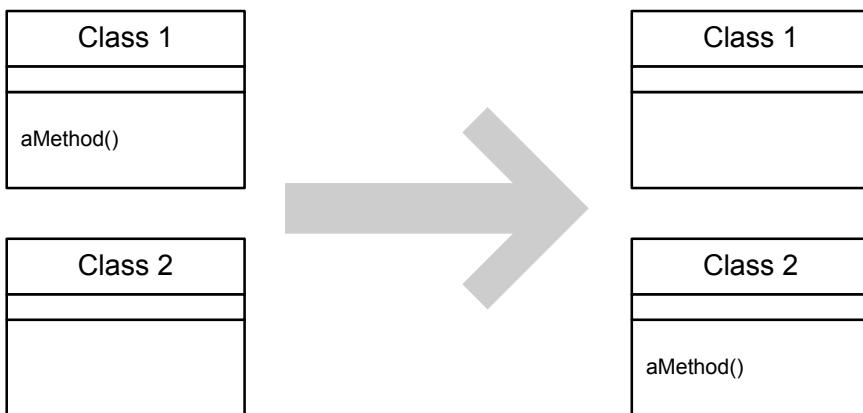
vereinfacht werden. (Versuchen Sie bitte nicht, dem Beispiel einen anderen Sinn als den der Demonstration zu geben.)

Methodenklassen decken Schwäche von JAVA auf

Dieses Refactoring mag in manchen Fällen sinnvoll sein, es stellt aber in gewisser Weise die Idee der objektorientierten Programmierung in Frage. Instanzen einer Methodenklaasse haben nämlich keine aus der Problemstellung (Domäne) heraus motivierte Existenzberechtigung — sie dienen lediglich der besseren Lesbarkeit eines Programms. Wenn eine Programmiersprache aber erforderlich macht, dass zur Laufzeit Dinge geschehen, die lediglich der besseren Lesbarkeit der Quelle dienen, dann ist dies ein starkes Indiz für Fehler (oder zumindest Schwächen) im Sprachentwurf. Im gegebenen Fall könnte eine Lösung z. B. so aussehen, dass JAVA geschachtelte Methodendeklarationen (also Methoden innerhalb von Methoden wie etwa in Pascal) erlaubt, wobei die formalen Parameter der äußeren Methoden globale Variablen aus Sicht der inneren Methoden wären. Das Problem der schlechten Lesbarkeit aufgrund der resultierenden Länge der (äußeren) Methode könnte durch eine geeignete Editorunterstützung (Zoom in/out) gelöst werden.

5.2.5.3 Feld oder Methode verlagern (MOVE FIELD/METHOD)

In welcher Klasse eine Methode angesiedelt ist, sollte eigentlich immer davon abhängen, wo der *Objektzustand* anzufinden ist (also wo die den Zustand haltenden Instanzvariablen deklariert sind), auf den diese Methode Zugriff haben muss. Wenn eine Methode gehäuft den Zustand von Objekten abfragt oder manipuliert, die zu einer anderen Klasse gehören, dann ist dies ein starker Indikator dafür, dass die Methode in die andere Klasse verlagert werden sollte, wo sie dann direkten Zugriff auf den Zustand hat. Dies ergibt sich aus dem Prinzip der objektorientierten Programmierung, nach dem Operationen den Zustand eines Objektes kapseln.



Beispiel

Im nachfolgenden Beispiel wird in der Methode `participate` der Klasse `Person` mehrheitlich auf den Zustand des Parameterobjekts `p` (vom Typ `Project`) zugegriffen:

```

1762 class Project {
1763     Person[] participants;
1764 }

1765 class Person {
1766     int id;

1767     boolean participate(Project p) {
1768         for (int i=0; i<p.participants.length; i++) {
1769             if (p.participants[i].id == id) return true;
1770         }
1771         return false;
1772     }
1773 }
  
```

Dies ist ein guter Indikator dafür, dass die Methode `participate` eigentlich in die Klasse `Project` gehört:

```

1774 class Project {
1775     Person[] participants;

1776     boolean participate(Person x) {
1777         for (int i=0; i<participants.length; i++) {
1778             if (participants[i].id == x.id) return true;
1779         }
1780         return false;
1781     }
1782 }

1783 class Person {
1784     int id;
1785 }
```

Die Verlagerung einer Methode von einer Klasse (die Klasse des ursprünglichen Empfängerobjekts) in eine andere (der Klasse des ursprünglichen Parameterobjekts) führt zu einer Vertauschung von Empfänger und Parameter. Sprachlich entspricht dies der Umformung eines Satzes vom Aktiv ins Passiv (oder umgekehrt). Wenn dies der Fall ist, sollte sich das in einer entsprechenden Umbenennung der Methode widerspiegeln. So sollte beispielsweise aus

```
1786 Drucker.drucken(Dokument)
```

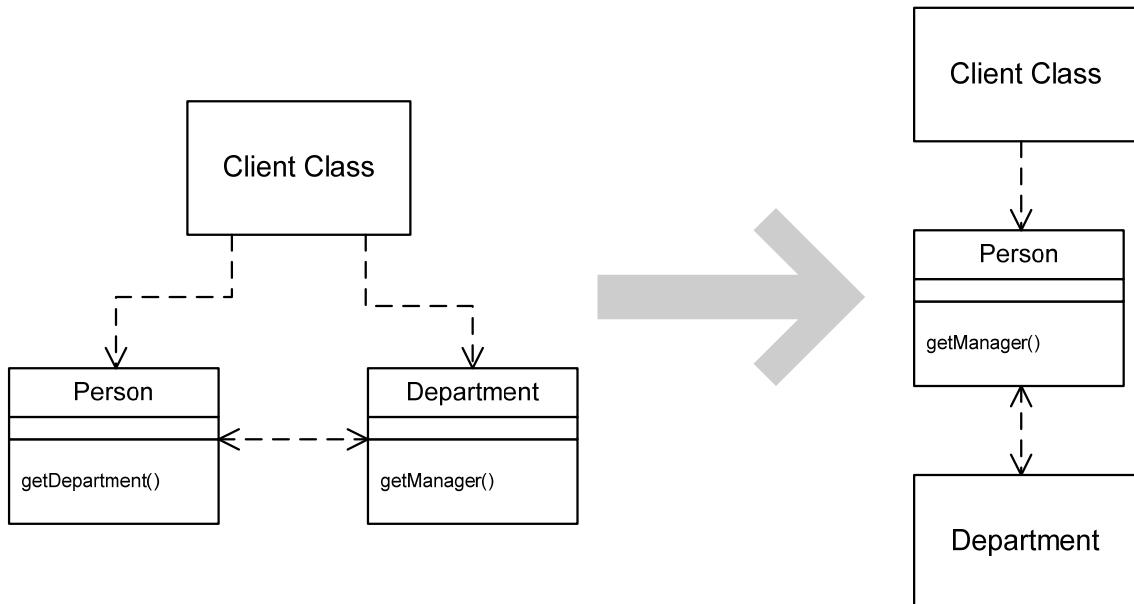
besser

```
1787 Dokument.gedrucktDurch(Drucker)
```

werden. Auf obiges Beispiel übertragen hieße das, dass man die Methoden in **participates** bzw. **participatesIn** umbenennen sollte.

5.2.5.4 Delegat verbergen (HIDE DELEGATE)

Häufig werden von einer Klientin Methoden auf Objekten aufgerufen, die die Klientin nicht selbst kennt, auf die sie also selbst keine Referenz hat. Dies geschieht dann durch verkettete Methodenaufrufe, wobei der jeweils erste Methodenaufruf das Objekt zurückgibt, auf dem die zweite Methode aufgerufen werden soll. Das ist insbesondere dann problematisch, wenn das Objekt, das der erste Methodenaufruf zurückgibt, eigentlich ein Geheimnis des liefernden Objekts (also des Empfängers der ersten Methode) sein sollte. Die Durchbrechung des Geheimnisprinzips mag zwar für einen gegebenen Fall des verketteten Methodenaufrufs vertretbar sein — die erste Methode, die das Geheimnis preisgibt, steht aber nun allgemein zur Verfügung und kann deswegen auch zur Verletzung missbraucht werden.



```

1788 class Person {
1789     Department department;
1790
1791     public Department getDepartment() {
1792         return department;
1793     }
1794
1795     public void setDepartment(Department arg) {
1796         department = arg;
1797     }
1798 }
1799
1800 class Department {
1801     private Person manager;
1802
1803     public Department(Person arg) {
1804         manager = arg;
1805     }
1806
1807     public Person getManager() {
1808         return manager;
1809     }
1810 }
  
```

Wenn man nun die Managerin einer Person `john` erfragen möchte, dann geht das mittels

```
1806     manager = john.getDepartment().getManager();
```

Damit wird aber von Klienten von `Person` verlangt, zu wissen, wie der Aufbau der Firma ist, dass man erst das Department erfragen muss und dieses

Struktur hinter der Schnittstelle einer Klasse verbergen

dann nach seiner Managerin (die dann zugleich die von John ist). Solches Wissen verkörpert aber Entwurfsentscheidungen, die sich im Laufe der Zeit ändern können, was dann auch unmittelbare Auswirkungen auf die Klientinnen von **Person** hat und das selbst dann, wenn sich am Inhalt der Anfrage (also dem Erfragen der Managerin) gar nichts ändert. Das ist nicht gut.

Die (refaktorierte) Lösung fügt einfach die folgende Methode

```
1807 public Person getManager() {
1808     return department.getManager();
1809 }
```

zur Klasse **Person** hinzu. Die Kopplung zwischen den Klientinnen von **Person** und **Department** wird damit entfernt:

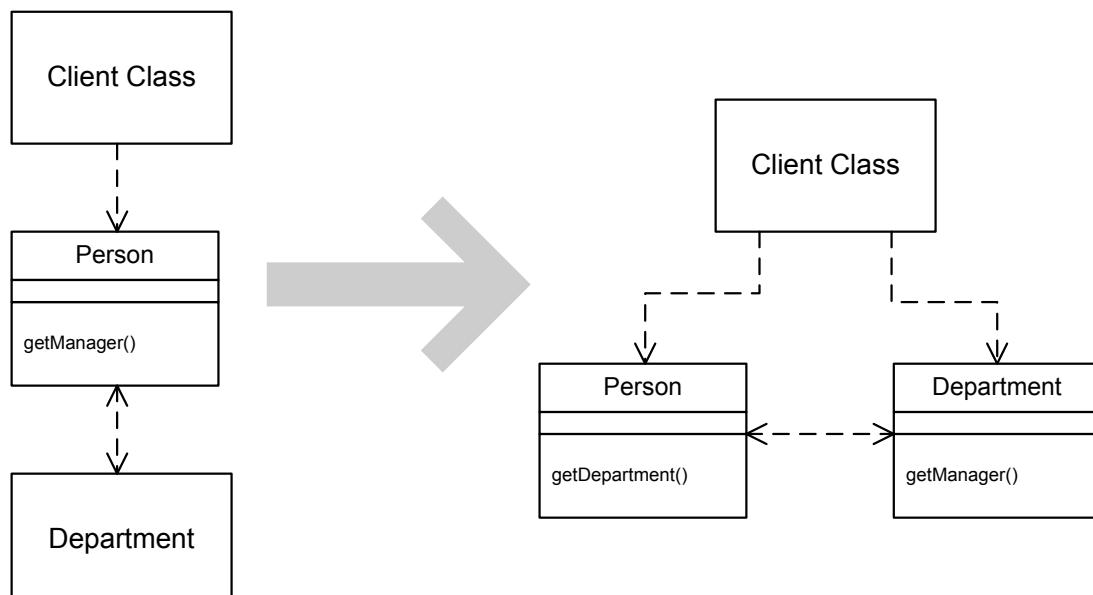
```
1810 manager = john.getManager();
```

Gesetz Demeters

Der Code ist somit leichter änderbar; die Methode `getDepartment()` kann aus **Person** entfernt werden, um das Geheimnis des Aufbaus der Firma vollkommen hinter dem Klasseninterface von **Person** zu verbergen. Das Entwurfsprinzip, das hinter diesem Refactoring steckt, ist unter dem Namen *Gesetz Demeters* (engl. *Law of Demeter*) bekannt (s. Kurs 01814).

5.2.5.5 Mittelsmann entfernen (REMOVE MIDDLEMAN)

Wenn auch inhaltlich nicht ganz, so ist zumindest formal die Umkehrung von „Delegat verbergen“ „Mittelsmann entfernen“: Anstatt eine Methode indirekt, also über eine Vermittlerin (die den Aufruf weiterdelegiert), aufzurufen, ruft die Klientin sie direkt auf dem implementierenden Objekt auf. Dies ist dann keine Verletzung des Gesetzes Demeters, wenn die Klientin das betreffende Objekt selbst kennt, es also nicht zuvor von der Vermittlerin anfragen muss.



5.2.5.6 Klassenfremde Methode einführen (INTRODUCE FOREIGN METHOD)

Manchmal fehlt einer Bibliotheksklasse, die man nicht selbst entwickelt hat bzw. deren Quellcode einem nicht zur Verfügung steht, eine Funktion, die man gern hätte. Da es in solchen Fällen nicht möglich ist, die Klasse um die fehlende Funktion zu erweitern, muss man sich irgendwie anders behelfen. Keine gute Lösung wäre es, die Funktion als eine Instanzmethode der Objekte vorzusehen, die sie brauchen; man würde dann sofort das Methode-verlagern-Refactoring anwenden wollen.

Was also tun? Man kann die neue Funktion als Klassenmethode (in JAVA, C# und C++: **static** deklariert) in der Klasse, deren Objekte sie benötigen, einführen und das Empfängerobjekt, an dessen Klasse man nicht herankommt, zu einem (dem ersten) Parameter dieser Methode machen. Statt

```
1811 Date newStart = new Date (previousEnd.getYear(), previousEnd.getMonth(), previousEnd.getDate() + 1);
```

hätte man dann beispielsweise

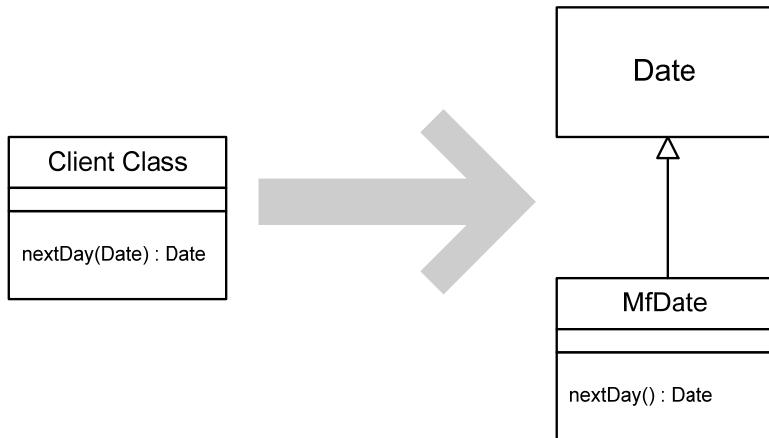
```
1812 // klassenfremde Methode - gehört eigentlich in Date
1813 private static Date nextDay(Date arg) {
1814     return new Date (arg.getYear(), arg.getMonth(),
1815                     arg.getDate() + 1);
1816 Date newStart = nextDay(previousEnd);
```

Der Kommentar dient als Erinnerung daran, dass es sich hier um ein Workaround handelt. Ähnlich wie das TODO-Tag kann er verwendet werden, um sich zu einem späteren Zeitpunkt des Problems noch einmal anzunehmen, z. B. nach dem man sich den fehlenden Zugriff verschafft hat.

Die eigentlich klassenfremde Methode zu einer Klassenmethode zu machen ist dann sinnvoll, wenn die Methode ausschließlich von Objekten in der einen Klasse benötigt wird. In allen anderen Fällen wäre es besser, eine eigenständige Klasse zu definieren, die als Behälter dieser und vielleicht noch anderer Hilfsfunktionen dient.

5.2.5.7 Lokale Erweiterung einführen (INTRODUCE LOCAL EXTENSION)

Eine Alternative zum vorigen Refactoring ist es, eine neue Klasse von der Klasse, die man nicht ändern kann, abzuleiten und dann für die eigenen Zwecke Instanzen dieser (lokalen) Erweiterung zu verwenden.



Anmerkung: Dies ist ein klassisches Beispiel für die *Wiederverwendung durch Vererbung*: Anstatt sich seine eigene Datumsklasse aufzubauen, nimmt man die vorhandene und erweitert sie um die fehlenden Methoden. Solange man nicht versucht, bestehende Methoden zu überschreiben, kann dabei auch nicht besonders viel schiefgehen.

5.3 Zusammenfassung und Ausblick

Refactorings sollten im Laufe einer Entwicklung (des Programmierens) regelmäßig durchgeführt werden. Sie unterstützen in hohem Maße eine wenig planvolle Vorgehensweise bei der Programmierung und stellen somit einen gar nicht hoch genug einzuschätzenden Beitrag dazu dar, die Qualität „real existierender Softwareentwicklung“ zu verbessern.

Für die Zukunft wäre es wünschenswert, dass sich neue Refactorings ohne großen Aufwand implementieren (das heißt im Wesentlichen als Automatismus in eine Entwicklungsumgebung integrieren) lassen. Hierzu wäre zweierlei notwendig:

1. Eine für alle Entwicklungsumgebungen (und Programmiersprachen) einheitliche Form des abstrakten Syntaxbaums, der alle für jedes mögliche Refactoring notwendigen Informationen bereithält.
2. Eine formale Sprache, mit der sich Refactorings als Verkettungen von Operationen auf dem obengenannten Syntaxbaum ausdrücken lassen.

Idealerweise wäre der Syntaxbaum auch noch sprachunabhängig, so dass ein einmal entwickeltes Refactoring ohne weiteren Aufwand auf andere Sprachen übertragen werden könnte (so denn die vom Refactoring betroffenen Sprachkonstrukte in den anderen Sprachen auch vorhanden sind). Bis dahin ist es aber noch ein weiter Weg.

5.4 Weiterführende Literatur

Refactorings sind ein sehr pragmatisches Thema, dessen theoretische Grundlagen dennoch anspruchsvoll sind. Bislang ist es allerdings nicht Gegenstand allzu vieler Forschungsarbeiten geworden, was vermutlich daran liegt, dass die damit verbundenen Probleme und Lösungen mehr oder weniger bekannt sind und man lediglich in Detailfragen noch einen Erkenntnisgewinn erzielen kann. Der Klassiker ist sicher das Buch von Fowler [40] (der einmal mehr den richtigen Reicher gehabt hat); die darin beschriebenen Refactorings, von denen ein Gutteil hier (inklusive der Beispiele) wiedergegeben werden, sind zum Teil jedoch nur recht oberflächlich beschrieben und werden sich in der Praxis nicht immer so (leicht) durchführen lassen. Dass das tatsächlich so ist, wurde in [41] klar gezeigt.

Interessant sind sicher noch die hier ausgelassenen sog. *großen Refactorings* (ebenfalls in [40] beschrieben) sowie der systematische Einsatz von Refactorings zur Verwendung von Entwurfsmustern im Code (*Refactoring to patterns*) [43] (wobei letzteres Werk wohl erst noch einer zweiten Ausgabe bedarf). Ein einsichtsreiches Buch ist auch [44], das allerdings, da es sich auf SMALLTALK bezieht, nur für wirklich Interessierte (und Fans von Kent Beck) sinnvoll zu lesen ist.

Das sehr mächtige Refactoring INFER TYPE, das Grundlage weiterer Refaktorisierungswerzeuge wie REPLACE INHERITANCE WITH DELEGATION ist, wurde in [45] und [46] ausführlich beschrieben.

- [39] W Opdyke Refactoring Object-Oriented Frameworks PhD thesis (Urbana-Champaign, IL, USA, 1992).
- [40] M Fowler Refactorings — Improving the Design of Existing Code (Addison-Wesley, 1999).
- [41] H Kegel, F Steimann „Systematically replacing inheritance with delegation in Java“ in: International Conference on Software Engineering (2008) 431–440.
- [42] E Gamma, R Helm, R Johnson, J Vlissides Design Patterns — Elements of Reusable Software (Addison-Wesley, 1995). Dublette zu [1].
- [43] J Kerievsky Refactoring to Patterns (Addison Wesley Professional, 2004).
- [44] K Beck Smalltalk Best Practice Patterns (Prentice Hall 1996).
- [45] F Steimann „The Infer Type refactoring and its use for interface-based programming“, in: Journal of Object Technology 6:2 (2007) 67–89.
- [46] H Kegel „Constraint-basierte Typinferenz für Java 5“ (Diplomarbeit, Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, 2007).

5.5 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 5.1 (Seite 171)

In der Regel auch der Dateiname, in der die Klasse gespeichert ist(!), sowie alle Verweise auf die Klasse, also alle Deklarationen, alle **extends** Klauseln, und alle Type casts, in denen die Klasse vorkommt. Dazu kommt auch noch die Verwendung des Namens bei anderen Namen (wenn z. B. eine Variable genauso heißen soll wie die Klasse), in Kommentaren und in introspektiven Ausdrücken (s. Abschnitt 6.1.2). Die Anpassung in den drei letztgenannten Fällen kann jedoch nur rudimentär automatisiert werden.

Hatten Sie all das auf der Liste? Selbst einfache Refactorings können extrem viel Aufwand sparen!

Selbsttestaufgabe 5.2 (Seite 185)

Eine neue Klasse hinzuzufügen reicht i. a. nicht, da von ihr ja auch Instanzen erzeugt und der Variable, auf der das dynamische Binden erfolgt, zugewiesen werden müssen. Nur wenn die Instanziierung automatisch erfolgt (z. B. durch Dependency injection oder eine Factory-Methode mit reflektiven Fähigkeiten; s. Abschnitte 1.6, 4.4.6 sowie 6.1), muss am Code nichts (im Falle der Dependency injection aber wohl an der Konfigurationsdatei) geändert werden.

Selbsttestaufgabe 5.3 (Seite 207)

Das Problem ist, dass wenn man die Methode verlagert, man dennoch eine Referenz auf Instanzen der ursprünglich beherbergenden Klasse mitnehmen muss. Dass Problem wird also mit dem Methode verlagern nur verlagert, nicht gelöst.

Eine Lösung, die ohne Über-Kreuz-Zugriffe auskommt, wäre, die Relation selbst als Klasse zu implementieren und Änderungen an den Beziehungen zwischen Objekten durch diese Klasse durchführen zu lassen.

Selbsttestaufgabe 5.4 (Seite 217)

```

1817  class Person {
1818      private final String type;
1819
1820      private Person (String type) {// private deklariert!
1821          this.type = type;
1822      }
1823
1824      static Person createMale() {// Factory
1825          return new Person("Male");

```

```
1824    }

1825    static Person createFemale() { // Factory
1826        return new Person("Female");
1827    }

1828    static Person createPersonOfType(String type) {
1829        return new Person(type);
1830    }

1831    boolean isType(String type) {
1832        return this.type.equals(type);
1833    }

1834    String getType() {
1835        return type;
1836    }
1837 }
```



6 Metaprogrammierung

Von Metaprogrammierung spricht man, wenn ein Programm, das Metaprogramm, ein anderes (oder sich selbst) oder seine Ausführung beobachtet oder sogar manipuliert. Metaprogrammierung ist rekursiv anwendbar, was so viel heißt wie dass Metaprogramme selbst der Metaprogrammierung unterliegen können.

Zu den bekanntesten Metaprogrammen zählen Programmierwerkzeuge wie Compiler, Interpreter, Optimierer, Werkzeuge zur Programmanalyse (Metrikwerkzeuge, Kontrollflussanalyse etc.) sowie Refactoring-Werkzeuge und Debugger. All diese Programme haben andere Programme zum Gegenstand — sie helfen sie zu erzeugen, zu analysieren und zu korrigieren. Tatsächlich wäre ohne Metaprogrammierung die Programmierung heute undenkbar, nur nehmen wir Metaprogramme in der Regel nicht als solche war.

Zu den interessantesten Metaprogrammen gehören solche, die auf sich selbst angewendet werden. Besonders umstritten ist dabei sog. selbstmodifizierender Code, da er in gewisser Weise zu einer Verselbständigung der Programmierung führt, die für eine Leserin des Programms nur noch sehr schwer nachzuvollziehen ist. Entsprechend sind auch sämtliche Möglichkeiten zur Verifikation von selbstmodifizierendem Code (Type checking etc.) stark eingeschränkt. Das Hauptanwendungsgebiet dieser Form der Metaprogrammierung ist daher vermutlich die Künstliche Intelligenz (KI).

Voraussetzung der Metaprogrammierung

Eine wichtige Voraussetzung der Metaprogrammierung ist, dass Programme selbst die Form von Daten haben. Dies ist trivialer Weise bei Quellcode immer der Fall: Ein Programm kann als Text eingelesen und in der Folge analysiert bzw. manipuliert werden. Voraussetzung ist dann allerdings, dass man aus dem Programm heraus Zugriff auf einen Compiler bzw. Interpreter hat, was nicht immer der Fall ist (es sei denn, beim Metaprogramm handelt es sich selbst um einen Compiler bzw. Interpreter). Wenn man diese erst nachbilden muss, wird die Metaprogrammierung zu einer recht beschwerlichen Angelegenheit.

Einfacher und direkter ist die Metaprogrammierung dann, wenn sich ein ausführbares Programm syntaktisch nicht von seinen Daten unterscheidet. Dies ist zum Beispiel in den Programmiersprachen LISP und PROLOG der Fall. Beide verwenden einheitlich Listen bzw. Terme, um sowohl Programme als auch Daten zu repräsentieren. Da die Möglichkeit besteht, in Variablen

gespeicherte Listen bzw. Terme zur Ausführung zu bringen, ist die Metaprogrammierung diesen Sprachen inhärent.

Die Metaprogrammierung ist so alt wie die Programmierung selbst. In einer frühen Auseinandersetzung zwischen Alan Turing und John von Neumann ging es darum, ob man eine bedingte Verzweigung als Maschineninstruktion benötigt. Turing vertrat dabei die Ansicht, dass man die von einer Bedingung abhängige Sprungadresse auch berechnen und hinter der Sprunganweisung in den Programmspeicher schreiben könne. Turing wollte also ausnutzen, dass im Maschinencode Daten und Programme das gleiche sind, nämlich einfach Zahlen. Heute ist man allerdings verbreitet der Ansicht, dass es sich zumindest beim schreibenden Zugriff eines Programms auf den Speicherbereich, in dem es abgelegt ist, um einen Programmierfehler (oder Angriff) handelt, und ist bemüht, diesen so früh wie möglich (durch das Betriebssystem oder gar durch eine physikalische Segmentierung des Speichers in Programm- und Datenbereiche) zu verhindern. Tatsächlich waren zumindest früher nicht wenige der sog. Systemabstürze darauf zurückzuführen, dass ein Programm statt in den Daten- in den Programmberreich schrieb und damit in der Regel unsinnige Anweisungen vorgab.

Ursprung der Metaprogrammierung

Neben den eben schon geschilderten Standardanwendungen der Metaprogrammierung, den Programmierwerkzeugen, hat sie noch weitere interessante Einsatzgebiete. Da sind zum einen Spracherweiterungen bzw. -anpassungen, die immer dann gewünscht werden, wenn einer verwendeten Programmiersprache für ein bestimmtes Anwendungsproblem geeignete Abstraktionen fehlen. Anstatt die Sprache selbst (und damit alle daran hängenden Werkzeuge) zu ändern, erlauben Sprachen mit Möglichkeit zur Metaprogrammierung, die neuen Sprachkonstrukte als eine Art Bibliotheksfunktionen hinzuzufügen. Unter den Sprachen, die in der kommerziellen Programmierung eingesetzt werden, seien hier insbesondere C++ und SMALLTALK genannt, wobei erstere das Problem der Spracherweiterung auf den Präprozessor abwälzt, während letztere, von wenigen Primitiven abgesehen, ganz in sich selbst definiert ist (und daher ohne die Metaprogrammierung gar nicht wiederzuerkennen wäre; vgl. Kurs 01814).

Einsatzmöglichkeiten der Metaprogrammierung

Zum anderen kommt die Metaprogrammierung im Kontext dynamisch konfigurierter Systeme und Dienste zum Einsatz. Überall dort, wo die Schnittstellen der zu verknüpfenden Komponenten zur Übersetzungs- (bzw. Binde-) Zeit nicht bekannt sind, kann auch kein Zueinanderpassen dieser Schnittstellen gewährleistet werden. Mit den Mitteln der Metaprogrammierung ist es jedoch möglich, die Schnittstellen zu inspizieren und bei Bedarf Adapter zur Verfügung zu stellen, die zwischen den Komponenten vermitteln.

Nicht zuletzt ist die bereits erwähnte Künstliche Intelligenz ein klassisches Anwendungsgebiet der Metaprogrammierung. Regelbasierte Systeme können selbst neue Regeln dazulernen — wie dies zu geschehen hat, kann wieder in Regeln ausgedrückt werden. In der sog. *genetischen Programmierung* werden bestehende Programme nach bestimmten Regeln, teilweise zufällig, verändert (mutiert) und zur Ausführung gebracht. Das Ergebnis der Ausführung wird mit dem erwarteten Ziel verglichen; das betreffende Programm bleibt abhängig davon weiter in der Population oder wird ausgesondert (Selektion). Sowohl für die Mutation als auch für die Selektion ist Metaprogrammierung notwendig.

6.1 Metaprogrammierung auf sich selbst: Reflektion

In der Programmierung versteht man unter Reflektion die Fähigkeit eines Programms, sich selbst zu (er)kennen und zu behandeln. Dazu enthält ein reflektives Programm seinen eigenen Interpreter oder zumindest Teile davon. Ein reflektives Programm ist also ein besonderes Metaprogramm, nämlich eines, bei dem Metaprogramm und Programm eins sind.

Ein vollständig reflektives System besitzt die Fähigkeit, seinen eigenen Aufbau und seine eigene Ausführung zu beobachten (die sog. *Introspektion*), seinen Ablauf zu beeinflussen (die sog. *Interzeption* oder *Interzession*) sowie ganz allgemein seine Bestandteile zu ändern, zu ergänzen oder auszutauschen (die *Modifikation*). So kann ein vollständig reflektives System beispielsweise Methodenaufrufe abfangen und bei Bedarf deren Ausführung verhindern oder auf andere, auch neue, Methoden umleiten.

6.1.1 Reflektieren ohne zu verändern: Introspektion

Weniger mächtig, aber einfacher zu realisieren als die allgemeine Reflektion ist die sog. Introspektion, also die Reflektion ohne die Möglichkeit der Interzession und Modifikation. Sie ist beispielsweise in JAVA umgesetzt. Introspektion wird oft mit Reflektion gleichgesetzt — so heißt beispielsweise das introspektive Sprachpaket JAVAs irreführenderweise `java.lang.reflect`.

Bei der Introspektion hat ein Programm die Möglichkeit, über seinen eigenen Aufbau zu reflektieren, ohne ihn jedoch verändern zu können. Das Programm weiß also beispielsweise, aus welchen Klassen es besteht, welche Methoden in diesen Klassen implementiert sind, welche Typen deren Parameter haben usw. So ermöglicht es die Introspektion einem Programm beispielsweise, sich ohne Zugriff auf den Quellcode zumindest teilweise selbst auszudrucken. Ein anderes Anwendungsbeispiel verlangt, dass eine Schnittstelle selbst darüber Auskunft gibt, welche Funktionen sie anbietet. Diese Selbstauskunft kann auf einfache Weise über Introspektion realisiert werden.

Indirekter Zugriff

Wirklich interessant wird die Introspektion jedoch erst, wenn sie nicht nur über Programmelemente Auskunft geben, sondern auch auf diese zugreifen oder diese zur Ausführung bringen kann. So wird es beispielsweise möglich, den Wert eines durch einen String benannten Feldes auszulesen oder zu verändern oder auch eine Methode nur anhand ihres Namens (wiederum als String vorliegend) auszuwählen und aufzurufen. Man beachte, dass dabei das vorliegende Programm und insbesondere der durch seine Strukturen vorgegebene Kontrollfluss nicht verändert werden. Im Falle eines solchen Methodenaufrufs ist lediglich aus dem Programmtext nicht ersichtlich, welche Methode aufgerufen wird; die aufgerufene Methode selbst muss jedoch existieren (und somit schon zur Übersetzungszeit existiert haben, ohne dass dieses jedoch geprüft werden könnte). Man kann allerdings mit Introspektion neue Kombinationen von bereits bestehenden Elementen erzeugen, also Systeme dynamisch konfigurieren.

6.1.2 Introspektion in JAVA

Um JAVA zu einer introspektiven Sprache zu machen, wurden der Klassenbibliothek des JDK Klassen hinzugefügt, die die Elemente eines Programms repräsentieren. So stehen Instanzen der Klasse **Class** für die Klassen eines Programms, Instanzen der Klasse **Field** für die Felder (Klassen- und Instanzvariablen) und Instanzen der Klasse **Method** für Methoden. Zusätzlich gibt es noch Klassen für Arrays (**Array**), für Konstruktoren (**Constructor**) sowie für die Zugriffsmodifizierer **public**, **private** etc. (**Modifier**). Nicht zuletzt besitzt die Klasse **Object** mit ihrer Methode **getClass()** einen introspektiven Anteil. Und schließlich ist auch die Tatsache, dass in JAVA ein Objekt seinen Typ kennt, in die Syntax der Sprache fest eingebaut: Der Operator **instanceof** erlaubt, zu testen, ob ein gegebenes Objekt Instanz einer bestimmten Klasse ist. Wenn im Programmtext selbst eine Klasse als Literal (also als ein Wert) auftauchen soll, dann schreibt man **<klassename>.class** (nicht jedoch bei **instanceof**). Tabelle 6.1 gibt einen Überblick.

Tabelle 6.1: JAVA-Klassen mit Beteiligung am Reflection API.

KLASSE	BESCHREIBUNG
Array	Erzeugung von und Zugriff auf Arrays variablen Typs
Class	Instanzen repräsentieren Klassen und Interfaces
Constructor	Aufruf von Konstruktoren variablen Typs
Field	Information über und Zugriff auf Felder variabler Typen
Method	Information über und Aufruf von Methoden variabler Typen
Modifier	Information über Access Modifier von Typen und Methoden
Object	getClass() Methode
-	instanceof testet den Typ eines Objekts
-	.class liefert das die Klasse repräsentierende Objekt

Beispiel

Nachfolgender Code zeigt, wie man in JAVA Objekte beliebiger, von **ReflectiveObject** abgeleiteter Klassen serialisieren, d. h. auf einem Stream ausgeben kann:

```

1838 class ReflectiveObject {
1839     void serializeOn(PrintStream aStream) {
1840         Class myClass = getClass();
1841         aStream.println(myClass.getName() + "(");
1842         for (Field field : myClass.getFields()) {
1843             aStream.print(field.getName() + " == ");
1844             try {
1845                 Object value = field.get(this);
1846                 if (field.getType().isPrimitive() || value == null)

```

```

1847         aStream.println(value);
1848     else
1849         ((ReflectiveObject) value).serializeOn(aStream);
1850     }
1851     catch ...
1852     }
1853     aStream.println(")");
1854   }
1855 }
```

Introspektive Zugriffe auf Programmteile entziehen sich der syntaktischen Prüfung durch den Compiler. Daher können Laufzeitfehler, die auf eine mangelnde Bindung von Namen zurückzuführen sind, nicht durch den Compiler ausgeschlossen werden. Introspektive Aufrufe müssen also immer durch Try-catch-Blöcke gesichert werden:

```

1856 try {
1857     o.getClass().getMethod("hello",new Class[] {String.class}).
1858         invoke(o, new Object[] {"world"});
1859     catch(NoSuchMethodException ex) {
1860         throw new IllegalArgumentException("Methode nicht gefunden");
1861     } catch (IllegalAccessException ex) {
1862         throw new Exception("Methode nicht zugreifbar");
1863     } catch (InvocationTargetException ex) {
1864         throw ex.getTargetException(); //Methode hat Exception ausgelöst
1865     }
```

Problem Refactoring

Des Weiteren werden introspektive Abfragen und Aufrufe nicht durch Refactoring-Werkzeuge erfasst: Das Umbenennen einer Methode geht an den Funktionen, die einen String zusammenbasteln, der eben diese Methode benennen soll, gänzlich vorbei. Vorsicht ist also auch bei der Introspektion geboten.

Die Einsatzmöglichkeiten der Introspektion sind vielseitig. Wie in Abschnitt 3.2.1 bereits vorweggenommen, macht sich z. B. JUNIT 3.8 die introspektiven Eigenschaften JAVAs zunutze, indem es Testfälle, von denen ja mehrere verschiedene als Instanzen einer Klasse realisiert werden können, über den Inhalt der Instanzvariable `fName` an jeweils genau eine Methode der Klasse bindet. Ebenso liefert die Standardimplementierung der Methode `suite()` über Introspektion für jede Methode einer Klasse, deren Name mit „test“ beginnt, einen neuen Testfall zurück.

Selbsttestaufgabe 6.1

In JAVA berücksichtigt das *dynamische Binden* eines Methodenaufrufs an eine Implementierung nur beim Empfängerobjekt den tatsächlichen (dynamischen) Typ. Bei den Parametern hingegen wird lediglich der deklarierte (statische) Typ herangezogen. Dieses Verhalten ist jedoch nicht immer erwünscht.

Versuchen Sie, mittels Introspektion eine Lösung vorzuschlagen, unter Zuhilfenahme derer das Dispatching bei Methodenaufrufen immer auf Empfänger- und Parametertypen stattfindet.

6.1.3 Interzession

Zwar erlaubt die Introspektion (Reflektion à la JAVA), auf Programmelemente indirekt, also ohne explizite Erwähnung der Elemente im Quellcode, zuzugreifen, doch bestehende, „fest verdrahtete“ Zugriffe lassen sich auf diese Weise ebenso wenig abändern wie neue Elemente einführen. Was man manchmal aber gerne hätte, ist die Möglichkeit, Methodenaufrufe oder Variablenzugriffe abzufangen und beispielsweise auf ihre Zulässigkeit zu überprüfen oder situationsbezogen umzulenken. Derartiges ist in JAVA zurzeit nur über entsprechende Entwurfsmuster (wie z. B. das *PROXY Pattern* [47]) möglich; da es sich dabei aber nicht um Metaprogrammierung im strengen Sinne handelt, werden wir an dieser Stelle auch nicht darauf eingehen. Die aspektorientierte Programmierung (Abschnitt 6.3) arbeitet aber sehr wohl auf Basis der Interzession.

6.1.4 Modifikation

Um ein Programm sich selbst um neuen Code ergänzen zu lassen, muss das Programm Zugriff auf seinen eigenen Compiler/Interpreter haben oder einen solchen extra bereitstellen (letzteres muss dann nicht unbedingt für den vollen Sprachumfang geschehen, ja kann sogar eine ganz andere Sprache verwenden). Für JAVA beispielsweise besteht die Möglichkeit, Klassen in Form von Quellcode-Dateien zu erzeugen, dann (über den Umweg eines Betriebssystemaufrufs) diese Dateien zu kompilieren und über einen Class loader zu importieren. Alternativ kann man natürlich auch direkt Bytecode erzeugen und laden (wer es mag). All das soll hier jedoch nicht weiterverfolgt werden.

6.1.5 Bewertung der Reflektion

Zweifellos bietet die Reflektion gewichtige Vorteile. Neben der Möglichkeit der Ergänzung von Funktionalität eines Programms zur Laufzeit (durch Ergänzung zusätzlicher Anweisungen oder Programmteile) ist hier vor allem die Erweiterung der Ausdrucksstärke (nicht in formalem Sinne — die Sprache bleibt natürlich Turing-äquivalent) zu nennen. So kann eine Programmiersprache mit den Mitteln der Reflektion vergleichsweise einfach um fehlende Konzepte erweitert werden. Aber auch unter theoretischen Gesichtspunkten ist die Reflektion durchaus interessant: Ohne Reflektion ließe sich zum Beispiel das Halteproblem nicht formulieren.

Diesen Vorteilen stehen aber auch einige gewichtige Nachteile gegenüber. Zum einen sind Programme, die die Reflektion verwenden (die also selbst Metaprogramme sind), schwerer lesbar.

Dies ist insbesondere dann der Fall, wenn der durch die Metaprogrammierung erstellte Code nicht im Klartext vorliegt, sondern durch das Programm eigenständig zusammengesetzt wird (was charakteristischerweise der Fall ist). Aus der schlechteren Lesbarkeit ergibt sich auch eine schlechtere Wart- und Testbarkeit. Zum anderen verwischt die Grenze zwischen Übersetzungszeit und Laufzeit: Während das Metaprogramm selbst noch zur Übersetzungszeit prüfbar ist, können syntaktische und andere Fehler in meta-, also automatisch programmierten Codeteilen, erst zur Laufzeit festgestellt werden.

6.2 Programmieren mit Metadaten: Annotationen und Attribute

Die Introspektion baut darauf, ein Programm so wie es ist auszulesen. Keinem Teil des Programms ist anzusehen, dass es speziell für die Introspektion geschrieben wurde; allenfalls das Vorhandensein von Namenskonventionen (wie etwa bei Verwendung von JUNIT) kann als Hinweis darauf gewertet werden, dass das Programm sich selbst als Eingabe dient.

Dies ist nicht ganz ungefährlich. Jemand, die von der Namenskonvention nichts weiß, ändert vielleicht Namen und zerstört damit das Programm; oder sie verwendet aus Unwissenheit selbst Namen, die unter die Konvention fallen, aber von der Metaprogrammierung gar nicht beglückt werden sollten. Implizite Vereinbarungen in einem Programm sind immer eine Gefahr.

Um den Zugriff auf Teile des Programms mit den Mitteln der Introspektion explizit zu machen, wurden zunächst in .NET sog. *Attribute*⁶¹ eingeführt, die inzwischen auch in JAVA zur Verfügung stehen und dort *Annotationen* genannt werden. Mit ihnen kann Source-Code mit zusätzlichen Informationen versehen werden, die zunächst für dessen Ausführung keine Bedeutung haben. Diese Informationen werden auch als *Metadaten* bezeichnet.

Was sind Metadaten?

Metadaten sind Daten über Daten. Ohne allzusehr ins Detail gehen zu wollen, muss man dabei aufpassen, dass man den Begriff des Metadatums nicht mit dem des Attributs (im herkömmlichen Sinne, also als Feld eines Records oder einer Klasse) verwechselt. So hat beispielsweise die Autorin eines Textes genau denselben Rang wie sein Titel oder auch sein Inhalt: Es sind alles Attribute eines Objektes vom Typ **Text** und niemand würde auf die Idee kommen, von einem oder gar allen als Metadaten zu sprechen. Nichtsdestotrotz wird **Autor** häufig als ein Metadatum angesehen, so z. B. das `@author` Javadoc tag als ein Metadatum, das eine JAVA-Klasse beschreibt, oder das `<META NAME="AUTHOR" CONTENT="xyz">` HTML tag als eines, das eine Webseite beschreibt. Metadaten sind aber nur dann meta, wenn das, was sie beschreiben, auf einer niedrigeren Ebene anzutreffen ist. Dies ist jedoch bei der Annotierung von Elementen eines Programms immer der Fall: Eine solche Annotation bezieht sich stets auf das annotierte Programmelement und nie auf das (gedachte oder realweltliche) Objekt, auf das sich das Programmelement bezieht. Den Unterschied soll folgendes Beispiel verdeutlichen:

⁶¹ Der Begriff Attribut ist hier insofern unglücklich gewählt, als er in der objektorientierten Programmierung manchmal und in der objektorientierten Modellierung immer für Felder (Instanzvariablen) steht. Die beiden Verwendungen sind aber klar voneinander zu trennen.

```
1865 /**
1866 * @author Mr. Genius
1867 */
1868 class Book {
1869     String text;
1870     String author;
1871 }
```

@author bezieht sich hier auf den nachfolgenden Programmtext, die Definition der Klasse Book, author auf ein Buch. Dabei ist @author auch keine Klassenvariable oder -konstante (obwohl die Abgrenzung dazu nicht ganz einfach ist).

Mit Hilfe von Annotationen (Metadaten) kann man also bestimmte Programmelemente mit zusätzlichen Informationen versehen, die gewissermaßen über dem Programm stehen. Konkret können in JAVA mittels Annotationen Typen (Klassen und Interfaces), Methoden, Konstruktoren, Variablen (Felder, formale Parameter und lokale Variablen), Pakete sowie Annotationstypen selbst annotiert werden. Annotiert wird immer an der Stelle der Deklaration eines Programmelements; daraus folgt, dass auch Annotationen deklariert werden müssen (denn sonst könnte man sie nicht annotieren).

6.2.1 Annotationstypen

In JAVA besteht die Deklaration einer benutzerdefinierten Annotation aus dem @-Zeichen und dem Schlüsselwort interface, gefolgt vom Namen der Annotation und einer optionalen Liste von Eigenschaften. Tatsächlich sind Annotationen (wie Interfaces) zunächst Typen; da man deren Instanzen jedoch ebenfalls Annotationen nennt, werden die Typen zur besseren Unterscheidung *Annotationstypen* genannt. Es ist mir dennoch nicht ganz klar, warum ausgerechnet das Schlüsselwort interface wiederverwendet wurde. Immerhin ist die JAVA-Sprachspezifikation insofern konsequent, als sie die Eigenschaften einer Annotation als Methoden bezeichnet, obwohl es sich eigentlich um Felder (wenn auch ohne Schreibzugriff) handelt. Auch erben alle Annotationen implizit von annotation.Annotation (einem Interface) und es kann von ihnen geerbt werden. Die Erben können jedoch selbst keine Annotationen sein.

Beispiel

Nachfolgendes Beispiel deklariert einen *Annotationstypen* TestCase, dessen Verwendung anzeigt, dass es sich bei dem annotierten Programmelement um einen Testfall handelt, und der zusätzlich über eine Eigenschaft vom Typ string verfügt, deren Wert den Namen einer Testsuite darstellen soll.

```
1872 public @interface TestCase {
1873     String testsuite();
1874 }
```

Das Schlüsselwort public gibt, wie bei Deklarationen allgemein üblich, die Sichtbarkeit der Deklaration an.

eingebaute Annotationen

Was dieser Deklaration noch fehlt, ist die Angabe dessen, was mit ihr annotiert werden kann. Zu diesem Zweck ist in JAVA eine eingebaute Annotation vorgesehen, die nur Annotationstypen annotiert (eine **Metaannotation**) und die eine Eigenschaft besitzt, mit der das oder die zu annotierenden Arten von Programmelementen festgelegt werden können. Fehlt diese Annotation bei der Deklaration einer Annotation, so wird angenommen, dass die Annotation auf alle Arten von Programmelementen angewendet werden kann. Dies ist bei Testfällen jedoch nicht der Fall. Die vollständige Deklaration sieht also wie folgt aus:

```
1875 @Target(ElementType.METHOD)
1876 public @interface TestCase {
1877     String testsuite();
1878 }
```

Die Deklaration des eingebauten *Annotationstypen Target* hingegen ist die folgende:

```
1879 @Target(ElementType.ANNOTATION_TYPE)
1880 public @interface Target {
1881     ElementType[] value();
1882 }
```

Man beachte, dass die Deklaration rekursiv (selbstbezüglich) ist.

Weitere eingebaute *Annotationstypen* sind **Retention**, **Overrides**, **Documented**, **Deprecated** und **Inherited**; man kann sich leicht vorstellen, dass diese Liste in kommenden Versionen fröhlich erweitert werden wird. Von den genannten ist hier vor allem der erste interessant: **Retention** erlaubt es, bei der Deklaration eines Annotationstypen anzugeben, bis wohin eine konkrete Annotation des damit annotierten Typs vordringen, d. h. verfügbar sein soll: auf den Quellcode beschränkt, bis ins Class file oder bis in die virtuelle Maschine. Die entsprechenden Werte heißen **SOURCE**, **CLASS** und **RUNTIME**. **Retention** ist wie **Target** eine *Metaannotation*.

Selbsttestaufgabe 6.2

Welche Retention hat **Retention**? Deklarieren Sie die Annotation in JAVA!

mögliche Typen der Eigenschaften

Die möglichen Typen der in Annotationstypen deklarierten Eigenschaften (also z. B. **testsuite()** oder **value()**) sind beschränkt. Erlaubt sind nur die primitiven Typen, Aufzählungstypen, Annotationstypen, die Typen **Class** und **String** sowie Arrays aller vorgenannten Typen.

Unter den Eigenschaften einer Annotation gibt es eine besondere mit Namen **value()**; sie erlaubt, wie wir am Beispiel **Target** oben schon und gleich noch sehen werden, eine abkürzende Schreibweise bei den konkreten Annotierungen. Außerdem sind Eigenschaften verboten, deren Signaturen Deklarationen in **Object** oder **annotation.Annotation** gleichen, da sonst die Eigenschaften mit den geerbten Methoden in Konflikt stünden.

6.2.2 Annotationsinstanzen und deren Verwendung

Kommen wir nun zur eigentlichen Annotierung. Um auszudrücken, dass eine Methode `abc()` einen Testfall einer Testsuite „XYZ“ darstellt, schreibt man:

```
1883 @TestCase(testsuite = "XYZ")
1884 public void abc() {...}
```

Die Methode `abc()` ist damit annotiert. Auswirkungen auf die Ausführung von `abc()` hat das zunächst keine.

Wozu dienen dann Annotationen? Für den Fall, dass sie der JVM zur Verfügung stehen, können sie mittels der erweiterten reflektiven Fähigkeiten von JAVA 5 abgefragt werden. Anstatt also zu prüfen, ob der Name einer Methode mit „test“ anfängt, kann man nun überprüfen, ob sie mit `TestCase` annotiert ist. Dazu wurde das Reflection API um ein Interface `AnnotatedElement` erweitert, das von den Klassen der annotierbaren Programmelemente (also `Class`, `Method` usw.) implementiert wird. Dieses Interface enthält unter anderem die Methode `getAnnotation(.)`, mit deren Hilfe sich die konkrete Annotierung (Instanz des Annotations-typs) zum Programmelement holen lässt. Im konkreten Beispiel sieht das so aus:

```
1885 Method m = ... // hier die reflektive Besorgung der Methode abc()
1886 TestCase tc = m.getAnnotation(TestCase.class);
```

Der Ausdruck `tc.testsuite()` liefert dann den Namen der Testsuite, zu der der Testfall gehören soll.

Es gibt übrigens zwei mögliche Kurzformen der Annotierung: Zum einen kann man, wie bereits oben angedeutet, den Namen der Eigenschaft weglassen, wenn diese (bei der Deklaration) `value()` genannt und keine andere angegeben wurde; zum anderen kann man, wenn bei der Deklaration des Annotationstyps keine Eigenschaften angegeben wurden, die Klammern auch weglassen. Man spricht dann von eingleitigen bzw. **Marker-Annotationen**.

abkürzende Schreibweisen

Marker-Annotationen sind ein gutes Stichwort: Die Verwendung von *Marker interfaces* wie beispielsweise `Serializable`, die man in früheren Versionen von JAVA eingesetzt hatte, um be-helfsweise Klassen zu annotieren, sollte nun eigentlich der Vergangenheit angehören. Stattdessen kann man ja nun eigenschaftslose Klassenannotierungen verwenden. Dagegen spricht jedoch die relativ umständliche Art und Weise, wie man die Annotationen zu einem Programm-element erfragen muss, und ihre mangelnde Abwärtskompatibilität.

Der Nutzen von Annotationen geht aber noch weit über den Ersatz von Namenskonventionen in der reflektiven Programmierung hinaus. So können Annotationen einen Präprozessor instruieren, der ein annotiertes Programm einliest und auf Basis der Annotationen modifiziert oder erweitert. Sich ständig wiederholende Codefragmente, die vielleicht nicht in ein Unterprogramm ausgelagert werden können oder sollen, können so automatisch in den Code eingepflegt werden. Dies bringt uns gleich zum nächsten Thema.

6.2.3 Annotationsverarbeitung zur Übersetzungszeit

Annotationen spezifizieren Informationen über Programmelemente auf Quellcodeebene. Da liegt es nahe, dass diese Informationen auch während der Übersetzungszeit berücksichtigt werden. Genau dies ermöglichen sog. *Annotation Processing Tools (APTs)*.

Was eine Annotation in einem Programm bewirken soll, kann solch ein APT natürlich nicht erkennen. Für die zu verarbeitenden Annotationen müssen daher speziell gefertigte Programme, die sog. *Annotationsprozessoren*, geliefert werden, die die notwendigen Verarbeitungsschritte durchführen. Diese Annotationsprozessoren werden dem APT übergeben, das sie dann aufruft, sobald es auf eine entsprechende Annotation stößt. Ein Annotationsprozessor kann dabei im Prinzip beliebige Aufgaben erfüllen; in der Regel wird er aber das Programm (den Quelltext) ändern oder neuen erzeugen. Ein Annotationsprozessor ist damit ein Metaprogramm.

Damit ein Annotationsprozessor das annotierte Programm sinnvoll verarbeiten kann, muss das Programm zunächst geparsert werden (so dass die Verarbeitung auf dem *abstrakten Syntaxbaum*, AST, stattfinden kann). Je nach Implementierung des APT ist dieses Parsen Teil des ganz normalen Kompilierens (und der AST ist der des Compilers) oder ein separater, vorgeschalteter Schritt. Die zweite Möglichkeit hat den Vorteil, dass der Compiler nicht geändert werden muss — er wird einfach nach der Annotationsverarbeitung aufgerufen. Der offensichtliche Nachteil ist der, dass dabei doppelte Arbeit geleistet werden muss (mehrfaches Parsen des Programms); wenn man allerdings bereit ist, gewisse Abstriche zu machen, genügt es, das APT lediglich die annotierbaren Programmelemente parsen zu lassen — in JAVA das sind das im Wesentlichen die Deklarationen.

Die Annotationsverarbeitung weckt viele neue Begehrlichkeiten. Insbesondere sog. *domänen-spezifische Sprachen* (engl. domain-specific languages, DSLs) können auf diese Weise relativ leicht angenähert werden. Dabei ist der Vorteil, dass die Syntax auf die Verwendung von Annotationen beschränkt ist (weswegen der Parser nicht angepasst werden muss), zugleich der Hauptnachteil — neue Kontrollstrukturen etwa lassen sich damit nur schwer umsetzen. Vollwertige Metasprachen wie etwa LISP oder PROLOG (und sogar SMALLTALK) sind da wesentlich flexibler. Gleichwohl werden die sich auftuenden Möglichkeiten dankbar angenommen und Erweiterungen gefordert — wie z. B. Annotierbarkeit bis hinunter zur einzelnen Anweisung. Es wird wohl noch ein paar Jahre dauern, bis die sinnvollen Möglichkeiten und Grenzen der Annotation ausgetestet sind.

6.3 Aspektorientierte Programmierung

Inzwischen sollte klargeworden sein, dass die Metaprogrammierung zwar ihre Reize hat, aber auch mit Vorsicht zu genießen ist. Wie häufig in solchen Fällen hat man versucht, Teile der Metaprogrammierung zu isolieren, die einen bestimmten Nutzen für die Programmierung haben, die aber selbst relativ wenig Schaden anrichten können. Ein Ergebnis solcher Versuche ist die sog. *aspektorientierte Programmierung*.

Bei der aspektorientierten Programmierung versucht man, bestimmte Anforderungen an ein Programm, die viele Stellen betreffen, die sich aber mit herkömmlichen Mitteln der Programm-

organisation nicht herausfaktorisieren lassen (die sog. *Crosscutting concerns*), zu isolieren und zusammen mit den sie implementierenden Funktionen an einer Stelle — in Form eines sog. *Aspekts* — zu lokalisieren. Da diese Funktionen zumindest im allgemeinen Fall an der Stelle ihres ursprünglichen Auftretens fest eingebunden sind, also insbesondere auch Zugriff auf den jeweiligen Programmkontext benötigen, muss dieser Kontext in der einen oder anderen Form an die neue Stelle mitgenommen werden. Im Gegensatz zum konventionellen Funktionsaufruf erfolgt die Spezifikation dieses Kontextes (die formalen Parameter der Funktionen) nur auf der Seite des Aspekts — an der „Aufrufstelle“ ist davon, genau wie vom Aufruf selbst, nichts zu sehen. Dies ist Segen und Fluch der aspektorientierten Programmierung zugleich.

6.3.1 Entwicklungsgeschichtliche Einordnung

Ziel der aspektorientierten Programmierung ist es, der Programmiererin zu erlauben, ihre Programme nach mehreren Gesichtspunkten gleichzeitig zu organisieren. Die aspektorientierte Programmierung steht damit in der Kontinuität früherer Anstrengungen, Programme besser zu strukturieren.

So erlauben zum Beispiel Unterprogramme, immer wieder gleiche (oder zumindest stark ähnliche) Codestrecken aus dem Programm zu isolieren und an einer Stelle zu lokalisieren. An den ursprünglichen Vorkommen dieser Codestrecken finden sich dann lediglich die Unterprogrammaufrufe, die, gemeinsam mit den formalen und aktuellen Parametern sowie der Return-Anweisung, die Verbindung zum Kontext herstellen. Der Preis für diese bessere Strukturierung ist eine Verletzung der sog. *Lokalitätseigenschaft* von Programmen: Während normalerweise chronologisch aufeinanderfolgende Anweisungen auch textuell in unmittelbarer Nachbarschaft zueinander stehen, verzweigt der Programmfluss bei Unterprogrammaufrufen temporär an eine andere Stelle. Dieser Nachteil kann allerdings leicht dadurch aufgewogen werden, dass man dem Unterprogramm einen Namen gibt, der seine Funktion hinreichend umschreibt, der also insbesondere an den Aufrufstellen des Unterprogramms dessen Beitrag hinreichend repräsentiert, so dass man beim Lesen des Programms zumindest in groben Zügen weiß, was an der Aufrufstelle passiert. Tatsächlich erlaubt die Zerlegung eines Programms in Unterprogramme eine stufenweise Abstraktion, die die Lesbarkeit erhöht und die deswegen auch zum Einsatz kommt, wenn es gar nicht um die Vermeidung redundanten Codes geht (vgl. Abschnitt 5.2.5.1).

**Bemühung um
Strukturierung von
Programmen**

Das nächste wichtige Ordnungsprinzip der Programmierung war die *Modularisierung*. Nach Parnas muss es Ziel der Modularisierung sein, jede wichtige

Entwurfsentscheidung in genau einem Modul zu kapseln [48]. Dadurch können sich Änderungen der Entwurfsentscheidung, die die Schnittstelle des Moduls unverändert lassen, nicht auf andere Module des Systems auswirken. Dabei können Module einzelne Prozeduren, aber auch ein Verband verschiedener Prozeduren und der dazugehörigen Daten, wie sie beispielsweise durch abstrakte Datentypen oder Klassen repräsentiert werden, sein. In der objektorientierten Programmierung ist die Modularisierung weitgehend mit der Klassenbildung gleichgesetzt (vgl. Abschnitt 1.1); in der komponentenbasierten Programmierung sind die Module (Komponenten) in der Regel größer.

Modularisierung

**Aspektorientierung
als nächste Stufe**

Die aspektorientierte Programmierung kritisiert an der Modularisierung, wie sie in z. B. objektorientierten Systemen durchgeführt wird, dass jedes Modul in

der Regel mehrere Aspekte eines zu realisierenden Systems berücksichtigen muss. Mit Aspekten sind dabei Dinge gemeint, die sich zwar konzeptuell isolieren lassen, die sich aber praktisch auf viele Funktionen (Module) eines Systems auswirken und dort entsprechend (in der Implementierung) wiederfinden. Wollte man einen solchen Aspekt zum Kriterium für die Organisation eines Programms machen, dann müsste man es, meistens in erheblichem Umfang, umstrukturieren. Da es in der Regel aber mehrere solche Aspekte gibt und die ursprüngliche Struktur des Programms ja auch nicht vom Himmel gefallen ist, sondern (hoffentlich) bestimmten, ernstzunehmenden Prinzipien folgt, braucht man ein Verfahren, mit dessen Hilfe man einem Programm mehrere verschiedene Ordnungen gleichzeitig aufzwingen kann. Ein solches Verfahren soll die aspektorientierte Programmierung liefern.

**Allgemeinheit
des Problems**

Nun ist das Problem der einen dominierenden Ordnung, die keine Rücksicht auf mögliche andere nimmt, kein spezielles der Programmierung — auch in

anderen Disziplinen, deren Artefakte linear geordnet sind, bieten sich häufig mehrere konkurrierende Ordnungen an. Die Aufgabe ist dann, diejenige auszuwählen, mit der sich unter Berücksichtigung aller Kriterien das beste Ergebnis erzielen lässt. So wird beispielsweise von Leibniz berichtet, er habe über Jahre niemanden in die ihm anvertraute Königliche Bibliothek gelassen, weil die Bücher darin noch nicht in rechter Ordnung stünden. Jedes Buch müsste, je nach Ordnungskriterium, andere nächste Nachbarn haben, aber die (lineare) Anordnung in Regalen verlangt die Präferenz eines einzelnen. Ein analoges Problem wird die eine oder andere von Ihnen auch schon vom Verfassen einer größeren schriftlichen Ausarbeitung her kennen: Spätestens, wenn man für eine bestimmte Aufgabenstellung zwei oder mehr verschiedene Lösungsansätze anhand zwei oder mehr verschiedener Beispiele diskutieren möchte, stellt sich die Frage, ob man den (linearen) Text anhand der alternativen Lösungen oder der Beispiele strukturiert. Eine allgemeine Faustregel gibt es nicht und die Vor- und Nachteile des einen oder anderen Ansatzes werden in der Regel erst offenbar, wenn man ihn durchgezogen hat. Wie schön wäre es da, man könnte alle Alternativen in einer Ordnung vereinigen!

Da dieses Problem kein neues ist, sollte es auch nicht verwundern, dass mehrere Lösungsansätze dazu existieren. Aus Büchern kennt man das Schlagwortverzeichnis, das Zusammenhänge über Kapitelgrenzen hinweg herstellt, aus Bibliotheken Kataloge, die die Bücher nach verschiedenen Kriterien ordnen und das Auffinden der Bücher in den Regalen unabhängig von dem jeweiligen (für Bibliotheksbenutzerinnen mitunter nur schwer zu erratenden) angewandten Ordnungskriterium machen, und aus der Programmierung die Ansätze zur sog. *mehrdimensionalen Trennung der Belange* (engl. multidimensional separation of concerns). Die meisten dieser Lösungsansätze wurden allerdings erst in jüngerer Zeit, im Zuge der großen Welle, die die aspektorientierte Programmierung verursacht hat, nach oben bzw. ins Bewusstsein der Programmieröffentlichkeit gespült. So werden heute unter anderem das *Subject Oriented Programming* (SOP, bei IBM entwickelt) und die sog. *Composition Filters* als Vorläufer bzw. Alternativen zur aspektorientierten Programmierung genannt. Auch wenn es für deren, sicher verdienten Protagonisten tragisch ist, sieht es im Moment so aus, als würde allein die aspektorientierte Programmierung, zumindest was den Namen angeht, davon übrig bleiben (auch wenn sie inzwischen um die eine oder andere Eigenschaft der anderen Ansätze ergänzt worden ist). Neben den Nachteilen, die eine solche

Monokultur für die Forschung und den Erkenntnisgewinn mit sich bringt, hat dies aber auch den immensen Vorteil, dass sich die Anstrengungen um die Bereitstellung von Werkzeugen zur aspektorientierten Programmierung konzentrieren, mit dem Ergebnis, dass schon heute jede die aspektorientierte Programmierung in ihren Projekten selbst ausprobieren kann, ohne dazu auf die Entwicklungsmethoden der Steinzeit (Texteditor, Compileraufruf auf Kommandozeile etc.) zurückgreifen zu müssen. Tatsächlich wird die aspektorientierte Programmierung heute hier und da auf ihre Tauglichkeit für die Praxis getestet — allerdings bislang mit wenig eindeutigen Ergebnissen.

Interessant ist übrigens auch, dass der als Leitfigur der aspektorientierten Programmierung bekannte Gregor Kiczales sich zuvor mit der Metaprogrammierung (vor allem in LISP) beschäftigt hatte. Die aspektorientierte Programmierung darf also auch personell als ein Abkömmling der Metaprogrammierung angesehen werden.

6.3.2 Inhalte von Aspekten

Welcher Art könnten nun diese Funktionen, die man durch einen Aspekt auslagern und an einer bestimmten Stelle konzentrieren möchte, sein? Die aspektorientierte Programmierung macht darüber zunächst keine Vorgaben. Wichtig (wenn auch nicht zwingend vorgeschrieben) ist lediglich, dass diese Funktionen nicht nur an einer, sondern an mehreren (am besten vielen und nicht genau vorherbestimmten) Stellen im Programm auftreten. Dabei können sich diese Funktionen in gewissen (engen) Grenzen von Fall zu Fall unterscheiden; unter theoretischen Gesichtspunkten ist es zunächst einmal nur notwendig, dass sie alle unter denselben konzeptuellen Aspekt fallen — unter praktischen Gesichtspunkten dann allerdings auch noch, dass es für alle diese Funktionen eine gemeinsame Implementierung gibt, die in Form eines Aspeks ausgelagert werden kann.

Wenn man allerdings die Literatur studiert, so fällt einem sofort auf, dass bezüglich der genannten Beispiele eine ziemliche Monokultur vorzuherrschen scheint: Es werden immer wieder (und mit geradezu ermüdender Monotonie) Tracing, Logging, Debugging, Exception handling, Persistenz, Transaktionsverwaltung, Authentisierung etc. genannt. Man könnte sofort einwenden, dass für diese tatsächlich immer wiederkehrenden Probleme auch (und besser) die Programmiersprache erweitert werden könnte. So ist das eine oder andere Problem ja auch schon in Form spezieller Sprachkonstrukte gelöst worden — man denke beispielsweise an das Exception handling. Auch werden zum Teil Entwicklungsumgebungen in die Richtung der die Programmentwicklung betreffenden Aspekte (z. B. Tracing und Debugging) erweitert, was entsprechende Aspekte ebenfalls überflüssig macht. Das Interesse an der aspektorientierten Programmierung hat das jedoch nicht beendet.

Monotonie der Beispiele

Dennoch ist es auffallend, dass sich zumindest unter den in der Literatur genannten Aspekten kaum solche befinden, die man mit Fug und Recht als *funktional* (also die Funktionen oder Daten einer Problemstellung betreffend) bezeichnen könnte. Tatsächlich scheinen Aspekte vorwiegend für die Sicherstellung der sog. *nichtfunktionalen Eigenschaften* eines Programms herangezogen zu werden. Nichtfunktionale Eigenschaften bezeichnen vor allem Qualitätseigenschaften wie Zuverlässigkeit, Wartbarkeit,

Aspekte präferieren nichtfunktionale Eigenschaften

Performanz, Sicherheit etc. All dies sind Eigenschaften, die die Lösung eines bestimmten Problems (in Form eines Programms) garantieren muss. Sie ergeben sich zwar aus der Aufgabenstellung, sind aber in der Regel nicht in deren üblichen Abstraktionen (Analyse- und Designmodelle, z. B. in UML) wiederzufinden. Aspekte scheinen also entgegen zahlreicher Verlautbarungen ein recht spezielles, auf bestimmte Aufrufe zugeschnittenes Konzept zu sein.

Natürlich kann man Aspekte (wie praktisch jedes andere Programmiersprachenkonstrukt auch) für nahezu beliebige Zwecke einsetzen (und dabei entsprechend missbrauchen). Man könnte sich sogar vorstellen, dass ein Programm ganz und gar nur noch aus Aspekten besteht, die auf ein leeres Programmgerüst oder auf sich gegenseitig angewendet werden. Man sollte aber neue Programmierkonstrukte nur einführen, wenn sie ein Problem lösen, das sich zuvor und mit anderen Mitteln nur behelfsweise lösen ließ. Dies bedingt im Umkehrschluss, dass das neue Konstrukt entweder einen bestimmten, eben diesen Verwendungszweck hat, oder aber so allgemein ist, dass es die zuvor behelfsweise verwendeten Konstrukte auch in anderen Kontexten vollständig umfasst und deswegen ersetzen kann. Letzteres ist beispielsweise den Modulen, mit denen sich abstrakte Datentypen emulieren ließen, durch die Einführung des Klassenkonzepts widerfahren.

6.3.3 Charakterisierung der aspektorientierten Programmierung

Die aspektorientierte Programmierung versucht, zwei störende Phänomene der objektorientierten Programmierung zu beseitigen: Das sog. *Scattering*, womit die Verstreutung von Code gleicher Funktionalität quer über das Programm gemeint ist, und das sog. *Tangling*, was so viel wie die Verwindung von Code für die Umsetzung einer Funktionalität mit dem für die Umsetzung einer anderen heißen soll. Etwas frei lässt sich das wie in Abbildung 6.1 veranschaulichen: Jede Sorte von Belag stellt die Implementierung einer Funktionalität an den verschiedenen Stellen des Programms dar.

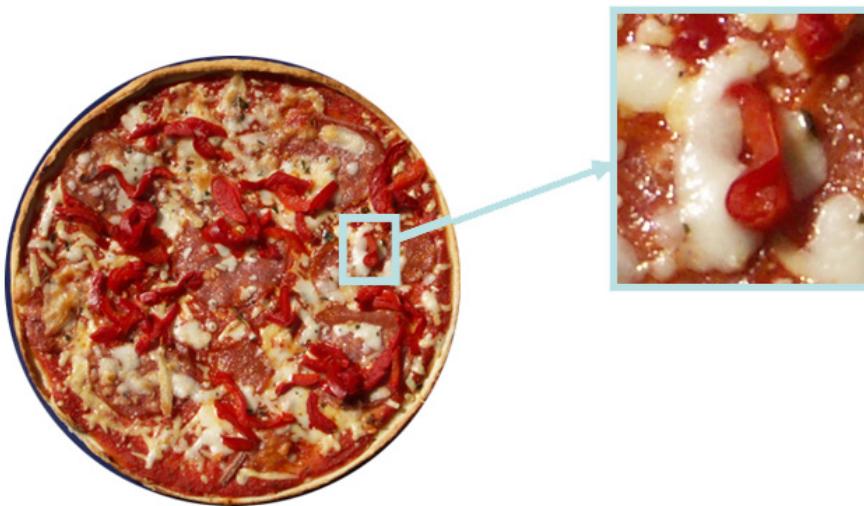


Abbildung 6.1: Ein monolithisches Programm mit mehreren Aspekten (Paprika, Salami, Käse etc.). Deren Implementierung ist über das Programm verstreut (Scattering) und ineinander verwunden (Tangling; s. Ausschnittvergrößerung).

Wenn man ein solches monolithisches Programm nun in Module, beispielsweise Klassen, aufteilt, dann wird die Situation dadurch kaum besser: Wie in Abbildung 6.2 zu sehen, bleibt so-

wohl die Verteilung als auch die Verwindung bestehen. Modularisiert wird nämlich lediglich das Basisprogramm (der Boden), in dem z. B. die Geschäftslogik untergebracht ist. Jedes Modul beherbergt damit neben den Teilen, die die Modulbildung bestimmten, auch die, die darüber verstreut und mit anderen Programmteilen verwunden sind.



Abbildung 6.2: Ein modularisiertes Programm. Die Verteilung und Verwindung des die verschiedenen Aspekte implementierenden Codes bleibt bestehen.

Nun sind verschiedene Ansätze vorstellbar, wie man auch die Verteilung und Verwindung von Programmteilen auflösen könnte. Eine Möglichkeit ist, die Teile nach Art auf Module aufzuteilen, wie in Abbildung 6.3 nahegelegt; da es hier keine Unterscheidung zwischen dem die Modularisierung bestimmenden Basisprogramm und den querschneidenden Funktionen gibt, nennt man einen solchen Ansatz auch symmetrisch. Im Gegensatz dazu steht die zweite, in Abbildung 6.4 dargestellte Möglichkeit, bei der die querschneidende Funktionalität in zusätzliche Programmeinheiten, die Aspekte, ausgelagert wird: Hier bleibt das „nackte“, d. h. um die querschneidende Funktionalität erleichterte Basisprogramm zurück, weswegen man auch von einem asymmetrischen Ansatz spricht.

In beiden Fällen stellt sich natürlich die Frage, wie in den separierten, aspektorientierten Programmen (Abbildung 6.3 und Abbildung 6.4) dieselbe Information stecken kann wie in dem ursprünglichen (Abbildung 6.2 oder gar Abbildung 6.1): Der Code wird ja wohl nicht aus lauter Ungeschicklichkeit so organisiert gewesen sein, wie vorgefunden, sondern weil es der Programmfluss (die Algorithmik, wenn so ein großes Wort hier passt) so verlangte (oder zumindest nahelegte). Wo und wie der nunmehr herausgelöste, neu organisierte Code ursprünglich verstreut und verwunden war, muss also irgendwie rekonstruierbar sein, wenn sich die Bedeutung des Programms nicht ändern soll. So ist es ja schließlich auch bei der Isolierung eines Unterpro-

gramms: An der Stelle, an der der Code ursprünglich stand, findet sich nach der Herauslösung ein Unterprogrammaufruf.⁶²



Abbildung 6.3: Aufteilung eines Programms durch Separierung der zuvor verstreut und verwunden implementierten Funktionalität. Da man hier nicht zwischen Basisprogramm und Aspekten unterscheidet, nennt man diese Art der Trennung der Belange auch eine symmetrische. Ein prominenter Vertreter einer solchen Strategie ist HYPERJ, die SOP-Sprache von IBM (Entwicklung inzwischen eingestellt).



Abbildung 6.4: Organisation des Programms in modularisiertes Basisprogramm und davon getrennte Aspekte. Da Basisprogramm und Aspekte unterschiedlich repräsentiert werden, spricht man auch von einem asymmetrischen Ansatz. Prominentestes Beispiel für diesen Ansatz ist ASPECTJ.

⁶² Man bedenke, dass sich das von der aspektorientierten Programmierung angegangene Problem durch Unterprogrammaufrufe nicht lösen lässt — tatsächlich handelt es sich bei dem Code, der durch Aspektorientierung herausfaktorisiert werden soll, nicht selten um Methodenaufrufe, die damit offensichtlich nicht zu der angestrebten Programmorganisation geführt haben (nämlich weil die Aufrufe selbst verstreut und verwunden sind).

Den Prozess, der die ursprünglichen Zusammenhänge wiederherstellt, nennt man im aspektorientierten Jargon *Weben* (engl. weaving). Er verlangt neben dem Basisprogramm und den Aspekten im asymmetrischen Fall und den Aspekten allein im symmetrischen Fall auch noch einen sog. *Webeplan* (engl. weaving plan), der einem geeigneten *Aspektmechanismus*, dem *Weber* (engl. weaver) als Basis dient, um das ursprüngliche Programm herstellen zu können (bzw. so zu kompilieren, dass sich eine zu seiner ursprünglichen Form äquivalente Bedeutung ergibt). Dieser Webeplan kann Bestandteil der Aspekte sein oder separat spezifiziert werden; in jedem Fall beinhaltet er Details von den Elementen, die verwoben werden sollen (da auch der schlauste Weber nicht erahnen kann, wie die Teile zusammengehören, damit sie gemeinsam den gewünschten Zweck erzielen). Daraus folgt, dass eine Änderung dieser Elemente potentiell auch eine Änderung des Webplans verlangt — erhebliche Wartungsprobleme sind die unmittelbare Konsequenz.

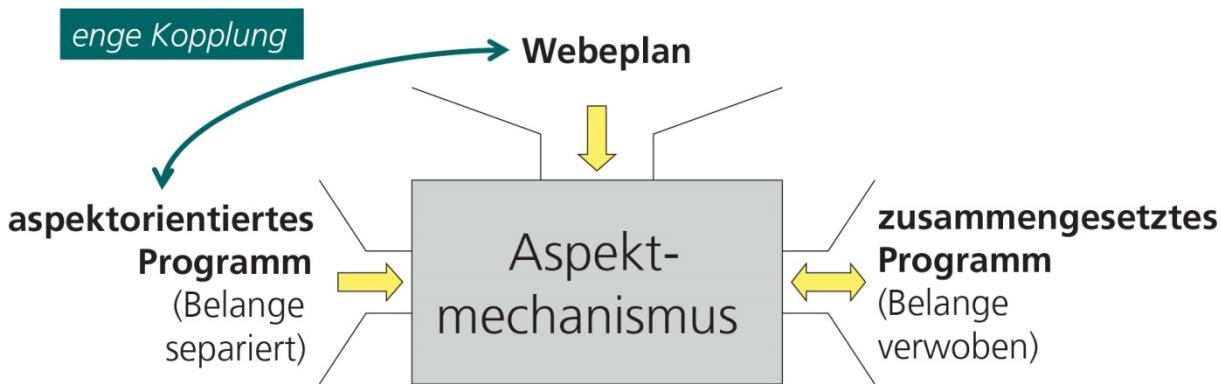


Abbildung 6.5: Zusammensetzung eines aus mehreren Aspekten bestehenden Programms anhand eines Webeplans. Da der Webeplan die Stellen benennen muß, an denen die Programmteile zusammengefügt werden müssen, besteht eine enge Kopplung zwischen Plan und Teilen.

An dieser Stelle lohnt es sich, sich noch ein wenig mit dem Vorgang des Webens zu beschäftigen. Abbildung 6.5 zeigt die prinzipielle Vorgehensweise: Ein aspektorientiertes Programm wird in einen Aspektmechanismus gefüttert, der, ähnlich wie ein Builder, die Teile zusammenfügt und daraus ein konventionelles Programm macht, das dann, da es nicht mehr Gegenstand der Beobachtung ist, die zu vermeidenden Eigenschaften aufweisen darf. Da der Webevorgang im Aspektmechanismus keine eigene Kenntnis von der Intention der Programmiererin hat, muss er über den Webeplan gesteuert werden, der dafür wiederum eine starke Abhängigkeit zum aspektorientierten Programm aufweisen muss. Abbildung 6.6 verdeutlicht dies für zwei populäre Formen der aspektorientierten Programmierung.

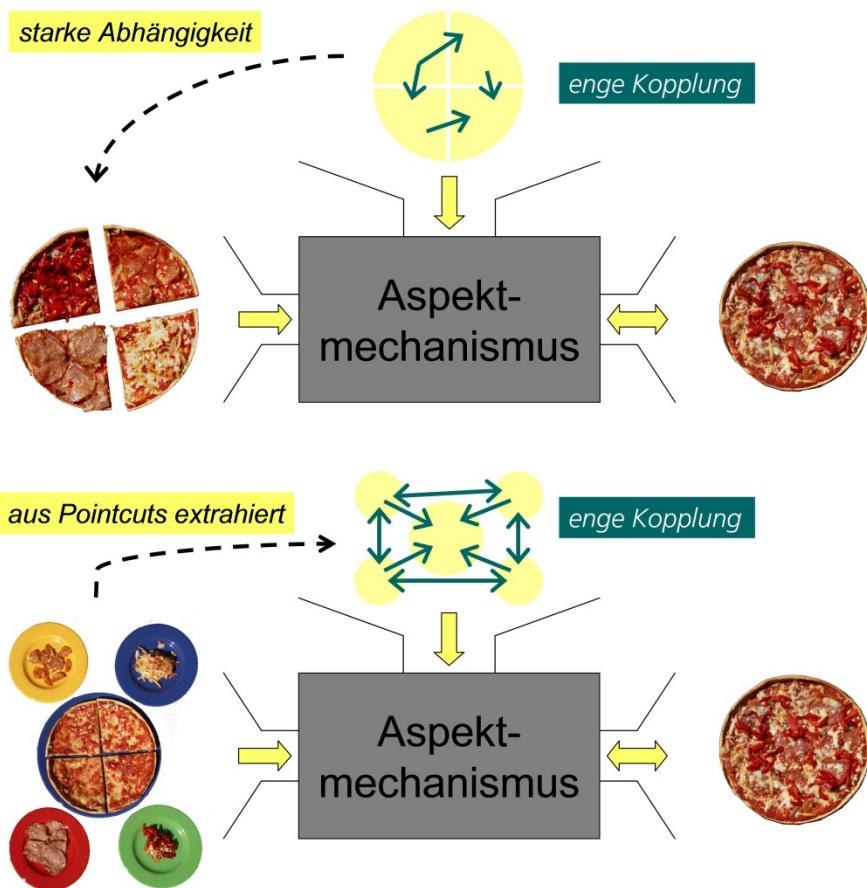


Abbildung 6.6: Konkrete Ausprägungen des Webepans. Im oberen Fall enthält der Webepan direkte Verweise auf Details der Aspekte und gibt an, wie sie zu verweben sind. Im unteren Fall sind die Webeinformationen in den Aspekten selbst enthalten und werden aus diesen extrahiert. In beiden Fällen werden die vermeintlich unabhängigen Teile (Aspekte sowie ggf. Basisprogramm) durch den Webepan stark verkoppelt und sind nicht unabhängig wartbar.

Formen des Webepans

Der Webepan enthält also Hinweise, wie die Programmteile zusammenzufügen sind, genauer, wie der mit den Aspekten verbundene Code über das Pro-

gramm verstreut und mit anderen Teilen verwunden werden soll. Grundsätzlich bieten sich zwei Möglichkeiten an, die Bezüge zwischen Programmteilen herzustellen: eine extensionale und eine intensionale Spezifikation. Bei der extensionalen Spezifikation werden einfach die Programmteile, die miteinander verbunden werden, aufgezählt; bei der intensionalen wird ein Prädikat gesucht, das, auf Programmstellen angewendet, aussagt, ob an der Stelle etwas gewoben werden soll oder nicht. Die Vor- und Nachteile der unterschiedlichen Verfahren liegen auf der Hand: Bei der extensionalen Spezifikation müssen die Programmstellen im Vorrhinein bekannt sein und müssen, da sie eine Art Index in das Programm darstellen, an jede Änderung angepasst werden; bei der intensionalen hängt es davon ab, ob die Änderungen im Programm das gewählte Prädikat berücksichtigen, also so gestaltet werden, dass das Prädikat auch nach der Änderung noch die richtigen Stellen im Programm meint. Aufgrund mangelnden automatischen Programmverstehens verwendet die intensionale Definition (zumindest heute noch) vor allem Ausdrücke, die sich auf die Namen von Methoden beziehen (also eine Form der Introspektion); die Änderungs-abhängig- und -anfälligkeit ist entsprechend hoch.

Wenn Ihnen das jetzt alles zu abstrakt ist, haben Sie vollkommen Recht; im Abschnitt über ASPECTJ (6.3.6) wird es konkreter. An dieser Stelle soll jedoch noch schnell die Charakterisierung

der aspektorientierten Programmierung abgeschlossen werden: Sie lässt sich (nach unter Anhängerinnen nicht unumstrittener Lehrmeinung) durch die Formel

$$\text{aspect orientation} = \text{quantification} + \text{obliviousness}^{63}$$

zusammenfassen. Dabei bedeutet *Obliviousness*, dass die Stellen, die von Aspekten betroffen werden, nichts von ihrem Glück wissen (und setzt voraus, dass dies zu ihrem Glück auch gar nicht notwendig ist). *Quantification* bedeutet, dass Aspekte an potentiell unendlich vielen Stellen im Programm zum Einsatz kommen können, was insbesondere dann von Bedeutung ist, wenn das Programm oder Teile davon noch gar nicht geschrieben sind. Sie ist natürlich nur bei intensiver Spezifikation des Webeplans gegeben

6.3.4 Aspektorientierte Programmierung und Modularisierung

Eine Zitierung zieht sich beinahe wie ein roter Faden durch die gesamte Literatur zur aspektorientierten Programmierung: die von Parnas' Arbeit über die Modularisierung [48]. Zur Modularisierung würden aber explizite Interfaces gehören, die das Zusammenspiel zweier oder mehrerer Module, im gegebenen Fall des Aspekts und der Programmteile, auf die sich der Aspekt bezieht, klar beschränken. In der aspektorientierten Programmierung bestimmt ausschließlich der Webeplan darüber, auf welche Teile eines (oder beliebiger) Programme Aspekte Zugriff haben — ein entsprechendes Interface der Teile bleibt implizit. Den zugegriffenen Programmteilen bleibt somit auch keine Möglichkeit, sich gegen einen Zugriff zu wehren. Umgekehrt können die Aspekte nicht verlangen, dass diese Programmteile ein ihnen nicht bekanntes Interface respektieren, es also insbesondere bei Änderungen unberührt lassen. Mit Modularisierung hat all das aber nichts zu tun — tatsächlich modularisieren Aspekte nicht, sie lokalisieren lediglich Funktionalität an einem Ort.

Abbildung 6.7 veranschaulicht die Problematik unter der Annahme der Modularität der beteiligten Programmelemente. Um den isolierten Code der Aspekte an den richtigen Stellen im Programm zur Wirkung zu bringen, müssen diese Stellen im Webeplan benannt werden. Solange diese Stellen nicht durch die Interfaces der Module veröffentlicht werden, muss der Webeplan sich auf Geheimnisse des Moduls beziehen. Eine vom Webeplan unabhängige Wartung der Module ist somit nicht mehr möglich — die Idee der Modularisierung wird konterkariert.

⁶³ „*oblivious*“ heißt hier etwa „gar nicht bemerken“ — gemeint ist damit, dass dem Quellcode nicht anzusehen ist, ob bzw. dass er durch einen Aspekt berührt wird.

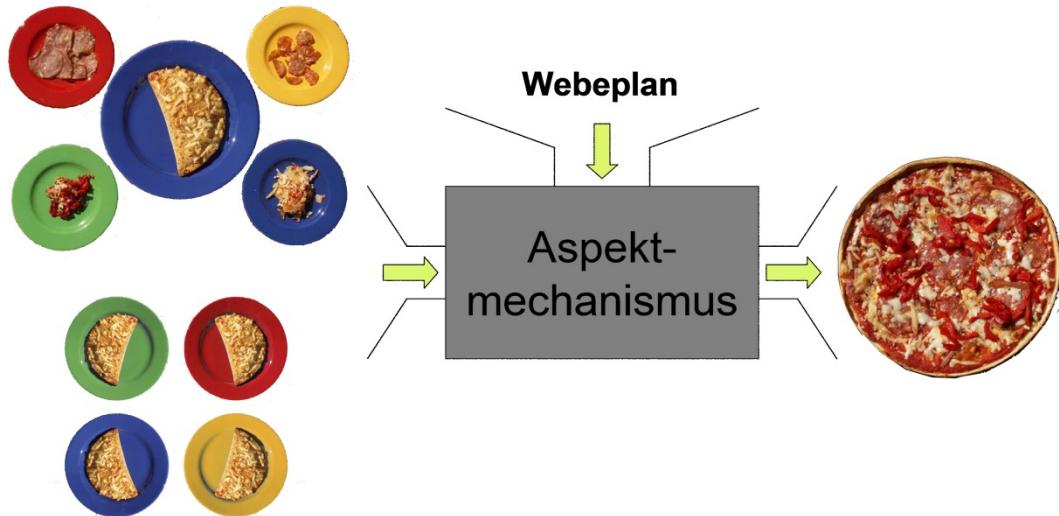


Abbildung 6.7: Weben unter Modularität der beteiligten Programmteile. Es ist unklar, wie ein Webeplan aussehen soll, der das Geheimnisprinzip der Module nicht durchbricht.

Damit aber noch nicht genug: Der Grund für die Verteilung und Verwindung von Code ist häufig — insbesondere in objektorientierten Programmen — nicht allein dessen Zugehörigkeit zu einer bestimmten Stelle im Programmfluss, sondern auch der benötigte Zugriff auf den Programmkontext, der nur an dieser Stelle gegeben ist. So muss solcher Code häufig auf Variablen zugreifen, deren Sichtbarkeit aus Gründen der Modularität auf eben diesen Kontext beschränkt ist. Wenn der Code nun aus diesem Kontext herausgelöst und an eine andere Stelle verfrachtet wird, dann muss er Referenzen auf den Kontext mitnehmen, um sich die Zugriffsmöglichkeit aufrecht zu erhalten. Dieser Sachverhalt ist in Abbildung 6.8 dargestellt.

Abhängigkeit vom Kontext

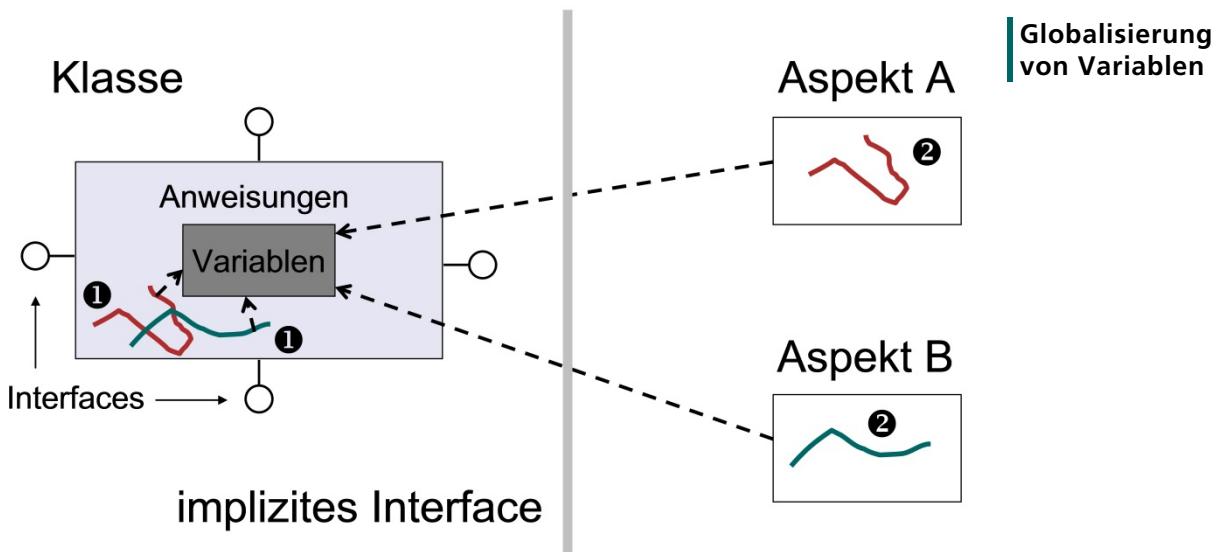


Abbildung 6.8: Ein Effekt der Verlagerung von verwundenem und verstreutem Code in Aspekte ist, dass neue Abhängigkeiten quer über Modulgrenzen und somit neue, allerdings implizite, Interfaces entstehen. (① und ② beziehen sich auf Vorher und Nachher.)

Dieser Umstand ist so wichtig, dass er noch einmal von einer anderen Seite beleuchtet werden soll. Ein gewöhnlicher Prozeduraufruf verwendet aktuelle und formale Parameter, um den Kontext des Aufrufs zu verpacken und an die entfernte Stelle (den Ort, an dem die Prozedur definiert ist) zu übergeben. Da die aspektorientierte Programmierung aber ohne explizite Aufrufe auskommen will (Ziel Obliviousness), muss sie den Zugriff auf Variablen, die den Kontext herstellen, von außen, nämlich für die Aspekte erlauben. Da diese Aspekte dem Kontext unbekannt sind (und somit beliebiger Natur sein können), werden die lokalen Variablen faktisch veröffentlicht, ohne dass man es diesen Variablen ansähe, was nicht nur dem Geheimnisprinzip der Modularisierung widerspricht, sondern auch der allgemein anerkannten Regel, nach der globale Variablen zu vermeiden sind.⁶⁴

Es gibt keine Möglichkeit, dieses Dilemma aufzulösen, ohne auf eines der charakteristischen Merkmale der aspektorientierten Programmierung zu verzichten. So könnte man zwar verlangen, dass die Programmteile, in die Aspekte hineingewoben werden sollen, die Stellen, an denen das möglich ist, über ein Interface veröffentlichen, aber damit ginge die Obliviousness verloren, also die Tatsache, dass die beglückten Programmteile nichts von ihrem Glück wissen müssen. Damit würde gleich eine ganze Klasse von Anwendungen der aspektorientierten Programmierung verlorengehen, nämlich all die, die wie Debugging, Logging etc. darauf beruhen, dass die debugten, geloggten etc. Programmstellen für eben diese Funktion nicht vorbereitet werden müssen.

**keine Lösung
in Sicht**

6.3.5 Aspektorientierte Programmierung und Lesbarkeit

Ein anderes, in der Literatur zur aspektorientierten Programmierung häufig anzutreffendes Zitat ist das von Dijkstras Separation of concerns (zu Deutsch: *Trennung der Belange*):

To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused. This is what I mean by focusing one's attention upon a certain aspect; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. Such a separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of. I usually refer to it as a separation of concerns because one tries to deal with the difficulties, the obligations, the desires and the constraints one by one. [49]

So muss Dijkstra, der als der Vater der sog. *strukturierten Programmierung* gilt, auch noch als Vordenker der aspektorientierten Programmierung herhalten, zumal er sogar schon das Wort Aspekt verwendete, und das durchaus im Sinne der aspektorientierten Programmierung. Ob er

⁶⁴ W Wulf, M Shaw „Global variable considered harmful“ SIGPLAN Not. 8:2 (1973) 28–34.

es aber tatsächlich auch so meinte, ist mehr als fraglich, wie die nachfolgenden Überlegungen zeigen.

strukturierte Programmierung

Dijkstra hatte mit seinem berühmten Artikel „Goto statement considered harmful“ von 1968 die Goto-Anweisungen praktisch im Handstreich zu einem Tabu der Programmierung gemacht. Abweichungen vom sequentiellen Programmfluss waren in der Folge nur noch als sauberer Programmierstil angesehen worden, wenn sie über die Kontrollstrukturen der strukturierten Programmierung, das sind im Wesentlichen die Verzweigung mit einem True- und einem False-Block, die Schleife (in verschiedenen Varianten, aber immer ohne Goto) und der Unterprogrammaufruf, umgesetzt werden. All diesen Kontrollstrukturen ist gemeinsam, dass sie sowohl im Programmtext (statisch) als auch im Programmfluss (dynamisch) genau einen Eingang und genau einen Ausgang haben. Man ist sich also beim Lesen derart strukturierter Programme immer im Klaren darüber, wo der Programmfluss herkommt und wo er hingeht. Letzteres wurde durch das dynamische Binden der objektorientierten Programmierung zumindest teilweise eingeschränkt, wofür eben diese auch viel Prügel bezogen hat (vgl. Kurs 01814).

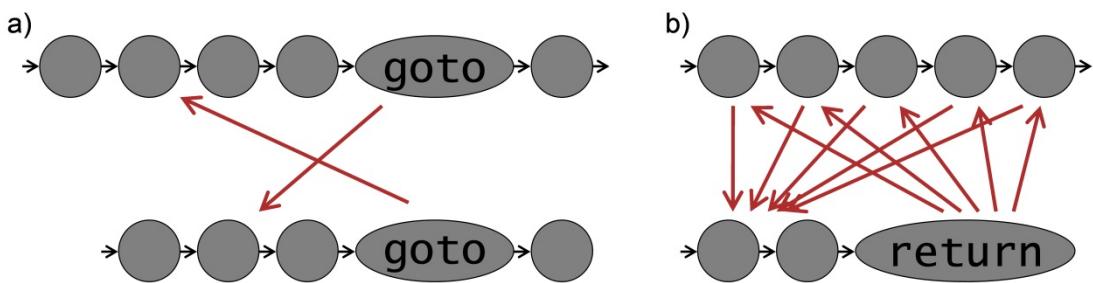


Abbildung 6.9: Bzgl. der Lesbarkeit eines Programms stellt die aspektorientierte Programmierung eine Verschlechterung gegenüber der Verwendung des schon schlechten Gotos dar: Während man beim Goto aus der näheren Umgebung einer Anweisung (hier dargestellt durch Kreise) nicht erkennen kann, welches das in der Programmausführung davorliegende Statement war (es könnte ein entferntes Goto gewesen sein), kennt man bei der aspektorientierten Programmierung weder das davorliegende noch das nachfolgende. Die Überraschung bei der Programmausführung ist somit perfekt.

Das Paradoxe an der aspektorientierten Programmierung ist nun, dass sie auch mit diesem Prinzip bricht, obwohl sie sich immer wieder darauf beruft. Tatsächlich ist der Kontrollfluss unter Anwendung von Aspekten noch viel obskurer, als er durch die Verwendung von Gotos jemals sein könnte, weil man einer einzelnen Anweisung und ihrer näheren Umgebung nicht nur ihren Vorgänger nicht ansehen kann, sondern auch nicht ihren Nachfolger (s. Abbildung 6.9) — man müsste hierfür schon den Webeplan konsultieren. Tatsächlich kann man unter aspektorientierter Programmierung beim Lesen eines bestimmten Quelltextes nicht erkennen, ob und gegebenenfalls welcher Aspekt in ihm greift. Man kann also insbesondere auch bei der Fehlersuche auf Basis des Programmtextes nicht erkennen, ob vielleicht durch einen Aspekt eingebrachte Funktionalität zum Fehlverhalten des Programms führt. Mindestens ebenso problematisch ist der Umstand, dass man beim Schreiben neuen Quellcodes diesen aus Versehen so verfassen könnte, dass er unter die intensionale Spezifikation des Webeplans eines Aspektes fällt, der Aspekt also zur Anwendung kommt, ohne dass man dies als Autorin so beabsichtigt hätte. Zwar bietet ak-

tuelle Werkzeugunterstützung an, beim Anzeigen bzw. Editieren des Quellcodes auf die Aspekte zu verweisen, die in ihm zur Anwendung kommen, doch setzt dies ein Vorliegen aller Aspekte voraus (und verbietet zudem, diese zu einem späteren Zeitpunkt zu ändern, ohne daraufhin den gesamten Quellcode noch einmal zu überprüfen). Und so kommt es, dass man bei Anwendung der aspektorientierten Programmierung in besonderem Maße auf die Existenz von guten Debuggern angewiesen ist (was ja schon insofern ironisch anmutet, als Debugging eine Hauptanwendung der aspektorientierten Programmierung ist).

6.3.6 ASPECTJ

Die derzeit am weitesten verbreitete (und wohl auch beliebteste) aspektorientierte Programmiersprache ist ASPECTJ, eine Erweiterung von JAVA um Aspekte. Für ASPECTJ gibt es neben dem unverzichtbaren Compiler auch Plug-ins für einige bekannte Entwicklungsumgebungen, darunter ECLIPSE, die es erlauben, unter den heute üblichen, komfortablen Bedingungen aspektorientiert zu programmieren. Aufgrund der hohen Popularität von ASPECTJ wird es oft mit der aspektorientierten Programmierung an sich gleichgesetzt. Dies ist jedoch genauso falsch, wie JAVA mit der objektorientierten Programmierung gleichzusetzen — beides sind nur (zugegebenermaßen ziemlich erfolgreiche) Exponenten einer jeweiligen Technologie. Man sollte aber nicht vergessen, dass sich wichtige Entwicklungen oft abseits des Mainstream abspielen. Wer also allein auf ASPECTJ setzt, läuft Gefahr, den einen oder anderen für sie interessanten Gesichtspunkt der aspektorientierten Programmierung zu verpassen.

6.3.6.1 Die wichtigsten Programmierkonstrukte von ASPECTJ

In ASPECTJ sind Aspekte Programmeinheiten, die formal starke Ähnlichkeit mit Klassen aufweisen. Anders als Klassen sind Aspekte jedoch weder Typen noch instanzierbar — neben technischen Gründen befreit einen das von der Notwendigkeit, sich vorstellen zu müssen, was die Instanz eines Aspekts sein soll. So stellt man sich einen Aspekt am besten als eine Sammlung von Daten und Methoden vor, die eine bestimmte, aus dem übrigen System ausgelagerte Funktionalität realisieren (vgl. Abbildung 6.4).

Zwar kann ein Aspekt seinen eigenen Zustand definieren, da er aber nicht instanzierbar ist (zumindest nicht durch die Programmiererin), teilen sich alle Anwendungen des Aspekts diesen Zustand (ausgedrückt durch die im Aspekt deklarierten Variablen). Es ist also zunächst nicht möglich, einen Aspekt einem Objekt beizufügen, also ihn auf einem Objekt zur Anwendung gebracht anders reagieren zu lassen als auf einem anderen.⁶⁵ Dies würde aber eine empfindliche Einschränkung der Ausdrucksstärke der aspektorientierten Programmierung bedeuten. Deshalb ist es in ASPECTJ möglich, dass ein Aspekt zusätzliche Instanzvariablen in die Klassen, auf die der Aspekt ausgerichtet ist, einbringt (sog. *Introductions*). Der Zustandsraum von Objekten dieser Klassen wird dadurch erweitert. Allerdings ist diese Erweiterung nur für den Aspekt sichtbar, denn der Quellcode der Klasse bleibt von der Erweite-

**Aspekte
und Zustand**

⁶⁵ Es ist es dann doch, aber dies ist ein weiterführendes Konzept, auf das hier nicht eingegangen werden soll.

nung unberührt. Lediglich die ebenfalls (per Introduction) durch den Aspekt in die Klasse eingebrachten Methoden haben Zugriff auf diesen zusätzlichen Zustand. Da diese Methoden aber auch Zugriff auf den nativen Zustand der Klassen haben, können sie auf einfache Weise das Geheimnisprinzip durchbrechen (vgl. Abschnitt 6.3.4).

Bestandteile eines Aspekts

In ASPECTJ besteht ein Aspekt neben seinem Namen sowie einer optionalen Menge von Variablen Deklarationen und Methodendefinitionen vor allem aus zwei Dingen: der Spezifikation dessen, was zu tun ist, und der Spezifikation dessen, wo bzw. wann es zu tun ist. Für beides wurden in ASPECTJ neue Namen eingeführt, die nicht unbedingt selbsterklärend sind: Die Spezifikation des Was wird *Advice* genannt, die des Wo/Wann *Pointcut*. Ein drittes neues Sprachkonstrukt bleibt implizit (d. h., ohne Schlüsselwort — dafür ist aber wenigstens sein Name selbsterklärend): der *Joinpoint*, der eine Stelle im Programm bezeichnet, an der theoretisch eine im Aspekt festgehaltene Funktion (Advice) eingebracht werden kann. Formal gesehen spezifiziert ein Pointcut eine Menge von Joinpoints, wobei diese Menge zumindest theoretisch in der Regel unendlich groß ist (so lange noch nicht bekannt ist, auf welche Programme sich der Aspekt bezieht, kommen potentiell alle möglichen Programme in Betracht; außerdem beziehen sich die Joinpoints in ASPECTJ immer auf Punkte in der Ausführung des Programms, von denen es — zumindest theoretisch — ebenfalls unendlich viele gibt).

Joinpoints und Joinpoint-Modell

Joinpoints bezeichnen also Stellen im Programm, auf die sich ein Aspekt beziehen kann. Typische Joinpoints sind z. B. Prozeduraufrufe (genauer: die Punkte vor und nach einem Prozederaufruf) oder auch die Zugriffe auf Variablen (lesend oder schreibend) eines Programms. Jedes Programm hat eine Menge von Joinpoints, unabhängig davon, ob es Aspekte hat. Woraus genau diese Menge besteht, ist eine Eigenschaft der betrachteten aspektorientierten Programmiersprache und unterliegt natürlich den Beschränkungen, die durch die Definition bzw. das Laufzeitsystem der zugrundeliegenden Programmiersprache (nicht selten JAVA) auferlegt werden. Man nennt diese Eigenschaft auch das *Joinpoint-Modell* der Sprache.

Pointcuts

Pointcuts sind demnach einem oder mehreren Aspekten zugeordnete, eingeschränkte Mengen von Joinpoints, die die Stellen im Programm bezeichnen, an denen die Aspekte greifen sollen. Pointcuts sind also Bestandteil der Definition konkreter Aspekte, weswegen sie eines syntaktischen Konstrukts bedürfen. Dieses sieht wie folgt aus:

```
1887 <access modifier> pointcut <pointcut-name>(<args>) :
    <pointcut-definition>
```

Dabei handelt es sich hier um die benannte Variante, also eine Deklaration von Pointcuts, die von der des Advice, also dessen, was an den bezeichneten Stellen im Programm zu tun ist (s. u.), unabhängig erfolgt. Der Access modifier kann einen der (in JAVA) üblichen Werte **public** etc. annehmen und regelt die Verwendbarkeit des Pointcuts außerhalb des beherbergenden Konstrukts (Aspekt oder Klasse). Der Pointcut-Name ist ein frei wählbarer Bezeichner, der den Pointcut referenziert.

Die Pointcut-Definition selbst besteht aus der Angabe der Kategorie des Joinpoints (z. B. Methodenaufruf oder -ausführung, Feldzugriff, etc.) sowie einem Muster, das eine Menge von Pro-

grammelementen (Joinpoints) spezifiziert. Ein Beispiel für einen solchen Pointcut ist die folgende Definition:

```
1888 public pointcut logPoints() : call(* Account.*(..));
```

Er erfasst alle Aufrufe von Methoden (`call`) der Klasse `Account`, und zwar unabhängig davon, wie sie heißen oder welche Parametertypen sie verlangen. Ein wesentlich konkreterer Pointcut wäre

```
1889 public pointcut logPoint() : call(void Account.setBalance(double));
```

Er umfasst aber alle Aufrufe der Methode `setBalance` der Klasse `Account` mit Parametertyp `double`.

Neben der Spezifikation der Stelle, an der ein Aspekt eingewoben werden soll, erfüllt der Pointcut noch eine andere wichtige Funktion: Wie in Abschnitt 6.3.4 bereits angedeutet, muss dem Aspekt Zugang zum Kontext, in dem seine Funktion anzusiedeln ist, gegeben werden. Da dieser Kontext, der in der Regel durch Variablen ausgedrückt wird, aber von Anwendungsstelle zu Anwendungsstelle recht unterschiedlich ausfallen kann, also insbesondere die Variablen nicht als einheitlich benannt vorausgesetzt werden dürfen (und die Möglichkeiten zum automatischen Programmverstehen zumindest heute noch fehlen), ist der Kontext in ASPECTJ im Wesentlichen auf eine einzige, immer gleichförmig vorhandene Information beschränkt: den Inhalt des Call stacks. So werden Aspekte in ASPECTJ, soweit sie sich nicht direkt auf Variablenzugriffe beziehen, praktisch immer vor, nach oder während der Ausführung einer Prozedur (Methode) zur Anwendung gebracht und haben als Kontext im Wesentlichen das Empfängerobjekt sowie die Parameter des Methodenaufrufs zur Verfügung.⁶⁶

Es bleibt noch die Spezifikation dessen, was ein Aspekt tun soll, also des **Advises**. Ein Advice besteht im Wesentlichen aus einem Block von Anweisungen; er ist damit dem Rumpf einer Methode vergleichbar. Da ein Advice immer implizit aufgerufen wird, wird er in ASPECTJ nicht benannt. Zugriffsbeschränkungen (in Form von Access modifiers) sind aus gleichem Grund ebenfalls nicht vorgesehen. Zugriff auf den Kontext, in dem er ausgeführt wird, erhält der Advice über die Verbindung mit einem Pointcut — dieser bezeichnet ja die Stellen im Programm, an denen der Advice zur Anwendung kommen soll. Diese Verbindung gibt auch an, wann der Advice im Verhältnis zu einem Joinpoint (einem Punkt der Ausführung des Programms) ausgeführt werden soll: davor (**before**), danach (**after**) oder beides (ein Teil davor, ein Teil danach; **around**). Die vollständige Spezifikation eines Advice sieht also beispielsweise wie folgt aus:

⁶⁶ Es gibt über die hier nur angedeuteten Mittel der Pointcut-Definition hinaus in ASPECTJ zahlreiche weitere Möglichkeiten, wie man die Menge der interessierenden Punkte in der Ausführung eines Programms spezifizieren kann. Neben Prozedurausführungen und Variablenzugriffen können auch Aussagen über den Programmablauf (Kontrollfluss) getroffen werden. Die Zusammenhänge sind teilweise recht kompliziert und nicht selten eher von den technischen Möglichkeiten als vom konzeptuellen wünschenswerten getrieben. Es soll daher an dieser Stelle nicht weiter darauf eingegangen werden; stattdessen wird auf die weiterführende Literatur, z. B. [50], verwiesen.

```

1890 void around(): logPoints() {
1891     // irgend etwas loggen
...
1892     proceed();
1893     // den Rest loggen
...
1894 }
```

Dabei ruft `proceed()` immer die Methoden auf, um die „herum“ der Advice ausgeführt werden soll; welche das sind, wird durch den mit dem Aspekt assoziierten Pointcut (hier `logPoints()`) bestimmt. Die leeren Klammern sind in anderen Beispielen natürlich mit Parametern gefüllt.

Ein vollständig definierter Aspekt folgt also folgendem Schema:

```

1895 public aspect tuMirGut {
1896     // eigene Variablendeklarationen und Methodendefinitionen
...
1897     // Pointcut-Definition(en)
1898     pointcut woEsPassierenSoll(...): ...
1899
1900     // Advice-Definition(en)
1901     ...
1902 }
```

6.3.6.2 Ein größeres Beispiel: Das **OBSERVER** Pattern als Aspekt

Eine interessante und zugegebenermaßen auch recht attraktive Anwendung der Möglichkeit, das Verhalten von Klassen durch Aspekte zu erweitern, ist die Umsetzung von Entwurfsmustern (Kurseinheit 4). Entwurfsmuster beschreiben ja Objektstrukturen, die in der einen oder anderen Form immer wiederkehren. Dabei hat jedes Entwurfsmuster einen konstanten Teil, der in allen Vorkommen gleich ist, und einen variablen Teil, der sich von Vorkommen zu Vorkommen unterscheidet. Bisher (mit konventionellen Mitteln der Programmierung) war eine Modularisierung des konstanten Teils und eine darauf basierende Wiederverwendung nicht oder nur schwer zu realisieren. Mittels ASPECTJ ist es nun möglich, den konstanten Teil bestimmter Entwurfsmuster aus deren Vorkommen herauszufaktorisieren und in einem Aspekt zu hinterlegen. Dieser Aspekt, auf verschiedenen Klassen angewendet, injiziert das den jeweiligen *Rollen* des Entwurfsmusters entsprechende Verhalten in diese Klassen.

Die konkrete Umsetzung von Entwurfsmustern mit ASPECTJ soll im Folgenden anhand des **OBSERVER** Patterns (s. Abschnitt 4.4.2) vorgeführt werden. Das Beispiel stammt direkt aus einer Arbeit, in der untersucht wurde, welche der Entwurfsmuster aus dem vielzitierten gleichnamigen Buch sich wie gut mit Aspekten umsetzen lassen [51].

Die Implementierung des allgemeinen, wiederverwendbaren Teils des Musters sieht danach wie folgt aus:

```
1903 public abstract aspect ObserverProtocol {  
1904     protected interface Subject {}  
1905     protected interface Observer {}  
1906     private WeakHashMap perSubjectObservers;  
1907     protected List getObservers(Subject s) {  
1908         if (perSubjectObservers == null) {  
1909             perSubjectObservers = new WeakHashMap();  
1910         }  
1911         List observers = (List)perSubjectObservers.get(s);  
1912         if (observers == null) {  
1913             observers = new LinkedList();  
1914             perSubjectObservers.put(s, observers);  
1915         }  
1916         return observers;  
1917     }  
1918     public void addObserver(Subject s,Observer o){  
1919         getObservers(s).add(o);  
1920     }  
1921     public void removeObserver(Subject s,Observer o){  
1922         getObservers(s).remove(o);  
1923     }  
1924     abstract protected pointcut subjectChange(Subject s);  
1925     abstract protected void updateObserver(Subject s, Observer o);  
1926     after(Subject s): subjectChange(s) {  
1927         Iterator iter = getObservers(s).iterator();  
1928         while (iter.hasNext()) {  
1929             updateObserver(s, ((Observer)iter.next()));  
1930         }  
1931     }  
1932 }
```

Auffällig ist, dass der Aspekt zunächst einmal zwei Interfaces, **Subject** und **observer**, einführt, die den Rollen des Musters entsprechen. Diese Interfaces sind innere Typen des Aspekts; da sie zudem **protected** deklariert sind, sind sie nur von Subaspekten (ja, so etwas gibt es auch; s. u.) zugreifbar.

Gleich danach (in den Zeilen 1906 –1923) wird eine Datenstruktur eingeführt, die die Zuordnung von Subjekten zu ihren Observern erfasst. Da der Aspekt nur einmal existiert, ist es notwendig, dass die Liste der Observer zu einem Subjekt in einem Dictionary (hier: **weakHashMap**) mit dem Subjekt als Schlüssel eingetragen wird. Die Implementierung der Methoden **getObservers()**, **addObserver()** und **removeObserver()** ist soweit Standard (und hat insbesondere nichts Aspektorientiertes an sich); man beachte lediglich die „faule Initialisierung“ (*lazy initialization*, s. Kurs 01814) der verwendeten Collections.

Als nächstes (in Zeile 1924) folgt die Deklaration des Pointcuts: Er ist in diesem Fall abstrakt, d. h., ohne Definition. Man ahnt, dass die Definition in einem konkreten Unterasperkt nachgeholt wird (und dass da dann die Anpassung des Aspekts an ein konkretes Vorkommen des Musters stattfindet). Ebenfalls abstrakt bleibt die nachfolgende Methodendeklaration (in Zeile 1925): Hier wird dann im konkreten Unterasperkt festgeschrieben werden, wie im konkreten Fall die Benachrichtigung der Observer stattzufinden hat. Nicht abstrakt ist hingegen der abschließende Advice (Zeilen 1926 ff.): Er ist an den Pointcut **subjectChange()** gebunden und übernimmt das übliche Iterieren über die Observer eines Subjekts sowie den Aufruf der (hier noch abstrakten) Notifikation. Man beachte, dass der Pointcut einen Parameter **s** vom Typ **Subject** spezifiziert; auf dieses **s** wird im Advice zugegriffen. Das Besondere ist hier, dass der Typ von **s**, **Subject**, im Aspekt selbst eingeführt wird und das auch noch als **protected**; man ahnt wiederum, dass im konkreten Subasperkt auch hier noch etwas passieren muss.

Ein möglicher konkreter Subasperkt ist nun der folgende:

```

1933 public aspect XYZObserver extends ObserverProtocol {
1934     declare parents: XYZ implements Subject;
1935     declare parents: ABC implements Observer;
1936     protected pointcut subjectChange(Subject s):
1937         call(void XYZ.set*(..)) && target(s);
1938     protected void updateObserver(Subject s, Observer o) {
1939         ((ABC) o).notify();
1940     }
}

```

Hier passiert nun etwas Ungeheuerliches: Die Klassen **XYZ** und **ABC** implementieren auf einmal Interfaces, die sie gar nicht kennen. Allerdings, und das macht die Sache etwas leichter, hat das für sie auch gar keine Auswirkungen, denn die Interfaces sind a) leer und b) außerhalb des Aspektes und seiner Subasperkte gar nicht sichtbar. Trotzdem wird man anerkennen, dass das Potential von ASPECTJ, vorliegende Programme zu verändern, beunruhigend ist.

Der Rest ist dann einfach: Der konkrete Pointcut definiert lediglich, dass er alle Setter der Klasse `XYZ` (in der Rolle des Subjekts) abfängt, und dass der Parameter `s` an das Empfängerobjekt der Setteraufrufe gebunden werden soll (mittels `target(s)`); die überschriebene Methode `updateObserver(.,.)` schließlich legt fest, dass der Aspekt die Methode `notify()` auf allen Observern, die im konkreten Fall ja vom Typ `ABC` sein müssen, ausführen soll.

Man hat es hier mit einer neuen, zunächst sicher unerwarteten Möglichkeit, Aspekte sinnvoll einzusetzen, zu tun. Ein Aspekt beschreibt auf einmal nicht mehr eine Funktion, sondern eine Kollaboration, wie sie z. B. in Entwurfsmustern, aber auch in der Definition von Komponenten anzutreffen ist. Tatsächlich gehen jüngere Arbeiten in die Richtung, aspektdefinierte Kollaborationen als Ausgangspunkt für die Definition von Komponenten zu sehen [52]. Dabei ist das Problem der Durchbrechung von Modulgrenzen (s. u.) in diesem Zusammenhang kein Problem mehr, da die Instanzen von Klassen, die gemeinsam eine Komponente bilden, voreinander keine Geheimnisse haben müssen. Wenn die Komponente als Aspekt definiert ist, bedeutet dies, dass die Modularität der Objekte nach außen, also über die Komponentengrenze hinweg, sehr wohl intakt bleibt.

6.4 Zusammenfassung und Ausblick

Die Metaprogrammierung ist ein mächtiges Konzept, das die gemeine Programmiererin allzu leicht überfordert. Da es keinen Quellcode im üblichen Sinne (also in Form eines zusammenhängend lesbaren Textes) gibt, ist Metaprogrammiertes nur schwer nachzuvollziehen. Überraschungen sind praktisch unvermeidlich und die Wahrscheinlichkeit, dass man metaprogrammierte Applikationen nur per Versuch und Irrtum richtig hinbekommt, ist groß. So ist es auch nicht verwunderlich, dass Sprachen, denen die Metaprogrammierung inhärent ist (wie z. B. PROLOG oder LISP), in der Wirtschaft keine besonders weite Verbreitung gefunden haben. Dem gegenüber stehen die vergleichsweise hohe Kompaktheit (niedrige Redundanz, d. h. wenig gleicher oder auch nur ähnlicher Code) sowie die extreme Flexibilität, die sich mit den Mitteln der Metaprogrammierung erzielen lässt.

In der Konsequenz dessen hat man versucht, wenig gefährliche Teile der *Metaprogrammierung* in „normale“ Programmiersprachen wie JAVA einzubauen. In einem ersten Schritt war das die Möglichkeit zur *Introspektion*, später dann die explizite Attributierung von Programmteilen mit *Metadaten (Annotationen)* und zuletzt die *aspektorientierte Programmierung*, mit deren Hilfe Abläufe im Programm aufgespürt, abgefangen und umgemünzt werden können (*Interzession*). Was noch fehlt, ist die Möglichkeit, per Programm neuen Code zu schreiben und diesen gleich (im selben Programm) zur Ausführung zu bringen. Spätestens dann wird sich jedoch wieder die Einsicht breit machen, dass die einfache Programmiererin mit der Sprache überfordert ist.

6.5 Weiterführende Literatur

Die Arbeit von Parnas [48] ist ein Stück Informatikgeschichte. Sie sollte jede gelesen haben, die eine Laufbahn auf dem Gebiet der Softwareentwicklung anstrebt. Die „Entwurfsmuster“ [47] kann man eigentlich immer heranziehen, so auch hier (wegen des *PROXY Patterns*). Die definitive Referenz zur aspektorientierten Programmierung mit ASPECTJ ist derzeit vermutlich noch immer das Buch von Laddad [50]. Da ich persönlich der ganzen Sache aber keine allzu lange Halbwertszeit zutraue [53], halte ich es für kein Muss, es im Regal stehen zu haben. Zudem ist ASPECTJ immer noch großen Veränderungen unterworfen (die Version 8 ist inzwischen erschienen, 1.8.13 von November 2017), so dass man auch auf eine zweite Auflage warten kann. Experimentierfreudige sollten sich aber auf jeden Fall einen ASPECTJ-Compiler besorgen und damit herumspielen, um sich ein eigenes Bild zu machen. Informationen hierzu gibt es unter [eclip-se.org/aspectj/](http://eclipse.org/aspectj/) (vormals www.aspectj.org).

- [47] E Gamma, R Helm, R Johnson, J Vlissides Design Patterns — Elements of Reusable Software (Addison-Wesley, 1995). Dublette zu [1].
- [48] DL Parnas „On the criteria to be used in decomposing systems into modules“ Communications of the ACM 15:12 (1972) 1053–1058. (Dublette zu [3])
- [49] EW Dijkstra A Discipline of Programming (Prentice Hall, Englewood Cliffs, New Jersey, 1976).
- [50] R Laddad AspectJ in Action (Manning, 2003).
- [51] J Hannemann, G Kiczales: „Design pattern implementation in Java and AspectJ“ OOPSLA (2002) 161–173.
- [52] KJ Lieberherr, DH Lorenz, J Ovlinger „Aspectual collaborations: combining modules and aspects“ The Computer Journal 46:5 (2003) 542–565.
- [53] F Steimann „The paradoxical success of aspect-oriented programming“ in: OOPSLA 2006 — Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (2006) 481–497.

6.6 Lösungen der Selbsttestaufgaben

Selbsttestaufgabe 6.1 (Seite 239)

```
1941 class Object {  
1942     public Object perform(String methodname, Object[] parameters) {  
1943         //lookup method, terminate if not available  
1944         ...  
1945         //create parameter type array  
1946         ...  
1947         //find best matching signature, terminate if not available  
1948         ...  
1949         //invoke method with parameters  
1950         ...  
1951     //return result  
1952     ...  
1953 }
```

Selbsttestaufgabe 6.2 (Seite 242)

```
1954 @Retention(value = RUNTIME)  
1955 @Target(value = ANNOTATION_TYPE)  
1956 public @interface Retention {  
1957     RetentionType value();  
1958 }
```



Copyright © 2003 United Feature Syndicate, Inc.

7 Extreme Programming

Nachdem jahrzehntelang die Verwendung strukturierter Softwareentwicklungsprozesse gelehrt und diese mit viel Energie in die Praxis eingebracht wurden, haben sich mit Ende der neunziger Jahre einige bekannte Persönlichkeiten aus dem Bereich der objektorientierten Programmierung (vor allem SMALLTALK) laut und deutlich für schlankere (und damit hoffentlich weniger aufwendige) Prozessmodelle stark gemacht. Quasi über Nacht bekannt geworden ist dabei vor allem das **Extreme Programming** [54], das gewissermaßen eine Vorreiterfunktion unter den sog. *agilen* Prozessmodellen einnimmt.

Programmiererin im Mittelpunkt

Gemeinsam ist diesen beweglicheren, eben agileren Prozessen, dass sie die Programmiererin und ihre Belange in den Mittelpunkt stellen. Im Ergebnis erinnern sie etwas an das berühmt-berüchtigte *Code and fix* (wofür die agilen Prozessmodelle auch häufig kritisiert werden); sie sind aber bei weitem nicht so unstrukturiert, wie sie auf den ersten Blick erscheinen mögen — es ist sogar eher das Gegenteil der Fall. Richtig ist in jedem Fall, dass die agilen Prozessmodelle auf die anderen Prozessmodelle dominierenden verschiedenen Formen der Dokumentation wenig Wert, sondern vielmehr nahelegen, soweit wie möglich darauf zu verzichten. Dies ist natürlich für jeden Anhänger eines sauber dokumentierten Softwareentwicklungsprozesses ein Graus.

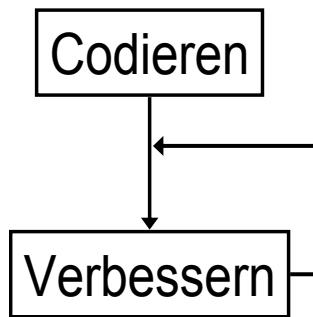


Abbildung 7.1: Das Code-and-fix-Modell. Wie das Diagramm nahelegt, bleibt der Softwareentwicklungsprozess beim Verbessern stehen — man wird praktisch nie fertig.

Natur der Programmiererin

Wie kam es zu dieser Revolution? Zunächst einmal ist festzuhalten, dass es in der Natur der meisten Programmiererinnen liegt, zu programmieren. Vielleicht werden Sie es an sich selbst erlebt haben: Wenn man mit einer neuen (Programmier-) Aufgabe konfrontiert wird, setzt bereits nach kurzer Zeit der Reflex ein, den Rechner einzuschalten, die

Entwicklungsumgebung nach Wahl zu starten und mit dem Codieren zu beginnen. Tatsächlich ist es einer der größten Reize der Programmierung, dass man jederzeit sofort damit beginnen kann, ohne erst — wie in anderen Ingenieursdisziplinen — langwierig die richtigen Materialien beschaffen und die entsprechenden Werkzeuge bereit- bzw. herstellen zu müssen. Sobald die Kiste läuft, sind der Kreativität keine Grenzen mehr gesetzt. Wer wollte sich da noch lange auf dem Papier Gedanken machen, wenn doch die Möglichkeit besteht, gleich mit der richtigen Arbeit zu beginnen und — vor allen Dingen — dabei sofort sehen zu können, wohin die Gedanken führen? Anstatt diesen Reflex, so schädlich er auch manchmal sein mag, zu bekämpfen, versucht man ihn beim Extreme Programming in den Prozess der Softwareentwicklung einzubauen. Die Kunst dabei ist, dafür zu sorgen, dass am Ende eben kein Code and fix herauskommt (also das Prozessmodell, bei dem man die meiste Zeit damit verbringt, die Fehler, die man bereits am Anfang gemacht hat, am Ende aufwendig zu reparieren), sondern ein zielführender, kontrollierbarer und — innerhalb gewisser Grenzen — wiederholbarer Prozess entsteht.

Während der Ansatz des Extreme Programming von Programmiererinnen vergleichsweise bereitwillig angenommen wird, gibt es von Seiten des Projektmanagements — verständlicherweise — doch erhebliche Bedenken. Besonders verdächtig erscheint der Umstand, dass — zumindest in den Anfängen der Bewegung — gefordert wurde, den vorgeschriebenen Prozess des Extreme Programming sklavisch einzuhalten, da es sonst angeblich nicht funktionieren würde. Dabei muss man sich doch im Klaren darüber sein, dass es völlig realitätsfern ist, irgendeinen Prozess akribisch einzuhalten, selbst wenn sich zeigt, dass das Projekt damit aus dem Ruder läuft. Es entsteht hier also nur allzu leicht der Verdacht, die Propheten des Extreme Programming hätten von Anfang an eine Hintertür eingebaut, mittels derer sie das Scheitern eines Extreme-Programming-Projekts jederzeit erklären können — man hat sich dann eben einfach nicht genau genug an die Vorschriften gehalten.

Natur der Managerin

7.1 Geschichte des Extreme Programming

Der Erfolg des Extreme Programming ist ein Lehrstück der Geschichte des Software Engineering. An dessen Anfang steht ein Buch, das sich im Wesentlichen auf den Erfolg eines einzigen Projektes stützt, eine Evidenz, die man in anderen Disziplinen (zum Beispiel der Medizin) allenfalls anekdotisch nennen würde. In diesem Projekt hat ein kleines Team sehr guter Leute (u. a. Kent Beck, Ward Cunningham, Ron Jeffries) in relativ kurzer Zeit ein Stück Software bis zur Betriebsreife entwickelt, an dem sich zuvor ein sehr viel größeres Team über einen sehr viel längeren Zeitraum erfolglos versucht hatte. Die genannten Personen dürfen aber mit Fug und Recht als Könner, wenn nicht als die Crème de la crème der objektorientierten Programmierung bezeichnet werden, weswegen man den Erfolg durchaus auch auf andere Faktoren (z. B. den „A-few-good-men-Ansatz“) zurückführen könnte. Was darauf folgte, ist eine Mischung aus gutem Marketing (allein der Name des Extreme Programming, der an Extremsport erinnert und den man eher bei MTV als in der Softwaretechnikliteratur erwarten würde) und höchst wirksamer Mund-zu-Mund-Propaganda, derer sich die Erschaffer allein aufgrund ihres Bekanntheitsgrades leicht bedienen konnten.

Die kritischen Anmerkungen sollen jedoch nicht davon ablenken, dass in der Branche der Softwareentwicklung durchaus ein gewisser Bedarf an der Abkehr von allzu schwergewichtigen (und damit schwerfälligen) Methoden besteht. So ist beispielsweise das *Wasserfallmodell* schon lange wegen seiner Unfähigkeit, Änderungen in den Anforderungen während der Projektaufzeit zu berücksichtigen, in Verruf geraten und allenfalls noch deswegen im Einsatz, weil es so leicht zu managen ist. Insgesamt haben Prozessmodelle dazu geführt, dass die Softwareentwicklung als Disziplin in die Gilde der papierproduzierenden Gewerbe eingetreten ist, und zwar in einem Ausmaß, das kaum noch zu rechtfertigen ist, dies umso weniger, als sich in der Praxis häufig herausstellt, dass die im Vorhinein angefertigte Dokumentation von Anfang an unrealistisch war oder von der Entwicklung überholt wurde und angesichts des üblichen Zeitdrucks, unter dem Projekte stehen, an eine entsprechende Anpassung nicht zu denken ist.⁶⁷ Doch genug der Vorrede — kommen wir jetzt zum Kern der Sache.

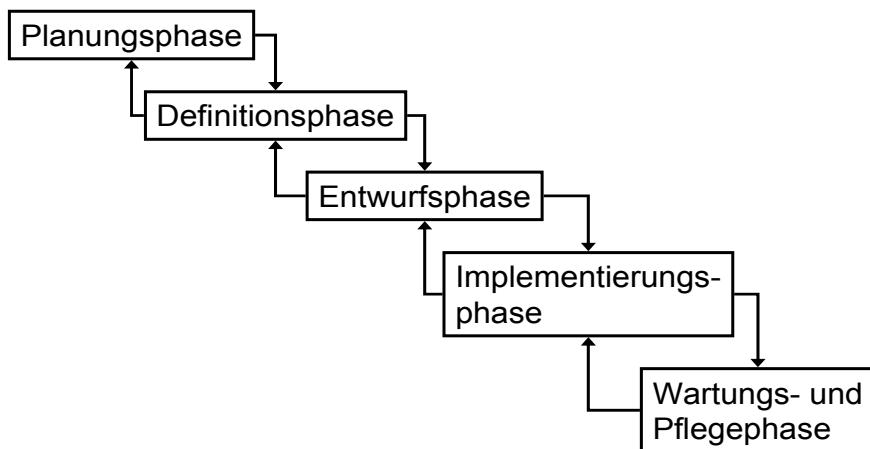


Abbildung 7.2: Das Wasserfallmodell — Traum jeder Managerin und Alptraum jeder Programmiererin. Wenn in einer Phase festgestellt wird, dass ein Ergebnis einer früheren Phase revidiert werden muß, müssen alle dazwischen liegenden Phasen durchlaufen werden. Änderungen sind also extrem schwerfällig und entsprechend teuer.

Extreme Programming ist das Zusammenspiel vieler einzelner Konzepte und Techniken. Einen Überblick gibt Abbildung 7.3. Sie mag wie eine Parodie erscheinen, aber sie drückt aus, wie eng das Zusammenspiel und die gegenseitigen Abhängigkeiten der Konzepte und Techniken des Extreme Programming sind.

⁶⁷ Gleichwohl hat natürlich auch das Wasserfallmodell seine guten Seiten. Es ist geradezu ideal in Fällen, in denen die Probleme reine Spezifikationsprobleme sind, in denen also die Programmieraufgaben bloße Schreibarbeiten sind, wenn erst einmal feststeht, was gemacht werden soll. So können beispielsweise Masken für die Bildschirmeingabe relativ simpel „herunterkodiert“ werden, sobald klar ist, welche (Datenbank-)Felder erfasst werden müssen. Leider wird dieser Vorteil häufig durch unrichtige/unvollständige Spezifikationen ad absurdum geführt.

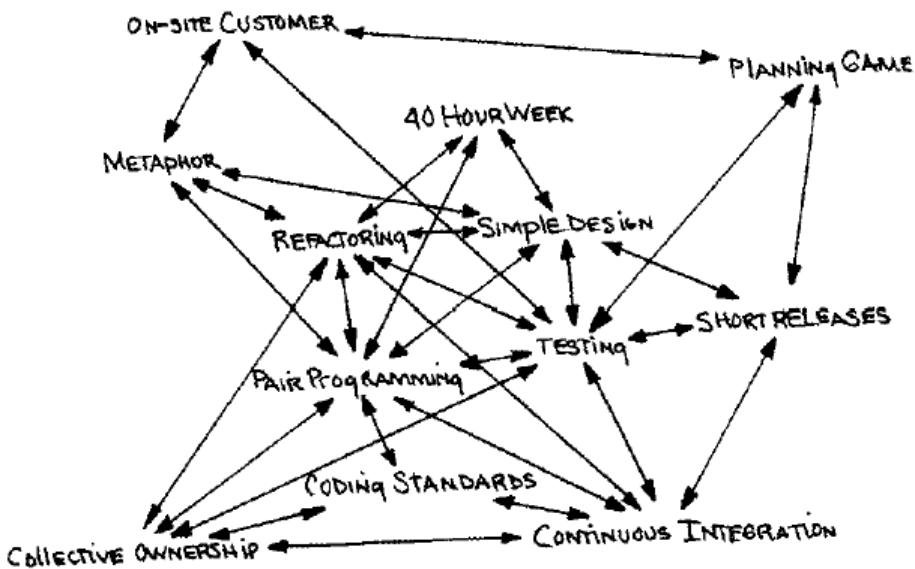


Abbildung 7.3: Die Kernkonzepte des Extreme Programming und ihr Zusammenhang (aus [54]).

7.2 Ziele des Extreme Programming

Oberstes Ziel des Extreme Programming (wie das jedes anderen Softwareentwicklungsprozesses) ist, Software in hoher Qualität im Zeit- und Kostenrahmen zu entwickeln. Dabei soll das Extreme Programming ein Prozessmodell für die Programmiererinnen sein und nimmt entsprechend wenig Rücksicht auf die Anforderungen eines klassischen Projektmanagements. Dies drückt sich unter anderem darin aus, dass die Verantwortung für das Gelingen des Projekts von Entwicklerinnen und Kundinnen geteilt wird; die Rolle einer Managerin ist dabei gar nicht vorgesehen.

Die angestrebte hohe Qualität wird im Extreme Programming im Wesentlichen durch zwei Maßnahmen erreicht: den Test-first-Ansatz und das Programmieren in Paaren.

7.3 Der Test-first-Ansatz

In größeren Firmen wird das Testen typischerweise nicht von den Programmiererinnen selbst durchgeführt. Der wohl wichtigste Grund hierfür ist die psychologische Hemmschwelle, eigene Fehler wahrzunehmen oder aufzudecken — man ist für die eigenen Fehler gewissermaßen blind. Das erklärt sich daraus, dass man sich wünscht, dass der eigene Code korrekt ist, was man mit dem Testen demonstrieren möchte. Man wird also — unbewusst — die Testfälle so formulieren, dass sie keine Fehler zutage befördern, und dies umso mehr, je geringer das eigene Vertrauen in die Korrektheit der abgelieferten Arbeit ist. Nur wenn man sich sicher ist, dass keine Fehler mehr im Programm sind, wird man den sportlichen Ehrgeiz entwickeln, wirklich harte Tests zu finden und durchzuführen, dies dann nämlich in der Gewissheit, dass auch diese keine Fehler aufzeigen (weil eben keine mehr drin sind) und die eigene Überzeugung nur bestätigt wird.

ungeliebtes Testen

Aus verständlichen Gründen ist das Berufsbild der Testerin wenig beliebt. Anstatt wie die Programmiererinnen große Entwürfe vorlegen, in die Tat umzusetzen und dabei die eigene Kreativität voll ausleben zu können, ist ihre Arbeit grundsätzlich destruktiver Natur: Sie besteht einzig und allein darin, Fälle aufzufinden und zu dokumentieren, die die Unzulänglichkeit anderer nachweisen. Auch wird die Arbeit einer Testerin, wenn sie ihre Arbeit gründlich macht, die Auslieferung eines Produkts eher verzögern denn beschleunigen — Punkt um, ihr Beitrag ist immer negativer Natur, und zwar umso mehr, je mehr sie sich angestrengt. Wen würde das wirklich befriedigen?

Phasen der Softwareentwicklung oder gar die Vorstellung, dass ein Produkt (Programm) bis zu seiner Fertigstellung mehrere Abteilungen durchläuft, sind dem Extreme Programming völlig fremd. Vielmehr liegt hier die Verantwortung für die Korrektheit des Codes bei allen und damit auch — und insbesondere — bei der Programmiererin selbst. Um nun dem Problem der Blindheit für die eigenen Fehler wirksam vorzubeugen, bedient man sich eines einfachen Tricks: Man dreht die Reihenfolge von Programmierung und Testen einfach um. Heraus kommt der sog. Test-first-Ansatz.

**gegen die Blindheit
gegenüber eigenen
Fehlern**

Nach dem Test-first-Ansatz ist jede Programmiererin verpflichtet, bevor sie eine neue Funktion implementiert, für diese einen oder mehrere Tests zu konzipieren und zu implementieren. Mit Hilfe eines geeigneten Test-Frameworks (häufig JUNIT; s. Abschnitt 3.2) ist die Programmiererin dann in der Lage, die neue Funktion unmittelbar nach der Fertigstellung und in der Folge immer wieder automatisch zu testen. Neben dem bereits zuvor beschriebenen Vorteil, dass die Programmiererin durch die umgekehrte Reihenfolge von Testen und Entwickeln nicht blind für die eigenen Fehler ist (es sei denn, sie hat unbewusste hellseherische Fähigkeiten), erlaubt die Existenz von Tests auch anderen Entwicklerinnen im Team, die Implementierung der Funktion zu einem späteren Zeitpunkt zu ändern, ohne sich dabei Gedanken machen zu müssen, ob sie immer noch das tut, was sie sollte. (Diese „Änderung durch andere“, oder auch die sog. *Collective code ownership*, ist ein weiterer Kernbestandteil des Extreme Programming.)

Da beim Test-first-Ansatz die zu testende Funktion noch gar nicht implementiert ist, kann die Programmiererin, die dann ja auch noch keine Fehler gemacht hat und sich entsprechend auch keine eingestehen muss, die Messlatte für die Qualität ihrer eigenen Arbeit selbst (hoch) anlegen, ohne dabei Gefahr zu laufen, sich als Schlechte ihrer Zunft zu entlarven. Tatsächlich befindet sie sich, wenn sie sich bei der Entwicklung der Tests Mühe gegeben hat, in einer Situation, in der sie nur gewinnen kann: Wenn der Test einen Fehler aufgedeckt, dann hat sich die Investition in den Test gelohnt; deckt er keinen auf, dann hat sich die Investition ebenfalls gelohnt, denn das Nachdenken über das, was schief gehen können hätte, hat vermutlich dazu geführt, dass es nicht schiefgegangen ist. Dies erklärt, warum viele Programmiererinnen dem Test-first-Ansatz durchaus etwas abgewinnen können, auch wenn so jede Aufgabe damit beginnt, zunächst einmal Code zu schreiben, der nicht produktiv ist in dem Sinne, dass er es nicht in das Endprodukt schafft.

Regressionstesten

Dadurch, dass JUNIT-Tests Programme sind, ist es leicht möglich, die Tests immer wieder auszuführen. Dies gestattet dem Extreme Programming ein konsequentes *Regressionstesten*. Dies wiederum erhöht das Vertrauen in die ständige Änderbarkeit

des Codes — nach jedem *Refactoring* werden einfach alle Tests noch einmal ausgeführt, wodurch man die Gewissheit erlangt, dass sich zumindest an der getesteten Funktion durch das Refactoring nichts geändert hat (was ja definitionsgemäß der Fall sein soll).

Zwar schaffen Unit-Tests ein Vertrauen in die Bausteine, aus denen das Programm besteht, aber korrekte Bausteine können auch falsch zusammengesetzt werden. Um die Korrektheit der gesamten Funktionalität eines Systems beurteilen zu können, werden sog. *Funktionstests* benötigt. Das Schreiben dieser Funktionstests setzt jedoch zwingend voraus, dass man die Anforderungen richtig verstanden hat bzw. — was noch viel schwieriger ist — dass man die tatsächlichen Anforderungen an das System — unabhängig von jeder Spezifikation derselben, die ja falsch sein kann — kennt. Dies ist in der Regel jedoch nur für die Kundin der Fall. Die konsequente Berücksichtigung dieser Erkenntnis bedeutet, dass die Kundin für die Formulierung der Funktionstests verantwortlich sein muss. Wenn sie nicht auch in der Lage ist, diese zu schreiben (oder gar zu programmieren), dann sollten ihr die Programmiererinnen des Teams hilfreich zur Seite stehen.

Funktionstests

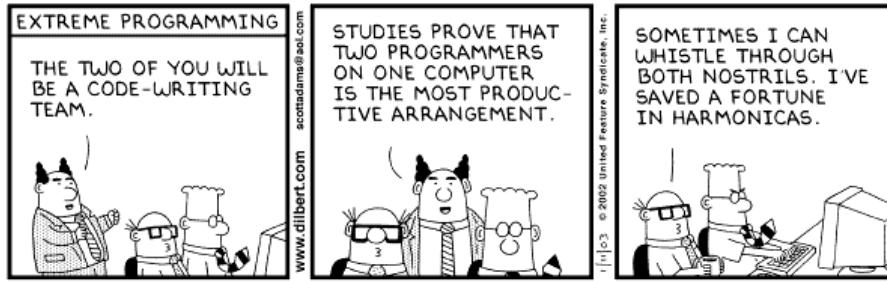
Da es im Extreme Programming keine herkömmliche Anforderungsspezifikation gibt, ist das Schreiben der Funktionstests nicht nur wesentlicher Bestandteil des gesamten Entwicklungsprozesses, sondern muss auch vor der Realisierung/Implementierung der eigentlichen Funktionen erfolgen. Nur so kann im Zweifel auch die Kundin entscheiden, ob das Ergebnis einer ihr vorgelegten Iteration (s. u.) die Anforderungen erfüllt. Dabei kann es natürlich vorkommen, dass ein Programm zwar die Funktionstests besteht, aber trotzdem nicht den Anforderungen genügt, oder dass ein Programm an den Funktionstests scheitert, aber dennoch die Anforderungen erfüllt. In beiden Fällen sind wahrscheinlich die Funktionstests falsch und müssen entsprechend überarbeitet werden.

Funktionstests ersetzen die Anforderungsspezifikation

7.4 Das Programmieren in Paaren

Neben dem Testen, das ja bekanntermaßen nur zeigen kann, dass Fehler im Programm sind, nicht, dass es fehlerfrei ist, sind sog. *Durchsichten* (engl. reviews) ein weiteres, höchst effizientes Mittel, Fehler zu entdecken und zu beseitigen. Während Durchsichten normalerweise eine unabhängige Aktivität sind, die in regelmäßigen Abständen (z. B. einmal wöchentlich) und bei speziell hierzu anberaumten Meetings durchgeführt werden, sind sie beim Extreme Programming quasi in den Programmierprozess integriert. Dies geschieht in der Praxis dadurch, dass die Programmiererinnen in Paaren programmieren, also zu zweit an einem Computer sitzen. Dabei schreibt die eine das Programm, so wie es ihr gerade einfällt, während die andere ihr dabei zuschaut und, da sie die Gedanken der anderen nicht kennt und entsprechend viel Abstand hat, sie auf mögliche Probleme oder Fehler hinweist.

ständige Durchsichten



Copyright © 2003 United Feature Syndicate, Inc.

In der Praxis muss man sich das etwa wie folgt vorstellen: Nach einer kurzen Diskussion, wie ein zu lösendes Problem (etwa das Schreiben einer Methode) angegangen werden soll, ergreift eine der beiden Programmiererinnen die Tastatur und beginnt mit dem Codieren. Die andere, die ihr dabei zuschaut, hat die Zeit und den mentalen Abstand, das, was gerade kodiert wird, kritisch zu hinterfragen und an mögliche Fälle zu denken, in denen die Lösung versagen könnte. So kann ein typischer Einwand der Beisitzerin sein, dass ein möglicher Testfall, der einen heiklen Punkt der Implementierung abdeckt, noch nicht geschrieben wurde. Auch kann sie darüber nachdenken, ob die gerade vorgenommene Ergänzung/Änderung Auswirkungen auf andere Teile des Codes hat und ob etwaige Änderungen im Design (*Refactorings*) sinnvoll wären, um eine saubere Struktur und gute Lesbarkeit des Codes insgesamt aufrechtzuerhalten. Es ist sogar erlaubt, dass die Beisitzerin der Programmiererin die Tastatur wegnimmt und selbst weitermacht, wenn sie meint, dass sie es besser könne.

Bedingungen des Programmierens in Paaren

Das Programmieren in Paaren setzt voraus, dass die beiden Partnerinnen sich gut verstehen, dass zumindest keine persönlichen Abneigungen bestehen. Dabei sollten die Paarzusammensetzungen jederzeit wechseln können. Wechseln ist insbesondere dann sinnvoll, wenn zu einer bestimmten Aufgabe jemand hinzugezogen werden soll, die damit schon gut vertraut ist (eine Expertin), oder wenn ein Stück Code geändert werden muss, das ursprünglich von einem anderen (Paar) geschrieben wurde. Programmieren in Paaren heißt insbesondere nicht, dass immer dieselben zusammenhocken.

Ökonomie des Programmierens in Paaren

Das Programmieren in Paaren mag wie eine Ressourcenverschwendug erscheinen: Da immer nur eine von zwei Programmiererinnen auch tatsächlich programmiert, könnte man meinen, dass die Produktivität sich ungefähr halbiert. Dem kann man folgendes entgegenhalten:

- Das Programmieren in Paaren führt zu einer höheren Qualität, da jede Zeile Code den Konsens mindestens zweier Programmiererinnen darstellt. Initial mehr verbrauchte Ressourcen werden daher gegen Ende des Projektes, wenn normalerweise die Fehler beseitigt werden, wieder eingespart.
- Das Wissen über den Code ist beim Programmieren in Paaren immer auf mindestens zwei Köpfe verteilt; der Ausfall oder Weggang eines einzelnen hat damit nicht die gleichen nachteiligen Konsequenzen wie im Normalfall. Dies ist insbesondere aufgrund der in der Branche immer noch üblichen hohen Mitarbeiterinnenfluktuation interessant.

- Durch das Programmieren in Paaren kann jede Programmiererin von anderen lernen. Langfristig steigt dadurch die Produktivität jeder einzelnen.

Auch wenn diese Punkte unmittelbar einleuchten, wird es in der Praxis doch schwierig sein, eine Projektverantwortliche davon zu überzeugen, dass sich das Programmieren in Paaren rentiert. Hierzu wird immer erst das Sammeln von eigenen Erfahrungen notwendig sein, die ein Vertrauen in diese Praxis aufzubauen. Dies ist umso mehr der Fall, als kontrollierte Studien, die die Produktivitätsunterschiede gemessen hätten, nicht verfügbar sind; es ist im Übrigen auch kaum vorstellbar, wie sich die dafür notwendigen Laborbedingungen (für eine Vergleichbarkeit der Ergebnisse) herstellen ließen.

7.5 Keine Planung

Sowohl der Test-first-Ansatz als auch das Programmieren in Paaren haben zur Folge, dass Fehler bereits bei ihrer Entstehung entdeckt und behoben werden können. Die Kosten von Änderungen, die sich aus der Beseitigung von Fehlern, die erst am Ende des Projekts entdeckt werden und die gegebenenfalls über mehrere Phasen des Projekts an ihren Ursprung zurückverfolgt werden müssen (wie es klassischerweise beim Wasserfallmodell der Fall ist; s. Abbildung 7.4), werden dadurch beseitigt oder zumindest deutlich reduziert. Tatsächlich ist der hohe Aufwand von Änderungen, die nicht nur auf Fehler im Code, sondern auch auf Fehler in der Spezifikation (in den Anforderungen) zurückzuführen sein können, einer der Gründe für das Nachdenken über immer neue Prozessmodelle.

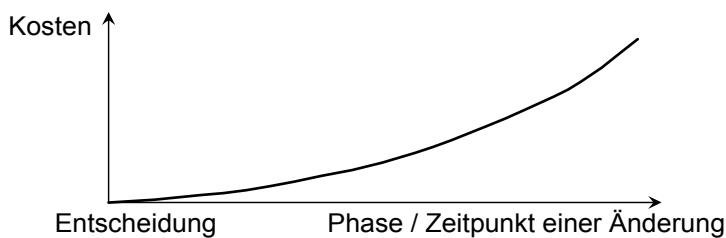


Abbildung 7.4: Kosten von Änderungen einer Entscheidung in Abhängigkeit von der Zeitspanne, die seit der Entscheidung vergangen ist. In der Literatur wird ein exponentieller Zusammenhang unterstellt [55].

Eine der Grundannahmen des Extreme Programming ist, dass Änderungen billig und jederzeit möglich sind. Dies wird einerseits dadurch erreicht, dass außer dem Code praktisch keinerlei Dokumentation angefertigt wird, die bei Änderungen entsprechend nachgepflegt werden müsste, andererseits dadurch, dass moderne Werkzeuge zur automatisierten Umgestaltung von Code (zum *Refactoring*) eingesetzt werden (soweit sie denn verfügbar sind). Wenn eine Änderung notwendig werden sollte (und zwar unabhängig davon, ob diese auf einen Programmierfehler oder auf eine Änderung der Anforderung zurückzuführen sein sollte), wird diese durchgeführt, selbst wenn man Gefahr laufen sollte, dass die Änderung nur vorläufig ist, also unter Umständen später durch eine weitere revidiert wird. Die (mutmaßliche) Kostenfunktion ist in Abbildung 7.5 dargestellt.

Die Kosten von Änderungen

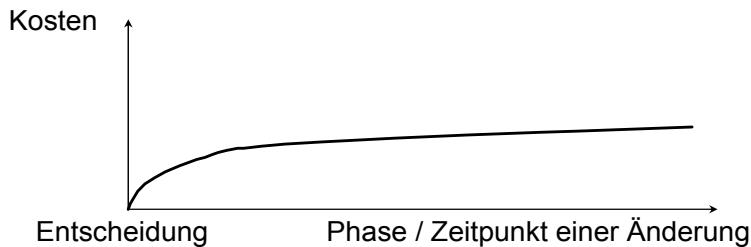


Abbildung 7.5: Kosten einer Änderung unter Verwendung geeigneter Werkzeuge (hypothetisch). Dies ist eine der Grundvoraussetzungen für die These des Extreme Programming, nämlich dass sich vorausschauende Planung nicht lohnt.

Einfachheit ersetzt Voraussicht

Tatsächlich ist eine der Herangehensweisen von Extreme Programming, nicht zu versuchen, in die Zukunft zu blicken und den Code für alle möglichen Eventualitäten passend zu schreiben. Vielmehr ist die Devise, alles so einfach wie möglich zu halten, zum einen, um keine unnötigen Energien zu vergeuden, zum anderen, um die einfache Änderbarkeit zu gewährleisten. Dies ist gar nicht so einfach, widerspricht es doch dem Naturell vieler Informatikerinnen, hinter jedem Problem, und sei es noch so klein, gleich das allgemeinere, umfassendere zu sehen und für dieses eine Lösung anzubieten.⁶⁸ Was aber, wenn die allgemeinere Lösung niemals gebraucht wird, oder — schlimmer noch — wenn sich schon das einfache Problem hinterher als falsch gestellt erweisen sollte? Dann war die ganze Mühe umsonst!

Und so verzichtet man im Extreme Programming weitgehend auf die Phasen der Projektplanung, der Anforderungserhebung und -analyse sowie des Softwareentwurfs. Selbst auf die heute so populäre Softwaremodellierung (z. B. mit UML) wird weitestgehend verzichtet. Diagramme sollen allenfalls in kurzen Besprechungen als Handskizze angefertigt werden, um einen Sachverhalt zu klären; sie sollen daraufhin aber sofort in Code umgesetzt werden. Dieser übernimmt dann auch die Funktion der Dokumentation — die Diagramme wandern in den Papierkorb.

Prinzip Rückkopplung

Dies bedarf natürlich der Erklärung. Anstatt ein Projekt im Vorhinein akribisch auf Papier zu planen, verfolgt das Extreme Programming den Ansatz der iterativen Softwareentwicklung: Das Produkt wird in einer Anzahl von Zyklen, Iterationen genannt, über mehrere Releases fertiggestellt (s. Abschnitt 7.8). Dabei setzt das Extreme Programming auf das Prinzip der Rückkopplung: Jede implementierte Funktionen wird bereits kurze Zeit nach Fertigstellung der Kundin vorgeführt, damit diese entscheiden kann, ob die Funktion ihren tatsächlichen Anforderungen genügt.

Anforderungen auf Karteikarten

Damit die Programmiererinnen wissen, welches die Anforderungen der Kundin sind, hat diese zuvor eine Menge von Karteikarten angefertigt, auf denen jeweils eine Systemfunktion geschrieben steht. Jedes Paar von Programmiererinnen nimmt sich, sobald es seine aktuelle Aufgabe abgeschlossen hat, eine neue Karteikarte und arbeitet die darauf befindliche Aufgabe ab. Die Karteikarten selbst übernehmen dabei gewissermaßen die Funktion der formlosen Anforderungsspezifikation (die noch durch Funktionstests formalisiert

⁶⁸ Das sog. Paradox des Erfinders (Inventor's Paradox) spricht sogar dafür: Das allgemeinere Problem ist manchmal das leichter lösbare.

werden müssen); die Reihung der Karteikarten in der Folge, in der sie abgearbeitet werden, ersetzt den Projektplan.

Um dennoch eine gewisse Kontrolle über den Fortschritt des Projekts etablieren zu können, wird das Produkt über eine (unbestimmte) Anzahl von Releases zur Vollendung gebracht. Jedes Release ist wiederum in eine Anzahl von sog. Iterationen aufgeteilt. Mit Iterationen ist allgemein eine bestimmte maximale Dauer (in der Regel ca. ein bis vier Wochen) verbunden. Aus dieser maximalen Dauer, gemeinsam mit der Produktivität der Programmiererinnen, ergibt sich das Pensum, das während einer Iteration bewältigt werden kann. Um dieses Pensum auf die Anzahl der Karteikarten umzurechnen, die während einer Iteration abgearbeitet werden können, schätzt jedes Programmiererinnenpaar den Aufwand je Karteikarte. Damit diese Schätzungen auch einigermaßen stimmen, ist es notwendig, dass jedes Paar in einer Art *persönlichem Softwareprozess* (PSP) [56] seine Produktivität ständig misst und damit seine Schätzungen überprüft sowie gegebenenfalls anpasst.

Aus der Vorgabe der festen Dauer einer Iteration und der Ableitung der Anzahl der zu bewältigen Anforderungen (Karteikarten) aus Iterationsdauer und Programmiererinnenproduktivität ergibt sich, dass der Umfang dessen, was innerhalb einer Iteration geschafft werden kann, nicht frei ansetzbar ist. Darin unterscheidet sich das Extreme Programming ganz wesentlich von klassischen Vorgehensweisen, bei denen zunächst der Funktionsumfang, dann der Zeitrahmen und schließlich die eingesetzten Ressourcen (Teamgröße) festgelegt werden. Tatsächlich sind beim Extreme Programming Überstunden verpönt: Was innerhalb einer Iteration nicht geschafft wird, wird eben nicht geschafft und wandert in die nächste Iteration. Die pünktliche Auslieferung des Produkts wird davon nicht ernsthaft gefährdet, da ja allenfalls der Termin für das nächste Release bereits geplant war. Sich ergebende schlechtere Produktivitätszahlen können dann bereits für das nächste Release berücksichtigt werden, so dass dessen Auslieferung wieder pünktlich erfolgen können sollte.

**Projektumfang wird
Produktivität und
Zeit angepaßt**

7.6 Die Kundin vor Ort

Die Kundin, von der eben die Rede war, ist beim Extreme Programming Teil des Projektteams. Da es in der Regel nicht nur eine Kundin gibt, hat sie in erster Linie Stellvertreterinnenfunktion: Sie muss die Kundinnen in ihrer Gesamtheit vertreten und mit diesen bei Bedarf, wenn sie selbst nicht weiter weiß, Rücksprache halten. Zwar ist auch diese Eigenheit des Extreme Programming nicht neu (manch eine Unternehmensberatung empfiehlt bereits seit längerem, bei größeren Projekten eine Kundin vor Ort zu haben), doch stellt sie eines der größten Hindernisse in der Praxis dar: Wenn es überhaupt die (prototypische) Kundin gibt (was bei Entwicklungen für den Massenmarkt ja gar nicht der Fall ist), dann ist diese häufig aufgrund ihrer Kompetenz nur schwer abkömmlig. Welche Firma könnte schon auf eine Person, die alles weiß, verzichten?

Als Kompromiss ist vorstellbar, dass die Kundin vor Ort in unmittelbarer Umgebung des Projektteams einen Arbeitsplatz eingerichtet bekommt, von dem aus sie ihrer angestammten Tätigkeit nachgehen kann. Sie könnte dann in der Zeit, in der das Projektteam sie nicht benötigt, normal arbeiten. Wichtig ist jedoch, dass sie diese Arbeit jederzeit unterbrechen kann, um bei etwaigen Unklarheiten, die auf Seiten des Projektteams bestehen, zur Klärung zur Verfügung zu stehen.

Weiß sie selbst keine Antwort, so hat sie die Möglichkeit, durch die Einbettung in ihren eigentlichen Arbeitsplatz jederzeit die benötigten Informationen abzurufen.

7.7 Gemeinsame Verantwortung

Das Prinzip der fehlenden Planung bedingt häufige Änderungen, und die bedingen wiederum, dass die Änderungen nicht immer von der ursprünglichen Autorin eines *Moduls* durchgeführt werden können. Tatsächlich gibt es im Extreme Programming strenggenommen nicht *die* Autorin eines Moduls. Vielmehr gehört der gesamte Code allen. Das hat allerdings mindestens zwei Konsequenzen:

1. Zwar kann die begnadete Programmiererin weiterhin brillieren, sie läuft jedoch Gefahr (und muss erdulden), dass eine andere, weniger brillante (in der Regel ignorante) Kollegin das eigene geniale Bauwerk wieder einreißt und durch ein viel profaneres ersetzt, nur weil sie es für zu unleserlich hält (oder vielleicht nur nicht verstanden hat). Welche leicht „Wer hat in meinem Code herumgepfuscht?“ denkt, für die ist Extreme Programming nichts.
2. Welche grundsätzlich der Ansicht ist, dass jede ihre Fehler selbst ausbaden sollte, die lässt ebenfalls am besten die Finger vom Extreme Programming. Sollte man nämlich, wenn man einen Fehler gefunden hat, diesen erst auf seine Verursacherin zurückverfolgen und ihn dann an diese verweisen wollen, so verliert man damit nicht nur unnötig viel Zeit — es steckt immer auch eine latente Schuldzuweisung dahinter, die besonders bei gehäuftem Auftreten die Stimmung im Team nicht gerade verbessert. Welche einen Fehler gefunden hat, die sollte ihn selbst verbessern, es sei denn, sie versteht die Zusammenhänge nicht und benötigt deswegen Hilfe. Ist letzteres der Fall, dann sollte sie dafür Sorge tragen, dass mit der Korrektur auch das Verständnisproblem behoben wird.

starkes Ego hinderlich

Ein stark ausgeprägtes Ego, wie es bei Programmiererinnen nicht gerade selten anzutreffen ist, ist beim Extreme Programming also eher hinderlich. Dies sollte man bei der Zusammenstellung eines Teams immer berücksichtigen. Allerdings — und da ist Extreme Programming durchaus konsequent — würden solche Persönlichkeiten in der Regel schon nicht die Prüfungen des Programmierens in Paaren bestehen, da sie vermutlich nicht allzu lange dabei zusehen könnten, wie jemand anderes als sie selbst programmiert.

Die gemeinsame Verantwortung bedingt auch, dass sich alle im Team auf gewisse Konventionen einigen. Insbesondere geht es nicht, dass sich jede, die ein Stück Code bearbeitet, dieses erst einmal so hinformatiert, dass es ihren Lesegewohnheiten oder Vorlieben entspricht. Stattdessen sollten sich alle an einheitliche Namenskonventionen und Formatierungsregeln halten. Für letzteres sind Werkzeuge vorhanden und äußerst hilfreich.

Einhalten von Konvention

7.8 Der Prozess des Extreme Programming

Der Softwareentwicklungsprozess des Extreme Programming ist ein iterativer.

Die iterative Softwareentwicklung hat allgemein den Vorteil, dass in (einiger-

**iterative Software-
entwicklung**

maßen) regelmäßigen Abständen (Zwischen-)Produkte vorgelegt werden können, anhand derer man Richtung und Fortschritt der Entwicklung kontrollieren kann. Das schafft Vertrauen und erlaubt zudem, Fehlentwicklungen frühzeitig zu erkennen, ihnen entgegenzuwirken oder sogar ein Projekt abzubrechen, bevor die gesamte veranschlagte Investitionssumme ausgegeben (und damit verloren) ist.⁶⁹

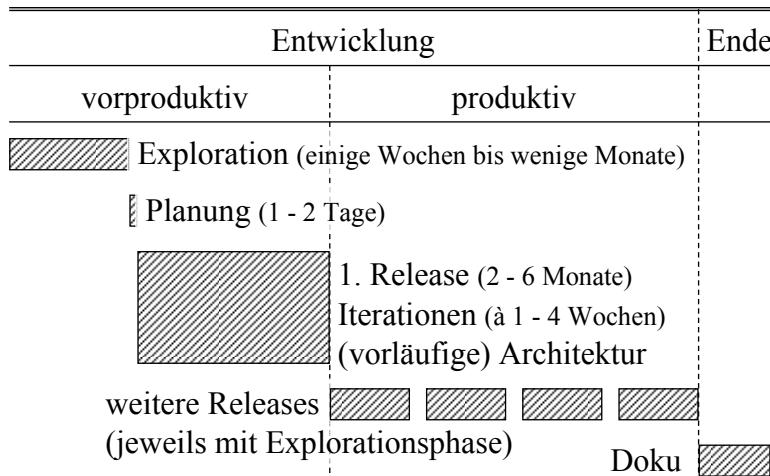


Abbildung 7.6: Lebenslauf eines Extreme-Programming-Projekts.

Das Extreme Programming nutzt die Möglichkeiten der iterativen Softwareentwicklung, um möglichst kurze Rückkopplungsschleifen in den Entwicklungsprozess einzubauen. So wird der Entwicklungszyklus eines Softwareprodukts in mehrere Releases eingeteilt, wobei deren Anzahl vorher nicht unbedingt genau feststeht. Dabei sollte ein Release mindestens zwei, in der Regel jedoch nicht länger als sechs Monate bis zu seiner Fertigstellung benötigen. Das Produkt wird dabei mit seinen wichtigsten Grundfunktionen beginnend (dem ersten Release) in seinem Funktionsumfang über die Folgereleases immer weiter ausgebaut (Abbildung 7.6). Neue Releases stehen also nicht für Modernisierung (wie das beispielsweise bei heutiger Standardsoftware der Fall ist), sondern für die stufenweise Erweiterung des Funktionsumfangs bis hin zum Endprodukt.

⁶⁹ Motto: „If you are going to fail, fail early!“

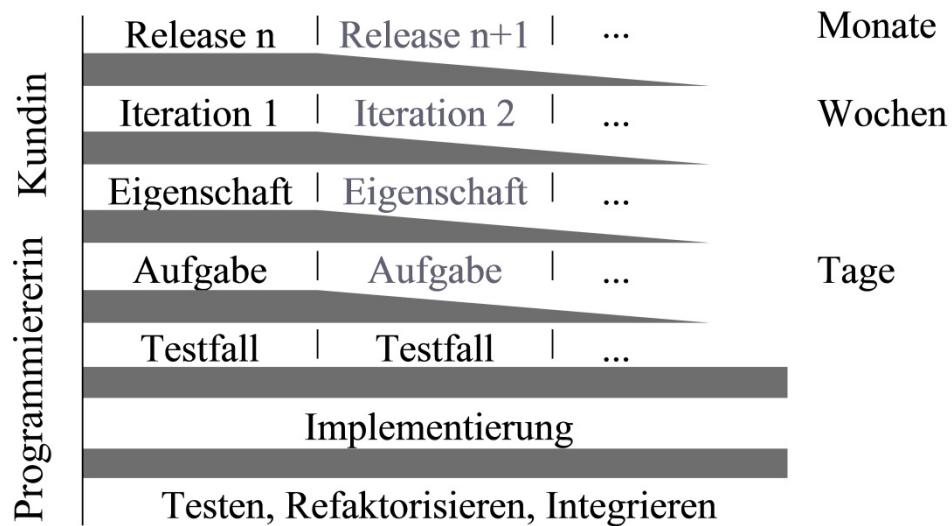


Abbildung 7.7: Gliederung des Prozesses.

Jedes Release ist selbst wieder in eine Anzahl Iterationen eingeteilt, wobei eine Iteration ca. ein bis vier Wochen dauern sollte. Innerhalb der Iterationen werden konkrete Eigenschaften des Produkts umgesetzt, wie sie von der Kundin in Form der sog. *User stories* festgehalten worden sind. Dabei zerfällt jede Eigenschaft in mehrere Teilaufgaben, die zur Implementierung neuer Methoden und Klassen führen. Im Rahmen der Abarbeitung der Aufgaben sind insbesondere auch die Unit-Tests zu formulieren und zu implementieren (Abbildung 7.7).

Vorteile kurzer Zyklen

Die relativ kurze Dauer der Iterationen hat den Vorteil, dass der Projektfortschritt für alle Beteiligten sichtbar wird. Dies wirkt vertrauensbildend auf die Kundin und motivierend auf die Entwicklerinnen. Außerdem verhindert es den Effekt, dass Aufgaben immer mehr in Richtung Abgabetermin verschoben werden, wodurch sich in dessen zeitlicher Nähe ein nicht mehr zu überwindender Berg von noch zu erledigendem ansammelt. Außerdem haben die kurzen Zyklen den bereits erwähnten Vorteil, dass Fehlentwicklungen sehr früh erkannt und mögliche Folgeschäden damit vermieden werden.

früh in den Produktionsbetrieb

Je nach Art des Projektes kann es zudem durchaus möglich sein, dass selbst frühe Releases schon „produktiv gehen“. Dies kann wiederum zur Folge haben, dass sich die ursprünglich geplanten Prioritäten zugunsten anderer Aufgabenstellungen verschieben. Dadurch, dass in die Folgereleases noch nichts investiert wurde, ist es dem Extreme Programming leicht möglich, den Plan zu ändern und die sich neu ergebende Aufgabenstellung sofort anzugehen.

Zusammenfassend lässt sich sagen, dass sich das Prozessmodell des Extreme Programming bemüht, so lebensnah wie möglich zu sein. Wie man aus eigener (Lebens-) Erfahrung jedoch weiß, ist manchmal Festlegung für den Fortschritt unentbehrlich. Es ist also grundsätzlich fraglich, ob die durch mangelnde Festlegung erkaufte Flexibilität immer und in allen Situationen von Vorteil ist.

7.9 Voraussetzungen für den Einsatz von Extreme Programming

Es wäre illusorisch, zu behaupten, dass man jedes beliebige Projekt nach der Methode des Extreme Programming bewältigen könnte. Vielmehr gibt es eine Reihe ganz harter Voraussetzungen, die erfüllt sein müssen, damit Extreme Programming überhaupt anwendbar ist. Diese Voraussetzungen ergeben sich zum guten Teil aus den Eigenheiten des Extreme Programming.

Zunächst einmal ist die Größe des Teams und damit indirekt auch die Größe des Projekts beschränkt. Die ideale Anzahl von Entwicklerinnen in einem Extreme-Programming-Projekt beträgt circa 10 Personen. Wenn man eine Produktentwicklungszeit von maximal drei Jahren ansetzt, ergibt sich daraus eine maximale Leistung von 30 Personenjahren. Wirklich große Softwareprojekte lassen sich damit nicht bewältigen. **Teamgröße**

Wodurch entsteht diese Obergrenze? Da es im Extreme Programming keine geschriebene Projektdokumentation gibt, setzt der Ansatz auf die Kommunikation aller Projektbeteiligten. Der Aufwand einer gleichmäßigen (das heißt, nicht hierarchischen) Kommunikation steigt aber im Quadrat mit der Anzahl der Teilnehmer — der Kommunikationsaufwand beschränkt also die Teamgröße (und damit indirekt auch die Projektgröße).

Eine Beschränkung der Teamgröße nach unten lässt sich aus den Vorgaben des Extreme Programming höchstens mit zwei ableiten — mit weniger Teilnehmern ist das Programmieren in Paaren nicht möglich. Allgemein gilt aber, dass Softwareentwicklungsprozessmodelle für größere Teams ausgelegt sind. Für kleinere Teams (wie etwa zwei Personen) ist der Verwaltungsaufwand in der Regel nicht gerechtfertigt. Auf der anderen Seite lassen sich viele Einzeltechniken des Extreme Programming, wie z. B. der Test-first-Ansatz, sehr gut auch allein (also in einem Ein-Personen-Projekt) einsetzen.

Eine weitere Grundvoraussetzung für den Einsatz des Extreme Programming ist, dass moderne Technik zum Einsatz kommen kann. So ist z. B. das Vorhandensein eines geeigneten Test-Frameworks (s. Kurseinheit 3) sowie von Werkzeugunterstützung für das Refactoring (s. Kurseinheit 5) unverzichtbar. In der Regel sind diese Werkzeuge heute vor allem für objektorientierte Programmiersprachen wie JAVA oder C# vorhanden. Die Ersteller des Extreme Programming behaupten darüber hinaus, dass objektorientierte Technologie zum Einsatz kommen muss — es ist jedoch nicht unmittelbar ersichtlich, welchen Einfluss die Objektorientierung, also das Vorhandensein von Klassen und Vererbung, auf den Entwicklungsprozess hat. **Einsatz moderner Technik**



keine Altlasten

Eng damit verwandt ist die Forderung, dass es sich nicht um ein Altlastenprojekt handeln darf. Altlasten zwingen in der Regel zu Kompromissen und widersetzen sich per se der leichten Änderbarkeit, deren Vorliegen eine der Grundvoraussetzungen des Extreme Programming ist. Gleichwohl ist vorstellbar, dass ein gut gekapseltes Altsystem in eine Weiterentwicklung, die auf Extreme Programming setzt, integrierbar ist.

Bereitschaft zum Schreiben von Unit-Tests

Zu den Voraussetzungen des Extreme Programming zählt auch die Bereitschaft, große Mengen an nicht produktivem Code zu schreiben. Tatsächlich sind die Unit-Tests, ähnlich wie die Vor- und Nachbedingungen aus dem Design by contract, nicht Bestandteil der Auslieferung bzw. des Codes, der die eigentliche Arbeit verrichtet. Außerdem ist für das Schreiben der Unit-Tests ein gewisser Sportgeist erforderlich: Jede ist selbst dafür verantwortlich, wie hoch sie die Messlatte, die der eigene Code nehmen muss, auflegt.

unklare Anforderungen

Im Übrigen sollte man sich darüber im Klaren sein, dass Extreme Programming aufgrund seiner Vorgehensweise dann besonders gut geeignet ist, wenn die Anforderungen zunächst noch unklar sind (und das Entwicklungsrisiko entsprechend hoch ist). Wenn das Produkt hingegen vollständig durchspezifiziert ist, z. B. weil die Aufgabe lautet, ein bestehendes Altsystem mit komplett identischem Funktionsumfang neu zu implementieren, dann besteht kaum Bedarf an Rückkopplung, weswegen man sich durch die iterative Anlage des Extreme Programming auch keinen Vorteil verschafft. In solchen Fällen können die Nachteile überwiegen.

Experimentierfreudige Kundin

Nicht zuletzt erfordert ein Vorgehen nach der Methode des Extreme Programming eine Kundin, die bereit ist, sich darauf einzulassen. Insbesondere die feste Einbindung der Kundin in die Iterationen bedingt eine Verlagerung der Verantwortung für das Gelingen des Produkts weg von der Auftragnehmerin allein hin zu einer Art Schicksalsgemeinschaft von Kundin und Lieferantin. Dies kann insbesondere dem Interesse der Kundin entgegenlaufen, und zwar immer dann, wenn sie es vorzieht, einen Auftrag zu vergeben, bis zur Lieferung nichts mehr damit zu tun zu haben und gegebenenfalls bei Nichterfüllung oder mangelhafter Erfüllung zu wandeln. Insbesondere die Forderung, für die Laufzeit des Projektes eine Mitarbeiterin abzustellen, die vorrangig für die Auftragnehmerin zur Verfügung steht, wird bei mancher Haushaltsverantwortlichen Stirnrunzeln auslösen. Aber auch eine Vorgehensweise ohne genaue Spezifikation, Planung, Analyse oder irgendeine Form der Dokumentation, die nicht in Code gegossen wäre, verlangt eine gewisse Experimentierfreude seitens derer, die das Geld dafür lockermachen soll.

Alternative zu Code and fix

Alles in allem bleibt in der Praxis vermutlich — abgesehen von Ausnahmefällen — hauptsächlich die Möglichkeit, Extreme Programming in einem Projekt einzusetzen, in dem keinerlei Vorgaben bezüglich des Entwicklungsprozesses gemacht werden. Da man dort im schlimmsten Fall auch nach dem Muster des Code and fix vorgehen könnte, dürfte ein sich an den Vorgaben des Extreme Programming Orientieren vermutlich zu vertretbaren Ergebnissen führen.

7.10 Werkzeuge des Extreme Programming

Wie bereits erwähnt, ist Extreme Programming in hohem Maße von der Verfügbarkeit geeigneter Werkzeuge abhängig. Im Einzelnen sind dies:

- *Syntaxeditoren mit Auto-Vervollständigung und automatischer Codeformatierung*: Da es im Extreme Programming nur das Prinzip der gemeinsamen Verantwortung für den Code gibt, also insbesondere niemand da ist, die allein bestimmte Standards wie z. B. die Benennung von Komponenten vorgeben kann, ist es nötig, dass jede Programmiererin flexibel auf Umstrukturierungen und Namensänderungen reagieren kann. Ein Syntaxeditor mit automatischer Vervollständigung von Bezeichnern, wie er in modernen Entwicklungsumgebungen üblich ist, bietet dabei unschätzbare Hilfe. Um nicht ständig über andere als die gewohnten Formen der Codeformatierung zu stolpern, ist die automatische (Überprüfung der) Einhaltung von Codeformatierungsrichtlinien ebenfalls wünschenswert.
- *Refactoring-Werkzeuge*: Wer planlos vorgeht, muss jederzeit zu Änderungen bereit sein. Dabei darf das planlose Vorgehen nebst den notwendig werdenden Änderungen im Mittel nicht mehr Zeit in Anspruch nehmen, als es ein planvolles Vorgehen getan hätte. Da der Zeitverbrauch eines planlosen Vorgehens kaum beeinflussbar ist, sind systematische Zeiteinsparungen nur bei den Änderungen zu erwarten. Diese Zeiteinsparungen lassen sich in der Praxis durch den systematischen Einsatz von Refactoring-Werkzeugen erzielen. Allerdings ist die Entwicklung von solchen Werkzeugen höchst aufwendig — auch wenn ihre Zahl ständig wächst, müssen doch immer noch viele Änderungen von Hand durchgeführt werden. Dies gilt insbesondere für die sog. *großen Refactorings*, deren Zerlegung in mehrere kleinere zeitaufwendig und fehleranfällig ist.
- *Versionskontrolle*: Für die Entwicklung im Team eigentlich eine Selbstverständlichkeit, jedoch längst noch nicht überall eingesetzt sind Werkzeuge zur Versionskontrolle wie z. B. CVS, SVN oder Git. Nur wenn sichergestellt ist, dass jede jederzeit Zugriff auf die aktuellste Version des gesamten Programms hat, kann das Prinzip der gemeinsamen Verantwortung für den Code (das ja jeder erlaubt, jederzeit beliebige Änderungen durchzuführen) überhaupt funktionieren. Auch verlangen die kurzen Iterationszyklen, dass in regelmäßigen Abständen vollständig kompilierbare Versionen des Produkts vorliegen. Auch das lässt sich mit Hilfe einer Versionskontrolle vergleichsweise einfach bewerkstelligen.



- *Builder*: Je nach eingesetzten Technologien sind auch Werkzeuge zur automatischen Erstellung einer ausführbaren Version des Produkts (sog. Builder) hilfreich. Dies ist insbesondere dann der Fall, wenn das Endprodukt aus mehreren Teilprodukten zusammengesetzt werden muss, die mit unterschiedlichen Werkzeugen oder in unterschiedlichen Umgebungen entwickelt werden. Da beim Extreme Programming regelmäßig und in kurzen Abständen vollständige Versionen des Produkts angefertigt werden müssen, ist ein immer wieder neues, manuelles Zusammenbauen nicht praktikabel.

**nur geringe
Investitionen nötig**

Man beachte, dass von all diesen benötigten Werkzeugen bereits heute sehr gute, freie Implementierungen erhältlich sind. Dazu kommt, dass Entwicklungsumgebungen wie ECLIPSE all diese Werkzeuge bündeln und unter einer gemeinsamen Oberfläche zur Verfügung stellen. Das Extreme Programming ist damit eine Methode, die mit sehr geringen Investitionen in die Infrastruktur zurechtkommt.

7.11 Extreme Programming als risikogetriebene Methode

Risk item	Risk-management technique
Personnel shortfalls	Staffing with top talent, job matching, team building, key personnel agreements, cross training.
Unrealistic schedules and budgets	Detailed multisource cost and schedule estimation, design to cost, incremental development, software reuse, requirements scrubbing.
Developing the wrong functions and properties	Organization analysis, operations-concept formulation, user surveys and user participation, prototyping, early users' manuals, off-nominal performance analysis, quality-factor analysis.
Developing the wrong user interface	Prototyping, scenarios, task analysis, user participation.
Gold-plating	Requirements scrubbing, prototyping, cost-benefit analysis, designing to cost.
Continuing stream of requirements changes	High change threshold, information hiding, incremental development (deferring changes to later increments).
Shortfalls in externally furnished components	Benchmarking, inspection, reference checking, compatibility analysis.
Shortfalls in externally performed tasks	Reference checking, preaward audits, award-fee contracts, competitive design or prototyping, team-building.
Real-time performance shortfalls	Simulation, benchmarking, modeling, prototyping, instrumentation, tuning.
Straining computer-science capabilities	Technical analysis, cost-benefit analysis, prototyping, reference checking.

Abbildung 7.8: Liste der 10 größten Risiken der Softwareentwicklung mit ihren möglichen Abhilfen im Original (aus [57])

Extreme Programming ist eine risikogetriebene Methode. Das bedeutet, dass alle Aktivitäten des Extreme Programming darauf ausgerichtet sind, Risiko zu vermeiden oder, wenn das nicht möglich sein sollte, flexibel genug zu bleiben, um auf etwaige ungünstige Verläufe (eben das Risiko) reagieren zu können. Eine solche Vorgehensweise ist der Realität geschuldet und war schon Grundlage eines früheren Prozessmodells, nämlich des Spiralmodells. Von dessen Autor, Barry Boehm, stammt auch die berühmte Liste der 10 häufigsten Risiken in der Softwareentwicklung [57] :

1. Personalknappheit
2. unrealistische Zeitpläne und Budgets
3. Entwicklung der falschen Funktionen
4. Entwicklung der falschen Benutzungsschnittstelle
5. Vergoldung (Überausstattung)
6. ständige Anforderungsänderungen
7. Mängel in zugelieferten Produkten
8. Mängel in externen Leistungen
9. Mängel im Echtzeitverhalten
10. Anforderungen jenseits des derzeit technisch Möglichen

Seit dem Ende der achtziger Jahre, zu dem diese Liste entstanden ist, ist ein weiteres, elftes Risiko hinzugekommen: das Risiko der sich ständig ändernden Basistechnologien. Wer heute ein Projekt mit einer Laufzeit von drei Jahren durchführt, muss damit rechnen, dass während dieser Zeit mindestens einmal das Betriebssystem, mindestens zweimal die Programmiersprache, mindestens dreimal die Entwicklungsumgebung und mindestens sechsmal die verwendeten Frameworks die Version wechseln (wenn Wartung und Pflege nicht ganz eingestellt werden). Zu guter Letzt bleibt natürlich noch das Risiko, dass das ausgelieferte Produkt nicht seiner Spezifikation genügt, also Fehler enthält, die von seinen Entwicklerinnen nicht aufgedeckt wurden.

neue Risiken

Extreme Programming hat nun für die meisten dieser Risiken eine Antwort parat. Personalproblemen allgemein soll durch eine gute Behandlung der Mitarbeiterinnen (keine Überstunden!) sowie durch die Doppelbesetzung bei der Programmierung entgegengewirkt werden. Unrealistische Zeitpläne und Budgets sowie die durch ständige Anforderungsänderungen verursachten Probleme werden durch eine inkrementelle Entwicklung ohne allumfassende, endgültige Zielvereinbarung vermieden. Die Entwicklung der falschen Funktionen sowie die der falschen Benutzungsschnittstelle werden durch die enge Anbindung der Kundin („Kundin vor Ort“) verhindert. Einer Vergoldung (Überausstattung) steht das Prinzip der Einfachheit (s. u.) entgegen. Anforderungen jenseits des derzeit technisch Möglichen können durch eine Explorationsphase zu Beginn des Projekts (Abbildung 7.6) aufgedeckt werden. Mängel im Echtzeitverhalten lassen sich durch ständiges Releasen in den Produktivbetrieb zumindest frühzeitig feststellen. Mängel in zugelieferten Produkten sowie Mängel in externen Leistungen hingegen lassen sich durch kein, wie auch immer geartetes Prozessmodell vermeiden.

**Antwort
des Extreme
Programming**



Copyright © 2003 United Feature Syndicate, Inc.

7.12 Zusammenfassung

Zusammenfassend lässt sich Extreme Programming durch eine Menge von Prinzipien beschreiben:

1. *Das Prinzip der kleinen Schritte*: Das Projekt wird projektbegleitend geplant und spezifiziert, die Tests sind die Spezifikation. Überprüfungen der Korrektheit (Verifikation und Validation) finden laufend statt.
2. *Das Prinzip der Einfachheit*: Es gibt keine Analyse-/Modellierungsphase und der Softwareentwurf (das Design) findet während der Implementierung statt. Etwaige Fehlentwicklungen werden durch ständiges Refaktorieren korrigiert.
3. *Das Prinzip der gemeinsamen Verantwortung*: Es gibt keine Autorin eines Moduls — jede kann alles jederzeit ändern. Bugs werden von der beseitigt, die sie findet.
4. *Das Prinzip des sprechenden Codes*: Es gibt keine separate Dokumentation, vielmehr dokumentieren die Tests und der Code sich selbst.
5. *Das Prinzip der Kundin vor Ort*: Eine Kundin steht während der gesamten Projektlaufzeit zur Verfügung, liefert die Anforderungen und nimmt die Abnahme einzelner Funktionen sowie des gesamten Systems vor.
6. *Das Prinzip keine Überstunden*: Das Arbeitspensum wird durch die verbleibende Zeit (bis zum nächsten Release) mal Produktivität bestimmt.
7. *Das Prinzip Rückkopplung*: Charakteristisch für das Extreme Programming ist, dass in ihm für die Softwareentwicklung außergewöhnlich kurze Rückkopplungsschleifen eingebaut sind. Die kürzeste Schleife ist die über die Unit-Tests: Jede Änderung an einem Programm, und sei sie noch so klein, wird sofort auf ihre Gültigkeit überprüft, indem alle Tests noch einmal ausgeführt werden. Die nächste Schleife ist die der Integration: Da in jedem Projekt regelmäßig integriert wird (zum Beispiel jede Nacht), werden Fehler, die zu einer mangelnden Integrierbarkeit des Projekts führen, ebenfalls sehr schnell entdeckt. Weitere Rückkopplungsschleifen ergeben sich auf natürliche Weise aus den Iterationen und den Releases, in denen jeweils die Kundin als rückkoppelndes Element eingebunden ist.

7.13 Übergang zu agilen Prozessen

Zwar nimmt das Extreme Programming für sich in Anspruch, Flexibilität und Wandel im Projekt perfekt zu unterstützen, doch dummerweise gilt das nicht für den Prozess selbst — das Prozessmodell „Extreme Programming“ ist starr festgeschrieben, ja man soll sich sklavisch daran halten, da sonst ein Erfolg nicht garantiert sei. Bei der klassischen Informatikübung der Selbstanwendung fällt das Extreme Programming somit glatt durch.

Vom allzu Dogmatischen des Extreme Programming abgesetzt hat sich inzwischen eine Art Bewegung der *agilen Prozesse*. Darin wird anerkannt, dass die klassischen Prozessmodelle zu starr sind und die grundlegende Idee, einen programmiererinnzentrierten Ansatz zu verfolgen, nicht wegen der Beschränkungen des Extreme Programming verworfen werden sollte. Um einen Claim für solche Ansätze abzustecken, wurde das sog. „Manifesto for Agile Software Development“⁷⁰ verfasst, dessen Kernaussage die folgende ist:

*We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:*

*Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan*

*That is, while there is value in the items on
the right, we value the items on the left more.*

Das liest sich doch schon deutlich kompromissbereiter.

Unter dem Sammelbegriff der agilen Prozesse tummeln sich inzwischen einige Prozessmodelle, von denen Scrum vielleicht das bekannteste ist. Mir ist jedoch nicht klar, ob und mit welchem Erfolg welche dieser Prozesse in der Praxis eingesetzt werden — hier sind sicher noch mehr Zeit und darin gesammelte Erfahrungen erforderlich. Bis dahin bleibt festzuhalten, dass den agilen Prozessen vor allem eines gelungen ist: das Bewusstsein vieler Projektverantwortlicher für die Probleme überliefelter Prozessmodelle zu wecken und zum Reflektieren über die eigenen Praktiken und Prinzipien anzuregen. So sagte denn auch ein Prozessverantwortlicher einer bekannten deutschen Beratungs- und Softwareentwicklungs firma unlängst zu mir: „Wir wollen agiler werden!“

70 <http://agilemanifesto.org/>

7.14 Weiterführende Literatur

Der Klassiker zum Thema ist sicher [54] von Kent Beck. Das Buch ist dünn und gut zu lesen, auch wenn man des Autors Stil dafür mögen muss. Allzu tiefe Reflektionen über Extreme Programming — insbesondere eine kritische Auseinandersetzung damit — darf man natürlich nicht erwarten. Insofern kann diese Kurseinheit in gewisser Weise als eine Gegenposition gelesen werden; wer mehr will, wird vielleicht mit [58] glücklich.

Ein Klassiker ganz anderen Kalibers, den jede, die mit Softwareentwicklung im größeren Stil zu tun hat (Softwareprojektmanagerinnen insbesondere), zumindest im Schrank stehen haben sollte, ist [55] : eine schier unerschöpfliche Quelle von Weisheiten ohne Verfallsdatum!

- [54] K Beck Extreme Programming Explained: Embrace Change (Addison-Wesley 2000).
- [55] FP Brooks The Mythical Man Month Jubiläumsausgabe (Addison-Wesley, 1995).
- [56] <http://www.sei.cmu.edu/tsp/psp.html>
- [57] B Boehm „Software risk management: principles and practices“ IEEE Software 8:1 (1991) 32–41.
- [58] http://www.softwarereality.com/lifecycle/xp/case_against_xp.jsp

Index

- Abhängigkeitsrelation 19
- abstrakter Syntaxbaum 162, 174, 244
- Access modifier 13
- Adapter 160
 - klasse 158
 - objekt 159
- Pattern s. Pattern
- Advice 258
- agile Prozeßmodelle 266
- agiler Prozeß 285
- Annotation 22, 94, 97, 102, 118, 240, 263
 - Marker- 243
 - Meta- 242
- Annotation Processing Tool 244
- Annotationen 14
- Annotationsinstanz 243
- Annotationsprozessor 244
- Annotationstyp 241
 - DataPoint 104
 - Retention 242
 - Target 242
 - TestCase 241
- APT s. Annotation Processing Tool
- Aspekt 168, 245
- Aspektmechanismus 251
- aspektorientierte Programmierung 168, 244, 263
- Attribute 14, 240
- Aufrufabhängigkeit 19
 - umgekehrte 130
 - von Subklasse zu Superklasse 130
- Aufrufabhängigkeiten von oben nach unten 45
- Bottom-up-Testen s. Testen
- Call dependency 19
- Callback 36
- Client für die Aufruferin 19
- Code and fix 266
- Collective code ownership 270
- Command 68
- Composite Pattern s. Pattern
- Composition Filters 246
- Constructor injection 40
- Cross cast 48, 142
- Crosscutting concern 161, 245
 - im Kontext des Visitor Patterns 161

Defect seeding 123
defensive Programmierung 188
Delegate 36, 147
Delegation **134, 150, 217**
 vs. Forwarding **133**
dependency 19
Dependency injection **25, 39, 46, 116, 152**
Dependency Inversion Principle **46, 51**
Design by contract **54, 107, 177**
 Verified 124, 125
DLL hell 22
domänenpezifische Sprache 244
Double dispatch **167**
Durchsichten 271
dynamisch typisierte Sprache 124
dynamische Sprache 124
dynamisches Binden **11, 94, 118, 184, 218, 239**
Einzelfehlerannahme 119
Entwurfsmuster **44, 87, 134**
ermöglichendes Interface 117
Error
 vs. Failure 92
Error seeding 123
Extreme Programming **172, 266**
Facade Pattern *s. Pattern*
Factories 31
Factory **43, 152**
Factory Method Pattern *s. Pattern*
Factory-Methode **108, 152**
Factory-Methoden 25
Failure
 vs. Error 92
falsch negatives Ergebnis 122
falsch positives Ergebnis 122
Fassadenklasse **160**
Fehlerlokalisierung **117**
 Granularität der 118
Fehlerlokator **119**
 abdeckungsbasierter **119**
 A-priori- 120
 perfekter 119
Fixture **108**
Fokussierung durch Interfaces 21
Forwarding **133, 150, 217**
 vs. Delegation **133, 176**
Fragile base class problem 11, 51
Framework **156**
 Black Box- 34
 Black-box- 147
 White-box- 132, 149
Frameworks
 White Box- 34
funktionale Eigenschaft 247
funktionalen Dekomposition, 45
Funktionalität 53
Funktionskonstante 221
Funktionstests 271
Generalisierung **18, 21, 50, 208**
 echte 155
 unzulässige 210
genetische Programmierung 235
Gesetz Demeters 228
Guarded command **181, 192**
Hollywood-Prinzip 34
Hook **33, 39**
 -Methode 148, 149
Implementation inheritance **14, 17**
impliziter Parameter **156**
Integrationstest 124
Interface
 Abstraktheit von 107
 allgemeines 31
 anbietendes **29, 30, 32, 33**
 anbietendes 117
 angebotenes 19
 angebotenes 157
 benötigt 46
 benötigtes 19
 benötigtes 157
 Client/Server **32**
 Client/Server- 158
 ermöglichendes **29, 33**
 ermöglichendes 147
 explizite Implementierung 32
 Familien- **31**
 Familien- 155
 Idiosynkratisches **30**
 Interpretation als Rolle **48**
 kontextspezifisches **32, 33**
 kontextspezifisches 213

Marker- 243
öffentlichtes 22
partielles 21, 32
provided 19
provided 157
public 22
published 22
required 19
required 157
Server/Client 34
Server/Client- 147
Server/Item- 36
Tagging- oder Marker 38
totales 20, 29, 31, 32
veröffentlichtes 22
Interface inheritance 14, 17
Interface injection 41
Interface Segregation Principle 21, 51
Interfaces
 allgemeines 29
 Interzeption s. Interzession
 Interzession 116, 236, 239, 263
 Introduction 257
 Introspektion 236, 263
 Invariante 56
 Inversion of control 33
 Joinpoint 258
 -Modell 258
 Klasseninterface 17, 22, 30, 31, 180, 228
 Klasseninterfaces. 13
 Klasseninvarianten 56
 komponentenbasierten Programmierung 19
Komposition
 vs. Vererbung 133
konstante Methode 115
kontinuierliches Testen 121
Law of Demeter 200, 228
Lokalitätseigenschaft
 von Programmen 245
Mehrfachvererbung 14, 21, 31
Metadaten 240, 263
Metaprogramm 173
Metaprogrammierung 42, 115, 116, 208, 216, 234
 wenig gefährliche Teile der 263
Mix in 145
Mock-Objekt 111
modellbasierte Diagnose 120
Modifikation 236, 239
Modul 11, 276
Modul Packages als 12
Modularisierung 245
 und aspektorientierte Programmierung 253
Module getrennte Kompilierung 13
Modulgrenze 11
Multi dispatch 167
Multicast 143
Mutation testing 123
Nachbedingung 55, 176
 eines Refactorings 173
nichtfunktionale Eigenschaften 247
nominale Typkonformität 17, 53
Normalisierung 205
objektbasiert 12
Objektschizophrenie 151
Obliviousness 253
Observer Pattern s. Pattern
offene Rekursion 34, 35, 130, 148
Package 12
Parameterized factory method 156
Parameterobjekt 149, 190
Pattern
 Abstract Factory 157
 Acyclic visitor 167
 Adapter 157
 Composite 89, 102, 135
 Extension object 167
 Facade 160
 Factory Method 152
 Observer 36
 Observer 92
 Observer 143
 Proxy 239
 Role Object 150, 161
 Singleton 188, 190
 Strategy 88, 149
 Template Method 93, 148
 Visitor 161, 163
persönlicher Softwareprozeß 275
Plug point 17, 33, 39
Pointcut 258

polymorphe Konstantenmethoden 215
Polymorphismus 11, 13, **156, 183, 211**
postcondition s. Nachbedingung
precondition s. Vorbedingung
primitive Methode 148
Properties 178
Protokolle 14
Proxy Pattern 264
Publish-subscribe-Verfahren 143
Quantification 253
Query 68, 74
Refactoring 26, 50, 272, 273
 Bedingung durch Polymorphismus ersetzen 156
 großes **173, 174, 231, 281**
 Replace Inheritance with Delegation **176**
 Self encapsulate field 198
 Testen nach dem 271
 to patterns **174, 231**
 Vererbung durch Delegation ersetzen 159
Refactoring-Browser 172
Refactorings s. Inhaltsverzeichnis
Referenzobjekt 202
Reflection 191
Regressionstest s. Test
Regressionstesten s. Testen
Requirements drift 172
Role Object Pattern s. Pattern
Rolle 19, 31, 32, 135, 136, 200, 215, 219
 in Entwurfsmustern 129, 150, 168, 260
 Realisierung durch das Role Object Pattern s. Pattern
 Realisierung durch ein Interface **48**
Rollenwechsel 142
Scattering 248
Schnittstelle /t 12
Separation of concerns s. Trennung der Belange
Server
 für die Aufgerufene 19
Service locator 43
Setter injection **41**
single-fault assumption s. Einzelfehlerannahme
Softwarefäulnis 171
Spezialisierungsabhängigkeit 46
Strategy Pattern s. Pattern
strukturelle Typkonformität 53
strukturierte Programmierung 255
Subclassing 38, 65, 129, 218
Subject Oriented Programming 246
Substituierbarkeit 14, 209, 217
Subtyp-Abhängigkeit 46
Subtyping 65, 155, 214, 217
Tagging- oder Marker-Interface 14
Tangling 248
Template Method Pattern s. Pattern
Test
 Abhängigkeit vom Zustand 105
 Regressions- **85, s. a. Testen**
 Test fixture **88, 107**
 Testabdeckung 118
 Testen
 Back-to-back- 174
 Bottom-up- 122
 Regressions- 270
 von Tests **121**
 Testfall **84**
 Testframework **85**
 Testorakel 117
 Testsuite **87**
 Theorie **104**
 Trennung der Belange 255
 mehrdimensionale 246
 Typfehler
 Laufzeit- 48
 -umwandlungs- 48
 Typinferenz 214
 Typkonformität
 strukturelle **17**
 Typprüfung
 starke 115
 statische 48, 124
 Zusammenhang mit Unit testen und Design by
 contract **123**
Typsystem 124
 Hindernis 115
 nominales 177
Umkehrung der Ausführungskontrolle 33, 35, 40, 46,
 132, 147, 149
User stories 278
Vererbung 11, 34, 130, 161, 279
 Ächtung der 51
 auskommen ohne 159

- durch Delegation ersetzen **150, 159, 217**
- Infragestellung der **129**
- Mehrfach **14**
- unter Objekten **134**
- von Fixtures **103**
- von Testfällen **107**
- von Vor- und Nachbedingung **65**
- vs. Komposition **133**
- Wiederverwendung durch **230**
- Vererbung zwischen Klassen **17**
- Vererbungsabhängigkeit **46**
- Vererbungsinterface **129**
 - Deklaration **130**
 - fehlendes oder implizites **133**
 - implizites **35**
- Vererbungsmechanismus **168**
- Vertrag **29, 54**
- eines Interfaces **107**
- virtueller Konstruktor **156**
- Visitor Pattern *s. Pattern*
- Vorbedingung **55, 176**
 - eines Refactorings **173**
- Wasserfallmodell **268**
- Weben **251**
- Webeplan **251**
- Weber **251**
- Wertobjekt **201, 202**
- Wiederholungsgruppen **205**
- Zusicherung **55**
 - trivialerweise gegeben **56**
- Zustand **219**
 - eines Mock-Objekts **114**
- Objekt- **225**
- änderungen **36**

002572400
(10/19)

01853-2-01-S 1



Alle Rechte vorbehalten
© 2019 FernUniversität in Hagen
Fakultät für Mathematik und Informatik