Skriptum zu 185.A01

Objektorientierte Programmiertechniken

im Wintersemester 2014/2015

Franz Puntigam
Andreas Krall

Technische Universität Wien
Institut für Computersprachen
# Inhaltsverzeichnis

1 Paradigmen der Programmierung 9
   1.1 Berechnungsmodell und Programmstruktur ................. 10
      1.1.1 Formaler Hintergrund .................................. 10
      1.1.2 Praktische Realisierung ................................ 14
      1.1.3 Evolution und Struktur im Kleinen .................... 17
   1.2 Programmorganisation ................................... 22
      1.2.1 Modularisierung .................................... 22
      1.2.2 Parametrisierung .................................. 27
      1.2.3 Ersetzbarkeit ..................................... 31
   1.3 Typisierung ............................................. 34
      1.3.1 Typkonsistenz, Verständlichkeit und Planbarkeit ... 34
      1.3.2 Abstraktion ....................................... 38
      1.3.3 Gestaltungsspielraum ................................ 42
   1.4 Objektorientierte Programmierung .......................... 45
      1.4.1 Basiskonzepte ...................................... 45
      1.4.2 Polymorphismus ................................... 50
      1.4.3 Vorgehensweisen in der Programmierung ............ 53
      1.4.4 Wiederverwendung und Paradigmenwahl ............ 57
   1.5 Wiederholungsfragen ..................................... 59

2 Untertypen und Vererbung 65
   2.1 Das Ersetzbarkeitsprinzip ................................ 65
      2.1.1 Untertypen und Schnittstellen ......................... 65
      2.1.2 Untertypen und Codewiederverwendung ............... 71
      2.1.3 Dynamisches Binden ................................ 74
   2.2 Ersetzbarkeit und Objektverhalten ......................... 77
      2.2.1 Client-Server-Beziehungen .......................... 78
      2.2.2 Untertypen und Verhalten ............................ 85
      2.2.3 Abstrakte Klassen .................................. 91
   2.3 Vererbung versus Ersetzbarkeit ............................ 93
      2.3.1 Reale Welt, Vererbung, Ersetzbarkeit ............... 93
Inhaltsverzeichnis

<table>
<thead>
<tr>
<th>Abschnitt</th>
<th>Seitennummer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.2 Vererbung und Codewiederverwendung</td>
<td>95</td>
</tr>
<tr>
<td>2.3.3 Fehlvermeidung</td>
<td>100</td>
</tr>
<tr>
<td>2.4 Exkurs: Klassen und Vererbung in Java</td>
<td>102</td>
</tr>
<tr>
<td>2.4.1 Klassen in Java</td>
<td>102</td>
</tr>
<tr>
<td>2.4.2 Vererbung und Interfaces in Java</td>
<td>107</td>
</tr>
<tr>
<td>2.4.3 Pakete und Zugriffskontrolle in Java</td>
<td>110</td>
</tr>
<tr>
<td>2.5 Wiederholungsfragen</td>
<td>115</td>
</tr>
<tr>
<td>3 Generizität und Ad-hoc-Polymorphismus</td>
<td>119</td>
</tr>
<tr>
<td>3.1 Generizität</td>
<td>119</td>
</tr>
<tr>
<td>3.1.1 Wozu Generizität?</td>
<td>119</td>
</tr>
<tr>
<td>3.1.2 Einfache Generizität in Java</td>
<td>121</td>
</tr>
<tr>
<td>3.1.3 Gebundene Generizität in Java</td>
<td>125</td>
</tr>
<tr>
<td>3.2 Verwendung von Generizität im Allgemeinen</td>
<td>131</td>
</tr>
<tr>
<td>3.2.1 Richtlinien für die Verwendung von Generizität</td>
<td>132</td>
</tr>
<tr>
<td>3.2.2 Arten der Generizität</td>
<td>136</td>
</tr>
<tr>
<td>3.3 Typabfragen und Typumwandlungen</td>
<td>140</td>
</tr>
<tr>
<td>3.3.1 Verwendung dynamischer Typinformation</td>
<td>140</td>
</tr>
<tr>
<td>3.3.2 Typumwandlungen und Generizität</td>
<td>145</td>
</tr>
<tr>
<td>3.3.3 Kovariante Probleme</td>
<td>152</td>
</tr>
<tr>
<td>3.4 Überladen versus Multimethoden</td>
<td>156</td>
</tr>
<tr>
<td>3.4.1 Deklarierte versus dynamische Argumenttypen</td>
<td>157</td>
</tr>
<tr>
<td>3.4.2 Simulation von Multimethoden</td>
<td>160</td>
</tr>
<tr>
<td>3.5 Wiederholungsfragen</td>
<td>163</td>
</tr>
<tr>
<td>4 Kreuz und quer</td>
<td>165</td>
</tr>
<tr>
<td>4.1 Ausnahmebehandlung</td>
<td>165</td>
</tr>
<tr>
<td>4.1.1 Ausnahmebehandlung in Java</td>
<td>166</td>
</tr>
<tr>
<td>4.1.2 Einsatz von Ausnahmebehandlungen</td>
<td>169</td>
</tr>
<tr>
<td>4.2 Nebenläufige Programmierung</td>
<td>174</td>
</tr>
<tr>
<td>4.2.1 Thread-Erzeugung und Synchronisation in Java</td>
<td>174</td>
</tr>
<tr>
<td>4.2.2 Nebenläufigkeit in der Praxis</td>
<td>178</td>
</tr>
<tr>
<td>4.2.3 Synchronisation und Objektorientiertheit</td>
<td>182</td>
</tr>
<tr>
<td>4.3 Annotationen und Reflexion</td>
<td>185</td>
</tr>
<tr>
<td>4.3.1 Annotationen und Reflexion in Java</td>
<td>185</td>
</tr>
<tr>
<td>4.3.2 Anwendungen von Annotationen und Reflexion</td>
<td>190</td>
</tr>
<tr>
<td>4.4 Aspektorientierte Programmierung</td>
<td>193</td>
</tr>
<tr>
<td>4.5 Wiederholungsfragen</td>
<td>199</td>
</tr>
</tbody>
</table>
5 Software-Entwurfs muster ........................................ 201
  5.1 Grundsätzliches ........................................ 201
    5.1.1 Bestandteile von Entwurfsmustern ............... 202
    5.1.2 Iterator als Beispiel .............................. 205
  5.2 Erzeugende Entwurfs muster .............................. 209
    5.2.1 Factory Method .................................... 209
    5.2.2 Prototype .......................................... 213
    5.2.3 Singleton .......................................... 216
  5.3 Entwurfs muster für Struktur und Verhalten ............ 219
    5.3.1 Decorator .......................................... 219
    5.3.2 Proxy .............................................. 223
    5.3.3 Template-Method .................................... 226
  5.4 Wiederholungsfragen .................................... 228
Inhaltsverzeichnis
**Vorwort**


Von Teilnehmern wird erwartet, dass Sie schon in Java programmieren können, die wichtigsten Algorithmen und Datenstrukturen kennen und Wissen über die objektorientierte Modellierung mitbringen. Auch Sprachkonzepte wie Interfaces, Subtyping, Klassen und Vererbung sollten in Ihren Grundzügen schon bekannt sein. Darauf aufbauend werden unter anderem folgende Themen behandelt:

- Übersicht über Programmierparadigmen
- Ziele und Eigenschaften des objektorientierten Paradigmas
- Gestaltung von Objektschnittstellen und Untertypbeziehungen
- Umgang mit statischer und dynamischer Typinformation
- Einsatz einiger gängiger Software-Entwurfsmuster

Viel Erfolg bei der Teilnahme an der Lehrveranstaltung!

Franz Puntigam
Andreas Krall

www.complang.tuwien.ac.at/franz/objektorientiert.html
VORWORT
1 Paradigmen der Programmierung


Häufig unterscheidet man das auf Maschinenbefehlen aufbauende *imperative* vom auf formalen Modellen beruhenden *deklarativen* Paradigma. Das imperative Paradigma unterteilt man nach der vorherrschenden Programmablaufstruktur in das *prozedurale* und *objektorientierte* Paradigma, das deklarative nach dem formalen Modell in das *funktionale* und *logikorientierte* Paradigma. Allerdings ist diese einfache Einteilung kaum strukturiert und lässt weniger etablierte Paradigmen unbeachtet.

1 Paradigmen der Programmierung

1.1 Berechnungsmodell und Programmstruktur

Hinter jedem Programmierparadigma steckt ein Berechnungsmodell. Berechnungsmodelle haben immer einen formalen Hintergrund. Sie müssen in sich konsistent und in der Regel Turing-vollständig sein, also alles ausdrücken können, was als berechenbar gilt. Ein Formalismus eignet sich aber nur dann als Grundlage eines Paradigmas, wenn die praktische Umsetzung ohne übermäßig großen Aufwand zu Programmiersprachen, Entwicklungsmethoden und Werkzeugen hinreichender Qualität führt. Es braucht Experimente und Erfahrungen um die praktische Eignung festzustellen. Mühsam gewonnene Erfahrungen sind es auch, die zur langsamen Weiterentwicklung von Paradigmen führen. So kommen bestimmte Strukturen in die Programme, die im ursprünglichen Formalismus nicht vorhanden waren. Häufig sind gerade diese Strukturen von entscheidender Bedeutung, nicht so sehr die ursprünglichen Eigenschaften der Formalismen.

1.1.1 Formaler Hintergrund

Es entstehen ständig neue Theorien, Formalismen, etc., die in einem Zweig der Informatik von Bedeutung sind. Wir stellen eine Auswahl davon ganz kurz vor und betrachten ihren Einfluss auf Programmierparadigmen. Details der Formalismen sind Gegenstand anderer Lehrveranstaltungen.

Funktionen: In fast jedem Programmierparadigma spielen Funktionen (oder ähnliche Konzepte) eine zentrale Rolle. Es gibt unterschiedliche Formalismen zur Beschreibung von Funktionen. Am einfachsten zu verstehen sind wohl primitiv-rekursive Funktionen, die von einer vorgegebenen Menge einfacher Funktionen ausgehen und daraus durch Komposition und Rekursion neue Funktionen bilden, so wie es Programmierer(inn)en geläufig ist. Primitiv-rekursive Funktionen können vieles berechnen, aber nicht alles, was berechenbar ist. Turing-Vollständigkeit erreicht man durch µ-rekursive Funktionen, wo der µ-Operator angewandt auf partielle Funktionen das kleinste aller möglichen Ergebnisse liefert. Der ebenfalls Turing-vollständige λ-Kalkül (von dem es mehrere Varianten gibt) wurde unabhängig von primitiv- und µ-rekursiven Funktionen entwickelt und beschreibt Funktionen ohne Notwendigkeit für Rekursion.

Historisch gesehen ist nicht ganz klar, wie groß der Einfluss dieser Formalismen, vor allem des λ-Kalküls, auf die Entwicklung der
1.1 Berechnungsmodell und Programmstruktur


Constraint-Programmierung: Ein Zweig der Programmiersprachen der 5. Generation verwendete Einschränkungen auf Variablen wie etwa „$x < 5$“ zusätzlich zu solchen wie „$A$ oder $B$ ist wahr“. Fortschrittliche Beweistechniken können solche Constraints vergleichsweise ef-


**Freie Algebren:** Algebra ist ein sehr altes und etabliertes Teilgebiet der Mathematik, das sich mit Eigenschaften von Rechenoperationen besetzt. Gleichzeitig ist eine (universelle) *Algebra* auch ein mathematisches Objekt, etwa eine Gruppe, ein Ring, ein Körper, etc. Von besonderer Bedeutung für die Informatik sind sogenannte freie Algebren. Diese universellen Algebren sind, stark vereinfacht formuliert, die allgemeinsten Algebren innerhalb von Familien von Algebren mit gemeinsamen Eigenschaften. Freie Algebren erlauben uns die Spezifikation beinahe beliebiger Strukturen (beispielsweise von Datenstrukturen) über einfache Axiome. Auch wenn freie Algebren im Zusammenhang mit Programmierparadigmen nicht so dominant sind wie z.B. Funktionen, so spielen Sie doch in einigen Bereichen eine wichtige Rolle, etwa im Zusammenhang mit Modulen und Typen. Auf freien Algebren basieren aber auch viele Spezifikationssprachen.

**Prozesskalküle:** Eine Familie speziell dafür entwickelter Algebren eignet sich gut zur Modellierung von **Prozessen** (vergleichbar mit Threads) in nebenläufigen Systemen. Die bekanntesten Prozesskalküle sind
1.1 Berechnungsmodell und Programmstruktur


WHILE, GOTO und Co: So manches typische Sprachkonstrukt imperativer Sprachen hatten zum Zeitpunkt seiner Entstehung keinerlei formalen Hintergrund. Entsprechende Formalismen mussten erst geschaffen und analysiert werden. Beispiele dafür sind die WHILE- und die GOTO-Sprache, ganz einfache formale Sprachen, in denen es außer Zuweisungen, ganz primitiven arithmetischen Operationen und bedingten Anweisungen nur entweder eine While-Schleife oder eine Goto-Anweisung (Sprung an eine beliebige andere Programmposition) gibt. Diese beiden Sprachen sind Turing-vollständig. Im Gegensatz dazu ist die LOOP-Sprache, in der es statt While bzw. Goto eine Schleife mit einer vorgegebenen Anzahl an Iterationen gibt, nur äqui-
valent zu primitiv-rekursiven Funktionen. *PRAM*-Sprachen (Parallel Random Access Memory) ändern obige Sprachen dahingehend ab, dass die dahinter stehenden Maschinenmodelle mehrere Operationen gleichzeitig auf unterschiedlichen Speicherzellen durchführen können. Es ist klar, dass derartige formale Sprachen eine starke Verbindung zu imperativen Programmierparadigmen haben.

Diese Aufzählung ist ganz und gar nicht vollständig. Es gibt viele weitere, oft eher exotische Modelle und Formalismen, die in dem einen oder anderen Paradigma eine größere Bedeutung erlangt haben.

1.1.2 Praktische Realisierung

Beim Programmieren denkt man kaum bewusst an Berechnungsmodelle, sondern eher an Programmierwerkzeuge und diverse Details der Syntax und Semantik einer Sprache. Es geht um die Lösung praktischer Aufgaben. Im Idealfall unterstützen uns die Werkzeuge und Sprachen bei der Lösung der Aufgaben in einer zum Berechnungsmodell passenden Weise. Trotz der Unterschiedlichkeit der Aufgaben, Sprachen, Modelle und Werkzeuge bestimmen immer wieder dieselben Eigenschaften den Erfolg oder Misserfolg eines Programmierparadigmas zu einem großen Teil mit:


**Konsistenz:** Programmiersprachen und -paradigmen müssen zahlreiche Erwartungen hinsichtlich verschiedenster Aspekte erfüllen. Ein ein-
1.1 Berechnungsmodell und Programmstruktur


Systemnähe: Programme müssen effizient auf realer Hardware ausführbar sein. Effizienz lässt sich scheinbar am leichtesten erreichen, wenn das Paradigma wesentliche Teile der Hardware und des Betriebssystems direkt widerspiegelt. Viele Programmierer(innen) bevorzugen systemnahe Paradigmen aus diesem Grund und weil sie ihr eigenes System gut kennen und nutzen wollen. Unverzichtbar ist Systemnähe dann, wenn Hardwarekomponenten direkt angesprochen werden müssen. Gelegentlich wird aus wirtschaftlichen Gründen eine starke


Wer danach sucht, wird viele weitere praktische Kriterien für den Erfolg von Paradigmen finden, vor allem solche, die für bestimmte Paradigmen bedeutend sind. Man kann die zukünftige Bedeutung vieler Kriterien nur schwer abschätzen. So manche Vermutung aus der Vergangenheit hat sich
1.1 Berechnungsmodell und Programmstruktur

nicht bewahrheitet. Die oben genannten Kriterien waren jedoch bis jetzt stets von Bedeutung, sodass man mit hoher Wahrscheinlichkeit davon ausgehen kann, dass Sie es auch in absehbarer Zukunft noch sein werden.


1.1.3 Evolution und Struktur im Kleinen

Widersprüchliche Ziele. Paradigmen entwickeln sich auch ohne Paradigmenwechsel langsam, aber stetig weiter. Die Entwicklung verläuft meist wellenförmig, selten geradlinig. Eine Triebfeder dafür sind immer wieder neue praktische Erfahrungen im Spannungsfeld zwischen zum Teil widersprüchlichen Forderungen. So ist es etwa unmöglich, folgende Forderungen gleichzeitig und in vollem Umfang zu erfüllen:

• Flexibilität und Ausdruckskraft sollen in kurzen Texten die Darstellung aller vorstellbaren Programmabläufe ermöglichen.
• Lesbarkeit und Sicherheit sollen Absichten hinter Programmteilen sowie mögliche Inkonsistenzen leicht erkennen lassen.
• Die Konzepte müssen verständlich bleiben und es muss klar sein, was einfach machbar ist und was nicht.

Erfolg verschwindet diese Zielsetzung aber rasch aus dem Blickfeld. Man kann auch erkennen, dass statische und dynamische Programmierung zwar praktisch schon seit Beginn der Informatik zusammen existieren, aber gelegentlich Pendelbewegungen einmal hin zu eher statischer und dann wieder hin zu eher dynamischer Programmierung stattfinden.

**Strukturierte Programmierung.** Selten kann sich in diesem Spannungsfeld eine Idee so eindeutig durchsetzen, dass nach einer gewissen Zeit alle Pendelbewegungen aufhören. Das Paradebeispiel dafür ist die *strukturierte Programmierung*, die mehr Struktur in die prozedurale Programmierung bringt. Jedes Programm bzw. jeder Rumpf einer Prozedur ist nur aus drei einfachen Kontrollstrukturen aufgebaut:

- Sequenz (ein Schritt nach dem anderen)
- Auswahl (Verzweigung im Programm, z.B. *if* und *switch*)
- Wiederholung (Schleife, Rekursion oder Ähnliches)

Heute erscheinen diese Kontrollstrukturen nicht nur in der prozeduralen Programmierung als so selbstverständlich, dass man sich kaum mehr etwas anderes vorstellen kann. Nur durch die genaue Ausformung dieser Kontrollstrukturen ergeben sich noch Unterschiede. Das mächtige und früher allgegenwärtige Goto, das einen Gegenpol zur strukturierten Programmierung bildet, wird heute fast gar nicht mehr verwendet.

1.1 Berechnungsmodell und Programmstruktur


Seiteneffekte und Querverbindungen. Ein Problem der imperativen Programmierung wird durch strukturierte Programmierung nicht gelöst: Programmfortschritte erzielt man über Seiteneffekte, vor allem durch Zuweisung neuer Werte an Variablen (sowie Ein- und Ausgaben, die letztlich wieder über Zuweisungen realisiert sind). Das wäre kein Problem, könnte man beim Programmieren alle Veränderungen der Programmzustände durch Zuweisungen überblicken. Leider ist so mancher Seiteneffekt gut versteckt. Nehmen wir als Beispiel einen Prozedurauftruf \( f(x) \), der die Zahlen im Array \( x \) sortiert. Vielleicht hat \( f \) weitere uns unbekannte Seiteneffekte, sammelt etwa statistische Daten über die sortierten Zahlen. Während Aufrufe wie \( f(x); f(y) \) mit verschiedenen Arrays so wie gedacht funktionieren, würde eine Aufrufsequenz \( f(x); f(x); f(y) \) durch das semantisch vermeintlich irrelevante doppelte Sortieren von \( x \) die statistischen Daten verfälschen. Diese Daten sind unerwartet mit \( x \) und \( y \) verbunden. Auswirkungen versteckter Seiteneffekte und entsprechender Querverbindungen sind in der Praxis oft gravierend.


Die deklarative Programmierung strebt referentielle Transparenz als Eigenschaft aller Ausdrücke im Programm an. Ein Ausdruck ist referentiell transparent, wenn er durch seinen Wert ersetzt werden kann, ohne die Semantik des Programms dadurch zu ändern. Man kann den Ausdruck auch durch einen anderen, ebenfalls referentiell transparenten Ausdruck desselben Werts ersetzen. Beispielsweise lässt sich \( 3 + 4 \) durch 7 oder \( 14/2 \) ersetzen, ohne die Semantik zu ändern. Auch \( f(x) + f(x) \) lässt sich durch \( 2 \times f(x) \) ersetzen, obwohl die Bedeutung der Funktion \( f \) unbekannt ist. Jedoch darf \( f \) keine Seiteneffekte haben und nicht von Variablen abhän-
1 Paradigmen der Programmierung

gen, deren Werte sich im Laufe der Zeit ändern könnten. Variablenwerte sind nur mittels Seiteneffekten änderbar. Daher erreicht man referentielle Transparenz durch gänzlichen Verzicht auf Seiteneffekte. Seiteneffektfreiheit ist Mittel zum Zweck, nicht das Ziel. Referentielle Transparenz ist das Ziel. Man kann Ausdrücke vereinfachen, ohne Angst vor versteckten Querverbindungen und dadurch verursachten Fehlern haben zu müssen. Ohne referentielle Transparenz dürfte man \( f(x) + f(x) \) nicht durch \( 2 \times f(x) \) ersetzen, weder gedanklich noch tatsächlich.


Auch imperative Paradigmen unterstützen Funktionen und erlauben bis zu einem gewissen Grad einen funktionalen Programmierstil. Wo es einfach geht setzt man überall, auch in der imperativen Programmierung auf referentielle Transparenz. So sind primitive Operationen wie \(+\), \(-\), \(*\) und \(/\) fast durchwegs referentiell transparent, aber nicht nur diese.


**First-Class-Entities.** Funktionen und ähnliche Konzepte gibt es in fast allen Paradigmen. Dennoch spielen Funktionen in funktionalen Sprachen eine wichtigere Rolle, die unter anderem dadurch sichtbar wird, dass Funktionen *First-Class-Entities* sind. Das bedeutet, dass Funktionen wie nor-
male Daten verwendet werden. Man kann Funktionen zur Laufzeit erzeu-
gen (z.B. λ-Ausdrücke), in Variablen ablegen, als Argumente an andere
Funktionen übergeben und als Ergebnisse von Funktionen zurückbekom-
men. Methoden in Java sind dagegen keine First-Class-Entities. Man kann
sie zwar aufrufen, aber nicht an Variablen zuweisen oder als Argumente
übergeben. Objekt-ähnliche Konzepte gibt es ebenso in vielen Paradigmen,
z.B. in Form von Modulen (siehe Abschnitt 1.2), aber im Wesentlichen nur
in der objektorientierten Programmierung sind Objekte als First-Class-
Entities wie alle anderen Daten erzeug- und verwendbar. Offensichtlich
bekommt so manches Konzept erst durch die gleiche Behandlung wie alle
anderen Daten jene überragende Bedeutung, die notwendig ist, um ein
Paradigma darauf aufzubauen.

First-Class-Entities sind häufig viel komplizierter als vergleichbare Kon-
zepte, weil die uneingeschränkte Verwendbarkeit eine große Zahl an zu
berücksichtigenden Sonderfällen nach sich zieht. Diesen Aufwand möchte
man zugunsten von Einfachheit und Effizienz vermeiden. Man akzeptiert
ihn nur, wenn die uneingeschränkte Verwendbarkeit echte Vorteile bringt.

Betrachten wir beispielsweise Funktionen. Solange wir nur rekursive
Aufrufe (und vordefinierte Schleifen) für Wiederholungen verwenden, bie-
ten Funktionen als First-Class-Entities keine Vorteile. Die Verwendung
entspricht der von primitiv-rekursiven Funktionen. Aber Funktionen hö-
gerer Ordnung (also Funktionen mit Funktionen als Parametern) bringen
eine neue Dimension in die Programmierung. Sie sind wie Kontrollstruktu-
ren verwendbar, etwa zur Realisierung eigener Arten von Schleifen. So er-
hält auch der λ-Kalkül seine Ausdrucksstärke. Ein häufiger Gebrauch von
Funktionen höherer Ordnung führt zu einem eigenen Paradigma, der appli-
kativen Programmierung, bei der man quasi Schablonen von Program-
teilen schreibt, die dann durch Übergabe von Funktionen (zum Füllen der
Löcher in den Schablonen) ausführbar werden. In diesem Stil lassen sich
sehr kompakte und trotzdem verständliche Programme schreiben. Funk-
tionen erster (= nicht-höherer) Ordnung eignen sich dafür nicht.

Erst die First-Class-Eigenschaft macht Objekte vielseitig verwendbar. Spezialisierte Objekte (z.B. Iteratoren) kann man wie Kontrollstruktu-
ren verwenden und dadurch etwas mit Funktionen höherer Ordnung Ver-
gleichbares erreichen. Im Detail ergeben sich aber große Unterschiede; ein
Konzept kann das andere nicht vollständig ersetzen. Daher gibt es Bemü-
hungen, Funktionen bzw. Methoden höherer Ordnung auch in der objekt-
orientierten Programmierung besser zu unterstützen. Das bedeutet jedoch
nicht, dass diese Methoden auch referentiell transparent sein müssen.
1 Paradigmen der Programmierung

1.2 Programmorganisation


1.2.1 Modularisierung

Durch Modularisierung bringen wir größere Strukturen in Programme. Wir zerlegen das Programm in einzelne Modularisierungseinheiten, die nur lose voneinander abhängen und daher relativ leicht gegeneinander austauschbar sind. Wenn man etwas genauer hinsieht, muss man verschiedene Formen von Modularisierungseinheiten unterscheiden.


Beim Programmieren zeigt sich ein weiterer Vorteil: Module lassen sich relativ unabhängig voneinander entwickeln, sodass mehrere Leute oder mehrere Teams gleichzeitig an unterschiedlichen Stellen eines Programms arbeiten können, ohne sich gegenseitig zu stark zu behindern.


Paradigmen der Programmierung


1.2 Programmorganisation

Es gibt eine breite Palette an Möglichkeiten zur Festlegung der Details. So ist Data-Hiding praktisch in jeder objektorientierten Sprache etwas anders realisiert, und neue Objekte werden auf ganz unterschiedliche Weise erzeugt und initialisiert. Beispielsweise entstehen neue Objekte in der Programmiersprache Self \[33\] nur durch Kopieren bereits bestehender Objekte, und man kommt ganz ohne Klassen und ähnliche Konzepte aus. Meist verwendet man aber Klassen und Konstruktoren für die Initialisierung.


Meist ist es möglich, Klassen von anderen Klassen abzuleiten und dabei Methoden zu erben. Auf eine bestimmte Art angewandt (siehe Abschnitte \[1.2.3\] und \[1.3\]) ergeben sich durch Klassenableitungen auf vielfältige Weise strukturierbare Klassifizierungen von Objekten, in denen ein einzelnes Objekt gleichzeitig mehrere Typen haben kann. Zusammen mit abgeleiteten Klassen sind auch *abstrakte Klassen* sinnvoll, von denen zwar andere Klassen ableitbar aber keine Objekte erzeugbar sind. Interfaces wie in Java sind einfach nur eine Spezialform von abstrakten Klassen.


Der klassische Lösungsansatz besteht darin, die globale Namensverwaltung ganz den Softwareentwicklern zu überlassen. Beim Anwenden von Werkzeugen (z.B. Compilern) muss man alle Dateien anführen, welche die zu bearbeitenden Modularisierungseinheiten enthalten. Das einzige Hilfsmittel sind oft standardmäßig vorgegebene Verzeichnisse, in denen automatisch nach verwendeten Modul- oder Klassennamen gesucht wird.


### 1.2.2 Parametrisierung


**Befüllen zur Laufzeit.** Am einfachsten ist es, wenn das Befüllen der Löcher erst zur Laufzeit erfolgt und die dazu verwendeten Daten First-Class-Entities sind. In diesem Fall kann man die Löcher durch einfache Variablen darstellen. Beim Befüllen werden ihnen Werte zugewiesen. Allerdings befinden sich die Variablen üblicherweise nicht an der Stelle im Programm, an der die zuzuweisenden Werte bekannt sind. Man kann die Werte auf unterschiedliche Weise zu den Variablen bringen, unter anderem so:

**Konstruktor:** Beim Erzeugen eines Objekts wird ein Konstruktor ausgeführt, der die Objektvariablen initialisiert. Diesen Konstruktor sieht man mit formalen Parametern. An der Stelle, an der die Ob-
jekterzeugung veranlasst wird, übergibt man Werte als aktuelle Parameter an den Konstruktor, die dann zur Initialisierung verwendet werden. Das ist die häufigste und einfachste Form der Parametrisierung, nicht nur in objektorientierten Sprachen.

**Initialisierungsmethode:** In einigen Fällen sind Konstruktoren nicht verwendbar, beispielsweise wenn Objekte durch Kopieren erzeugt werden oder zwei zu erzeugende Objekte voneinander abhängen; man kann ja das später erzeugte Objekt nicht an den Konstruktor des zuerst erzeugten Objekts übergeben. Solche Probleme kann man durch Methoden lösen, die unabhängig von der Objekterzeugung zur Initialisierung eines bereits bestehenden Objekts aufgerufen werden. Objekte werden also in einem ersten Schritt erzeugt und in einem zweiten initialisiert bevor sie verwendbar sind. Diese Technik funktioniert nur in imperativen Paradigmen.

**Zentrale Ablage:** Man kann Werte an zentralen Stellen (etwa in globalen Variablen oder als Konstanten) ablegen, von wo sie bei der Objekterzeugung oder erst bei der Verwendung abgeholt werden. In letzterer Variante ist diese Technik auch für statische Modularisierungseinheiten verwendbar, die bereits zur Übersetzungszeit feststehen. Zum Abholen der Werte kann man direkt auf die Variablen oder Konstanten zugreifen oder Methoden verwenden. Bei Klassen kann man Konstanten z.B. von einem Interface erben um Werte „abzuholen“.

Von diesen Techniken sind unzählige Verfeinerungen vorstellbar, die gerade in der objektorientierten Programmierung häufig ausgereizt werden.

Diese Techniken eignen sich auch zur *Dependency-Injection*. Dabei überträgt man die Verantwortung für das Erzeugen und Initialisieren von Objekten an eine zentrale Stelle (z.B. eine Klasse), von der aus man die Abhängigkeiten zwischen den Objekten überblicken und steuern kann.

**Generizität.** Unter Generizität versteht man eine Form der Parametrisierung, bei der Löcher (zumindest konzeptuell) bereits zur Übersetzungszeit befüllt werden. Daher können alle Arten von Modularisierungseinheiten generisch sein (also Löcher enthalten, die mittels Generizität befüllt werden), aber beispielsweise auch Funktionen und Ähnliches. Das, womit die Löcher gefüllt werden sollen, wird zunächst durch generische Parameter bezeichnet. In allen Löchern, die mit Denselben befüllt werden sollen, stehen auch dieselben generischen Parameter. Später, aber noch vor der
1.2 Programmorganisation

Programmausführung werden die Parameter durch das Einzufüllende ersetzt. Bevorzugt stehen generische Parameter nicht für gewöhnliche Werte, sondern für Konzepte, die keine First-Class-Entities sind. Häufig sind das Typen. In diesem Fall nennt man generische Parameter auch \textit{Typparameter}. Generizität ist also vorwiegend für solche Fälle gedacht, wo das Befüllen der Löcher zur Laufzeit nicht funktioniert.


\textbf{Annotationen.} Annotationen sind optionale Parameter, die man zu unterschiedlichsten Sprachkonstrukten hinzufügen kann. Ein Beispiel dafür ist die Annotation überschriebener Methoden in Java mittels \texttt{@Override}. Annotationen werden von Werkzeugen verwendet oder aber auch einfach ignoriert. So gibt ein Compiler, der \texttt{@Override} versteht, in manchen Situationen Warnungen aus, während andere Compiler und Werkzeuge, die die Annotation nicht kennen, diese einfach ignorieren. Diese Form der Annotationen wirkt sich statisch, also zur Übersetzungszeit aus. Man kann Annotationen meist auch dynamisch, also zur Laufzeit abfragen. Über spezielle Funktionen (oder Ähnliches) lässt sich erfragen, mit welchen Annotationen ein Sprachkonstrukt versehen ist. Alles funktioniert so, als ob keine Annotationen vorhanden wären, solange die Annotationen nicht explizit abgefragt und entsprechende Aktionen gesetzt werden. Sie ähneln also Kommentaren, die aber bei Bedarf auch zur Steuerung des Programmablaufs herangezogen werden können.

Wie Generizität eignen sich Annotationen nur für statisch bekannte Informationen. Die Löcher, die durch Annotationen befüllt werden, sind im Gegensatz zur Generizität nirgends im Programm festgelegt. Daher ist die Art und Weise, wie die mitgegebenen Informationen zu verwenden sind, ebenso unterschiedlich wie die Anwendungsgebiete. In der Praxis
setzt man Annotationen oft in Situationen ein, wo Informationen nicht nur von lokaler Bedeutung sind, sondern auch System-Werkzeuge (wie einen Compiler) oder das Betriebssystem steuern oder zumindest beeinflussen.


Bestimmte Aufgaben lassen sich durch Aspekte überzeugend rasch und einfach lösen. Für andere Aufgaben sind Aspekte dagegen kaum geeignet. Ein Problem besteht darin, dass die Bestimmung der betroffenen Punkte im Programm gelegentlich Wissen über so manches Implementierungsdetail voraussetzt. Wenn sich solche Details ändern, müssen auch die Aspekte entsprechend geändert werden.

Parametrisierung steigert zwar die Flexibilität von Modularisierungseinheiten, aber ein Problem bleibt bei allen Formen der Parametrisierung bestehen: Die Änderung einer Modularisierungseinheit macht mit hoher Wahrscheinlichkeit auch Änderungen an allen Stellen nötig, an denen diese Modularisierungseinheit verwendet wird. Konkret: Wenn die Löcher sich ändern, dann muss sich auch das ändern, was zum Füllen der Löcher verwendet wird. Solche notwendigen Änderungen behindern die Wartung gewaltig. Vor allem muss man für die Änderungen alle Stellen kennen,
1.2 Programmorganisation

an denen eine Modularisierungseinheit verwendet wird. Bei Modularisierungs einheiten, die in vielen unterschiedlichen Programmen über die ganze Welt verteilt zum Einsatz kommen, ist das so gut wie unmöglich. Nachträgliche Änderungen der Löcher in solchen Modularisierungseinheiten sind dadurch praktisch kaum durchführbar.

1.2.3 Ersetzbarkeit


Leider ist es recht kompliziert im Detail festzustellen, unter welchen Bedingungen $A$ durch $B$ ersetzbar ist. Diese Bedingungen hängen nicht nur von $A$ und $B$ selbst ab, sondern auch davon, was man sich, von außen betrachtet, von $A$ und $B$ erwartet. Daher ist Ersetzbarkeit nur für Modularisierungseinheiten anwendbar, die alle erlaubten Betrachtungsweisen von außen klar festlegen. Das geht Hand in Hand mit klar definierten Schnittstellen. Ersetzbarkeit zwischen $A$ und $B$ ist dann gegeben, wenn die Schnittstelle von $B$ dasselbe beschreibt wie die von $A$. Jedoch kann die Schnittstelle von $B$ mehr Details festlegen als die von $A$, also etwas festlegen, was in $A$ noch offen ist. Man kann entsprechende Schnittstellen auf verschiedene Weise spezifizieren:

**Signatur:** In der einfachsten Form spezifiziert eine Schnittstelle nur, welche Inhalte der Modularisierungseinheit von außen zugreifbar sind. Diese Inhalte werden über ihre Namen und gegebenenfalls die Typen von Parametern und Ergebnissen beschrieben. Die Bedeutung der Inhalte bleibt offen. Man nennt eine solche Schnittstelle Signatur der Modularisierungseinheit. In Kapitel 2 werden wir sehen, dass Ersetzbarkeit für Signaturen einfach und klar definiert ist und auch von einem Compiler überprüft werden kann. Im Wesentlichen muss $B$ alles enthalten und von außen zugreifbar machen, was auch in $A$ von außen zugreifbar ist, kann aber mehr enthalten als $A$. Wenn man sich hinsichtlich der Ersetzbarkeit jedoch nur auf Signaturen verlässt, kommt es leicht zu Irrtümern. Ein Inhalt von $B$ könnte eine ganz andere Bedeutung haben als der gleichnamige Inhalt von $A$. Es
passiert etwas Unerwartetes, wenn man statt des Inhalts von $A$ den entsprechenden Inhalt von $B$ verwendet. Dennoch verlässt man sich eher auf Signaturen als ganz auf Ersetzbarkeit zu verzichten.


**Zusicherungen:** Um Fehler auszuschließen ist eine genaue Beschreibung der erlaubten Erwartungen an eine Modularisierungseinheit nötig. Diese Beschreibung bezieht sich auf die Verwendungsmöglichkeiten aller nach außen sichtbaren Inhalte. In der objektorientierten Programmierung hat sich für solche Beschreibungen der Begriff *Design-by-Contract* etabliert. Dabei entspricht die Schnittstelle einem Vertrag zwischen einer Modularisierungseinheit (als Server) und ihren Verwendern (Clients). Der Vertrag legt in *Zusicherungen* fest, was sich der Server von den Clients erwarten kann (das sind *Vorbedingungen*), was sich die Clients vom Server erwarten können (*Nachbedingungen*), welche Eigenschaften in konsistenten Programmzuständen immer erfüllt sind (*Invarianten*) und wie – vor allem in welchen Aufruf-Reihenfolgen – Clients mit dem Server interagieren können (*History-Constraints*). Theoretisch sind über diese Arten von Zusicherungen die erlaubten Erwartungen beliebig genau beschreibbar. Es ist auch klar geregelt, wie sich Zusicherungen aus unterschiedlichen Schnittstellen zueinander verhalten müssen, damit Ersetzbarkeit gegeben ist. In der Praxis ergeben sich jedoch Probleme. Häufig sind Zusicherungen nur informal und nicht präzise. Vor allem komplexere Vorbedingungen stehen nicht selten in Konflikt zu Data-Hiding, weil sie von Programmzuständen abhängen, die eigentlich nach au-
1.2 Programmorganisation

ßen nicht sichtbar werden sollten. Meist wird die Einhaltung von Zusicherungen, wenn überhaupt, erst zur Laufzeit überprüft; dann ist es dafür eigentlich schon zu spät.


In der objektorientierten Programmierung steht die Ersetzbarkeit ganz zentral im Mittelpunkt. Man versucht stets, Programme so zu gestalten, dass jeder Programmteil möglichst problemlos durch einen anderen Programmteil ersetzbar ist. Einerseits achtet man auf Ersetzbarkeit von Objekten innerhalb eines Programms. Programmteile sind vielfältig einsetzbar, wenn sie zwar Objekte einer bestimmten Art erwarten, aber trotzdem auf allen Objekten operieren können, durch die die erwarteten Objekte ersetzbar sind. Dies ermöglicht einfache Erweiterungen des Programms, ohne dabei schon existierenden Code ständig anpassen zu müssen. Andererseits bietet die Ersetzbarkeit (nicht nur in der objektorientierten Programmierung) eine Grundlage für die Erzeugung neuer Programmversionen, die mit Ihrer Umgebung trotz Erweiterungen kompatibel bleiben.

In anderen Programmierparadigmen ist Ersetzbarkeit zur Erzeugung neuer Programmversionen genauso wichtig, aber meist fehlen Sprachmechanismen um Ersetzbarkeit sicherzustellen. Im besten Fall gibt es noch eine Unterstützung zur Überprüfung von Signaturen.

1.3 Typisierung


1.3.1 Typkonsistenz, Verständlichkeit und Planbarkeit


Sprachen unterscheiden sich erheblich hinsichtlich des Zeitpunkts, an dem Typen bekannt sind und überprüft werden. In dynamischen Spra-
1.3 Typisierung


Paradigmen der Programmierung

**Typen und Entscheidungsprozesse.** Berechnungsmodelle beschreiben ausschließlich die Programmausführung. Alle bedeutenden Entscheidungen hinsichtlich der Programmstruktur und des Programmablaufes werden jedoch schon vorher, bei der Programmerstellung getroffen. Hier ist eine Grobeinteilung von Entscheidungszeitpunkten:


- Zum Zeitpunkt der *Erstellung von Übersetzungseinheiten* werden die meisten wichtigen Entscheidungen getroffen, auf die man beim Programmieren Einfluss hat. Hierzu braucht man viel Flexibilität.

- Manche wichtigen Entscheidungen werden durch Parametrisierung erst bei der *Einbindung* vorhandener Module, Klassen oder Komponenten getroffen. Dies geht jedoch nur, wenn die eingebundenen Modularisierungseinheiten dafür vorgesehen sind.

- Vom *Compiler* getroffene Entscheidungen sind eher von untergeordneter Bedeutung und betreffen meist nur Optimierungen. Alles Wichtige ist bereits im Programmcode festgelegt oder liegt erst zur Laufzeit vor.


Typen verknüpfen die zu unterschiedlichen Zeitpunkten vorliegenden Informationen miteinander. Vor allem statisch geprüfte Typen helfen dabei, einmal getroffene Entscheidungen über den gesamten folgenden Zeitraum konsistent zu halten. Einige Beispiele in Java sollen dies verdeutlichen:

1.3 Typisierung


Diese Beispiele zeigen Folgendes: Je früher Entscheidungen getroffen werden, desto weniger ist zur Laufzeit zu tun und desto weniger Programmcode (Fallunterscheidungen) braucht man. Ohne statisch geprüfte Typen wäre mehrfacher Aufwand nötig: Sogar wenn man weiß, dass die Variable eine ganze Zahl enthält, muss der Compiler eine beliebige Referenz annehmen und zur Laufzeit eine dynamische Typprüfung durchführen.


**Planbarkeit.** Letztendlich erleichtern frühe Entscheidungen die Planbarkeit weiterer Schritte. Wenn man weiß, dass eine Variable vom Typ `int` ist, braucht man kaum mehr Überlegungen darüber anzustellen, welche Werte in der Variablen enthalten sein könnten. Statt auf Spekulationen baut man auf Wissen auf. Um sich auf einen Typ festzulegen muss man voraussehen (also planen), wie bestimmte Programmteile im fertigen Pro-


1.3.2 Abstraktion

Wie wir in Abschnitt 1.2.3 gesehen haben, stellen Abstraktionen der realen Welt eine in der Praxis wichtige Grundlage für die Ersetzbarkeit und damit für die Wartbarkeit von Software dar. Typen spielen eine entscheidende Rolle im Umgang mit Abstraktionen.

Abstrakte Datentypen. Der Begriff abstrakter Datentyp bezieht sich in erster Linie auf die Trennung der Innenansicht von der Außenansicht einer Modularisierungseinheit (Data-Hiding) zusammen mit Kapselung. Die
1.3 Typisierung


scheidlichen nominalen Typen. Die Konzepte dahinter können sich unter-

scheiden. Zwei nominale Typen sind äquivalent wenn sie denselben Namen

haben, zwei strukturelle Typen wenn sie dieselbe Struktur haben.

Obwohl hinter abstrakten Datentypen oft komplexe Konzepte stecken,

können auch elementare Typen wie \texttt{int} als abstrakte Datentypen be-

trachtet werden. Diese Typen abstrahieren über Implementierungsdetails,

etwa die genaue Repräsentation in der Maschine.

Untertypen. Untertypbeziehungen werden durch das Ersetzbarkeitsprin-

zip \cite{8, 25} definiert:

Ein Typ $U$ ist Untertyp eines Typs $T$ wenn jedes Objekt von

$U$ überall verwendbar ist wo ein Objekt von $T$ erwartet wird.

Ohne Ersetzbarkeit (siehe Abschnitt \cite{12, 3}) gibt es also keine Untertypen.

Wie wir in Kapitel 2 sehen werden, sind Untertypbeziehungen durch

das Ersetzbarkeitsprinzip für strukturelle Typen ganz eindeutig definiert.

Die Theorie lässt keinen Spielraum: Sind zwei strukturelle Typen gegeben,

kann man durch Anwendung fixer Regeln automatisch ermitteln, ob ein

Typ Untertyp des anderen ist \cite{2}.

Für nominale Typen reichen einfache Regeln nicht aus. Abstrakte und
daher die Regeln nicht zugängliche Konzepte lassen sich ja nicht auto-
matisch vergleichen. In der Praxis muss man beim Programmieren expli-

zit hinschreiben, welcher Typ Untertyp von welchem anderen ist. Meist

verwendet man dazu abgeleitete Klassen: Wird eine Klasse $U$ von einer

Klasse $T$ abgeleitet, so nimmt man an, dass $U$ Untertyp von $T$ ist. Dazu

müssen auch die entsprechenden strukturellen Typen, also die Signaturen

von $U$ und $T$, in einer durch die Regeln überprüfbaren Untertypbeziehung

stehen. Man soll $U$ nur dann von $T$ ableiten, wenn das Konzept hinter

$T$ durch das Konzept hinter $U$ vollständig ersetzbar ist, sodass statt ei-

dem Objekt von $T$ stets auch ein Objekt von $U$ verwendbar ist. Beim

Programmieren liegt die Einhaltung dieser Bedingung zur Gänze in un-

serer Verantwortung. Kein Compiler oder anderes Werkzeug kann sie uns

abnehmen. Falschlich angenommene Untertypbeziehungen zählen zu den

folgenschwersten Fehlern in der objektorientierten Programmierung.

So wie in Abschnitt \cite{1, 2, 3} beschrieben ist es durchaus möglich, Zusii-

cherungen in die Entscheidung von Untertypbeziehungen einzureihen.

Einige wenige Programmiersprachen wie Eiffel \cite{28} machen das. Allerdings

zeigt die Erfahrung, dass Zusicherungen praktisch nie so präzise und voll-

ständig sind, dass dadurch abstrakte Konzepte bei der Entscheidung von
Untertypbeziehungen außer Acht gelassen werden könnten. Das heißt, die Einbeziehung von Zusicherungen ändert nichts daran, dass man Untertypbeziehungen explizit hinschreiben und damit die Verantwortung für die Kompatibilität der Konzepte übernehmen muss. Daher betrachten die meisten Programmiersprachen Zusicherungen nicht als zu den Typen gehörig. Trotzdem empfiehlt es sich, beim Programmieren die wichtigsten Zusicherungen zumindest in Form von Kommentaren hinzuschreiben und dadurch die größten Fehler bei Untertypbeziehungen zu verhindern.


1 Paradigmen der Programmierung

sind. Dieser Ansatz wird auf unterschiedliche Weise beispielsweise in C++
aber auch in der funktionalen Sprache Haskell verwendet. Beide Ansätze
können gut mit Fällen umgehen, mit denen Untertypen nicht oder nur
schwer zureckkommen. Deswegen unterstützen die meisten statischen ob-
jectorientierten Sprachen Generizität. Jedoch ist Generizität hinischlich
der Wartung prinzipiell kein vollwertiger Ersatz für Ersetzbarkeit.

Wie wir in Kapitel 3 sehen werden gibt es auch mehrere Möglichkeiten,
wie der Compiler generische Programme in nichtgenerische umformt. Es
gibt Unterschiede im Umfang und Spezialisierungsgrad des generierten
Codes, in der im generischen Programm verfügbaren Information und im
Zeitpunkt, zu dem diese Information verwendbar ist.

1.3.3 Gestaltungsspielraum

Zahlreiche Entscheidungen im Entwurf einer Sprache spiegeln sich in den
Typen wider. Der Gestaltungsspielraum scheint endlos. Aber die Theorie
zeigt klare Grenzen auf. Beispielsweise haben wir schon angedeutet, dass es
im Zusammenhang mit Untertypen eine bedeutende Einschränkung gibt;
in den Kapiteln 2 und 3 werden wir uns näher damit beschäftigen. Im
Folgenden betrachten wir kurz einige weitere Bereiche, teilweise mit für
Uneingeweihte überraschenden Möglichkeiten und Grenzen.

Rekursive Datenstrukturen. Viele Datenstrukturen wie Listen und Bäume
sind in ihrer Größe unbeschränkt. Nicht selten spricht man von potentiell
unendlichen Strukturen. Aber tatsächlich unendlich große Strukturen sind
im Computer niemals darstellbar. Wenn man in Typen versucht, rekursive
Datenstrukturen als möglicherweise unendlich zu betrachten, wird man
scheitern, weil Typprüfungen in eine Endlosschleife geraten.

Der einzig sinnvolle Weg zur Beschreibung unbeschränkter Strukturen
führt über eine induktive Konstruktion: Man beginnt mit einer endlichen
Menge $M_0$, die nur einfache Werte enthält, z.B. $M_0 = \{\text{end}\}$ wenn man
nur einen einzigen Wert $\text{end}$ braucht. Dann beschreibt man, wie man über
endlich viele Möglichkeiten aus einer Menge $M_i$ die Menge $M_{i+1}$ \((i \geq 0)\)
genenerieren kann, wobei $M_{i+1}$ zumindest alle Elemente von $M_i$ enthält, z.B.
$M_{i+1} = M_i \cup \{\text{elem}(n, x) \mid n \in \text{Int}; x \in M_i\}$. Die meist unendliche Menge
$M = \bigcup_{i=0}^{\infty} M_i$, also die Vereinigung aller $M_i$, enthält dann alle beschriebe-
nen Strukturen. Die als Beispiel konstruierte Menge enthält die Elemente
$\text{end}, \text{elem}(3, \text{end}), \text{elem}(7, \text{elem}(3, \text{end}))$ und so weiter, also alle
Listen ganzer Zahlen. Vor allem in neueren funktionalen Sprachen werden
Typen von Datenstrukturen tatsächlich auf diese Weise konstruiert. Der Typ \texttt{Lst}, dessen Werte die Elemente der oben beschriebenen Menge sind, wird in Haskell so definiert (wobei „|“ Alternativen trennt):

\begin{verbatim}
data Lst = end | elem(Int, Lst)
\end{verbatim}

Aus der Syntax geht auf den ersten Blick hervor, dass \texttt{Lst} ein rekursiver Typ ist, weil \texttt{Lst} sowohl links als auch in zumindest einer Alternative rechts von = vorkommt. Rekursion beschreibt die Mengenkonstruktion nur in leicht abgewandelter Form (so wie $M_i$ in $M_{i+1}$ verwendet wird).


Man muss klar zwischen $M_0$ (nicht-rekursiv) und der Konstruktion aller $M_i$ mit $i > 0$ (rekursiv) unterscheiden, wobei $M_0$ nicht leer sein darf. Diese Eigenschaft nennt man Fundiertheit. In Haskell muss es in jeder Typdefinition zumindest eine nicht-rekursive Alternative (z.B. \texttt{end} in \texttt{Lst}) geben.

Viele Sprachen verwenden einen einfacheren Ansatz für nicht-elementare Typen: Die Menge $M_0$ ist etwa in Java schon in der Sprachdefinition vorgegeben und enthält nur den speziellen Wert \texttt{null}, und Klassen beschreiben die $M_i$ mit $i > 0$. Da statt einem Objekt immer auch \texttt{null} verwendet werden kann, ist Fundiertheit immer gegeben. Der Nachteil dieser Vereinfachung liegt auf der Hand: Man muss beim Programmieren stets damit rechnen, statt einem Objekt nur \texttt{null} zu erhalten, auch wenn man es gar nicht mit rekursiven Datenstrukturen zu tun hat.

### Typinferenz

Die Techniken hinter der Typinferenz sind heute weitgehend ausgereift, und für bestimmte Aufgaben wird Typinferenz sehr breit eingesetzt, auch in Java, C# und C++. Ein technisches Problem ist bis heute jedoch ungelöst: Typinferenz funktioniert nicht, wenn gleichzeitig (also an derselben Stelle im Programm) Ersetzbarkeit durch Untertypen verwendet wird. Aus diesem Grund wird Typinferenz in neueren funktionalen Sprachen, in denen keine Untertypen verwendet werden, in größtmöglichem Umfang eingesetzt, aber nur sehr lokal und für wenige Sprachkonstrukte in objektorientierten Sprachen.
Im Gegensatz zu Untertypen verträgt sich Generizität sehr gut mit Typinferenz. Das gilt auch für Einschränkungen auf Typparameter, trotz großer Ähnlichkeiten der Einschränkungen zu Untertypen.


Manche sinnvolle Information ist nicht beliebig propagierbar. Beispielsweise wäre es unsinnig, die Information „das ist die einzige Referenz auf das Objekt“ durch Zuweisung an mehrere Variablen zu propagieren, weil dies die Richtigkeit der Information zerstören würde; danach gäbe es ja mehrere Referenzen. Mit solcher Information kann man umgehen, indem man das Propagieren kontrolliert: Man kann die Information zwar weitergeben, etwa vom Argument zum formalen Parameter, aber nicht duplizieren; das Argument hätte die Information durch die Weitergabe an den Parameter verloren. Derartiges funktioniert gut, wenn man Typen nicht als unveränderlich, sondern als zustandsbehaftet ansieht; je nach Zustand des Typs ist die Information vorhanden oder nicht. Solche Typen gibt es heute in experimentellen Sprachen. Diese Typen sind sehr mächtig, weil sie auch Eigenschaften ausdrücken können, die für die Synchronisation nebeneinanderlaufender Programme benötigt werden (z.B. Prozestypen).

Zusammenfassend kann man sagen, dass man mit statisch geprüften Typen heute viel realisieren kann, was noch vor wenigen Jahren als gänzlich unmöglich erschien. Umgekehrt kennt man aber auch unerwartete Einschränkungen bei statischen Typprüfungen, die es notwendig machen, manche Aspekte erst zur Laufzeit zu prüfen.

### 1.4 Objektorientierte Programmierung

Wenden wir uns nun verstärkt der objektorientierten Programmierung zu. Der Inhalt dieses Abschnitts sollte großteils schon aus anderen Lehrveranstaltungen bekannt sein. Hier wird also eine kurze Wiederholung gegeben, vor allem um die verwendete Terminologie wieder in Erinnerung zu rufen.

#### 1.4.1 Basiskonzepte

Die objektorientierte Programmierung will Softwareentwicklungsprozesse unterstützen, die auf inkrementelle Verfeinerung aufbauen. Gerade bei diesen Entwicklungsprozessen spielt die leichte Wartbarkeit eine große Rolle. Im Wesentlichen geben objektorientierte Sprachen Entwickler(inne)n Werkzeuge in die Hand, die sie zum Schreiben leicht wiederverwendbarer und änderbarer Software brauchen.

**Objekt.** Ein Objekt ist eine grundlegende Modularisierungseinheit in der Ausführung eines Programms – siehe Abschnitt 1.2.1. Zur Laufzeit besteht
die Software aus einer Menge von Objekten, die einander teilweise kennen und untereinander Nachrichten (Messages) austauschen.

Die folgende Abbildung veranschaulicht ein Objekt:

**Objekt: einStack**

private Variablen:

<table>
<thead>
<tr>
<th>elems:</th>
<th>&quot;a&quot;</th>
<th>&quot;b&quot;</th>
<th>&quot;c&quot;</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>size:</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

öffentlich sichtbare Methoden:

<table>
<thead>
<tr>
<th>push:</th>
<th>Implementierung der Methode</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop:</td>
<td>Implementierung der Methode</td>
</tr>
</tbody>
</table>


Jedes Objekt besitzt folgende Eigenschaften [35]:

**Identität (Identity):** Ein Objekt ist durch seine unveränderliche Identität eindeutig gekennzeichnet. Über seine Identität kann man das Objekt ansprechen, ihm also eine Nachricht schicken. Vereinfacht kann man sich die Identität als die Adresse des Objekts im Speicher vorstellen. Dies ist aber nur eine Vereinfachung, da die Identität erhalten bleibt, wenn sich die Adresse ändert – zum Beispiel beim Verschieben des Objekts bei der Garbage-Collection oder beim Auslagern in eine Datenbank. Jedenfalls gilt: Gleichzeitig durch zwei Namen bezeichnete Objekte sind *identisch (identical)* wenn sie am selben Speicherplatz liegen, es sich also um nur ein Objekt mit zwei Namen handelt.


Schnittstelle und Implementierung. Unter der Implementierung einer Methode verstehen wir den Programmcode, der festlegt, was bei Ausführung der Methode zu tun ist. Die Implementierungen aller Methoden eines Objekts und die Deklarationen der Objektvariablen bilden zusammen die Implementierung des Objekts. Die Implementierung beschreibt das Verhalten bis ins kleinste Detail. Für die Programmausführung ist die genaue Beschreibung essenziell; sonst wüsste der Computer nicht was zu tun ist. Für die Wartung ist es günstiger, wenn das Verhalten nicht jedes Detail widerspiegelt. Statt der Implementierung haben wir beim Programmieren nur eine abstrakte Vorstellung im Kopf, die viele Details offen lässt. Beispielsweise gibt die Methodenbeschreibung

```
push fügt beim Empfang der Nachricht `push("d")` das Argument "d" in den Stack ein (falls es einen freien Platz gibt)
```

eine abstrakte Vorstellung davon wider, was tatsächlich passiert. Es bleibt offen, wie und wo "d" eingefügt wird und wann Platz frei ist. Menschen können mit solchen Abstraktionen ganz selbstverständlich und viel einfacher umgehen als mit Implementierungen. Bei Computern ist es genau umgekehrt. Daher wollen wir Beschreibungen des Objektverhaltens so weit
wie möglich abstrakt halten und erst dann zur Implementierung übergehen, wenn dies für den Computer notwendig ist.

Wirfordern eine weitere Objekteigenschaft, die den Abstraktionsgrad des Verhaltens nach Bedarf steuern lässt:


**Klasse.** Viele, aber nicht alle objektorientierten Sprachen beinhalten ein Klassenkonzept: Jedes Objekt gehört zu der Klasse, in der das Objekt implementiert ist. Die Klasse beschreibt auch Konstruktoren zur Initialisierung neuer Objekte. Alle Objekte, die zur Klasse gehören, wurden durch Konstruktoren dieser Klasse initialisiert. Man nennt diese Objekte auch

Anmerkung: Man sagt auch, ein Objekt gehöre zu mehreren Klassen, der spezifischsten und deren Oberklassen. Im Skriptum verstehen wir unter der „Klasse des Objekts“ immer die spezifischste Schnittstelle und sprechen allgemein von der „Schnittstelle“ wenn wir eine beliebige meinen.


In einer objektorientierten Sprache mit Klassen schreibt man beim Programmieren hauptsächlich Klassen. Objekte werden nur zur Laufzeit durch Verwendung von Konstruktoren oder durch Clonen erzeugt.

Ein Stack könnte in Java so implementiert sein:

```java
public class Stack {
    private String[] elems;
    private int size = 0;

    public Stack (int sz) { // new stack of size sz
        elems = new String[sz];
    }

    // push pushes elem onto stack if not yet full
    public void push(String elem) {
        if (size < elems.length)
            elems[size++] = elem;
    }

    // pop returns element taken from stack if
    // not empty; otherwise pop returns null
    public String pop() {
        if (size > 0)
            return elems[--size];
        return null;
    }
}
```

Die Kommentare sind Zusicherungen, also mehr als nur Erläuterungen.
zum besseren Verständnis. Sie beschreiben das abstrakte Verhalten soweit dies für Aufrufer relevant ist. Wer einen Aufruf in das Programm einfügt, soll sich auf die Kommentare verlassen und nicht die Implementierung betrachten müssen. Außerdem können Details der Implementierung problemlos nachträglich geändert werden, solange die unveränderten Kommentare die geänderte Implementierung treffend beschreiben.

### 1.4.2 Polymorphismus

Das Wort *polymorph* kommt aus dem Griechischen und heißt „vielgestaltig“. Im Zusammenhang mit Programmiersprachen spricht man von *Polymorphismus*, wenn eine Variable oder Methode gleichzeitig mehrere Typen haben kann. Ein formaler Parameter einer polymorphen Methode kann an Argumente von mehr als nur einem Typ gebunden werden. Objektorientierte Sprachen sind polymorph. Im Gegensatz dazu sind konventionelle statisch typisierte Sprachen wie C und Pascal *monomorph*: Jede Variable oder Funktion hat einen eindeutigen Typ.

Man kann verschiedene Arten des Polymorphismus unterscheiden:

\[
\text{Polymorphismus} \begin{cases} 
\text{universeller} & \text{Generizität} \\
\text{Polymorphismus} & \text{Untertypen} \\
\text{Ad-hoc-} & \text{Überladen} \\
\text{Polymorphismus} & \text{Typumwandlung}
\end{cases}
\]

1.4 Objektorientierte Programmierung

In der objektorientierten Programmierung sind Untertypen von überraschender Bedeutung, die anderen Arten des Polymorphismus existieren eher nebenbei. Daher nennt man alles, was mit Untertypen zu tun hat, oft auch \textit{objektorientierten Polymorphismus} oder nur kurz Polymorphismus.

In einer objektorientierten Sprache hat eine Variable (oder ein formaler Parameter) gleichzeitig folgende Typen:

\textbf{Deklarierter Typ:} Das ist der Typ, mit dem die Variable deklariert wurde. Dieser existiert natürlich nur bei expliziter Typdeklaration.


1 Paradigmen der Programmierung


In populären objektorientierten Programmiersprachen können bei der Vererbung Unterklassen im Vergleich zu Oberklassen aber nicht beliebig geändert werden. Eigentlich gibt es nur zwei Änderungsmöglichkeiten:

Erweiterung: Die Unterkasse erweitert die Oberklasse um neue Variablen, Methoden und Konstruktoren.


Diese beiden Änderungsmöglichkeiten sind beliebig kombinierbar.


Wir verwenden in diesem Skriptum für das Sprachkonstrukt in Java den englischen Begriff, während wir mit dem gleichbedeutenden deutschen Begriff Schnittstellen im Allgemeinen bezeichnen.

1.4.3 Vorgehensweisen in der Programmierung


**Faustregel:** Gute Faktorisierung kann die Wartbarkeit eines Programms wesentlich erhöhen.

Die objektorientierte Programmierung bietet viele Möglichkeiten zur Faktorisierung und erleichtert damit das Finden einer guten Zerlegung. Bei Weitem nicht jede Faktorisierung ist gut. Es ist unsere Aufgabe beim Programmieren die Qualität von Faktorisierungen zu beurteilen.

der realen Welt simulieren, die für die entwickelte Software bedeutungslos sind. Die Einfachheit ist wichtiger.

**Faustregel:** Man soll die reale Welt simulieren, aber nur so weit, dass die Komplexität dadurch nicht erhöht wird.


**Faustregel:** Zykli sche Prozesse verkraften Anforderungsänderungen besser, aber Zeit und Kosten sind schwer planbar.


Gerade in der objektorientierten Programmierung ist der gezielte Einsatz von Erfahrungen essenziell. Objektorientierte Sprachen bieten viele...
1.4 Objektorientierte Programmierung

unterschiedliche Möglichkeiten zur Lösung von Aufgaben. Jede Möglich-
keit hat andere Eigenschaften. Mit ausreichend Erfahrung wird man jene
Möglichkeit wählen, deren Eigenschaften später am ehesten hilfreich sind.
Mit wenig Erfahrung wählt man einfach nur die Möglichkeit, die man zu-
erst entdeckt. Damit verzichtet man auf einen wichtigen Vorteil. Generell
kann man sagen, dass die objektorientierte Programmierung durch erfah-
rene Leute derzeit wahrscheinlich das erfolgversprechendste Programmier-
paradigma darstellt, andererseits aber Gelegenheitsprogrammierer(innen)
und noch unerfahrene Softwareentwickler(innen) oft überfordert.

Zusammenhalt und Kopplung. Bei der Softwareentwicklung müssen wir
in jeder Phase wissen, wie wir vorgehen müssen um möglichst hochwer-
tige Software zu produzieren. Vor allem eine gute Faktorisierung ist ein
extscheidendes Kriterium, aber leider erst gegen Ende der Entwicklung
beurteilbar. Daher gibt es Faustregeln, die uns beim Finden guter Fakto-
risierungen unterstützen. Wir wollen hier einige Faustregeln betrachten,
die in vielen Fällen einen Weg zu guter Faktorisierung weisen

Verantwortlichkeiten (Responsibilities): Die Verantwortlichkeiten einer
Klasse können wir durch drei w-Ausdrücke beschreiben:

• „was ich weiß“ – Beschreibung des Zustands der Objekte
• „was ich mache“ – Verhalten der Objekte
• „wen ich kenne“ – sichtbare Objekte, Klassen, etc.

Das Ich steht dabei jeweils für die Klasse. Wenn etwas geändert wer-
den soll, das in den Verantwortlichkeiten einer Klasse liegt, dann sind
dafür die Entwickler(innen) dieser Klasse zuständig.

Klassen-Zusammenhalt (Class-Cohesion): Darunter versteht man den
Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse.
Dieser Grad ist zwar nicht einfach messbar, oft aber intuitiv einfach
fassbar. Der Zusammenhalt ist hoch, wenn alle Variablen und Me-
thoden eng zusammenarbeiten und durch den Namen der Klasse gut
beschrieben sind. Das heißt, einer Klasse mit hohem Zusammenhalt
fehlt etwas Wichtiges, wenn man beliebige Variablen oder Methoden
entfernt. Außerdem wird der Zusammenhalt niedriger, wenn man die
Klasse sinnändernd umbenennt.

Objekt-Kopplung (Object-Coupling): Das ist die Abhängigkeit der Ob-
jeekte voneinander. Die Objekt-Kopplung ist stark, wenn
1 Paradigmen der Programmierung

- viele Methoden und Variablen nach außen sichtbar sind,
- im laufenden System Nachrichten und Variablenzugriffe zwischen unterschiedlichen Objekten häufig auftreten
- und die Anzahl der Parameter dieser Methoden groß ist.

**Faustregel:** Der Klassen-Zusammenhalt soll hoch sein.


**Faustregel:** Die Objekt-Kopplung soll schwach sein.


**Faustregel:** Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen.

### 1.4.4 Wiederverwendung und Paradigmenwahl

Es ist sinnvoll, bewährte Software so oft wie möglich wiederzuverwenden. Das spart Entwicklungsaufwand. Wir müssen zwischen verschiedenen Arten der Wiederverwendung unterscheiden:

**Programme:** Die meisten Programme werden im Hinblick darauf entwickelt, dass sie häufig (wieder)verwendet werden. Dadurch zahlt es sich erst aus, einen großen Aufwand in die Entwicklung zu stecken.

**Daten:** Auch Daten in Datenbanken und Dateien werden in vielen Fällen häufig wiederverwendet. Nicht selten haben Daten eine längere Lebensdauer als die Programme, die sie benötigen oder manipulieren.

**Erfahrungen:** Häufig unterschätzt wird die Wiederverwendung von Konzepten und Ideen in Form von Erfahrungen. Diese können zwischen sehr unterschiedlichen Projekten ausgetauscht werden.

**Code:** Viele Konzepte von Programmiersprachen, wie zum Beispiel Untertypen, Vererbung und Generizität, wurden im Hinblick auf die Wiederverwendung von Code entwickelt. Man kann mehrere Arten der Codewiederverwendung unterscheiden:
Paradigmen der Programmierung

**Bibliotheken:** Einige Klassen in Klassenbibliotheken werden sehr häufig (wieder)verwendet. Allerdings kommen nur wenige, relativ einfache Klassen für die Aufnahme in Bibliotheken in Frage. Komplexere Klassen sind dafür meist zu stark projektbezogen.

**Projektinterne Wiederverwendung:** Hochspezialisierte Programmteile sind nur innerhalb eines Projekts in unterschiedlichen Programmversionen wiederverwendbar. Wegen der Komplexität erspart bereits eine einzige Wiederverwendung viel Arbeit.

**Programminterne Wiederverwendung:** Code in einem Programm kann zu unterschiedlichen Zwecken oft wiederholt ausgeführt werden. Durch den Einsatz eines Programmteils in mehreren Aufgaben wird das Programm einfacher und leichter wartbar.

Die Erfahrung zeigt, dass durch objektorientierte Programmierung tatsächlich Code-Wiederverwendung erzielbar ist. Kosteneinsparungen ergeben sich aber normalerweise nur

- mit ausreichend Erfahrung um die Möglichkeiten gut zu nutzen
- und wenn viel Zeit in die Wiederverwendbarkeit investiert wird.

Mit wenig Erfahrung investiert man oft zu wenig, zu viel oder an falscher Stelle in die Wiederverwendbarkeit. Fehlentscheidungen rächen sich durch lange Entwicklungszeiten oder sogar das Scheitern eines Projekts. Im Zweifel soll man anfangs eher weniger in die Wiederverwendbarkeit investieren, diese Investitionen aber nachholen, sobald sich ein Bedarf ergibt.

**Fautregel:** Code-Wiederverwendung erfordert beträchtliche Investitionen in die Wiederverwendbarkeit. Man soll diese tätigen, wenn ein tatsächlicher Bedarf absehbar ist.


In vielen Programmierparadigmen stehen Algorithmen zentral im Mittelpunkt, etwa in der prozeduralen, funktionalen und logikorientierten Programmierung. Für die objektorientierte Programmierung gilt das nicht. Hier steht die Datenabstraktion im Mittelpunkt, aber Algorithmen müssen unter Umständen aufwendig auf mehrere Objekte verteilt werden. Das
kann den Entwicklungsaufwand von Algorithmen erhöhen und deren Verständlichkeit verringern. Diese Unterschiede haben einen Einfluss darauf, wofür die Paradigmen gut geeignet sind.

**Faustregel:** Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

Wenn die Wiederverwendung und Wartbarkeit von untergeordneter Bedeutung ist, beispielsweise weil man die Software nur einmal einsetzt, kann die objektorientierte Programmierung ihre Vorteile nicht ausspielen. Diese kommen erst durch einen langen Einsatz- und Wartungszeitraum zum Tragen. In anderen Paradigmen kann man wesentlich rascher programmieren, da man sich nicht um die Wiederverwendbarkeit kümmern muss. Soll Software aber über einen längeren Zeitraum gewartet werden, zahlt es sich aus, durch objektorientierte Programmierung etwas mehr in die Entwicklung zu investieren. Sonst zahlt es sich nicht aus.

### 1.5 Wiederholungsfragen


1. Was versteht man unter einem Programmierparadigma?
2. Wozu dient ein Berechnungsmodell?
3. Welche Berechnungsmodelle werden in Programmierparadigmen verwendet, und welche charakteristischen Eigenschaften haben sie?
4. Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?
5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?
6. Was ist die strukturierte Programmierung? Wozu dient sie?
7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?
1 Paradigmen der Programmierung

8. Was bedeutet referentielle Transparenz, und wo findet man referentielle Transparenz?

9. Wieso passt referentielle Transparenz nicht gut mit Ein- und Ausgabe zusammen, und wie kann man das Dilemma lösen?

10. Welchen Zusammenhang gibt es zwischen Seiteneffekten und der objektorientierten Programmierung?

11. Was sind First-Class-Entities? Welche Gründe sprechen für deren Verwendung, welche dagegen?

12. Was haben Funktionen höherer Ordnung mit einem applikativen Programmierstil zu tun?

13. Welche Modularisierungseinheiten gibt es, was sind ihre charakteristischen Eigenschaften, und wodurch unterscheiden sie sich?

14. Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten? Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?

15. Was ist und wozu dient ein Namensraum?

16. Warum können Module nicht zyklisch voneinander abhängen, Komponenten aber schon?

17. Was versteht man unter Datenabstraktion, Kapselung und Data-Hiding?

18. Warum und inwiefern ist die Einbindung von Komponenten komplizierter als die von Modulen?

19. Wie kann man globale Namen verwalten und damit Namenskonflikte verhindern?

20. Was versteht man unter Parametrisierung? Wann kann das Befüllen von „Löchern“ durch welche Techniken erfolgen?

21. Warum braucht man zur Parametrisierung in der Objekterzeugung neben Konstruktoren gelegentlich auch Initialisierungsmethoden?

22. Welche Vor- und Nachteile hat die zentrale Ablage von Werten zum Zweck der Parametrisierung?
23. Was unterscheidet Generizität von den verschiedenen Formen der Parametrisierung zur Laufzeit?

24. Was sind Annotationen und wozu kann man sie verwenden? Wodurch unterscheiden sie sich von Generizität?

25. Was versteht man unter aspektorientierter Programmierung?

26. Wodurch unterscheiden sich die verschiedenen Formen der Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit in der objektorientierten Programmierung von so zentraler Bedeutung?

27. Wann ist $A$ durch $B$ ersetzbar?

28. Wodurch kann festgelegt sein, ob $A$ durch $B$ ersetzbar ist?

29. Was ist die Signatur einer Modularisierungseinheit?

30. Wie verhält sich die Signatur einer Modularisierungseinheit zur Abstraktion, die durch diese Modularisierungseinheit gebildet wird?

31. Was sind Zusicherungen, und welche Rolle spielen sie für Modularisierungseinheiten?

32. Wann sind Typen miteinander konsistent, und was sind Typfehler?

33. Wie schränken Typen die Flexibilität ein, und warum verwendet man Typen trotzdem?

34. Welche Gründe sprechen für den Einsatz statischer Typprüfungen, welche dagegen?

35. Was versteht man unter Typinferenz? Welche Gründe sprechen für bzw. gegen deren Einsatz?

36. Zu welchen Zeitpunkten können Entscheidungen getroffen werden (Typen und Entscheidungsprozesse)?

37. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?

38. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?

39. Was ist ein abstrakter Datentyp?

40. Was unterscheidet strukturelle von nominalen Typen?
1 Paradigmen der Programmierung

41. Warum verwenden wir in Programmiersprachen meist nominale Typen, in theoretischen Modellen aber hauptsächlich strukturelle?

42. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?

43. Warum kann ein Compiler ohne Unterstützung durch Programmierer(innen) nicht entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?

44. Erklären Sie Einschränkungen bei Untertypbeziehungen zusammen mit statischer Typprüfung.

45. In welchem Zusammenhang verwendet man Higher-Order-Subtyping und F-gebundene Generizität?

46. Wie konstruiert man rekursive Datenstrukturen?

47. Was versteht man unter Fundiertheit rekursiver Datenstrukturen? Welche Ansätze dazu kann man unterscheiden?

48. Warum wird Typinferenz in objektorientierten Sprachen meist nur lokal beschränkt eingesetzt?

49. Wie können statisch geprüfte Typen beliebige Eigenschaften von Werten propagieren?

50. Erklären Sie folgende Begriffe:
   - Objekt, Klasse, Vererbung
   - Identität, Zustand, Verhalten, Schnittstelle
   - deklarierter, statischer und dynamischer Typ
   - Faktorisierung, Refaktorisierung
   - Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung

51. Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?

52. Wann sind zwei gleiche Objekte identisch und wann sind zwei identische Objekte gleich?

53. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?

54. Was besagt das Ersetzbarkeitsprinzip? (Häufige Prüfungsfrage!)
1.5 Wiederholungsfragen

55. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig (mehrere Gründe)?

56. Wann und warum ist gute Wartbarkeit wichtig?

57. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?

58. Welche Arten von Software kann man wiederverwenden, und welche Rolle spielt jede davon in der Softwareentwicklung?

59. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?

60. Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?
1 Paradigmen der Programmierung
2 Untertypen und Vererbung

In Abschnitt 2.1 untersuchen wir Grundlagen von Untertypbeziehungen. In Abschnitt 2.2 gehen wir auf wichtige Aspekte des Objektverhaltens ein, die man bei der Verwendung von Untertypen beachten muss. Danach betrachten wir in Abschnitt 2.3 die Vererbung im Zusammenhang mit direkter Codewiederverwendung. Schließlich beleuchten wir in Abschnitt 2.4 einige Details zu Klassen und Interfaces in Java.

2.1 Das Ersetzbarkeitsprinzip

Untertypbeziehungen sind durch das Ersetzbarkeitsprinzip definiert:

\textbf{Definition:} Ein Typ $U$ ist Untertyp eines Typs $T$, wenn jedes Objekt von $U$ überall verwendbar ist, wo ein Objekt von $T$ erwartet wird.

Per Definition ist ein Objekt eines Untertyps überall verwendbar, wo ein Objekt eines Obertyps erwartet wird. Insbesondere benötigt man das Ersetzbarkeitsprinzip für

- den Aufruf einer Methode mit einem Argument, dessen Typ ein Untertyp des Typs des entsprechenden formalen Parameters ist
- und für die Zuweisung eines Objekts an eine Variable, wobei der Typ des Objekts ein Untertyp des deklarierten Typs der Variable ist.

Beide Fälle kommen in der objektorientierten Programmierung häufig vor.

2.1.1 Untertypen und Schnittstellen

Die Frage danach, wann das Ersetzbarkeitsprinzip erfüllt ist, wurde in der Fachliteratur intensiv behandelt [2, 4, 24]. Wir wollen diese Frage hier nur so weit beantworten, als es in der Praxis relevant ist. Fast alles, was wir anhand von Java untersuchen, gilt auch für andere objektorientierte
Sprachen, zumindest für solche mit statischer Typprüfung wie C# und C++. Wir gehen davon aus, dass Typen Schnittstellen von Objekten sind, die in Klassen beziehungsweise Interfaces spezifiziert wurden. Es gibt in Java auch elementare Typen wie `int`, die keiner Klasse entsprechen. Aber auf solchen Typen haben wir keine Untertypbeziehungen. Deshalb werden wir sie hier nicht näher betrachten.


**Strukturelle Untertypbeziehungen:** Unter den folgenden Bedingungen stehen strukturelle Typen in einer Untertypbeziehung. Diese Bedingungen gelten allgemein und sind nicht auf eine bestimmte Programmiersprache bezogen. Alle Untertypbeziehungen sind stets

- reflexiv – jeder Typ ist Untertyp von sich selbst,
- transitiv – ist ein Typ $U$ Untertyp eines Typs $A$ und ist $A$ Untertyp eines Typs $T$, dann ist $U$ auch Untertyp von $T$,

Beliebige strukturelle Typen bezeichnen wir mit $U$ und $T$ sowie $A$ und $B$. Es gilt $U$ ist Untertyp von $T$ wenn folgende Bedingungen erfüllt sind:

- Für jede Konstante (also jede spezielle Variable, die nach der Initialisierung nur lesende Zugriffe erlaubt) in $T$ gibt es eine entsprechende Konstante in $U$, wobei der deklarierte Typ $B$ der Konstante in $U$ ein Untertyp des deklarierten Typs $A$ der Konstante in $T$ ist.

- Für jede Variable in T gibt es eine entsprechende Variable in U, wobei die deklarierten Typen der Variablen äquivalent sind.


- Für jede Methode in T gibt es eine entsprechende gleichnamige Methode in U, wobei
  - der deklarierte Ergebnistyp der Methode in U ein Untertyp des deklarierten Ergebnistyps der Methode in T ist,
  - die Anzahl der formalen Parameter beider Methoden gleich ist
  - und der deklarierte Typ jeden formalen Parameters in U ein Obertyp des deklarierten Typs des entsprechenden formalen Parameters in T ist (gilt nur für Eingangsparameter).

Begründung: Für die Ergebnistypen der Methoden gilt dasselbe wie für Typen von Konstanten beziehungsweise lesende Zugriffe auf Variablen: Der Aufrufer einer Methode möchte ein Ergebnis des in T versprochenen Ergebnistyps bekommen, auch wenn die entsprechende Methode in U ausgeführt wird. Für die Typen der formalen Parameter gilt dasselbe wie für schreibende Zugriffe auf Variablen: Der
2 Untertypen und Vererbung

Aufrufer möchte alle Argumente der Typen an die Methode übergeben können, die in $T$ deklariert sind, auch wenn die entsprechende Methode in $U$ ausgeführt wird. Daher dürfen die Parametertypen in $U$ nur Obertypen der Parametertypen in $T$ sein.


Alle diese Bedingungen gelten nur für Variablen und Methoden, die außerhalb eines Objekts sichtbar sind, die also zur Schnittstelle gehören. Private Inhalte haben keinen Einfluss auf Untertypbeziehungen.

Diese Bedingungen hängen nur von den Strukturen der Typen ab. Sie berücksichtigen das Verhalten in keiner Weise. Solche Untertypbeziehungen sind auch gegeben, wenn die Strukturen nur zufällig zusammenpassen.

Obigen Regeln entsprechend kann ein Untertyp einen Obertyp nicht nur um neue Elemente erweitern, sondern auch deklarierte Typen der Elemente (z.B. Parameter) ändern; das heißt, die deklarierten Typen der Elemente können variieren. Man unterscheidet folgende Arten der Varianz:


**Kontravarianz:** Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp. Zum Beispiel sind deklarierte Typen von formalen Eingangsparametern kontravariant. Typen und die betrachteten darin enthaltenen Elementtypen variieren in entgegengesetzte Richtungen.

Möglichkeiten und Grenzen: Ein Beispiel in einer Java-ähnlichen Sprache:

```java
public class A {
    public A meth(B par) { ... }
}

public class B {
    public B meth(A par) { ... }
    public void foo() { ... }
}
```


Das alles gilt in einer Java-ähnlichen Sprache, die auf strukturellen Typen beruht. Java verwendet jedoch nominale Typen, und in Java ist B kein Untertyp von A. Wenn B mit der Klausel „extends A“ deklariert wäre, würde meth in B die Methode in A nicht überschreiben; stattdessen würde meth von A geerbt und überladen, so dass es in Objekten von B beide Methoden nebeneinander gäbe.


```java
public class Point2D {
    protected int x, y; // von außen sichtbar
    public boolean equal(Point2D p) {
        return x == p.x && y == p.y;
    }
}
```
2 Untertypen und Vererbung

public class Point3D {
    protected int x, y, z;
    public boolean equal(Point3D p) {
        return x == p.x && y == p.y && z == p.z;
    }
}

Wegen der zusätzlichen von außen sichtbaren Variable in Point3D sind die beiden Typen nicht äquivalent und kann Point2D kein Untertyp von Point3D sein. Außerdem kann Point3D kein Untertyp von Point2D sein, da equal nicht die Kriterien für Untertypbeziehungen erfüllt. Der Parametertyp wäre ja kovariant und nicht, wie gefordert, kontravariant.

Eine Methode wie equal, bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt binäre Methode. Die Eigenschaft binär bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zweimal vorkommt – einmal als Typ von this und mindestens einmal als Typ eines formalen Parameters. Binäre Methoden werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) aber prinzipiell nicht realisierbar.


Untertypbeziehungen in Java setzen entsprechende Vererbungsbeziehungen voraus. Vererbung ist in Java so eingeschränkt, dass zumindest alle Bedingungen für Untertypbeziehungen auf strukturellen Typen erfüllt sind. Die Bedingungen werden bei der Übersetzung eines Java-Programms fast lückenlos überprüft. (Eine Ausnahme bezogen auf Arrays und Generizität werden wir in Abschnitt 3.1.3 kennenlernen.)

Untertypbeziehungen sind in Java nicht nur aufgrund nominaler Typen stärker eingeschränkt als durch obige Bedingungen notwendig wäre. In Java sind alle Typen invariant, abgesehen von kovarianten Ergebnistypen ab Version 1.5. Der Grund dafür liegt darin, dass bei Zugriffen auf Variablen und Konstanten nicht dynamisch sondern statisch gebunden wird (dafür also keine Ersetzbarkeit nötig ist), und darin, dass Methoden überladen sein können. Da überladene Methoden durch die Typen der formalen Parameter unterschieden werden, wäre es schwierig, überladene Methoden von Methoden mit kontravariant veränderten Typen auseinanderzuhalten.
2.1.2 Untertypen und Codewiederverwendung

Die wichtigste Entscheidungsgrundlage für den Einsatz von Untertypen ist die erzielbare Wiederverwendung. Der richtige Einsatz eröffnet Möglichkeiten, die auf den ersten Blick gar nicht so leicht zu erkennen sind.


An diesem Bild fällt die Version 3b auf: Sie vereinigt die zwei inkompatiblen Vorgängerversionen 2a und 2b. Ein Untertyp kann mehrere Obertypen haben, die zueinander in keiner Untertypbeziehung stehen. Das ist ein Beispiel für Mehrfachvererbung, während in den anderen Fällen nur Einfachvererbung nötig ist. Diese Hierarchie kann in Java nur realisiert werden, wenn die Treiberschnittstellen Interfaces sind.

**Faustregel:** Man soll auf Ersetzbarkeit achten, um Codewiederverwendung zwischen Versionen zu erreichen.

Die Wiederverwendung zwischen verschiedenen Versionen funktioniert nur dann gut, wenn die Schnittstellen bzw. Typen zwischen den Versionen stabil bleiben. Das heißt, eine neue Version darf die Schnittstellen nicht beliebig ändern, sondern nur so, dass die in Abschnitt 2.1.1 beschriebenen Bedingungen erfüllt sind. Im Wesentlichen kann die Schnittstelle also nur erweitert werden. Wenn die Aufteilung eines Programms in einzelne Objekte gut ist, bleiben Schnittstellen normalerweise recht stabil.

**Faustregel:** Schnittstellen sollen stabil bleiben. Gute Faktorisierung hilft dabei.

Das, was in obigem Beispiel für verschiedene Versionen einer Klasse funktionierte, kann man genauso gut innerhalb eines einzigen Programms nutzen, wie wir an einem modifizierten Beispiel sehen. Wir wollen ein Programm zur Verwaltung der Personen an einer Universität entwickeln. Die dafür verwendete Klassenstruktur könnte so aussehen:

Entsprechend diesen Strukturen sind Tutor(inn)en sowohl Universitätslehrer(innen) als auch Student(inn)en, und Werkstudent(inn)en an der Universität gehören zum Verwaltungspersonal und sind Student(inn)en. Wir benötigen im Programm eine Komponente, die Serienbriefe – Einladungen zu Veranstaltungen, etc. – an alle Personen adressiert. Für das Erstellen einer Anschrift benötigt man nur Informationen aus der Klasse Person.
2.1 Das Ersetzbarkeitsprinzip


**Faustregel:** Man soll auf Ersetzbarkeit achten um interne Codiewiederverwendung im Programm zu erzielen.


**Faustregel:** Man soll auf Ersetzbarkeit achten um Programmänderungen lokal zu halten.

Falls bei der nötigen Programmänderung alle Schnittstellen der Klasse unverändert bleiben, betrifft die Änderung keine Programmstellen, an denen Student und dessen Unterklassen verwendet werden. Lediglich diese Klassen selbst sind betroffen. Auch daran kann man sehen, wie wichtig es ist, dass Schnittstellen und Typen stabil sind. Eine Programmänderung führt möglicherweise zu vielen weiteren nötigen Änderungen, wenn dabei eine Schnittstelle geändert wird. Die Anzahl wahrscheinlich nötiger Änderungen hängt auch davon ab, wo in der Klassenstruktur die geänderte Schnittstelle steht. Eine Änderung ganz oben in der Struktur hat wesentlich größere Auswirkungen als eine Änderung ganz unten. Eine Schlussfolgerung aus diesen Überlegungen ist, dass man möglichst nur von solchen...
Klassen ableiten soll, deren Schnittstellen bereits – oft nach mehreren Refaktorisierungsschritten – recht stabil sind.

**Faustregel:** Die Stabilität von Schnittstellen an der Wurzel der Typhierarchie ist wichtiger als an den Blättern. Man soll nur Untertypen von stabilen Obertypen bilden.


**Faustregel:** Man soll Parametertypen vorausschauend und möglichst allgemein wählen.


### 2.1.3 Dynamisches Binden


Wir demonstrieren dynamisches Binden an einem kleinen Beispiel:
Die Ausführung von DynamicBindingTest liefert folgende Ausgabe:

foo1A
foo2A
foo1B
foo2B


Der zweite Aufruf von test übergibt ein Objekt von B als Argument. Dabei ist A der deklarierte Typ von x, aber der dynamische Typ ist B. Wegen dynamischen Bindens werden diesmal foo1 und foo2 in B ausgeführt. Die dritte Zeile der Ausgabe enthält das Ergebnis des Aufrufs von foo1 in einem Objekt von B.

Die letzte Zeile der Ausgabe lässt sich folgendermaßen erklären: Da die Klasse B die Methode foo2 nicht überschreibt, wird foo2 von A geerbt. Der Aufruf von foo2 in B ruft fooX in der aktuellen Umgebung auf, das
2 Untertypen und Vererbung

ist ein Objekt von B. Die Methode fooX liefert als Ergebnis die Zeichenkette "foo2B", die in der letzten Zeile ausgegeben wird.

Bei dieser Erklärung muss man vorsichtig sein: Man macht leicht den Fehler anzunehmen, dass foo2 und daher auch fooX in A aufgerufen wird, da foo2 ja nicht explizit in B steht. Tatsächlich wird aber fooX in B aufgerufen, da B der spezifischste Typ der Umgebung ist.

Dynamisches Binden ist mit switch-Anweisungen und geschachtelten if-Anweisungen verwandt. Wir betrachten als Beispiel eine Methode, die eine Anrede in einem Brief, deren Art auf konventionelle Weise über eine ganze Zahl bestimmt ist, in die Standardausgabe schreibt:

```java
public void gibAnredeAus(int art, String name) {
    switch(art) {
        case 1: System.out.print("S.g. Frau " + name);
            break;
        case 2: System.out.print("S.g. Herr " + name);
            break;
        default: System.out.print(name);
    }
}
```

In der objektorientierten Programmierung wird man die Art der Anrede eher durch die Klassenstruktur zusammen mit dem Namen beschreiben:

```java
public class Adressat {
    protected String name;
    public void gibAnredeAus() {
        System.out.print(name);
    }
    ... // Konstruktoren und weitere Methoden
}
public class WeiblicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print("S.g. Frau " + name);
    }
}
public class MaennlicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print("S.g. Herr " + name);
    }
}
```


**Faustregel:** Dynamisches Binden ist `switch`-Anweisungen und geschachtelten `if`-Anweisungen vorzuziehen.

### 2.2 Ersetzbarkeit und Objektverhalten

In Abschnitt 2.1 haben wir Bedingungen kennengelernt, unter denen ein struktureller Typ Untertyp eines anderen ist. Für nominale Typen gilt daneben noch die Bedingung, dass der Untertyp explizit vom Obertyp abgeleitet sein muss. Die Erfüllung dieser Bedingungen wird vom Compiler
überprüft. In Java und den meisten anderen Sprachen mit statischer Typprüfung werden sogar etwas strengere Bedingungen geprüft, die nicht für Untertypen, sondern z.B. für das Überladen von Methoden sinnvoll sind.

Jedoch sind alle prüfbaren Bedingungen nicht ausreichend, um die uningeschränkte Ersetzbarkeit eines Objekts eines Obertyps durch ein Objekt eines Untertyps zu garantieren. Dazu müssen weitere Bedingungen hinsichtlich des Objektverhaltens erfüllt sein, die von einem Compiler nicht überprüft werden können. Wir müssen diese Bedingungen beim Programmieren selbst (ohne Werkzeugunterstützung) sicherstellen.

2.2.1 Client-Server-Beziehungen


Für die Ersetzbarkeit von Objekten sind Client-Server-Beziehungen bedeutend. Man kann ein Objekt gegen ein anderes austauschen, wenn das neue Objekt als Server allen Clients zumindest dieselben Dienste anbietet wie das ersetzte Objekt. Um das gewährleisten zu können, brauchen wir eine Beschreibung der Dienste, also das Erhalten der Objekte.

Das *Objektverhalten* beschreibt, wie sich das Objekt beim Empfang einer Nachricht verhält, das heißt, was das Objekt beim Aufruf einer Methode macht. Diese Definition von Objektverhalten lässt etwas offen: Es ist unklar, wie exakt die Beschreibung des Verhaltens sein soll. Einerseits beschreibt die Signatur das Objekt nur sehr unvollständig. Eine genauere Beschreibung wäre wünschenswert. Andererseits enthält die Implementierung, also der Programmcode in der Klasse, oft zu viele Implementierungsdetails, die bei der Betrachtung des Verhaltens hinderlich sind. Im Programmcode gibt es meist keine Beschreibung, deren Detailiertheitsgrad zwischen dem der Signatur und dem der Implementierung liegt. Wir haben es beim Objektverhalten also mit einem abstrakten Begriff zu tun. Er wird vom Programmcode nicht notwendigerweise widerspiegelt.

Es hat sich bewährt, das Verhalten eines Objekts als einen Vertrag zwischen dem Objekt als Server und seinen Clients zu sehen – *Design-by-Contract*. Der Server muss diesen Vertrag ebenso einhalten wie jeder
2.2 Ersetzbarkeit und Objektverhalten

einzelne Client, in einigen Fällen auch die Gemeinschaft aller Clients zusammen. Generell sieht der Softwarevertrag folgendermaßen aus [25, 29]:

Jeder Client kann Dienste des Servers in Anspruch nehmen, wenn die festgeschriebenen Bedingungen dafür erfüllt sind. Im Falle einer Inanspruchnahme setzt der Server die festgeschriebenen Maßnahmen und liefert dem Client ein Ergebnis, das die festgeschriebenen Bedingungen erfüllt.

Im einzelnen regelt der Vertrag für jeden vom Server angebotenen Dienst, also für jede aufrufbare Methode (unter der Annahme, dass auf Objektvariablen nur über Methoden zugegriffen wird), folgende Details:


**Invariante (Invariant):** Für die Erfüllung von Invarianten auf Objektvariablen sowohl vor als auch nach Ausführung jeder Methode ist grundsätzlich der Server zuständig. Direkte Schreibzugriffe von Clients auf Variablen des Servers kann der Server aber nicht kontrollieren; dafür sind die Clients verantwortlich. Zum Beispiel darf das Guthaben auf einem Sparbuch nie kleiner 0 sein, egal welche Operationen auf dem Sparbuch durchgeführt werden. Eine Invariante impliziert eine Nachbedingung auf jeder Methode des Servers.
**History-Constraint:** Diese Bedingungen schränken die Entwicklung von Objekten im Laufe der Zeit ein. Wir unterscheiden zwei Unterarten:

**Server-kontrolliert:** Sie ähneln Invarianten, schränken aber zeitliche Veränderungen der Variableninhalte eines Objekts ein. Z.B. kann der ganzzahlige Wert einer Variablen, die als Zähler verwendet wird, im Laufe der Zeit immer größer, aber niemals kleiner werden. Wie bei Invarianten ist für die Einhaltung der Serverzuständig. Wenn jedoch Clients die betroffenen Variablen direkt schreiben können, sind auch die Clients verantwortlich.

**Client-kontrolliert:** Über History-Constraints kann man auch die Reihenfolge von Methodenaufrufen einschränken. Beispielsweise darf man eine Methode namens `initialize` in jedem Objekt nur einmal aufrufen, und davor sind keine Aufrufe anderer Methoden erlaubt. Methodenaufrufe erfolgen durch Clients. Nur Clients können die Aufrufreihenfolge bestimmen und sind für die Einhaltung der Bedingungen verantwortlich. Manchmal ist es gar nicht möglich, die Aufrufreihenfolge im Objektzustand abzubilden, z.B. wenn `initialize` in einem durch Kopieren (`clone`) erzeugten Objekt ausgeführt werden soll; der kopierte Objektzustand sagt darüber ja nichts aus.

Da History-Constraints noch nicht etabliert und die Regeln dahinter weniger einheitlich sind, werden sie oft nicht als Möglichkeit zur Gestaltung des Softwarevertrags wahrgenommen. Dennoch steckt hinter ihnen viel Potenzial.

Vorbedingungen, Nachbedingungen, Invarianten und History-Constraints sind verschiedene Arten von *Zusicherungen (Assertions)*.

2.2 Ersetzbarkeit und Objektverhalten

Anmerkungen wie diese gehören nicht zum Prüfungsstoff. Folgendes Beispiel in Eiffel veranschaulicht Zusicherungen:

```eiffel
class KONTO feature {ANY}
  guthaben: Integer;
  ueberziehungsrahmen: Integer;

einzahlen (summe: Integer) is
  require summe >= 0
  do guthaben := guthaben + summe
  ensure guthaben = old guthaben + summe
  end; -{}- einzahlen

abheben (summe: Integer) is
  require summe >= 0;
  guthaben + ueberziehungsrahmen >= summe
  do guthaben := guthaben - summe
  ensure guthaben = old guthaben - summe
  end; -{}- abheben

  invariant guthaben >= -ueberziehungsrahmen
end -{}- class KONTO
```


Diese Klasse sollte bis auf einige syntaktische Details selbsterklärend sein. Die Klausel feature {ANY} besagt, dass die danach folgenden Variablen- und Methodendeklarationen überall im Programm sichtbar sind. Nach dem Schlüsselwort end und einem (in unserem Fall leeren) Kommentar kann zur besseren Lesbarkeit der Name der Methode oder der Klasse folgen.
Hier ist ein Java-Beispiel für Kommentare als Zusicherungen:

```java
public class Konto {
    public long guthaben;
    public long ueberziehungsrahmen;
    // guthaben >= -ueberziehungsrahmen

    // einzahlen addiert summe zu guthaben; summe >= 0
    public void einzahlen (long summe) {
        guthaben = guthaben + summe;
    }

    // abheben zieht summe von guthaben ab;
    // summe >= 0; guthaben+ueberziehungsrahmen >= summe
    public void abheben (long summe) {
        guthaben = guthaben - summe;
    }
}
```


2.2 Ersetzbarkeit und Objektverhalten

Bisher haben wir die Begriffe Typ (bzw. Schnittstelle) und Signatur (bei nominalen Typen zusammen mit Namen) als im Wesentlichen gleichbedeutend angesehen. Ab jetzt betrachten wir Zusicherungen, unabhängig davon, ob sie durch eigene Sprachkonstrukte oder in Kommentaren beschrieben sind, als zum Typ (und zur Schnittstelle) eines Objekts gehörend. Ein nominaler Typ besteht demnach aus

- dem Namen einer Klasse, eines Interfaces oder elementaren Typs,
- der entsprechenden Signatur
- und den dazugehörenden Zusicherungen.


In Abschnitt 2.1 haben wir gesehen, dass Typen wegen der besseren Wartbarkeit stabil sein sollen. Solange eine Programmänderung den Typ der Klasse unverändert lässt oder nur auf unbedenkliche Art und Weise erweitert (siehe Abschnitt 2.2.2), hat die Änderung keine Auswirkungen auf andere Programmenteile. Das betrifft auch Zusicherungen. Eine Programmänderung kann sich sehr wohl auf andere Programmenteile auswirken, wenn dabei eine Zusicherung (= Kommentar) geändert wird.

**Faustregel:** Zusicherungen sollen stabil bleiben. Das ist für Zusicherungen in Typen an der Wurzel der Typierarchie ganz besonders wichtig.

Wir können die Genauigkeit der Zusicherungen selbst bestimmen. Dabei sind Auswirkungen der Zusicherungen zu beachten: Clients dürfen sich nur auf das verlassen, was in der Signatur und in den Zusicherungen vom Server zugesagt wird, und der Server auf das, was von den Clients zugesagt wird. Beispiele dafür folgen in Abschnitt 2.2.2. Sind die Zusicherungen sehr genau, können sich die Clients auf viele Details des Servers verlassen, und auch der Server kann von den Clients viel verlangen. Aber Programmänderungen werden mit größerer Wahrscheinlichkeit dazu führen, dass Zusicherungen geändert werden müssen, wovon alle Clients betroffen sind.
2 Untertypen und Vererbung

Steht hingegen in den Zusicherungen nur das Nötigste, sind Clients und Server relativ unabhängig voneinander. Der Typ ist bei Programmmänderungen eher stabil. Aber vor allem die Clients dürfen sich nur auf Weniges verlassen. Wenn keine Zusicherungen gemacht werden, dürfen sich Clients auf nichts verlassen, was nicht in der Signatur steht.

**Faustregel:** Zur Verbesserung der Wartbarkeit sollen Zusicherungen keine unnötigen Details festlegen.


**Faustregel:** Alle benötigten Zusicherungen sollen (explizit als Kommentare oder zumindest durch sprechende Namen impliziert) im Programm stehen.
2.2 Ersetzbarkeit und Objektverhalten


2.2.2 Untertypen und Verhalten

Zusicherungen, die zu Typen gehören, müssen auch bei der Verwendung von Untertypen beachtet werden. Auch für Zusicherungen gilt das Ersetzbarkeitsprinzip bei der Feststellung, ob ein Typ Untertyp eines anderen Typs ist. Neben den Bedingungen, die wir in Abschnitt 2.1 kennengelernt haben, müssen folgende Bedingungen gelten, damit ein Typ `U` Untertyp eines Typs `T` ist [24]:

**Vorbedingung:** Jede Vorbedingung auf einer Methode in `T` muss eine Vorbedingung auf der entsprechenden Methode in `U` implizieren. Das heißt, Vorbedingungen in Untertypen können schwächer, dürfen aber nicht stärker sein als entsprechende Vorbedingungen in Obertypen. Der Grund liegt darin, dass ein Aufrufer der Methode, der nur `T` kennt, nur die Erfüllung der Vorbedingungen in `T` sicherstellen kann, auch wenn die Methode tatsächlich in `U` statt `T` aufgerufen wird. Daher muss die Vorbedingung in `U` automatisch erfüllt sein, wenn sie in `T` erfüllt ist. Wenn Vorbedingungen in `U` aus `T` übernommen werden, können sie mittels Oder-Verknüpfungen schwächer werden. Ist die Vorbedingung in `T` zum Beispiel „`x > 0`“, kann die Vorbedingung in `U` auch „`x > 0` oder `x = 0`“, also abgekürzt „`x ≥ 0`“ lauten.

**Nachbedingung:** Jede Nachbedingung auf einer Methode in `U` muss eine Nachbedingung auf der entsprechenden Methode in `T` implizieren. Das heißt, Nachbedingungen in Untertypen können stärker, dürfen aber nicht schwächer sein als entsprechende Nachbedingungen in Obertypen. Der Grund liegt darin, dass ein Aufrufer der Methode, der nur `T` kennt, sich auf die Erfüllung der Nachbedingungen in `T` verlassen kann, auch wenn die Methode tatsächlich in `U` statt `T` aufgerufen wird. Daher muss eine Nachbedingung in `T`
Untertypen und Vererbung

automatisch erfüllt sein, wenn ihre Entsprechung in $U$ erfüllt ist. Wenn Nachbedingungen in $U$ aus $T$ übernommen werden, können sie mittels Und-Verknüpfungen stärker werden. Lautet die Nachbedingung in $T$ beispielsweise „result > 0“, kann sie in $U$ auch „result > 0 und result > 2“, also „result > 2“ sein.


Die Begründung geht davon aus, dass Objektvariablen nicht durch andere Objekte verändert werden. Ist dies doch der Fall, so müssen Invarianten, die sich auf von außen änderbare Variablen beziehen, in $U$ und $T$ übereinstimmen. Beim Schreiben einer solchen Variablen muss die Invariante vom Client überprüft werden, was dem generellen Konzept widerspricht. Außerdem kann ein Client die Invariante gar nicht überprüfen, wenn in der Bedingung vorkommende Variablen und Methoden nicht öffentlich zugänglich sind. Daher sollen Objektvariablen möglichst nicht durch andere Objekte verändert werden.

**Server-kontrollierter History-Constraint:** Dafür gilt im Prinzip dasselbe wie für Invarianten. Es ist jedoch nicht so einfach, von stärkeren oder schwächeren Bedingungen zu sprechen, da viel von der konkreten Formulierung der Bedingungen abhängt. Einfacher und klarer ist es, die Konsequenzen gegenüberzustellen. Für alle Objektzustände $x$ und $y$ in $U$ und $T$ (wobei ein Objektzustand die Werte aller gemeinsamen Variablen der Objekte von $U$ und $T$ widerspiegelt) soll gelten: Wenn Server-kontrollierte History-Constraints in $T$ ausschließen, dass ein Objekt von $T$ im Zustand $x$ durch Veränderungen im Laufe der Zeit in den Zustand $y$ kommt, dann müssen auch Server-

**Client-kontrollierter History-Constraint:** Dafür gilt im Prinzip dasselbe wie für Vorbedingungen, jedoch bezogen auf Einschränkungen in der Reihenfolge von Methodenaufrufen. Eine Reihenfolge von Methodenaufrufen nennt man Trace, die meist unendlich große Menge aller möglichen (= erlaubten) Traces ist ein *Trace-Set*. Jede entsprechend $T$ erlaubte Aufrufreihenfolge muss auch entsprechend $U$ erlaubt sein. Es ist jedoch möglich, dass $U$ mehr Aufrufreihenfolgen erlaubt als $T$, wodurch die Einschränkungen in $U$ schwächer sind als in $T$. Das durch Client-kontrollierte History-Constraints in $T$ beschriebene Trace-Set muss also eine Teilmenge des durch Client-kontrollierte History-Constraints in $U$ beschriebenen Trace-Sets sein.

Wenn die Clients an ein Objekt Nachrichten in einer durch $T$ erlaubten Reihenfolge schicken, so ist sichergestellt, dass das Objekt vom Typ $U$ die entsprechenden Methoden auch in dieser Reihenfolge ausführen kann. Im Allgemeinen müssen wir bei der Überprüfung Client-kontrollierter History-Constraints die *Menge aller Clients* betrachten, nicht nur einen einzelnen Client, da der Server ja alle Methodenaufrufe nur in eingeschränkter Reihenfolge ausführen kann, nicht nur die Aufrufe, die von einem Client kommen.

Im Einzelnen muss sichergestellt werden, dass

- obige Bedingungen für Untertypbeziehungen eingehalten werden,

- die Implementierungen der Server die Nachbedingungen, Invarianten und Server-kontrollierten History-Constraints erfüllen und nichts voraussetzen, was nicht in Vorbedingungen, Invarianten und History-Constraints (beider Arten) festgelegt ist

- und Clients die Vorbedingungen und Client-kontrollierten History-Constraints der Aufrufe erfüllen und nichts voraussetzen, was nicht durch Nachbedingungen, Invarianten und History-Constraints zugesichert wird.

Es kann sehr aufwendig sein, alle solchen Überprüfungen vorzunehmen. Einfacher geht es, wenn man während der Codeerstellung und bei Änderungen stets an die einzuhaltenden Bedingungen denkt, die Überprüfungen also nebenbei erfolgen. Wichtig ist darauf zu achten, dass die Zusicherungen unmissverständlich formuliert sind. Nach Änderung einer Zusicherung ist die Überprüfung besonders schwierig, und die Änderung einer Zusicherung ohne gleichzeitige Änderung aller betroffenen Programmteile ist eine häufige Fehlerursache in Programmen.

**Faustregel:** Zusicherungen sollen unmissverständlich formuliert sein und während der Programmentwicklung und Wartung ständig bedacht werden.

Betrachten wir ein Beispiel:

```java
public class Set {
    public void insert(int x) {
        // inserts x into set iff not already there;
        // x is in set immediately after invocation
        ...
    }
    public boolean inSet(int x) {
        // returns true if x is in set, otherwise false
        ...
    }
}
```
Die Methode `insert` fügt eine ganze Zahl genau dann („iff“ ist eine übliche Abkürzung für „if and only if“) in ein Objekt von `Set` ein, wenn sie noch nicht in dieser Menge ist. Unmittelbar nach Aufruf der Methode ist die Zahl in jedem Fall in der Menge. Die Methode `inSet` stellt fest, ob eine Zahl in der Menge ist oder nicht. Dieses Verhalten der Objekte von `Set` ist durch die Zusicherungen in den Kommentaren festgelegt. Wenn man den Inhalt dieser Beschreibungen von Methoden genauer betrachtet, sieht man, dass es sich dabei um Nachbedingungen handelt. Da Nachbedingungen festlegen, was sich ein Client vom Aufruf einer Methode erwartet, lesen sich Nachbedingungen oft tatsächlich wie Beschreibungen von Methoden.

Folgende Klasse unterscheidet sich von `Set` nur durch einen zusätzlichen Server-kontrollierten History-Constraint:

```java
public class SetWithoutDelete extends Set {
    // elements in the set always remain in the set
}
```

Eine Zahl, die einmal in der Menge war, soll stets in der Menge bleiben. Offensichtlich ist `SetWithoutDelete` ein Untertyp von `Set`, da nur ein vom Server kontrollierter History-Constraint dazugefügt wird, welcher die zukünftige Entwicklung des Objektzustands gegenüber `Set` einschränkt.

Sehen wir uns eine kurze Codesequenz für einen Client an:

```java
Set s = new Set();
s.insert(41);
doSomething(s);
if (s.inSet(41)) { doSomeOtherThing(s); } else { doSomethingElse(); }
```

Während der Ausführung von `doSomething` könnte `s` verändert werden. Es ist nicht ausgeschlossen, dass `41` dabei aus der Menge gelöscht wird, da die Nachbedingung von `insert` in `Set` ja nur zusichert, dass `41` unmittelbar nach dem Aufruf von `insert` in der Menge ist. Bevor wir die Methode `doSomeOtherThing` aufrufen (von der wir annehmen, dass sie ihren Zweck nur erfüllt, wenn `41` in der Menge ist), stellen wir sicher, dass `41` tatsächlich in der Menge ist. Dies geschieht durch Aufruf von `inSet`.

Verwenden wir ein Objekt von `SetWithoutDelete` anstatt einem von `Set`, ersparen wir uns den Aufruf von `inSet`. Wegen der stärkeren Zusicherung ist `41` sicher in der Menge:
2 Untertypen und Vererbung

```java
SetWithoutDelete s = new SetWithoutDelete();
s.insert(41);
dosomething(s);
doSomeOtherThing(s); // s.inSet(41) returns true
```

Von diesem kleinen Vorteil von SetWithoutDelete darf man sich nicht dazu verleiten lassen, generell starke Einschränkungen in Zusicherungen zu verwenden. Solche Einschränkungen erschweren die Wartung (siehe Abschnitt 2.2.1). Als triviales Beispiel können wir Set leicht um eine Methode delete (zum Löschen einer Zahl aus der Menge) erweitern:

```java
public class SetWithDelete extends Set {
    public void delete(int x) {
        // deletes x from the set if it is there
        ...
    }
}
```


2.2 Ersetzbarkeit und Objektverhalten

2.2.3 Abstrakte Klassen


Nehmen wir als Beispiel folgende Klassenstruktur:

```
public abstract class Polygon {
    public abstract void draw();
    // draw a polygon on the screen
}
```


In Java sieht die abstrakte Klasse etwa so aus:

```
public abstract class Polygon {
    public abstract void draw();
    // draw a polygon on the screen
}
```

2 Untertypen und Vererbung


Die konkrete Klasse Triangle könnte so aussehen:

```java
public class Triangle extends Polygon {
    public void draw() { // draw a triangle on the screen
        ...
    }
}
```

Auch Square und Hexagon müssen die Methode draw implementieren.


**Faustregel:** Es ist empfehlenswert, als Obertypen und Parametertypen hauptsächlich abstrakte Klassen (ohne Implementierungen) und Interfaces zu verwenden.

Vor allem Parametertypen sollen keine Bedingungen an Argumente stellen, die nicht benötigt werden. Konkrete Klassen legen aber oft zahlreiche Bedingungen in Form von Zusicherungen und Methoden in der Schnittstelle fest. Diesen Konflikt kann man leicht lösen, indem man für die Typen der Parameter nur abstrakte Klassen verwendet. Es ist ja leicht, zu jeder konkreten Klasse eine oder mehrere abstrakte Klassen als Oberklassen zu schreiben, die die benötigten Bedingungen möglichst genau angeben. Damit werden unnötige Abhängigkeiten vermieden.
2.3 Vererbung versus Ersetzbarkeit


2.3.1 Reale Welt, Vererbung, Ersetzbarkeit

In der objektorientierten Softwareentwicklung begegnen wir zumindest drei verschiedenen Arten von Beziehungen zwischen Klassen [22]:

Untertypen: Diese Beziehung, die auf dem Ersetzbarkeitsprinzip beruht, haben wir bereits untersucht.


Beziehungen in der realen Welt verlieren stark an Bedeutung, sobald genug Details bekannt sind, um sie zu Untertyp- und Vererbungsbeziehungen weiterzuentwickeln. Deshalb konzentrieren wir uns hier nur auf die Unterscheidung zwischen Untertypen und Vererbung.

In Java und ähnlichen objektorientierten Sprachen setzen Untertypen Vererbung voraus und sind derart eingeschränkt, dass die vom Compiler überprüfbaren Bedingungen für Untertypen erfüllt sind. Das heißt, als wesentliches Unterscheidungskriterium verbleibt nur die Frage, ob Zusicherungen zwischen Unter- und Oberklasse kompatibel sind. Diese Unterscheidung können nur Personen treffen, die Bedeutungen von Namen und Kommentaren verstehen. In allen anderen Kriterien sind in Java reine Vererbungs- von Untertypbeziehungen nicht unterscheidbar.

Man kann leicht erkennen, ob reine Vererbungs- oder Untertypbeziehungen angestrebt werden. Betrachten wir dazu ein Beispiel:

![Diagramm der Klassenrelationen](image)

Das Ziel der reinen Vererbung ist es, so viele Teile der Oberklasse wie möglich direkt in der Unterkasse wiederverwenden. Angenommen, die Implementierungen von `LargeSet` und `Bag` zeigen so starke Ähnlichkeiten, dass sich die Wiederverwendung von Programmenteilen lohnt. In diesem Fall erbt `Bag` große Teile der Implementierung von `LargeSet`. Für diese Entscheidung ist nur der pragmatische Gesichtspunkt, dass sich `Bag` einfacher aus `LargeSet` ableiten lässt als umgekehrt, ausschlaggebend. Für `SmallSet` wurde eine von `LargeSet` unabhängige Implementierung gewählt, das bei kleinen Mengen effizienter ist als `LargeSet`.

Wenn wir uns von Konzepten und Typen leiten lassen, schaut die Hierarchie anders aus. Wir führen eine zusätzliche (abstrakte) Klasse `Set` ein, da die Typen von `LargeSet` und `SmallSet` dieselbe Bedeutung haben sollen. Wir wollen im Programmcode nur selten zwischen `LargeSet` und `SmallSet` unterscheiden. `Bag` und `LargeSet` stehen in keinem Verhältnis zueinander, da die Methoden für das Hinzufügen von Elementen einander ausschließende Bedeutungen haben, obwohl `Set` und `Bag` dieselbe
Signatur haben können. Einander ausschließende Bedeutungen kommen daher, dass ein Objekt von \texttt{Set} höchstens ein Vorkommen eines Objekts enthalten kann, während in einem Objekt von \texttt{Bag} mehrere Vorkommen erlaubt sind. Entsprechend darf eine Methode nur dann ein Element zu einem Objekt von \texttt{Set} hinzufügen, wenn das Element noch nicht vorkommt, während die Methode zum Hinzufügen in ein Objekt von \texttt{Bag} jedes gewünschte Element akzeptieren muss.

Obiges Beispiel demonstriert unterschiedliche Argumentationen für die reine Vererbung im Vergleich zu Untertypbeziehungen. Die Unterschiede zwischen den Argumentationen sind wichtiger als jene zwischen den Hierarchien, da die Hierarchien selbst letztendlich von Details und beabsichtigten Verwendungen abhängen.

### 2.3.2 Vererbung und Codewiederverwendung


| Faustregel: | Wiederverwendung durch das Ersetzbarkeitsprinzip ist wesentlich wichtiger als direkte Wiederverwendung durch Vererbung. |
2 Untertypen und Vererbung

Der allgemeine Ratschlag ist daher ganz klar: Ein wichtiges Ziel ist die Entwicklung geeigneter Untertypbeziehungen. Vererbung ist ein Mittel zum Zweck. Die soll sich Untertypbeziehungen unterordnen. Im Allgemeinen soll es keine Vererbungsbeziehung geben, die nicht auch eine Untertypbeziehung ist, bei der also alle Zusicherungen kompatibel sind.

Wie die Erfahrung zeigt, vergessen Programmieranfänger allzu leicht das Ersetzbarkeitsprinzip und konzentrieren sich ganz und gar auf direkte Codewiederverwendung durch Vererbung. Daher soll noch einmal klar gesagt werden, dass die Menge des aus einer Oberklasse ererbten Codes für die Codewiederverwendung nur sehr geringe Bedeutung hat. Viel wichtiger für die Wiederverwendung ist das Bestehen von Untertypbeziehungen.


Direkte Codewiederverwendung durch Vererbung erspart uns nicht nur das wiederholte Schreiben desselben Codes, sondern hat auch Auswirkungen auf die Wartbarkeit. Wenn ein Programmteil nur einmal statt mehrmals implementiert ist, brauchen Änderungen nur an einer einzigen Stelle vorgenommen werden, wirken sich aber auf alle Programmteile aus, in denen der veränderte Code verwendet wird. Nicht selten muss man alle gleichen oder ähnlichen Programmteile gleichzeitig ändern, wenn sich die Anforderungen ändern. Gerade dabei kann Vererbung sehr hilfreich sein.

Faustregel: Auch reine Vererbung kann sich positiv auf die Wartbarkeit auswirken.

Es kommt vor, dass nicht alle solchen Programmteile geändert werden sollen, sondern nur einer oder einige wenige. Dann ist es nicht möglich, eine Methode unverändert zu erben. Glücklicherweise ist es in diesem Fall sehr einfach, eine geerbte Methode durch eine neue Methode zu überschreiben. In Sprachen wie Java ist es sogar möglich, die Methode zu überschreiben und trotzdem noch auf die überschriebene Methode in der Oberklasse zuzugreifen. Ein Beispiel soll das demonstrieren:
public class A {
    public void foo() { ... }
}
public class B extends A {
    private boolean b;
    public void foo() {
        if (b) { ... }
        else { super.foo(); }
    }
}


In komplizierten Situationen ist geschickte Faktorisierung notwendig, um direkte Codewiederverwendung zu erreichen:

public class A {
    public void foo() {
        if (...) { ... }
        else { ...; x = 1; ... }
    }
}
public class B extends A {
    public void foo() {
        if (...) { ... }
        else { ...; x = 2; ... }
    }
}

Die Methode foo muss gänzlich neu geschrieben werden, obwohl der Unterschied minimal ist. Man muss ja immer eine ganze Methode überschreiben, nicht nur eine Anweisung der Methode. Eine Aufspaltung von foo in mehrere Methoden kann helfen:

public class A {
    public void foo() {
        if (...) { ... }
        else { fooX(); }
    }
    void fooX() { ...; x = 1; ... }
}
Untertypen und Vererbung

```java
public class B extends A {
    void fooX() { ...; x = 2; ... }
}
```


Unterschiede zwischen Unter- und Oberklassen kann man auch durch zusätzliche Parameter beschreiben und nach außen sichtbare Methoden nur zum Setzen der Parameter verwenden:

```java
public class A {
    public void foo() { fooY(1); }
    void fooY (int y) {
        if (...) { ... }
        else { ...; x = y; ... }
    }
}
public class B extends A {
    public void foo() { fooY(2); }
}
```


In der Sprache Sather gibt es zwei komplett voneinander getrennte Hierarchien auf Klassen: die Vererbungshierarchie für direkte Codewiederverwendung und die Typhierarchie für indirekte Codewiederverwendung. Da die Vererbungshierarchie nicht den Einschränkungen des Ersetzbarkeitsprinzips unterliegt, gibt es zahlreiche Möglichkeiten der Codeveränderung bei der Vererbung, z.B. Einschränkungen der Sichtbarkeit von Methoden und Variablen (Kommentare beginnen mit \texttt{--}):

```sather
class A is ~ Definition einer Klasse A
  ...; ~ Routinen und Variablen von A
end;

class B is ~ Definition einer Klasse B
  include A ~ B erbt von A
    a->b, ~ wobei a aus A in B b heisst,
    c->, ~ c aus A in B nicht sichtbar
    d->private d; ~ und d aus A in B private ist
  ...; ~ Routinen und Variablen von B
end;
```

Neben den konkreten Klassen gibt es in Sather (wie in Java) auch abstrakte Klassen. Deren Namen müssen mit \$ beginnen:

```sather
abstract class $X is ...; end;
```

Abstrakte Klassen spielen in Sather eine ganz besondere Rolle, da nur sie als Obertypen in Untertypdeklarationen verwendbar sind:

```sather
abstract class $Y <$X is ...; end;
  -- $Y ist Untertyp von $X
class C <$Y, $Z is ...; end;
  -- C ist Untertyp von $Y und $Z
```

Damit sind Objekte von \texttt{C} überall verwendbar, wo Objekte von \texttt{$X, $Y oder $Z} erwartet werden. Anders als \texttt{extends in Java bedeutet} \texttt{< in Sather jedoch nicht, dass die Unterklass}e von der Oberklasse erbt, sondern nur, dass der Compiler die statisch überprüfbaren Bedingungen für eine Untertypbeziehung prüft und dynamisches Binden ermöglicht. Für Vererbung ist eine separate include-Klausel notwendig.

99
2 Untertypen und Vererbung

2.3.3 Fehlervermeidung

Im Zusammenhang mit Untertypen und Vererbung passieren immer wieder schwere Programmierfehler, die sich erst später nach Programmänderungen äußern. Häufig ist Programmierer(inne)n gar nicht bewusst, dass sie etwas falsch machen. Die Fehler fallen ja bei einfachem Testen nicht gleich auf und werden, wenn überhaupt, erst viel später entdeckt.

Der Kardinalfehler besteht darin, eine Untertypbeziehung anzunehmen, wo keine besteht. Nachdem der Compiler (oder das Laufzeitsystem) garantiert, dass die Bedingungen für strukturelle Untertypbeziehungen eingehalten werden, fallen Fehler in der Struktur fast immer rasch auf.


Die Einhaltung folgender Regel sollte fast alle Probleme mit falschen Untertypbeziehungen vermeiden:

**Faustregel:** Man muss beim Programmieren stets prüfen, ob sich ein Objekt tatsächlich immer so verhält, wie in jedem Typ des Objekts beschrieben.

Leider enthält diese Regel einige Schwierigkeiten, nämlich die Wörter „stets“, „immer“ und „jedem“. Man muss also sehr häufig recht komplexe und umfangreiche Prüfungen vornehmen. Das geht nur, wenn man während des Programmierens ein recht genaues Modell des Objektverhaltens und der Verhaltensbeschreibungen (also der Typen) im Kopf hat, und das für alle Objekte und Typen, mit denen man sich gerade beschäftigt. Dazu ist höchste Konzentration erforderlich. Es reicht ein Moment der Unachtsamkeit oder Ermüdung, und schon ist ein schwerwiegender, aber kaum zu entdeckender Fehler eingebaut. Man muss sich sehr tief in die Verhaltensbeschreibungen einarbeiten um keine Details zu übersehen. Es ist viel Übung erforderlich, bis man sich die notwendigen Überprüfungen so gut eingeprägt hat, dass sie im Unterbewusstsein während des Programmierens quasi automatisch ablaufen. Zusammengefasst kann man folgende Ratschläge geben:
2.3 Vererbung versus Ersetzbarkeit

- Gründlich in vorgegebene Verhaltensbeschreibungen einarbeiten.
- Auf eine Umgebung achten, in der man sich gut konzentrieren kann.
- Bei Ermüdung unverzüglich eine Pause einlegen.
- Viel selbst programmieren, auch komplizierte Aufgaben selbst lösen.


**Faustregel:** Durchdachte Zusicherungen sind bei der Überprüfung des Objektverhaltens sehr hilfreich.


Manchmal ist aber gerade die Konzentration auf Nebensächlichkeiten schuld an falschen Untertypbeziehungen. Wenn man etwa zu sehr an einfache Änderbarkeit durch Vererbung denkt, übersieht man vielleicht wesentliche Kriterien für Untertypbeziehungen. Man braucht schon etwas Erfahrung, um sich stets auf das Wichtigste zu konzentrieren.

### 2.4 Exkurs: Klassen und Vererbung in Java


#### 2.4.1 Klassen in Java

In Java wird streng zwischen Groß- und Kleinschreibung unterschieden, A und a sind daher verschieden. Namen von Klassen werden per Konvention mit großen Anfangsbuchstaben geschrieben, Namen von Konstanten dagegen oft nur mit Großbuchstaben und alle anderen Namen mit kleinen
2.4 Exkurs: Klassen und Vererbung in Java


Klassen können mehrere explizit definierte Konstruktoren enthalten:

```java
public class Circle {
    private int r;
    public Circle(int r) { this.r = r; } // 1
    public Circle(Circle c) { this.r = c.r; } // 2
    public Circle() { r = 1; } // 3
    ...
}
```

Die Klasse `Circle` hat drei verschiedene Konstruktoren, die sich in der Anzahl oder in den Typen der formalen Parameter unterscheiden. Das ist ein typischer Fall von Überladen.

Beim Erzeugen eines neuen Objekts werden dem Konstruktor Argumente übergeben. Anhand der Anzahl und den deklarierten Typen der Argumente wird der geeignete Konstruktor gewählt:

```java
Circle a = new Circle(2); // Konstruktor 1
Circle b = new Circle(a); // Konstruktor 2
Circle c = new Circle(); // Konstruktor 3
```

In zwei Konstruktoren haben wir `this` wie den Namen einer Variable verwendet. Tatsächlich bezeichnet `this` immer das aktuelle Objekt der Klasse. In Konstruktoren ist dies das Objekt, das gerade erzeugt wurde. Im ersten Konstruktor benötigen wir `this`, um die Variable `r` im neuen Objekt, das ist `this.r`, vom formalen Parameter `r` des Konstruktors zu unterscheiden. Wie in diesem Beispiel können formale Parameter (oder lokale Variablen) Variablen im aktuellen Objekt der Klasse verdecken, die denselben Namen haben. Über `this` kann man dennoch auf die Objektvariablen zugreifen. Wie im zweiten Konstruktor gezeigt, kann man `this` immer verwenden, auch wenn es gar nicht nötig ist. Außerdem benötigt man `this` bei der Verwendung des aktuellen Objekts der Klasse als Argument. Zum Beispiel liefert `new Circle(this)` innerhalb der Klasse `Circle` eine Kopie des aktuellen Objekts.
Falls in einer Klasse kein Konstruktor explizit definiert ist, enthält die Klasse automatisch einen Defaultkonstruktor:

```java
public Klassenname() { super(); }
```


**Objektvariablen**, auch *Instanzvariablen* genannt, sind Variablen, die zu Objekten (= Instanzen einer Klasse) gehören. Wenn in der Deklaration einer Objektvariablen keine Initialisierung angegeben ist, wird je nach Typ eine Defaultinitialisierung mit 0 bzw. 0.0 oder `null` vorgenommen; für lokale Variablen erfolgt dagegen keine Defaultinitialisierung. Jedes Objekt der Klasse enthält eigene Kopien der Objektvariablen.


Eine Methode, die durch `static` gekennzeichnet wird, gehört ebenfalls zur Klasse und nicht zu einem Objekt. Ein Beispiel ist die Methode `main`:

```java
static void main (String[] args) { ... }
```

Konstruktoren machen es uns leicht, komplexe Initialisierungen von ObjektvARIABLEN vorzunehmen. Static-Initializers bieten eine derartige Möglichkeit auch für KLASSENvariablen:

static { ... }


Das Gegenteil von Konstruktoren sind Destruktoren, die festlegen, was unmittelbar vor der endgültigen Zerstörung eines Objekts gemacht werden soll. In Java sind Destruktoren Methoden mit Namen finalize, die keine formalen Parameter haben und kein Ergebnis zurückgeben. Wir werden nicht näher auf Destruktoren eingehen, da sie auf Grund einiger Eigenschaften von Java kaum sinnvoll einsetzbar sind.

Geschachtelte Klassen sind innerhalb anderer Klassen definiert. Sie können überall definiert sein wo Variablen deklariert werden dürfen und kommen vorwiegend wegen folgender Eigenschaft zum Einsatz: Innerhalb geschachtelter Klassen kann man private Variablen und Methoden aus der Umgebung verwenden. Es gibt zwei Arten geschachtelter Klassen:

**Statische geschachtelte Klassen:** Diese werden mit dem Schlüsselwort static versehen und gehören zur umschließenden Klasse selbst:

```java
class EnclosingClass {
    ...
    static class StaticNestedClass { ... }
    ...
}
```

Wie statische Methoden dürfen statische geschachtelte Klassen nur auf Klassenvariablen der umschließenden Klasse zugreifen und statische Methoden der umschließenden Klasse aufrufen. Dabei kann
untypen und Vererbung


**Innere Klassen**: Jede innere Klasse wird ohne `static`-Modifier deklariert und gehört zu einem *Objekt* der umschließenden Klasse:

```java
class EnclosingClass {
    ...
    class InnerClass { ... }
    ...
}
```

Objektvariablen und nichtstatische Methoden aus der umschließenden Klasse (`EnclosingClass`) können in `InnerClass` uneingeschränkt verwendet werden, auch private. Innere Klassen dürfen jedoch keine statischen Methoden und keine statischen geschachtelten Klassen enthalten, da diese von einem Objekt der äußeren Klasse abhängen würden und dadurch nicht mehr statisch wären. Ein Objekt der inneren Klasse wird z.B. durch `a.new InnerClass()` erzeugt, wobei `a` eine Variable vom Typ `EnclosingClass` ist.


2.4 Exkurs: Klassen und Vererbung in Java

2.4.2 Vererbung und Interfaces in Java


Beim Erzeugen eines neuen Objekts wird nicht nur ein Konstruktor der entsprechenden Klasse aufgerufen, sondern auch mindestens ein Konstruktor jeder Oberklasse. Wenn die erste Anweisung in einem Konstruktor „super(a,b,c);“ lautet, wird in der Oberklasse, von der direkt geerbt wird, ein entsprechender Konstruktor mit den Argumenten a, b und c aufgerufen. Sonst wird automatisch ein Konstruktor der Oberklasse ohne Argumente aufgerufen. Eine Ausnahme stellen Konstruktoren dar, deren erste Zeile beispielsweise „this(a,b,c);“ lautet. Solche Konstruktoren rufen einen Konstruktor der eigenen Klasse mit den angegebenen Argumenten auf. Im Endeffekt werden auch in diesem Fall Konstruktoren aller Oberklassen aufgerufen, da irgendein Konstruktor nicht mehr mit this beginnt. Sonst hätten wir eine Endlosrekursion.


Es soll noch einmal betont werden, dass eine Methode der Unterklasse
eine der Oberklasse nur überschreibt, wenn Name, Parameteranzahl und
Parametertypen gleich sind. Sonst sind die Methoden überladen, das heißt,
in der Unterklasse existieren beide Methoden gleichzeitig. Die deklariert-
ten Typen der übergebenen Argumente entscheiden, welche überladene
Methode aufgerufen wird.

In Java gibt es eine Möglichkeit zu verhindern, dass eine Methode in
einer Unterklasse überschrieben wird. Das Überschreiben ist unmöglich,
wenn die Methode mit dem Schlüsselwort final definiert wurde. Da ein
Überschreiben nicht möglich ist, werden solche Methoden durch statisches
Binden aufgerufen. Dadurch erfolgt der Aufruf (meist unmerklich) schnel-
ler als durch dynamisches Binden. Trotzdem soll man final im Nor-
malfall eher nicht verwenden, da das Verbieten des Überschreibens die
Wartbarkeit vermindern kann. Nicht überschreibbare Methoden sind für
spezielle Fälle vorgesehen, in denen man das Überschreiben aus Sicher-
heitsgründen, z. B. zur Vermeidung der Umgehung einer Passwortabfrage,
verbieten will.

**Faustregel:** Methoden sollen nur in Spezialfällen als final
deklariert sein.

Man kann auch ganze Klassen mit dem Schlüsselwort final versehen.
Solche Klassen haben keine Unterklassen. Dadurch ist es auch nicht mög-
lich, die Methoden der Klassen zu überschreiben. In manchen objektori-
entierten Programmierstilen verwendet man solche Klassen um klarzustel-
len, dass das Verhalten der Objekte durch die Implementierung festgelegt
ist. Clients können sich auf alle Implementierungsdetails verlassen, oh-
ne auf mögliche Ersetzungen Rücksicht nehmen zu müssen. Änderungen
der Klassen können aber aufwendig sein, da alle Clients zu überprüfen
und gegebenenfalls ebenfalls zu ändern sind. Abstrakte Klassen dürfen
natürlich nicht final sein. Die Nachteile der Verwendung von final
Klassen sind durch den konsequenten Einsatz von abstrakten Klassen und
Interfaces als Typen von formalen Parametern und Variablen vermeidbar.
In diesem Fall kann die konsequente Verwendung von final Klassen
für alle nicht-abstrakten Klassen vorteilhaft sein. Damit erhält man einen
Programmierstil ähnlich dem von Sather (siehe Abschnitt 2.3.2).

Interfaces sind in Java eingeschränkte abstrakte Klassen, die (im Gegen-
zug zu den Einschränkungen) Mehrfachvererbung unterstützen. Interfaces
unterscheiden sich von normalen abstrakten Klassen wie folgt:
• Beginnen mit interface statt abstract class.
• Alle Methoden sind abstrakt, Modifier abstract nicht notwendig.
• Enthält außer static-final-Konstanten keine Variablen.
• In allen anderen Fällen sind die Modifier static, final, private und protected verboten.
• Methoden und Konstanten sind immer public, auch ohne Modifier.
• Nach extends können mehrere, durch Komma getrennte Namen von Interfaces, aber nicht von Klassen stehen (Mehrfachvererbung).

Auch Klassen können von mehreren Interfaces erben:

```java
public interface X {
    static final double PI = 3.14159;
    double fooX();
}

public interface Y {
    double fooY();
}

public interface Z extends X, Y {
    double fooZ();
}

public class A implements X, Y {
    protected double factor = 2.0;
    public double foo() { return PI; }
    public double fooX() { return factor * PI; }
    public double fooY() { return factor * fooX(); }
}

public class B extends A implements Z {
    public double fooY() { return 3.3 * foo(); }
    public double fooZ() { return factor / fooX(); }
}
```
2 Untertypen und Vererbung

Interface Z erbt von X und Y. Somit enthält Z die Konstante PI sowie die Methoden fooX, fooY und fooZ. Die Klasse A erbt ebenfalls von X und Y. Interfaces, von denen eine Klasse erbt, stehen nach implements um anzudeuten, dass die in den Interfaces deklarierten Methoden in der Klasse zu implementieren sind, beziehungsweise die Klasse die durch die Interfaces spezifizierten Schnittstellen implementiert. In einer Klassendefinition kann nach extends nur eine Klasse stehen.

Interfaces sind als Typen verwendbar. In dieser Hinsicht unterscheiden sie sich nicht von Klassen. Wie für abstrakte Klassen ohne Implementierungen gilt die Faustregel, dass Interfaces stabiler sind als Klassen mit Implementierungen. Durch Mehrfachvererbung sind sie oft flexibler einsetzbar als abstrakte Klassen. Daher sollten Interfaces immer verwendet werden, wo dies möglich ist, das heißt, wo die oben genannten Einschränkungen zu keinen Nachteilen führen.

**Faustregel:** Interfaces sind abstrakten Klassen vorzuziehen.


### 2.4.3 Pakete und Zugriffskontrolle in Java

Jede compilierte Java-Klasse wird in einer eigenen Datei gespeichert. Der Dateiname entspricht dem Namen der Klasse mit der Endung .class. Das Verzeichnis, das diese Datei enthält, entspricht dem Paket, zu dem die Klasse gehört; das ist ein Namensraum. Der Name des Verzeichnisses

Namen im Quellcode müssen den Namen des Paketes enthalten, in dem die Namen definiert sind, außer wenn sie im selben Paket definiert sind. Diese Namen sind relativ zu einer Basis, dem Class-Path, der vom System vorgegeben ist und über Parameter des Compilers bzw. Interpreters geändert werden kann. Nehmen wir an, wir wollen die statische Methode foo in einer Klasse AClass aufrufen, deren Quellcode in der Datei

    myclasses/examples/test/AClass.java

steht. Dann lautet der Aufruf folgendermaßen:

    myclasses.examples.test.AClass.foo();

Solche langen Namen bedeuten einen hohen Schreibaufwand und sind auch nur schwer lesbar. Daher bietet Java eine Möglichkeit, Klassen oder ganze Dateien zu importieren. Enthält der Quellcode zum Beispiel die Zeile

    import myclasses.examples.test;

dann kann man foo durch „test.AClass.foo();“ aufrufen, da der Paketname test lokal bekannt ist. Enthält der Quellcode sogar die Zeile

    import myclasses.examples.test.AClass;

kann man foo noch einfacher durch „AClass.foo();“ aufrufen. Häufig möchte man alle Klassen in einem Paket auf einmal importieren. Das geht beispielsweise dadurch:

    import myclasses.examples.test.*;

Auch nach dieser Zeile ist „AClass.foo();“ direkt aufrufbar.

Beliebig viele solche Zeilen mit dem Schlüsselwort import dürfen am Anfang einer Datei mit Quellcode stehen, sonst aber nirgends. Vor diesen Zeilen darf höchstens eine einzelne Zeile

    package paketName;
2 Untertypen und Vererbung

stehen, wobei \texttt{paketName} den Namen und Pfad des Paketes bezeichnet, zu dem die Klasse in der Quelldatei gehört. Ist eine solche Zeile in der Quelldatei vorhanden, muss der Aufruf von javac zur Compilation der Datei oder java zur Ausführung der übersetzten Datei im Dateinamen den Pfad enthalten, der in \texttt{paketName} vorgegeben ist (wobei Punkte in \texttt{paketName} je nach Betriebssystem durch / oder \ ersetzt sind). Wenn die Quelldatei oder compilierte Datei in einem anderen Verzeichnis steht, lässt sie sich nicht compilieren beziehungsweise verwenden. Die Zeile mit dem Schlüsselwort \texttt{package} stellt also – zumindest zu einem gewissen Grad – sicher, dass die Datei nicht einfach aus dem Kontext gerissen und in einem anderen Paket verwendet wird.

Nun kommen wir zur Sichtbarkeit von Namen. Generell sind alle Einheiten wie Klassen, Variablen, Methoden, etc. in dem Bereich (Scope), in dem sie definiert wurden, sichtbar und verwendbar, zumindest, wenn sie nicht durch eine andere Einheit mit demselben Namen verdeckt sind. Einheiten, die mit dem Schlüsselwort \texttt{private} definiert wurden, sind sonst nirgends sichtbar. Sie werden auch nicht vererbt.


Einheiten, die mit dem Schlüsselwort \texttt{public} definiert wurden, sind dagegen überall sichtbar und werden vererbt. Man kann

\begin{verbatim}
myclassesexamples.test.AClass.foo();
\end{verbatim}
aufrieren, wenn sowohl die Klasse \texttt{AClass} als auch die statische Methode \texttt{foo} mit dem vorangestellten Schlüsselwort \texttt{public} definiert wurden. In allen anderen Fällen darf man \texttt{foo} nicht aufrufen.

Neben diesen beiden Extremfällen gibt es noch zwei weitere Möglichkeiten zur Steuerung der Sichtbarkeit. Bei diesen Möglichkeiten sind Ein-

Wir fassen diese Sichtbarkeitsregeln in einer Tabelle zusammen:

<table>
<thead>
<tr>
<th></th>
<th>public</th>
<th>protected</th>
<th>—</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>sichtbar im selben Paket</td>
<td>ja</td>
<td>ja</td>
<td>ja</td>
<td>nein</td>
</tr>
<tr>
<td>sichtbar in anderem Paket</td>
<td>ja</td>
<td>nein</td>
<td>nein</td>
<td>nein</td>
</tr>
<tr>
<td>ererbbar im selben Paket</td>
<td>ja</td>
<td>ja</td>
<td>ja</td>
<td>nein</td>
</tr>
<tr>
<td>ererbbar in anderem Paket</td>
<td>ja</td>
<td>ja</td>
<td>nein</td>
<td>nein</td>
</tr>
</tbody>
</table>

Andere Sichtbarkeitseigenschaften als die in der Tabelle angeführten werden von Java derzeit nicht unterstützt.

Für weniger geübte Java-Programmierer(innen) ist es gar nicht leicht, stets die richtigen Sichtbarkeitseigenschaften zu wählen. Hier sind einige Ratschläge, die diese Wahl erleichtern sollen:

- Alle Methoden, Konstruktoren und Konstanten (und in ganz seltenen Fällen auch Variablen – ist aber verpönt), die man bei der Verwendung der Klasse oder von Objekten der Klasse benötigt, sollen `public` sein.

- Man verwendet `private` für alles, was nur innerhalb der Klasse verwendet werden soll und außerhalb der Klasse nicht verständlich zu sein braucht. Die Bedeutung von Variablen ist außerhalb der Klasse in der Regel ohnehin nicht verständlich, `private` daher ideal.


- In einem Paket sollen alle Klassen stehen, die eng zusammenarbeiten. Methoden und Konstruktoren (und ganz seltenen Variablen),
2 Untertypen und Vererbung


**Faustregel:** Fast alle Variablen sollten private sein.


**Faustregel:** Methoden zum direkten Setzen bzw. Abfragen von Variablenwerten sind im Normalfall zu vermeiden.

Wenn unklar ist, wo etwas sichtbar sein soll, verwendet man zu Beginn die am stärksten eingeschränkte Variante. Erst wenn sich herausstellt, dass eine weniger restriktive Variante nötig ist, erlaubt man weitere Zugriffe. Es ist viel einfacher, Restriktionen der Sichtbarkeit aufzuheben, als neue Einschränkungen einzuführen. Aber auch beim Ausweiten der Sichtbarkeit können sich Probleme ergeben.

(für Interessierte)

Java bietet eine weitere Möglichkeit, die Sichtbarkeit von Klassen gezielt einzuschränken: Da jede übersetzte Java-Klasse in einer eigenen Datei steht, kann über die Zugriffsrechte des Dateisystems geregelt werden, wer darauf zugreifen darf. Leider sind diese Kontrollmöglichkeiten durch ganz unterschiedliche Dateisysteme nicht portabel und werden, auch wegen der umständlichen Realisierung, kaum verwendet.

2.5 Wiederholungsfragen

1. In welchen Formen (mindestens zwei) kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?

2. Wann ist ein struktureller Typ Untertyp eines anderen strukturellen Typs? Welche Regeln müssen dabei erfüllt sein? Welche zusätzlichen Bedingungen gelten für nominale Typen bzw. in Java? (Hinweis: Häufige Prüfungsfrage!)

3. Sind die in Punkt 2 angeschnittenen Bedingungen (sowie das, was Compiler prüfen können) hinreichend, damit das Ersetzbarkeitsprinzip erfüllt ist? Wenn nicht, was muss noch beachtet werden?

4. Was bedeutet Ko-, Kontra- und Invarianz, und für welche Typen in einer Klasse trifft welcher dieser Begriffe zu?

5. Was sind binäre Methoden, und welche Schwierigkeiten verursachen sie hinsichtlich der Ersetzbarkeit?

6. Wie soll man Typen formaler Parameter wählen um gute Wartbarkeit zu erzielen?

7. Warum ist dynamisches Binden gegenüber switch- oder geschachtelten if-Anweisungen zu bevorzugen?
8. Welche Rolle spielt dynamisches Binden für die Ersetzbarkeit und Wartbarkeit?

9. Welche Arten von Zusicherungen werden unterschieden, und wer ist für die Einhaltung verantwortlich? (Hinweis: Häufige Prüfungsfrage!)

10. Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? (Hinweis: Häufige Prüfungsfrage!)

11. Warum sollen Signaturen und Typen stabil bleiben? Wo ist Stabilität besonders wichtig?

12. Was ist im Zusammenhang mit allgemein zugänglichen (= möglicherweise nicht nur innerhalb des Objekts geschriebenen) Variablen und Invarianten zu beachten?

13. Wie genau sollen Zusicherungen spezifiziert sein?

14. Wozu dienen abstrakte Klassen und abstrakte Methoden? Wo und wie soll man abstrakte Klassen einsetzen?

15. Ist Vererbung dasselbe wie das Ersetzbarkeitsprinzip? Wenn Nein, wo liegen die Unterschiede?


17. Was bedeuten folgende Begriffe in Java?
   - Objektvariable, Klassenvariable, statische Methode
   - Static-Initializer
   - geschachtelte und innere Klasse
   - final Klasse und final Methode
   - Paket

18. Wo gibt es in Java Mehrfachvererbung, wo Einfachvererbung?


20. Wozu benötigt man eine package-Anweisung?
21. Welche Möglichkeiten zur Spezifikation der Sichtbarkeit gibt es in Java, und wann soll man welche Möglichkeit wählen?

22. Wodurch unterscheiden sich Interfaces in Java von abstrakten Klassen? Wann soll man Interfaces verwenden? Wann sind abstrakte Klassen besser geeignet?
2 Untertypen und Vererbung
3 Generizität und Ad-hoc-Polymorphismus


3.1 Generizität


Generizität ist ein statischer Mechanismus, der von Java erst ab Version 1.5 unterstützt wird. Dynamisches Binden wie bei Untertypen ist nicht nötig. Dieses wichtige Unterscheidungsmerkmal zu Untertypen verspricht einen effizienten Einsatz in vielen Bereichen, schränkt uns beim Programmieren aber manchmal auch auf unerwartete Weise ein.

3.1.1 Wozu Generizität?

An Stelle expliziter Typen werden im Programm Typparameter verwendet. Das sind einfach nur Namen, die später durch Typen ersetzt werden. Anhand eines Beispiels wollen wir zeigen, dass eine Verwendung von Typparametern und die spätere Ersetzung durch Typen sinnvoll ist:

Beispiel. Wir entwickeln Programmcode für Listen, in denen alle Listen-Elemente vom Typ String sind. Bald stellt sich jedoch heraus, dass wir
3 Generizität und Ad-hoc-Polymorphismus


3.1 Generizität

3.1.2 Einfache Generizität in Java

Generische Klassen und Interfaces haben ein oder mehrere Typparameter, die in spitze Klammern (durch Beistriche voneinander getrennt) deklariert sind. Innerhalb der Klassen und Interfaces sind diese Typparameter beinahe wie normale Referenztypen verwendbar. Das erste Beispiel in Java verwendet zwei generische Interfaces mit je einem Typparameter $A$:

```java
public interface Collection<A> {
    void add(A elem);  // add elem to collection
    Iterator<A> iterator(); // create new iterator
}

public interface Iterator<A> {
    A next();         // get the next element
    boolean hasNext();  // further elements?
}
```

Mit diesen Definitionen bezeichnet Collection<String> ein Interface, das durch Ersetzung aller Vorkommen des Typparameters A im Rumpf von Collection<A> generiert wird. So enthält Collection<String> die entsprechenden generierten Methoden void add(String elem) und Iterator<String> iterator(), wobei Iterator<String> die Methoden String next() und boolean hasNext() enthält. Der Typparameter kann durch Referenztypen ersetzt werden, aber nicht durch elementare Typen wie int, char oder boolean.


Das Programmrstück in Abbildung 3.2 zeigt den Umgang mit generischen Klassen. An ListTest fällt auf, dass statt einfacher Werte von int Objekte der Standardklasse Integer verwendet werden müssen, da gewöhnliche Zahlen keine Referenzobjekte sind. In Java gibt es zu jedem elementaren Typ wie int, char oder boolean einen Referenztyp wie Integer, Character oder Boolean, weil in einigen Sprachkonstrukten nur Referenztypen erlaubt sind. Sie bieten dieselbe Funktionalität wie
public class List<A> implements Collection<A> {
    private class Node {
        private A elem; // element in node
        private Node next = null; // next node in list
        private Node(A elem) {
            this.elem = elem;
        }
    }
    private Node head = null; // first node of list
    private Node tail = null; // last list node

    private class ListIter implements Iterator<A> {
        private Node p = head; // iterator position

        public A next() { // get next list element
            if (p == null)
                return null;
            A elem = p.elem;
            p = p.next;
            return elem;
        }
        public boolean hasNext() { // more elements?
            return p != null;
        }
    }
    public void add(A x) { // add element to list
        if (head == null)
            tail = head = new Node(x);
        else
            tail = tail.next = new Node(x);
    }
    public Iterator<A> iterator() { // new list iter.
        return new ListIter();
    }
}

Abbildung 3.1: Generische Listenklasse
class ListTest {
    public static void main(String[] args) {
        List<Integer> xs = new List<Integer>();
        xs.add(new Integer(0));  // oder xs.add(0);
        Integer x = xs.iterator().next();

        List<String> ys = new List<String>();
        ys.add("zero");
        String y = ys.iterator().next();

        List<List<Integer>> zs =  
                new List<List<Integer>>();
        zs.add(xs);
        // zs.add(ys);  !! Compiler meldet Fehler !!
        List<Integer> z = zs.iterator().next();
    }
}

Abbildung 3.2: Verwendung generischer Listen

elementare Typen. Ein Nachteil ist der im Vergleich zu elementaren Werten weniger effizientere Umgang mit Objekten.

Java unterstützt Autoboxing und Autounboxing. Dabei erfolgt die Umwandlung zwischen Typen wie `int` und `Integer` bei Bedarf automatisch in beide Richtungen. Statt `xs.add(new Integer(0))` schreiben wir einfach `xs.add(0)`. Die automatische Umwandlung verringert nur den Schreibaufwand, nicht die dadurch bedingte Ineffizienz zur Laufzeit.


Generizität bietet statische Typsicherheit. Bereits der Compiler garantiert, dass in ein Objekt von `List<String>` nur Zeichenketten eingefügt werden können. Der Versuch, ein Objekt eines inkompatiblen Typs einzufügen, wird erkannt und als Fehler gemeldet. Wer den Umgang mit Collections, Listen und ähnlichen Datenstrukturen ohne Unterstützung durch Generizität gewohnt ist, kennt die Probleme mangelnder statischer Typ-
sicherheit, bei der Typfehler (in Form von Typkonvertierungsfehlern) erst zur Laufzeit auftreten. Generizität kann solche dynamischen Typfehler beseitigen und gleichzeitig die Lesbarkeit von Programmen verbessern.

Nicht nur Klassen und Interfaces können generisch sein, sondern auch Methoden, wie das nächste Beispiel zeigt:

```java
public interface Comparator<A> {
    int compare(A x, A y); // result < 0 if x < y
    // result == 0 if x == y
    // result > 0 if x > y
}

public class CollectionOps {
    public static <A> A max(Collection<A> xs,
                            Comparator<A> c ) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (c.compare(w, x) < 0)
                w = x;
        }
        return w;
    }
}
```


Generische Methoden haben den Vorteil, dass man die für Typparameter zu verwendenden Typen nicht explizit angeben muss:
3.1 Generizität

List<Integer> xs = ...;
List<String> ys = ...;

Comparator<Integer> cx = ...;
Comparator<String> cy = ...;

Integer rx = CollectionOps.max(xs, cx);
String ry = CollectionOps.max(ys, cy);
// Integer rz = CollectionOps.max(xs, cy); !Fehler!


Zum Abschluss sei hier noch ein Beispiel für die Implementierung eines sehr einfachen Komparators gezeigt:

class IntComparator implements Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return x.intValue() - y.intValue();
    }
}

Aufgrund von Autounboxing kann man die dritte Zeile auch einfach durch `return x - y;` ersetzen. Ein Komparator für Zeichenketten wird zwar etwas komplizierter, aber nach demselben Schema aufgebaut sein.

3.1.3 Gebundene Generizität in Java

Die einfache Form der Generizität ist zwar elegant und sicher, aber für einige Verwendungszwecke nicht ausreichend: Im Rumpf einer einfachen generischen Klasse oder Methode ist über den Typ, der den Typparameter ersetzt, nichts bekannt. Insbesondere ist nicht bekannt, ob Objekte dieser Typen bestimmte Methoden oder Variablen haben.

Schranken. Über manche Typparameter benötigt man mehr Information um auf Objekte der entsprechenden Typen zugreifen zu können. Gebundene Typparameter liefern diese Information: In Java kann man für jeden
Typparameter eine Klasse und beliebig viele Interfaces als *Schranken* angeben. Nur Untertypen der Schranken dürfen den Typparameter ersetzen. Damit ist statisch bekannt, dass in jedem Objekt des Typs, für den der Typparameter steht, die in den Schranken festgelegten öffentlich sichtbaren Methoden und Variablen verwendbar sind. Man kann Instanzen des Typparameters wie Instanzen der Schranken verwenden:

```java
public interface Scalable {
    void scale(double factor);
}

public class Scene<T extends Scalable> implements Iterable<T> {
    public void addSceneElement(T e) { ... }
    public Iterator<T> iterator() { ... }
    public void scaleAll(double factor) {
        for (T e : this)
            e.scale(factor);
    }
    ... }     
```

Die Klasse *Scene* hat einen Typparameter *T* mit einer Schranke. Jeder Typ, der *T* ersetzt, ist Untertyp von *Scalable* und unterstützt damit die Methode *scale*. Diese Methode wird in *scaleAll* aufgerufen (für jedes Element des aktuellen Objekts von *Scene*).


In obigem Beispiel erweitert *Scene* das in den Java-Bibliotheken vordefinierte Interface *Iterable<T>* , welches die Methode *iterator* zur Erzeugung eines Iterators beschreibt. In *Scene* wird der Iterator benötigt um einfach mittels *for*-Schleife über alle Elemente des aktuellen Objekts von *Scene* zu iterieren.
Rekursion. Diese Beispiels-Variante verwendet Typparameter rekursiv:

```java
public interface Comparable<A> {
    int compareTo(A that);  // res. < 0 if this < that
    // res. == 0 if this == that
    // res. > 0 if this > that
}

public class Integer implements Comparable<Integer> {
    private int value;
    public Integer(int value) { this.value = value; }
    public int intValue() { return value; }
    public int compareTo(Integer that) {
        return this.value - that.value;
    }
}

public class CollectionOps2 {
    public static <A extends Comparable<A>>
            A max(Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
}
```

3 Generizität und Ad-hoc-Polymorphismus

Keine impliziten Untertypen. Generizität unterstützt keine impliziten Untertypbeziehungen. So besteht zwischen \texttt{List\langle X\rangle} und \texttt{List\langle Y\rangle} keine Untertypbeziehung wenn \texttt{X} und \texttt{Y} verschieden sind, auch dann nicht, wenn \texttt{Y} von \texttt{X} abgeleitet ist oder umgekehrt. Natürlich gibt es die expliziten Untertypbeziehungen, wie beispielsweise die zwischen \texttt{Integer} und \texttt{Comparable\langle Integer\rangle}. Man kann Klassen wie üblich ableiten:

\begin{verbatim}
class MyList\langle A\rangle extends List\langle List\langle A\rangle\rangle { ... }
\end{verbatim}

Dann ist \texttt{MyList\langle String\rangle} ein Untertyp von \texttt{List\langle List\langle String\rangle\rangle}, aber \texttt{MyList\langle X\rangle} ist kein Untertyp von \texttt{List\langle Y\rangle} wenn \texttt{Y} möglicherweise ungleich \texttt{List\langle X\rangle} ist. Die Annahme impliziter Untertypbeziehungen ist ein häufiger Anfängerfehler. Man muss stets bedenken, dass es weder in Java noch in irgendeiner anderen Sprache, die auf F-gebundener Generizität beruht, sichere implizite Untertypbeziehungen dieser Art gibt.

In Java können bei Verwendung von Arrays Typfehler zur Laufzeit auftreten, da Arrays implizite Untertypbeziehungen unterstützen:

\begin{verbatim}
class Loophole {  
  public static String loophole(Integer y) {  
    String[] xs = new String[10];  
    Object[] ys = xs;  // no compile-time error  
    ys[0] = y;  // throws ArrayStoreException  
    return xs[0];  
  }  
}
\end{verbatim}

Diese Klasse wird unbeanstandet übersetzt, da in Java für jede Untertypbeziehung auf Typen automatisch eine Untertypbeziehung auf Arrays von Elementen solcher Typen angenommen wird, obwohl Ersetzbarkeit verletzt sein kann. Im Beispiel nimmt der Compiler an, dass \texttt{String[]} Untertyp von \texttt{Object[]} ist, da \texttt{String} ein Untertyp von \texttt{Object} ist. Diese Annahme ist falsch. Generizität schließt solche Fehler durch das Verbot impliziter Untertypbeziehungen aus:

\begin{verbatim}
class NoLoophole {  
  public static String loophole(Integer y) {  
    List\langle String\rangle xs = new List\langle String\rangle();  
    List\langle Object\rangle ys = xs;  // compile-time error  
    ys.add(y);  
    return xs.iterator().next();  
  }  
}
\end{verbatim}

128
3.1 Generizität

Wildcards. Die Sicherheit durch Nichtunterstützung impliziter Untertypbeziehungen hat auch einen Nachteil. Zum Beispiel kann die Methode

```java
void drawAll(List<Polygon> p) {
    ... // draws all polygons in list p
}
```

nur mit Argumenten vom Typ `List<Polygon>` aufgerufen werden, nicht aber mit Argumenten vom Typ `List<Triangle>` und `List<Square>` (entsprechend dem Beispiel in Abschnitt 2.2.3). Dies ist bedauerlich, da `drawAll` nur Elemente aus der Liste liest und nie in die Liste schreibt, Sicherheitsprobleme durch implizite Untertypbeziehungen wie bei Arrays aber nur beim Schreiben auftauchen. Für solche Fälle unterstützt Java *gebundene Wildcards* als Typen, die Typparameter ersetzen:

```java
void drawAll(List<? extends Polygon> p) { ... }
```


Gelegentlich gibt es auch Parameter, deren Inhalte in einer Methode nur geschrieben und nicht gelesen werden:

```java
void addSquares (List<? extends Square> from,
        List<? super Square> to ) { ...
    ... // add squares from 'from' to 'to'
}
```

In `to` wird nur geschrieben, von `to` wird nicht gelesen. Als Argument für `to` können wir daher `List<Square>`, aber auch `List<Polygon>` und `List<Object>` angeben. Als Schranke spezifiziert das Schlüsselwort `super`, dass jeder Obertyp von `Square` erlaubt ist. Der Compiler erlaubt die Verwendung von `to` nur an Stellen, für deren Typen in Untertypbeziehungen Kontravarianz gefordert ist; das sind Schreibzugriffe.

Flexibilität durch Wildcards. In der Praxis kann die Verwendung solcher Wildcards recht kompliziert werden, wie folgendes Beispiel zeigt:

```java
public class MaxList<A extends Comparable<? super A>>
    extends List<A> {
    public A max() {
        Iterator<A> i = this.iterator();
        A w = i.next();
        while (i.hasNext()) {
            A x = i.next();
            if (w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
}
```


le genungen zwar nachvollziehen, wie die Richtung mit den Klammerebenen zusammenhängt, aber intuitiv ist das nicht. Maschinen können damit einfacher umgehen. Am besten verlässt man sich dabei auf den Compiler, der zuverlässig warnt, wenn man einen falschen Wildcard verwendet.


### 3.2 Verwendung von Generizität im Allgemeinen

Wir wollen nun betrachten, wie man Generizität in der Praxis einsetzt. Abschnitt [3.2.1] gibt einige allgemeine Ratschläge, in welchen Fällen sich die Verwendung auszahlt. In Abschnitt [3.2.2] werden wir uns mit möglichen Übersetzungen generischer Klassen beschäftigen und einige Alternativen zur Generizität vorstellen, um ein etwas umfassenderes Bild davon zu bekommen, was Generizität leisten kann.
3 Generizität und Ad-hoc-Polymorphismus

3.2.1 Richtlinien für die Verwendung von Generizität

Generell ist der Einsatz von Generizität immer sinnvoll, wenn er die Wartbarkeit verbessert. Aber oft ist nur schwer entscheidbar, ob diese Voraussetzung zutrifft. Wir wollen hier einige typische Situationen als Entscheidungshilfen bzw. Faustregeln anführen:


**Faustregel:** Containerklassen sollen generisch sein.

Es zahlt sich aus, Generizität bereits beim ersten Verdacht, dass eine Containerklasse auch für andere Elementtypen sinnvoll sein könnte, zu verwenden. Beim Erstellen der Klasse ist es leicht zwischen Elementtypen und anderen Typen zu unterscheiden. Im Nachhinein, also wenn eine nichtgenerische Klasse in eine generische umgewandelt wird, ist nicht immer gleich zu erkennen ob ein Typ als Elementtyp angesehen werden soll oder nicht. Die Lesbarkeit und Laufzeiteffizienz wird durch die Verwendung von Generizität kaum oder überhaupt nicht beeinträchtigt. Es zahlt sich daher aus, Generizität bereits frühzeitig zu verwenden.

Es kann passieren, dass man die Sinnhaftigkeit von Typparametern erst spät erkennt. In diesen Fällen soll man das Programm gleich refaktorisieren, also die Klasse oder Methode mit Typparametern versehen. Ein Hinauszögern der Refaktorisierung führt leicht zu unnötigem Code.

Üblicher Programmcode enthält nur relativ wenige generische Containerklassen. Der Grund dafür liegt einfach darin, dass die meisten Programmierungsumgebungen mit umfangreichen Bibliotheken ausgestattet sind, welche die am häufigsten verwendeten, immer wieder gleich strukturierten
Klassen und Methoden bereits enthalten. Man braucht diese Klassen und Methoden also nur zu verwenden statt sie neu schreiben zu müssen.

**Faustregel:** Klassen und Methoden in Bibliotheken sollten generisch sein.

In aktuellen Java-Versionen sind die Standardbibliotheken durchwegs generisch. Generizität ist in Java so gestaltet, dass der Umstieg auf Generizität möglichst leicht ist. Übersetzte generische Klassen können auch in älteren, nicht-generischen Java-Versionen verwendet werden, und generisches Java kann mit nicht-generischen Klassen umgehen.

**Abfangen erwarteter Änderungen.** Mittels Generizität kann man erwartete Programmänderungen vereinfachen. Das gilt vor allem auch für Typen von formalen Parametern, die sich entsprechend dem Ersetzbarkeitsprinzip nicht beliebig ändern dürfen. Man soll von Anfang an Typparameter für die Typen formaler Parameter verwenden, wenn man erwartet, dass sich diese Typen im Laufe der Zeit ändern. Das gilt auch dann, wenn es sich nicht offensichtlich um Elementtypen in Containerklassen handelt. Es brauchen nicht gleichzeitig mehrere gleich strukturierte Klassen oder Methoden sinnvoll sein, sondern es reicht wenn zu erwarten ist, dass sich Typen in unterschiedlichen Versionen voneinander unterscheiden.

**Faustregel:** Man soll Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der Parametertypen absehbar sind.

daher gebundene Typparameter verwenden, und alle Währungen werden Untertypen eines bestimmten Typs, sagen wir Currency, sein.


Faustregel: Generizität und Untertyprelationen ergänzen sich. Man soll stets überlegen, ob man eine Aufgabe besser durch Ersetzbarkeit, durch Generizität, oder (häufig sinnvoll) eine Kombination aus beiden Konzepten löst.

Es sollte aber auch stets klar sein, dass nur Untertypen Ersetzbarkeit gewährleisten können. Bei Generizität bedingt eine Änderung im Servercode in der Regel auch Änderungen im Code aller Clients. Sinnvoll eingesetzte Untertypen können das verhindern.

Verwendbarkeit. Generizität und Untertypbeziehungen sind oft gegeneinander austauschbar. Das heißt, man kann ein und dieselbe Aufgabe mit Generizität oder über Untertypbeziehungen lösen. Es stellt sich die Frage, ob nicht generell ein Konzept das andere ersetzen kann. Das geht nicht, wie man an folgenden zwei Beispielen sieht:

3.2 Verwendung von Generizität im Allgemeinen

statisch sichergestellt, dass alle Elemente in der Liste denselben Typ haben. Daher kann man mit Hilfe von Generizität etwas machen, was ohne Generizität, also nur durch Untertypbeziehungen, nicht machbar wäre.

Mit Generizität ohne Untertypbeziehungen ist es nicht möglich, eine Listenklasse zu schreiben, in der Elemente unterschiedliche Typen haben können. Solche Listen sind heterogen. Daher kann man mit Hilfe von Untertypbeziehungen etwas machen, was ohne sie, also nur durch Generizität, nicht machbar wäre. Generizität und Untertypbeziehungen ergänzen sich.

Diese Beispiele zeigen, was man mit Generizität oder Untertypbeziehungen alleine nicht machen kann. Sie zeigen damit auf, in welchen Fällen man Generizität bzw. Untertypen zur Erreichung des Ziels unbedingt braucht.


**Faustregel:** Man soll Effizienzüberlegungen in der Entscheidung, ob man Generizität oder Untertypbeziehungen einsetzt, beiseitelassen.
Solche Optimierungen auf der untersten Ebene sind nur etwas für Exper-
ten, die Details ihrer Compiler und ihrer Hardware sehr gut kennen, und auch dann sind die Optimierungen nicht portabel. Viel wichtiger ist es, auf die Einfachheit und Verständlichkeit zu achten. Wenn Effizienz entschei-
dend ist, muss man vor allem die Effizienz der Algorithmen betrachten.

**Natürlichkeit.** Häufig bekommt man auf die Frage, ob man in einer be-
stimmten Situation Generizität oder Subtyping einsetzen soll, die Ant-
wort, dass der *natürlichere* Mechanismus am besten geeignet sei. Für erfah-
rene Leute ist diese Antwort zutreffend: Mit einem gewissen Erfahrungs-
schatz kommt es ihnen selbstverständlich vor, den richtigen Mechanismus zu wählen ohne die Entscheidung begründen zu müssen. Hinter der Na-
türlichkeit verbirgt sich der Erfahrungsschatz. Mit wenig Erfahrung sieht man kaum, was natürlicher ist. Es zahlt sich in jedem Fall aus genau zu überlegen, was man mit Generizität erreichen will und erreichen kann. Wenn man sich zwischen Generizität und Subtyping entscheiden soll, ist es angebracht, auch eine Kombination von Generizität und Subtyping ins Auge zu fassen. Erst wenn diese Überlegungen zu keinem eindeutigen Ziel führen entscheidet man sich für die natürlichere Alternative.

### 3.2.2 Arten der Generizität

Bisher haben wir Generizität als ein einziges Sprachkonzept betrachtet. Tatsächlich gibt es zahlreiche Varianten mit unterschiedlichen Eigenschaf-
ten. Wir wollen hier einige Varianten miteinander vergleichen.

Für die Übersetzung generischer Klassen und Methoden in ausführba-
ren Code gibt es im Großen und Ganzen zwei Möglichkeiten, die homo-
gene und die heterogene Übersetzung. In Java wird die *homogene* Über-
schnitt 5.3.2 sehen werden. Dies entspricht der Simulation einiger Aspekte von Generizität. Im Unterschied zur simulierten Generizität wird die Typ-
kompatibilität vom Compiler garantiert.
3.2 Verwendung von Generizität im Allgemeinen


---

(für Interessierte)

Folgendes Beispiel zeigt einen Ausschnitt aus einer generischen Klasse in C++:

137
Die Klasse `Pair` verwendet `T` als Typparameter. Für `T` kann jeder Typ eingesetzt werden, auf dessen Instanzen der Operator `+` definiert ist, da `+` in der Methode `sum` verwendet wird. Im Kopf der Klasse ist diese Einschränkung nicht angeführt. Hier sind einige Beispiele für die Verwendung von `Pair`:

```cpp
class Person { // Beispielklasse für Personen
    std::string name;
    // Konstruktor...
};

Pair<int> anIntPair(2, 3);
Pair<Person> aPersonPair(Person("Susi"),
                        Person("Strolchi"));
Pair<char*> aStringPair("Susi", "Strolchi");
```


Das nächste Beispiel zeigt eine generische Funktion in C++:

```cpp
template<class T> T max(T a, T b) {
    if (a > b)
        return a;
    return b;
}
```

```cpp
...  
int i, j;
char *x, *y;
Pair<int> p, q;
...
i = max(i, j);  // maximum of integers
x = max(x, y);  // maximum of pointers to characters
p = max(p, q);  // maximum of integer pairs
```

138
3.2 Verwendung von Generizität im Allgemeinen

Wie in Java erkennt der Compiler anhand der Typen der Argumente, welcher Typ für den Typparameter zu verwenden ist. Im Beispiel wird vorausgesetzt, dass der Operator > auf allen Typen, die für T verwendet werden, definiert ist, ohne diese Eigenschaft explizit zu machen. Für künftige Standards werden Wege gesucht, solche Eigenschaften unter Beibehaltung von Templates auszudrücken.


Eine generische Funktion in Ada soll zeigen, welche Flexibilität Einschränkungen auf Typparametern bieten können:

```ada
generic
  type T is private;
  with function "<" (X, Y: T) return Boolean is (<>);
function Max (X, Y: T) return T is begin
  if X < Y
    return Y
  else
    return X
  end if
end Max;

function IntMax is new Max (Integer);
function IntMin is new Max (Integer, ">");
```

Die Funktion Max hat zwei generische Parameter: den Typparameter T und den Funktionsparameter <, dessen Parametertypen mit dem Typparameter in

In Java kann man Ähnliches wie in Ada durch Komparatoren (siehe Abschnitt 3.1.2) auch erzielen, jedoch nur mit hohem Aufwand.

3.3 Typabfragen und Typumwandlungen


3.3.1 Verwendung dynamischer Typinformation

Jedes Objekt hat eine Methode getClass, welche die Klasse des Objekts als Ergebnis zurückgibt. Diese Methode bietet die direkteste Möglichkeit des Zugriffs auf den dynamischen Typ. Davon wird aber eher selten Gebrauch gemacht, da diese Typinformation an Programmstellen, an denen
die Klasse eines Objekts nicht ohnehin bekannt ist, kaum benötigt wird. In Spezialfällen ist getClass jedoch sehr wertvoll.

Etwas häufiger möchte man wissen, ob der dynamische Typ eines Referenzenobjekts Untertyp eines gegebenen Typs ist. Dafür bietet Java den instanceof-Operator an, wie folgendes Beispiel zeigt:

```java
int calculateTicketPrice(Person p) {
    if (p.age < 15 || p instanceof Student)
        return standardPrice / 2;
    return standardPrice;
}
```

Eine Anwendung des instanceof-Operators liefert true, wenn das Objekt links vom Operator eine Instanz des Typs rechts vom Operator ist. Im Beispiel liefert die Typabfrage true wenn p vom dynamischen Typ Student, Studienassistent oder Werkstudent ist (entsprechend der Typhierarchie aus Abschnitt 2.1.2). Die Abfrage liefert false wenn p gleich null oder von einem anderen dynamischen Typ ist. Solche dynamischen Typabfragen können, wie alle Vergleichsoperationen, an beliebigen Programmstellen stehen, an denen Boolesche Ausdrücke erlaubt sind.

Mittels Typabfragen lässt sich zwar der Typ eines Objektes zur Laufzeit bestimmen, aber Typabfragen reichen nicht aus, um Eigenschaften des dynamisch ermittelten Typs zu nutzen. Wir wollen auch Nachrichten an das Objekt senden können, die nur Instanzen des dynamisch ermittelten Typs verstehen, oder das Objekt als aktuellen Parameter verwenden, wobei der Typ des formalen Parameters dem dynamisch ermittelten Typ entspricht. Für diese Zwecke gibt es in Java explizite Typumwandlungen als Casts auf Referenzenobjekten. Folgendes (auf den ersten Blick überzeugende, tatsächlich aber fehlerhafte – siehe Abschnitt 3.3.3) Beispiel modifiziert ein Beispiel aus Abschnitt 2.1.1:

```java
class Point3D extends Point2D {
    private int z;
    public boolean equal(Point2D p) {
        // true if points are equal in all coordinates
        if (p instanceof Point3D)
            return super.equal(p) && ((Point3D)p).z == z;
        return false;
    }
}
```
3 Generizität und Ad-hoc-Polymorphismus

In **Point3D** liefert `equal` als Ergebnis `false` wenn der dynamische Typ des Arguments kein Untertyp von **Point3D** ist. Sonst wird die Methode aus **Point2D** aufgerufen, und die zusätzlichen Objektvariablen `z` werden verglichen. Vor dem Zugriff auf `z` von `p` ist eine explizite Typumwandlung nötig, die den deklarierten Typ von `p` von **Point2D** (wie im Kopf der Methode angegeben) nach **Point3D** umwandelt, da `z` in Objekten von **Point2D** nicht zugreifbar ist. Syntaktisch wird die Typumwandlung als `(Point3D)p` geschrieben. Um den Ausdruck herum sind weitere Klammern nötig, damit klar ist, dass der Typ von `p` umgewandelt werden soll, nicht der Typ des Ergebnisses von `p.z == z` wie in `(Point3D)p.z == z`.


Dynamische Typabfragen und Typumwandlungen sind sehr mächtige Werkzeuge. Man kann damit einiges machen, was sonst nicht oder nur

---

1 Versuchen Sie Fehler in dieser Lösung selbst zu finden. Die Verwendung von dynamischer Typinformation und Typumwandlung ist hier korrekt. Aber vielleicht wird eine implizite Zusicherung verletzt, wie die, dass `a.equal(b)` dasselbe Ergebnis liefert wie `b.equal(a)`. Am schnellsten wird man den Fehler finden, indem man Zusicherungen auf **Point3D** und **Point2D** explizit macht.
3.3 Typabfragen und Typumwandlungen


Die schlechte Wartbarkeit ist ein weiterer Grund um Typabfragen (auch ohne Typumwandlungen) nur sehr sparsam zu nutzen:

```java
if (x instanceof T1)
    doSomethingOfTypeT1((T1)x);
else if (x instanceof T2)
    doSomethingOfTypeT2((T2)x);
...
else
    doSomethingOfTypeAnyType(x);
```

Wie allgemein bekannt, lassen sich switch- und if-Anweisungen durch dynamisches Binden ersetzen. Das soll man tun, da dynamisches Binden wartungsfreundlicher ist. Dasselbe gilt für dynamische Typabfragen, welche die möglichen Typen im Programmcode fix verdrahten und daher bei Änderungen der Typhierarchie ebenfalls geändert werden müssen. Oft sind dynamische Typabfragen wie im Beispiel durch x.doSomething() ersetzbar. Die Auswahl des auszuführenden Programmcodes erfolgt durch dynamisches Binden. Die Klasse des deklarierten Typs von x implementiert doSomething entsprechend doSomethingOfTypeAnyType, die Un-
Generizität und Ad-hoc-Polymorphismus

terklassen T1, T2 und so weiter entsprechend doSomethingOfTypeT1, doSomethingOfTypeT2 und so weiter.

Manchmal ist es nicht einfach, dynamische Typabfragen durch dynamisches Binden zu ersetzen. Dies trifft vor allem in diesen Fällen zu:

- Der deklarierte Typ von x ist zu allgemein; die einzelnen Alternativen decken nicht alle Möglichkeiten ab. Das ist genau die oben erwähnte gefährliche Situation, in der die statische Typsicherheit von Java umgangen wird. In dieser Situation ist eine Refaktorisierung des Programms angebracht, die Typabfragen vermeidet.


- Manchmal ist die Verwendung dynamischen Bindens schwierig, weil die einzelnen Alternativen auf private Variablen und Methoden zugreifen. Methoden anderer Klassen haben diese Information nicht. Oft lässt sich die fehlende Information durch Übergabe geeigneter Argumente beim Aufruf der Methode oder durch „Call-Backs“ (also Rückfragen an das aufrufende Objekt – sinnvoll wenn die Information nur selten benötigt wird) verfügbar machen.

- Der deklarierte Typ von x kann sehr viele Untertypen haben. Wenn doSomething nicht in einer gemeinsamen Oberklasse in der Bedeutung von doSomethingOfTypeAnyType implementierbar ist, beispielsweise weil der deklarierte Typ von x ein Interface ist, muss doSomething in vielen Klassen auf gleiche Weise implementiert werden. Das bedeutet einen Mehraufwand für die Wartung. Der Grund dafür liegt in der fehlenden Unterstützung der Mehrfachvererbung in Java. Durch Refaktorisierung und Verwendung geeigneter Entwurfsmuster lassen sich diese Probleme abschwächen oder vermeiden. Typabfragen ermöglichen eine einfache Lösung.

| Faustregel: | Typabfragen und Typumwandlungen sollen nach Möglichkeit vermieden werden. |
In wenigen Fällen ist es nötig und durchaus angebracht diese mächtigen, aber unsicheren Werkzeuge zu verwenden, wie wir noch sehen werden.


### 3.3.2 Typumwandlungen und Generizität

Die homogene Übersetzung einer generischen Klasse oder Methode in eine Klasse oder Methode ohne Generizität ist im Prinzip sehr einfach, wie wir bereits in Abschnitt 3.2.2 gesehen haben:

- Alle Ausdrücke in spitzen Klammern werden weggelassen.
- Jedes andere Vorkommen eines Typparameters wird durch `Object` oder, falls vorhanden, die erste Schranke ersetzt.
- Ergebnisse und Argumente werden in die nötigen deklarierten Typen umgewandelt, wenn die entsprechenden Ergebnistypen und formalen Parametertypen Typparameter sind, die durch Typen ersetzt wurden – siehe weiter unten.

Für das Beispiel aus Abschnitt 3.1.2 wird folgender Code generiert:

```java
public interface Collection {
    void add(Object elem);
    Iterator iterator();
}

public interface Iterator {
    Object next();
    boolean hasNext();
}
```
public class List implements Collection {
    private class Node {
        private Object elem;
        private Node next = null;
        private Node(Object elem) {
            this.elem = elem;
        }
    }
    private Node head = null;
    private Node tail = null;
    private class ListIter implements Iterator {
        private Node p = head;
        public Object next() {
            if (p == null)
                return null;
            Object elem = p.elem;
            p = p.next;
            return elem;
        }
        public boolean hasNext() {
            return p != null;
        }
    }
    public void add(Object x) {
        if (head == null)
            tail = head = new Node(x);
        else
            tail = tail.next = new Node(x);
    }
    public Iterator iterator() {
        return new ListIter();
    }
}

In diesem übersetzten Code wurden keine Typumwandlungen eingeführt, da nirgends ein Typparameter durch einen Typ ersetzt wurde. Solche Typumwandlungen kommen beispielsweise durch die Übersetzung des Codes in Abbildung 3.2 ins Spiel:
class ListTest {
    public static void main(String[] args) {
        List xs = new List();
        xs.add((Integer)new Integer(0));
        Integer x = (Integer)xs.iterator().next();

        List ys = new List();
        ys.add((String)"zero");
        String y = (String)ys.iterator().next();

        List zs = new List();
        zs.add((List)xs);
        List z = (List)zs.iterator().next();
    }
}


Abgesehen von einigen unbedeutenden Details, auf die wir hier nicht näher eingehen, ist die Übersetzung so einfach, dass man sie auch selbst ohne Unterstützung durch den Compiler durchführen kann. Das ist vermutlich ein Grund dafür, dass Java ursprünglich keine Generizität unterstützte. Wir können gleich direkt Programmcode ohne Generizität schreiben. Allerdings hat das auch einen schwerwiegenden Nachteil: Statt Fehlermeldungen, die bei Verwendung von Generizität der Compiler generiert, werden ohne Generizität erst zur Laufzeit Ausnahmebehandlungen ausgelöst. Zum Beispiel liefert der Java-Compiler für

```java
List<Integer> xs = new List<Integer>();
x.add (new Integer(0));
String y = xs.iterator().next();
```

Eine Fehlermeldung, nicht aber für den daraus generierten Code:

```java
List xs = new List();
x.add (new Integer(0));
String y = (String)xs.iterator().next();
```

Die Vorteile von Generizität liegen also in erster Linie in der besseren Lesbarkeit und höheren Typsicherheit.

Viele nicht-generische Java-Bibliotheken verwenden Klassen, die so aussehen, als ob sie aus generischen Klassen erzeugt worden wären. Das heißt, Objekte, die in Listen etc. eingefügt werden, müssen in der Regel nur Untertypen von `Object` sein, und vor der Verwendung von aus solchen Datenstrukturen gelesenen Objekten steht meist eine Typumwandlung. Die durchgehende Verwendung von Generizität würde den Bedarf an Typumwandlungen vermeiden oder zumindest erheblich reduzieren.

**Faustregel:** Man soll nur sichere Formen der Typumwandlung einsetzen.

**Sichere Typumwandlungen.** Dieser Argumentation folgend ist es leicht sich auch bei der Programmierung in einer Sprache ohne Generizität anzugewöhnen nur „sichere“ Typumwandlungen einzusetzen: Typumwandlungen sind sicher (lösen keine Ausnahmebehandlung aus) wenn
3.3 Typabfragen und Typumwandlungen

• in einen Obertyp des deklarierten Objekttyps umgewandelt wird,
• oder davor eine dynamische Typabfrage erfolgt, die sicherstellt, dass das Objekt einen entsprechenden dynamischen Typ hat,
• oder man das Programmstück so schreibt, als ob man Generizität verwenden würde, dieses Programmstück händisch auf mögliche Typfehler, die bei Verwendung von Generizität zu Tage treten, untersucht und dann die homogene Übersetzung durchführt.

Im ersten Fall handelt es sich um eine völlig harmlose Typumwandlung nach oben in der Typhierarchie (genannt *Up-Cast*), die aber kaum gebraucht wird. Nur auf Argumenten ist sie zur Auswahl zwischen überladenen Methoden manchmal sinnvoll – wie die Typumwandlungen, die bei der Übersetzung der Generizität auf Argumenten eingeführt wurden.

Die beiden anderen Fälle sind wichtiger, beziehen sich aber auf weniger harmlose Typumwandlungen nach unten – sogenannte *Down-Casts*.

Der zweite Punkt in obiger Aufzählung sicherer Formen der Typumwandlungen impliziert, dass es einen sinnvollen Programmzweig geben muss, der im Falle des Scheiterns des Typvergleichs ausgeführt wird. Wenn man in einem alternativen Zweig einfach nur eine Ausnahmebehandlung anstösst, kann man nicht mehr von einer sicheren Typumwandlung sprechen. Leider erweisen sich gerade falsche Typannahmen in alternativen Zweigen zu Typabfragen als häufige Fehlerquelle.

**Faustregel:** Bei Zutreffen des zweiten Punktes ist besonders darauf zu achten, dass alle Annahmen im alternativen Zweig (bei Scheitern des Typvergleichs) in Zusicherungen stehen.

Außerdem gibt es oft mehrere alternative Zweige, die sich in geschachtelten *if*-Anweisungen zeigen. Aufgrund der damit verbundenen Wartungsprobleme sollte man auf solche Lösungen verzichten.

Bei Zutreffen des dritten Punktes treten keine solchen Probleme auf. Stattdessen sind aufwendige händische Programmanalysen notwendig. Es muss vor allem sichergestellt werden, dass

• wirklich alle Ersetzungen eines (gedachten) Typparameters durch einen Typ gleichförmig erfolgen – das heißt, jedes Vorkommen des Typparameters tatsächlich durch denselben Typ ersetzt wird,
• und keine impliziten Untertypbeziehungen vorkommen.
Vor allem hinsichtlich impliziter Untertypbeziehungen kann die Intuition manchmal in die Irre führen, da beispielsweise sowohl `List<Integer>` als auch `List<String>` in der homogenen Übersetzung durch `List` dargestellt werden, obwohl sie nicht gegeneinander ersetzbar sind.

**Faustregel:** Wenn die Programmiersprache Generizität unterstützt, soll die dritte Möglichkeit nicht verwendet werden. Generizität ist einer dynamischen Typumwandlung immer vorzuziehen. Wenn Generizität nicht unterstützt wird, ist der dritte Punkt dem zweiten vorzuziehen. Unnötige dynamische Typvergleiche (z.B. zur Absicherung einer Typumwandlung obwohl die Voraussetzungen dafür gemäß dem dritten Punkt händig überprüft wurden) sollen vermieden werden, da sie die Sicherheit nicht wirklich erhöhen, aber die Wartung erschweren können.

**Umgang mit Einschränkungen der Generizität.** Generizität ist, mit einigen Einschränkungen, auch in dynamischen Typabfragen und Typumwandlungen einsetzbar:

```java
<A> Collection<A> up(List<A> xs) {
    return (Collection<A>)xs;
}
<A> List<A> down(Collection<A> xs) {
    if (xs instanceof List<A>)
        return (List<A>)xs;
    else { ... } // Was macht man hier?
}
List<String> bad(Object o) {
    if (o instanceof List<String>) // error
        return (List<String>)o; // error
    else { ... } // Was macht man hier?
}
```

In der Methode `bad` werden vom Compiler Fehlermeldungen ausgegeben, da es zur Laufzeit keine Information über den gemeinsamen Typ der Listenelemente gibt. Es ist daher unmöglich, in einer Typabfrage oder Typumwandlung dynamisch zu prüfen, ob `o` den gewünschten Typ hat. Die Methoden `up` und `down` haben dieses Problem nicht, weil der bekannte Unter- beziehungsweise Obertyp den Typ aller Listenelemente bereits statisch festlegt, falls es sich tatsächlich um eine Liste handelt. Der Compiler
ist intelligent genug, solche Situationen zu erkennen. Bei der Übersetzung werden einfach alle spitzen Klammern (und deren Inhalte) weggelassen. Im übersetzten Programm sind die strukturellen Unterschiede zwischen down und bad nicht mehr erkennbar, aber bad kann zu einer Ausnahmebehandlung führen. Bei der händischen Programmüberprüfung ist auf solche Feinheiten besonders zu achten.

Java erlaubt die gemischte Verwendung von generischen Klassen und Klassen, die durch homogene Übersetzung daraus erzeugt wurden:

```java
public class List<A> implements Collection<A> {
    ...
    public boolean equals(Object that) {
        if (!(that instanceof List))
            return false;
        Iterator<A> xi = this.iterator();
        Iterator yi = ((List)that).iterator();
        while (xi.hasNext() && yi.hasNext()) {
            A x = xi.next();
            Object y = yi.next();
            if (!(x == null ? y == null : x.equals(y)))
                return false;
        }
        return !(xi.hasNext() || yi.hasNext());
    }
}
```


3 Generizität und Ad-hoc-Polymorphismus

3.3.3 Kovariante Probleme


abstract class Futter { ... }
class Gras extends Futter { ... }
class Fleisch extends Futter { ... }
abstract class Tier {
    public abstract void friss(Futter x);
}
class Rind extends Tier {
    public void friss(Gras x) { ... }
    public void friss(Futter x) {
        if (x instanceof Gras)
            friss((Gras)x);
        else
            erhoeheWahrscheinlichkeitFuerBSE();
    }
}
class Tiger extends Tier {
    public void friss(Fleisch x) { ... }
    public void friss(Futter x) {
        if (x instanceof Fleisch)
            friss((Fleisch)x);
        else
            fletscheZaehne();
    }
}

In Rind und Tiger ist friss überladen, es gibt also mehrere Methoden desselben Namens. Die Methoden mit Futter als Parametertyp überschreiben jene aus Tier, während die anderen nicht in Tier vorkommen. Es wird die Methode ausgeführt, deren formaler Parametertyp der spezifischste Oberotyp des deklarierten Argumenttyps ist. Im Beispiel wird der gewünschte deklarierte Argumenttyp durch Typumwandlung bestimmt.


**Faustregel:** Kovariante Probleme soll man vermeiden.


**Binäre Methoden.** Einen häufig vorkommenden Spezialfall kovarianter Probleme stellen binäre Methoden dar. Wie in Abschnitt 2.1 eingeführt, hat eine binäre Methode mindestens einen formalen Parameter, dessen
3.3 Typabfragen und Typumwandlungen

Typ stets gleich der Klasse ist, welche die Methode enthält. Im Prinzip kann man binäre Methoden auf dieselbe Weise behandeln wie alle anderen kovarianten Probleme. Das heißt, man könnte (wie in Abschnitt 3.3.1) dynamische Typabfragen verwenden um den dynamischen Parametertyp zu bestimmen. Das ist aber problematisch, wie wir gleich sehen werden. Hier ist eine weitere, bessere Lösung für die binäre Methode equal in Point2D und Point3D:

```java
abstract class Point {
    public final boolean equal(Point that) {
        if (that != null &&
            this.getClass() == that.getClass())
            return uncheckedEqual(that);
        return false;
    }
    protected abstract boolean uncheckedEqual(Point p);
}
class Point2D extends Point {
    private int x, y;
    protected boolean uncheckedEqual(Point p) {
        Point2D that = (Point2D)p;
        return x == that.x && y == that.y;
    }
}
class Point3D extends Point {
    private int x, y, z;
    protected boolean uncheckedEqual(Point p) {
        Point3D that = (Point3D)p;
        return x==that.x && y==that.y && z==that.z;
    }
}
```

Anders als in allen vorangegangenen Lösungsansätzen ist Point3D kein Untertyp von Point2D, sondern sowohl Point3D als auch Point2D sind von einer gemeinsamen abstrakten Oberklasse Point abgeleitet. Dieser Unterschied hat nichts direkt mit binären Methoden zu tun, sondern verdeutlicht, dass Point3D in der Regel keine Spezialisierung von Point2D ist. Die Rolle, die bisher Point2D hatte, spielt jetzt Point. Die Methode equal ist in Point definiert und kann in Unterklassen nicht überschrieben werden. Wenn die beiden zu vergleichenden Punkte genau den
generischen Typ haben, wird in der betreffenden Unterkasse von `Point` die Methode `uncheckedEqual` aufgerufen, die den eigentlichen Vergleich durchführt. Im Unterschied zur in Abschnitt 3.3.1 angerissenen Lösung vergleicht diese Lösung, ob die Typen wirklich gleich sind, nicht nur, ob der dynamische Typ des Arguments ein Untertyp der Klasse ist, in der die Methode ausgeführt wird. Die Lösung in Abschnitt 3.3.1 ist falsch, da ein Aufruf von `equal` in `Point2D` mit einem Argument vom Typ `Point3D` als Ergebnis `true` liefern kann.


### 3.4 Überladen versus Multimethoden


Generell, aber nicht in Java, ist es möglich, dass dynamisches Binden auch den dynamischen Typ von `y` in die Methodenauswahl einbezieht. Dann legt nicht bereits der Compiler anhand des deklarierten Typs fest, welche überladene Methode auszuwählen ist, sondern erst zur Laufzeit des Programms wird die auszuführende Methode durch die dynamischen
3.4 Überladen versus Multimethoden

Typen von \( x \) und \( y \) bestimmt. In diesem Fall spricht man nicht von Überladen sondern von Multimethoden [9]; es wird bei einem Methodenaufruf mehrfach dynamisch gebunden.

Leider wird Überladen viel zu oft mit Multimethoden verwechselt. Das führt zu schweren Fehlern. In Abschnitt 3.4.1 werden wir uns die Unterschiede deshalb deutlich vor Augen führen. In Abschnitt 3.4.2 werden wir sehen, dass man Multimethoden auch in Sprachen wie Java recht einfach simulieren kann.

3.4.1 Deklarierte versus dynamische Argumenttypen

Folgendes Beispiel soll klar machen, dass bei der Auswahl zwischen überladenen Methoden nur der deklarierte Typ eines Arguments entscheidend ist, nicht der dynamische. Wir verwenden das Beispiel zu kovarianten Problemen aus Abschnitt 3.3.3:

\[
\begin{align*}
\text{Rind} & \quad \text{rind} = \text{new Rind}(); \\
\text{Futter} & \quad \text{gras} = \text{new Gras}(); \\
\text{rind.fris} & \quad (\text{gras}); \quad \quad \quad \text{// Rind.fris(Futter x)} \\
\text{rind.fris} & \quad ((\text{Gras})\text{gras}); \quad \text{// Rind.fris(Gras x)}
\end{align*}
\]

Wegen dynamischem Binden werden die Methoden \text{friss} auf jeden Fall in der Klasse \text{Rind} ausgeführt, unabhängig davon, ob \text{rind} als \text{Tier} oder \text{Rind} deklariert ist. Der Methodenaufruf in der dritten Zeile führt die überladene Methode mit dem Parameter vom Typ \text{Futter} aus, da \text{gras} mit dem Typ \text{Futter} deklariert ist. Für die Methodenauswahl ist es unerheblich, dass \text{gras} tatsächlich ein Objekt von \text{Gras} enthält; der dynamische Typ von \text{gras} ist \text{Gras}, da \text{gras} direkt vor dem Methodenaufruf mit einem Objekt von \text{Gras} initialisiert wird. Es zählt aber nur der deklarierte Typ. Der Methodenaufruf in der vierten Zeile führt die überladene Methode mit dem Parameter vom Typ \text{Gras} aus, weil der deklarierte Typ von \text{gras} wegen der Typumwandlung an dieser Stelle \text{Gras} ist. Typumwandlungen ändern ja den deklarierten Typ eines Ausdrucks.

Häufig weiß man in solchen Fällen, dass \text{futter} ein Objekt von \text{Gras} enthält und nimmt an, dass die Methode mit dem Parameter vom Typ \text{Gras} gewählt wird. Diese Annahme ist aber falsch! Es wird stets der deklarierte Typ verwendet, auch wenn man den dynamischen Typ kennt.

 Nehmen wir an, die erste Zeile des Beispiels sehe so aus:

\[
\text{Tier} \quad \text{rind} = \text{new Rind}();
\]
Generizität und Ad-hoc-Polymorphismus

Wegen dynamischen Bindens würde \texttt{friss} weiterhin in \texttt{Rind} ausgeführt. Aber zur Auswahl überladener Methoden kann der Compiler nur deklarierte Typen verwenden. Das gilt auch für den Empfänger einer Nachricht. Die überladenen Methoden werden in \texttt{Tier} gesucht, nicht in \texttt{Rind}. In \texttt{Tier} ist \texttt{friss} nicht überladen, sondern es gibt nur eine Methode mit einem Parameter vom Typ \texttt{Futter}. Daher wird in \texttt{Rind} auf jeden Fall die Methode mit dem Parameter vom Typ \texttt{Futter} ausgeführt, unabhängig davon, ob der deklarierte Typ des Arguments \texttt{Futter} oder (nach einem Cast) \texttt{Gras} ist. Wie das Beispiel zeigt, kann sich die Auswahl zwischen überladenen Methoden stark von der Intuition unterscheiden und ist von vielen Details abhängig. Daher ist besondere Vorsicht geboten.

Die Methoden \texttt{friss} in \texttt{Rind} und \texttt{Tiger} sind so überladen, dass es (außer für die Laufzeiteffizienz) keine Rolle spielt, welche der überladenen Methoden aufgerufen wird. Wenn der dynamische Typ des Arguments \texttt{Gras} ist, wird im Endeffekt immer die Methode mit dem Parametertyp \texttt{Gras} aufgerufen. Es ist empfehlenswert, Überladen nur so zu verwenden.

\begin{faustregel}
Man soll Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden.
\end{faustregel}

Für je zwei überladene Methoden gleicher Parameteranzahl

- soll es zumindest eine Parameterposition geben, an der sich die Typen der Parameter unterscheiden, nicht in Untertyprelation zueinander stehen und auch keinen gemeinsamen Untertyp haben,

- oder alle Parametertypen der einen Methode sollen Obertypen der Parametertypen der anderen Methode sein, und bei Aufruf der einen Methode soll nichts anderes gemacht werden, als auf die andere Methode zu verzweigen, falls die entsprechenden dynamischen Typen der Argumente dies erlauben.

Unter diesen Bedingungen ist die strikte Unterscheidung zwischen deklarierten und dynamischen Typen bei der Methodenauswahl nicht wichtig.

Das Problem der Verwechslung von dynamischen und deklarierten Typen könnte man nachhaltig lösen, indem man zur Methodenauswahl generell die dynamischen Typen aller Argumente verwendet. Statt überladener Methoden hätte man dann Multimethoden. Würde Java Multimethoden unterstützen, könnte man die Klasse \texttt{Rind} im Beispiel aus Abschnitt 3.3.3 kürzer und ohne dynamische Typabfragen und -umwandlungen schreiben:
3.4 Überladen versus Multimethoden

class Rind extends Tier {
    public void friss(Gras x) { ... }
    public void friss(Futter x) {
        erhoeheWahrscheinlichkeitFuerBSE();
    }
} // Achtung: In Java ist diese Lösung falsch !!!!

Die Abfrage, ob x den dynamischen Typ Gras hat, hätte man sich erspart, da friss mit dem Parametertyp Futter bei Multimethoden nur aufgerufen wird, wenn der dynamische Argumenttyp nicht Gras ist.


Unter der höheren Komplexität der Methodenauswahl versteht man etwas anderes als die Laufzeiteffizienz: Für Programmierer(innen) ist nicht gleich erkennbar, unter welchen Bedingungen welche Methode ausgeführt wird. Eine Regel besagt, dass immer jene Methode mit den speziellsten Parametertypen, die mit den dynamischen Typen der Argumente kompatibel sind, auszuführen ist. Wenn wir friss mit einem Argument vom Typ Gras (oder einem Untertyp davon) aufrufen, sind die Parametertypen beider Methoden mit dem Argumenttyp kompatibel. Da Gras spezieller ist als Futter, wird die Methode mit dem Parametertyp Gras ausgeführt. Diese Regel ist für die Methodenauswahl aber nicht hinreichend wenn Multimethoden mehrere Parameter haben, wie folgendes Beispiel zeigt:

    public void frissDoppelt(Futter x, Gras y) { ... }
    public void frissDoppelt(Gras x, Futter y) { ... }

Mit einem Aufruf von frissDoppelt mit zwei Argumenten vom Typ Gras sind beide Methoden kompatibel. Aber keine Methode ist spezieller

```java
public void frissDoppelt(Gras x, Gras y) { ... }
```

In Java kommt es zu Fehlern, wenn man unbewusst Überladen statt Überschreiben verwenden, wenn man also eine Methode überschreiben will, die überschreibende Methode sich aber in den Parametertypen von der zu überschreibenden Methode unterscheidet. Beispielsweise werden in C++ die häufigsten derartigen Probleme vom Compiler erkannt, weil es Einschränkungen beim Überladen ererbter Methoden gibt. Nicht so in Java. Man muss speziell darauf achten, dass Parametertypen der überschreibenden und überschriebenen Methode wirklich gleich sind. Nur Ergebnistypen dürfen ab Java 1.5 kovariant verschieden sein.

### 3.4.2 Simulation von Multimethoden

Multimethoden verwenden mehrfaches dynamisches Binden: Die auszuführende Methode wird dynamisch durch die Typen mehrerer Argumente bestimmt. In Java gibt es nur einfaches dynamisches Binden. Trotzdem ist es nicht schwer, mehrfaches dynamisches Binden durch wiederholtes einfaches Binden zu simulieren. Wir nutzen mehrfaches dynamisches Binden für das Beispiel aus Abschnitt [3.3.3](#) und eliminieren damit dynamische Typabfragen und Typumwandlungen:

Statt vonRindGefressen und vonTigerGefressen hätten wir auch einen gemeinsamen Namen wählen können (= Überladen), da sich die Typen der formalen Parameter eindeutig unterscheiden.


Diese Lösung kann auch dadurch erzeugt worden sein, dass in der ursprünglichen Lösung aus Abschnitt 3.3.3 if-Anweisungen mit dynamischen Typabfragen durch dynamisches Binden ersetzt wurden. Nebenbei sind auch die Typumwandlungen verschwunden. Auch diese Umformung ist automatisch durchführbar. Wir haben damit die Möglichkeit, dynamische Typabfragen genauso wie Multimethoden aus Programmen zu entfernen und damit die Struktur des Programms zu verbessern.


Das Visitor-Pattern hat einen bedeutenden Nachteil: Die Anzahl der benötigten Methoden wird schnell sehr groß. Nehmen wir an, wir hätten m unterschiedliche Tierarten und n Futterarten. Zusätzlich zu den m Methoden in den Elementklassen werden m · n Visitormethoden benötigt. Noch rascher steigt die Methodenanzahl mit der Anzahl der dynamischen Bindungen. Bei k ≥ 2 dynamischen Bindungen mit n_i Möglichkeiten für die i-te Bindung (i = 1...k) werden Σ n_1 · n_2 · ... · n_k−1 inhalts liche Methoden und zusätzlich Σ n_1 + n_1 · n_2 + ... + n_1 · n_2 · ... · n_k−1 Methoden für
die Verteilung benötigt, also sehr viele. (Wir vermeiden hier die Begriffe Element- bzw. Visitormethode, da die Klassenhierarchien mit den Indizes $i = 2 \ldots k - 1$ gleichzeitig Element- und Visitorklassen sind.) Für $k = 4$ und $n_1, \ldots, n_4 = 10$ kommen wir auf 11.110 Methoden. Außer für sehr kleine $k$ und kleine $n_i$ ist diese Technik nicht sinnvoll einsetzbar. Vererbung kann die Zahl der nötigen Methoden meist nur unwesentlich verringern.


Lösungen mit Multimethoden oder dynamischen Typabfragen haben meist den Vorteil, dass die Anzahl der nötigen Methoden deutlich kleiner bleibt. Dies trifft besonders dann zu, wenn die Multimethode aus einigen speziellen Methoden mit uneinheitlicher Struktur der formalen Parametersätze und ganz wenigen allgemeinen Methoden, die den großen Rest behandeln, auskommt. Bei Verwendung dynamischer Typabfragen ist in diesen Fällen der große Rest in wenigen else-Zweigen versteckt.

### 3.5 Wiederholungsfragen

1. Was ist Generizität? Wozu verwendet man Generizität?

2. Was ist gebundene Generizität? Was kann man mit Schranken auf Typparameter machen, das ohne Schranken nicht geht?

3. In welchen Fällen soll man Generizität einsetzen, in welchen nicht?
3 Generizität und Ad-hoc-Polymorphismus

4. Was bedeutet statische Typsicherheit in Zusammenhang mit Generizität, dynamischen Typabfragen und Typumwandlungen?

5. Welche Arten von Generizität kann man hinsichtlich ihrer Übersetzung und ihrem Umgang mit Schranken unterscheiden? Welche Art wird in Java verwendet, und wie flexibel ist diese Lösung?

6. Was sind (gebundene) Wildcards als Typen in Java? Wozu kann man sie verwenden?

7. Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?

8. Was wird bei der heterogenen bzw. homogenen Übersetzung von Generizität genau gemacht?

9. Was muss der Java-Compiler überprüfen um sicher zu sein, dass durch Generizität keine Laufzeitfehler entstehen?

10. Welche Möglichkeiten für dynamische Typabfragen gibt es in Java, und wie funktionieren sie genau?

11. Was wird bei einer Typumwandlung in Java umgewandelt – der deklarierte, dynamische oder statische Typ? Warum?

12. Welche Gefahren bestehen bei Typumwandlungen?

13. Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden? In welchen Fällen kann das schwierig sein?

14. Welche Arten von Typumwandlungen sind sicher? Warum?

15. Was sind kovariante Probleme und binäre Methoden? Wie kann man mit ihnen umgehen oder sie vermeiden?

16. Wie unterscheidet sich Überschreiben von Überladen, und was sind Multimethoden?

17. Wie kann man Multimethoden simulieren? Welche Probleme können dabei auftreten?

18. Was ist das Visitor-Entwurfsmuster?

19. Wodurch ist Überladen problematisch, und in welchen Fällen ergeben sich kaum Probleme?
4 Kreuz und quer


4.1 Ausnahmebehandlung

4 Kreuz und quer

4.1.1 Ausnahmebehandlung in Java

Eigentlich sollten Teilnehmer an „Objektorientierte Programmier-techniken“ mit den Grundlagen der Ausnahmebehandlung in Java schon vertraut sein. Da es diesbezüglich immer wieder Schwierigkeiten gibt und um Missverständnissen hinsichtlich der Terminologie vorzubeugen sind hier die wichtigsten Fakten (noch einmal) zusammengestellt.


Vom Java-Laufzeitsystem werden nur Objekte der vordefinierten Unterklassen von Error und RuntimeException als Ausnahmen ausgelöst. Programme können explizit Ausnahmen auslösen, die Objekte jeder beliebigen Unterkasse von Throwable sind. Dies geschieht mit Hilfe der throw-Anweisung, wie im folgenden Beispiel:

```java
public class Help extends Exception { ... }

...  
if (helpNeeded())
    throw new Help();
```
4.1 Ausnahmebehandlung


Zum Abfangen von Ausnahmen gibt es try-catch-Blöcke:

```
try { ... }  
catch(Help e) { ... }  
catch(Exception e) { ... }
```


Normalerweise ist nicht klar, an genau welcher Stelle im try-Block die Ausnahme ausgelöst wurde. Man weiß daher nicht, welche Befehle bereits ausgeführt wurden und ob die Werte in den Variablen konsistent sind. Man muss catch-Klauseln so schreiben, dass sie mögliche Inkonsistenzen beseitigen. Variablen, die in einem try-Block deklariert wurden, sind in entsprechenden catch-Blöcken nicht sicht- und verwendbar.

Falls ein try-catch-Block eine Ausnahme nicht abfangen kann oder während der Ausführung einer catch-Klausel eine weitere Ausnahme ausgelöst wird, wird nach dem nächsten umschließenden try-Block gesucht. Wenn es innerhalb der Methode, in der die Ausnahme ausgelöst wurde,
keinen geeigneten try-catch-Block gibt, so wird die Ausnahme von der Methode statt einem regulären Ergebnis zurückgegeben und die Suche nach einem passenden try-catch-Block im Aufrufer fortgesetzt, solange bis die Ausnahme abgefangen ist oder es (für die statische Methode main) keinen Aufrufer mehr gibt, an den die Ausnahme weitergereicht werden könnte. Letzterer Fall führt zum Programmabbruch.

Methoden dürfen nicht Ausnahmen beliebiger Typen zurückgeben, sondern nur Objekte von Error und RuntimeException sowie Ausnahmen von Typen, die im Kopf der Methode ausdrücklich angegeben sind – daher der Begriff überprüfte Ausnahmen. Soll eine Methode foo beispielsweise auch Objekte von Help als Ausnahmen zurückgeben können, so muss dies durch eine throws-Klausel deklariert sein:

```java
void foo() throws Help { ... }
```

Alle Ausnahmen, die im Rumpf der Methode ausgelöst, aber mangels Eintrag in der throws-Klausel nicht zurückgegeben werden können, müssen im Rumpf der Methode durch einen geeigneten try-catch-Block abgefangen werden. Das wird vom Compiler überprüft.

Die im Kopf von Methoden deklarierten Ausnahmetypen sind für das Bestehen von Untertypbeziehungen relevant. Das Ersetzbarkeitsprinzip verlangt, dass die Ausführung einer Methode eines Untertyps nur solche Ausnahmen zurückliefern kann, die bei Ausführung der entsprechenden Methode des Obertyps erwartet werden. Daher dürfen Methoden in einer Unterklasse in der throws-Klausel nur Typen anführen, die auch in der entsprechenden throws-Klausel in der Oberklasse stehen. Selbiges gilt natürlich auch für Interfaces. Der Java-Compiler überprüft diese Bedingung. In Unterklassen dürfen Typen von Ausnahmen aber weggelassen werden, wie das folgende Beispiel zeigt:

```java
class A {
    void foo() throws Help, SyntaxError { ... }
}
class B extends A {
    void foo() throws Help { ... }
}
```

Der Compiler kann natürlich nur das Vorhandensein von Typnamen in throws-Klauseln prüfen, nicht das tatsächliche Werfen von Ausnahmen. Ersetzbarkeit verlangt, dass die Ausführung einer Methode im Untertyp
4.1 Ausnahmebehandlung

niemals eine Ausnahme zurückpropagieren darf, die nicht auch bei Ausführung der entsprechenden Methode im Obertyp in derselben Situation zurückgegeben werden könnte und daher vom Aufrufer erwartet wird. Beschreibt eine Nachbedingung einer Methode im Obertyp beispielsweise, dass in einer konkreten Situation eine Ausnahme ausgelöst und propagiert wird, darf die entsprechende Methode im Untertyp diese Ausnahme nicht auch in gänzlich anderen Situationen auslösen.

Zum Abschluss seien **finally**-Blöcke erwähnt: Nach einem **try**-Block und beliebig vielen **catch**-Klauseln kann ein **finally**-Block stehen:

```java
try { ... }
catch(Help e) { ... }
catch(Exception e) { ... }
finally { ... }
```

Wird der **try**-Block ausgeführt, so wird in jedem Fall auch der **finally**-Block ausgeführt, unabhängig davon, ob Ausnahmen aufgetreten sind oder nicht. Tritt keine Ausnahme auf, wird der **finally**-Block unmittelbar nach dem **try**-Block ausgeführt. Tritt eine Ausnahme auf, die abgefangen wird, erfolgt die Ausführung des **finally**-Blocks unmittelbar nach der der **catch**-Klausel. Tritt eine nicht abgefangene Ausnahme im **try**-Block oder in einer **catch**-Klausel auf, wird der **finally**-Block vor Weitergabe der Ausnahme ausgeführt. Tritt während der Ausführung des **finally**-Blocks eine nicht abgefangene Ausnahme auf, wird der **finally**-Block nicht weiter ausgeführt und die Ausnahme weitergegeben. Allenfalls vorher angefallene Ausnahmen werden in diesem Fall vergessen.

Solche **finally**-Blöcke eignen sich dazu, Ressourcen auch beim Auftreten von Ausnahmen freizugeben. Dabei ist aber Vorsicht geboten, da oft nicht klar ist, ob eine bestimmte Ressource bereits vor dem Auftreten einer Ausnahme angefordert war.

### 4.1.2 Einsatz von Ausnahmebehandlungen

Ausnahmen werden in folgenden Fällen eingesetzt:

**Unvorhergesehene Programmabbrüche:** Wird eine Ausnahme nicht abgefangen, kommt es zu einem Programmabbruch. Die entsprechende Bildschirmausgabe (Stack-Trace) enthält genaue Informationen über Art und Ort des Auftretens der Ausnahme. Damit lassen sich die Ursachen von Programmfehlern leichter finden.

Ausstieg aus Sprachkonstrukten: Ausnahmen sind nicht auf den Umgang mit Programmfehlern beschränkt. Sie erlauben ganz allgemein das vorzeitige Abbrechen der Ausführung von Blöcken, Kontrollstrukturen, Methoden, etc. in außergewöhnlichen Situationen. Das Auftreten solcher Ausnahmen wird erwartet (im Gegensatz zum Auftreten unbekannter Fehler). Es ist daher relativ leicht, entsprechende Ausnahmebehandlungen durchzuführen, die eine sinnvolle Weiterführung des Programms ermöglichen.

Rückgabe alternativer Ergebniswerte: In Java und vielen anderen Sprachen kann eine Methode nur Ergebnisse eines bestimmten Typs liefern. Wenn in der Methode eine unbehandelte Ausnahme auftritt, wird an den Aufrufer statt eines Ergebnisses die Ausnahme zurückgegeben, die er abfangen kann. Damit ist es möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurückgibt, die nicht den deklarierten Ergebnistyp der Methode haben.

4.1 Ausnahmebehandlung

**Faustregel:** Aus Gründen der Wartbarkeit soll man Ausnahmen und Ausnahmebehandlungen nur in echten Ausnahmesituationen und sparsam einsetzen.


**Faustregel:** Man soll Ausnahmen nur einsetzen, wenn dadurch die Programmlogik vereinfacht wird.

Es gibt aber auch Fälle, in denen der Einsatz von Ausnahmen und deren Behandlungen die Programmlogik wesentlich vereinfachen kann, beispielsweise wenn viele bedingte Anweisungen durch eine einzige *catch*-Klausel ersetzbar sind. Wenn das Programm durch Verwendung von Ausnahmebehandlungen einfacher lesbar undverständlicher wird, ist der Einsatz durchaus sinnvoll. Das gilt vor allem dann, wenn die Ausnahmen lokal abgefangen werden. Oft sind aber gerade die nicht lokal abfangbaren Ausnahmen jene, die die Lesbarkeit am ehesten erhöhen können.

Wir wollen einige Beispiele betrachten, die Grenzfälle für den Einsatz darstellen. Im ersten Beispiel geht es um eine einfache Iteration:

```java
while (x != null)
    x = x.getNext();
```

Die Bedingung in der *while*-Schleife kann man vermeiden, indem man die Ausnahme, dass *x* gleich null ist, abfangt:

```java
try {
    while(true)
        x = x.getNext();
} catch (NullPointerException e) {} 
```

171

**Faustregel:** Bei der Verwendung von Ausnahmen müssen nicht-lokale Effekte beachtet werden.

Das nächste Beispiel zeigt geschachtelte Typabfragen:

```java
if (x instanceof T1) { ... }
else if (x instanceof T2 { ... }
...
else if (x instanceof Tn { ... }
else { ... }
```

Diese sind durch eine trickreiche, aber durchaus lesbare Verwendung von catch-Klauseln ersetzbar, die einer switch-Anweisung ähnelt:

```java
try { throw x; } 
catch(T1 x) { ... }
catch(T2 x) { ... }
...
catch(Tn x) { ... }
catch(Exception x) { ... }
```

Die zweite Variante funktioniert natürlich nur wenn T1 bis Tn Unterklassen von Exception sind. Da der try-Block nur eine throw-Klausel enthält und spätestens in der letzten Zeile jede Ausnahme gefangen wird, kann es zu keinen nicht-lokalen Effekten kommen. Nach obigen Kriterien steht einer derartigen Verwendung von Ausnahmebehandlungen nichts im Weg. Allerdings entspringen beide Varianten einem schlechten Programmierstil: Typabfragen sollen, soweit es möglich ist, vermieden werden um
4.1 Ausnahmebehandlung

Das folgende Beispiel zeigt einen Fall, in dem die Verwendung von Ausnahmen uneingeschränkt sinnvoll ist. Angenommen, die statische Methode addA addiert zwei beliebig große Zahlen, die durch Zeichenketten bestehend aus Ziffern dargestellt werden. Wenn eine Zeichenkette auch andere Zeichen enthält, gibt die Funktion die Zeichenkette "Error" zurück:

```java
public static String addA(String x, String y) {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else
        return "Error";
}
```

Diese Art des Umgangs mit Fehlern ist problematisch, da das Ergebnis jedes Aufrufs mit "Error" verglichen werden muss, bevor es weiter verwendet werden kann. Wird ein Vergleich vergessen, pflanzt sich der Fehler in andere Programmzweige fort. Ausnahmen lösen das Problem:

```java
public static String addB(String x, String y)
    throws NoNumberString {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else
        throw new NoNumberString();
}
```

In dieser Variante kann sich der Fehler nicht leicht fortpflanzen. Wenn ein bestimmter Ergebniswert fehlerhafte Programmzustände anzeigt, ist es fast immer ratsam, statt diesem Wert eine Ausnahme zu verwenden. Diese Verwendung ist zwar nicht lokal, aber spezielle Ergebniswerte würde ebenso nicht-lokale Abhängigkeiten im Programm erzeugen.

4 Kreuz und quer

4.2 Nebenläufige Programmierung

Bisher sind wir implizit davon ausgegangen, dass jedes Programm schrittweise, ein Befehl nach dem anderen, in einem Thread ausgeführt wird. Java bietet, wie die meisten anderen Programmiersprachen auch, minimale Unterstützung für die **nebenläufige Programmierung**, bei der mehrere Threads gleichzeitig nebeneinander laufen und Befehle aus verschiedenen Threads entweder (auf mehreren Prozessor-Kernen) tatsächlich oder (auf nur einem Prozessor-Kern) scheinbar gleichzeitig ausgeführt werden.

Die Programmierung wird durch nebenläufige Threads aufwendiger, da wir gelegentlich neue Threads erzeugen und kontrollieren müssen, vor allem aber, da wir nebenläufige Threads **synchronisieren** müssen um zu verhindern, dass durch gleichzeitige Zugriffe die aus Variablen gelesenen und in Variablen geschriebenen Werte inkonsistent werden.

4.2.1 Thread-Erzeugung und Synchronisation in Java

Folgendes Beispiel soll ein Synchronisationsproblem demonstrieren:

```java
public class Zaehler {
    private int i = 0, j = 0;
    public void schnipp() { i++; j++; }
}
```

Die Variablen `i` und `j` sollten stets die gleichen Werte enthalten. Wenn wir jedoch in mehreren nebenläufigen Threads `schnipp` in demselben Objekt von `Zaehler` wiederholt aufrufen, kann es vorkommen, dass sich `i` und `j` plötzlich voneinander unterscheiden. Den Grund dafür finden wir in der fehlenden Synchronisation: Bei Ausführung des `++`-Operators wird der Wert der Variablen aus dem Speicher gelesen, um eins erhöht und wieder in den Speicher geschrieben. Wird nun `schnipp` in zwei Threads annähernd gleichzeitig ausgeführt, wird von beiden Threads der gleiche Wert aus der Variablen gelesen, jeweils um eins erhöht, und von beiden Threads derselbe Wert zurückgeschrieben. Das ist nicht das, was wir haben wollen, da sich ein Variablenwert bei zwei Aufrufen nur um eins erhöht hat. Unterschiede zwischen den Werten von `i` und `j` ergeben sich, wenn das nur beim Ändern einer der beiden Variablen passiert.

Die Ausführungen von `schnipp` lassen sich in Java sehr einfach synchronisieren, indem wir `schnipp` als `synchronized` definieren:
4.2 Nebenläufige Programmierung

public synchronized void schnipp() { i++; j++; }

In jedem Objekt wird zu jedem Zeitpunkt höchstens eine synchronized Methode ausgeführt. Wenn mehrere Threads schnipp annähernd gleichzeitig aufrufen, werden alle bis auf einen Thread solange blockiert, bis dieser eine aus schnipp zurückkehrt. Dann darf der nächste Thread schnipp ausführen, und so weiter. Die oben beschriebenen Synchronisationsprobleme sind damit beseitigt. Die Methode wird atomar; also wie eine nicht weiter in Einzelteile zerlegbar Einheit ausgeführt.

So wie schnipp sollen alle Methoden, die auf Objekt- oder Klasse-variablen zugreifen, in nebendläufigen Programmen oder Programmenteilen als synchronized definiert sein, um Inkonsistenzen bei Variabenzugriffen zu verhindern. Das gilt vor allem für ändernde Zugriffe wie in obigem Beispiel. Auch bei nur lesenden Zugriffen ist häufig Synchronisation notwendig um zu verhindern, dass inkonsistente Daten gelesen werden (z.B. i schon erhöht, j aber noch nicht).


```java
public void schnipp() {
    synchronized(this) { i++; }
    synchronized(this) { j++; }
}
```

Die Ausführungen der Befehle i++ und j++ werden getrennt voneinan- der synchronisiert. Die Methode als ganze braucht nicht synchronisiert zu werden, da in ihr außerhalb von synchronized-Blöcken nirgends auf Objekt- oder Klassevariablen zugegriffen wird. In dieser Variante von schnipp ist es zwar möglich, dass i und j kurzfristig unterschiedliche
Werte enthalten (z.B. weil mehrere Threads, die im nächsten Schritt i erhöhen, früher an die Reihe kommen als jene, die j erhöhen), aber am Ende des Programms sind i und j gleich; es wird keine Erhöhung vergessen.

Zur Synchronisation verwendet Java Locking. Ein „Lock“ kann in jedem Objekt auf einen bestimmten Thread gesetzt sein um zu verhindern, dass ein anderer als dieser Thread auf das Objekt zugreift. Das Argument des synchronized-Blocks bestimmt das Objekt, dessen Lock zu setzen ist. Bei synchronized Methoden ist das immer das Objekt, in dem die Methode aufgerufen wird, also this. Dieser Mechanismus erlaubt rekursive Aufrufe: Da Locks bereits auf die richtigen Threads gesetzt sind, brauchen sich rekursive Aufrufe nicht mehr um Synchronisation zu kümmern.

Zugriffe auf eine Variable, die als volatile deklariert ist, sind seit Java 5 atomar. Mit Hilfe der Klasse AtomicInteger könnte man obiges Beispiel auch ohne synchronisierte Blöcke und ohne Locking schreiben.

Manchmal soll die Ausführung von Threads von weiteren Bedingungen abhängen, die Threads unter Umständen für längere Zeit blockieren. Die Methode onOff in folgender Klasse schaltet einen Drucker online bzw. offline und steuert damit, ob Druckaufträge an den Drucker weitergeleitet oder Threads, die den Drucker verwenden wollen, blockiert werden:

```java
public class Druckertreiber {
    private boolean online = false;
    public synchronized void drucke (String s) {
        while (!online) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        ... // schicke s zum Drucker
    }
    public synchronized void onOff() {
        online = !online;
        if (online) notifyAll();
    }
    ...
}
```

Die Methode drucke stellt sicher, dass online den Wert true hat, bevor das Argument an den Drucker weitergeleitet wird. Falls online nicht true ist, wird wait aufgerufen. Diese in Object vordefinierte Methode blockiert den aktuellen Thread so lange, bis dieser explizit wieder


Objekte der folgenden Klasse erzeugen nach Aufruf von run in einer Endlosschleife immer wieder neue Zeichenketten und schicken diese an den im Konstruktor festgelegten Druckertreiber:

```java
public class Produzent implements Runnable {
    private Druckertreiber t;
    public Produzent(Druckertreiber t) { this.t = t; }
    public void run() {
        String s = ....
        for (;;) {
            ... // produziere neuen Wert in s
            t.drucke(s); // schicke s an Druckertreiber
        }
    }
}
```
Das vordefinierte Interface Runnable spezifiziert nur run. Objekte von Klassen wie Produzent, die Runnable implementieren, können wie in folgendem Codestück zur Erzeugung neuer Threads verwendet werden:

```java
Druckertreiber t = new Druckertreiber(...);
for (int i = 0; i < 10; i++) {
    Produzent p = new Produzent(t);
    new Thread(p).start();
}
```


### 4.2.2 Nebenläufigkeit in der Praxis

In der Praxis verwendet man die grundlegenden Sprachkonzepte der Nebenläufigkeit nicht sehr häufig. Gründe dafür sind einerseits prinzipielle Schwierigkeiten im Umgang mit Nebenläufigkeit, andererseits aber auch eine Reihe vorgefertigter Lösungen für die häufigsten Aufgaben, die man nur mehr zu verwenden braucht. Damit wird die Entwicklung nebenläufiger Programme auf eine höhere Ebene verschoben.

### Aufgaben und Threads.


### Thread-sichere Datenstrukturen.

4 Kreuz und quer

griffen werden (würde zu Fehlern führen), aber bei nur einem Thread ist HashMap natürlich effizienter. Es gibt noch weitere Varianten: So erzeugt

```
Collections.synchronizedMap(new HashMap(...))
```


Für die meisten Datenstrukturen gilt Ähnliches. Auf in java.util definierte Datenstrukturen dürfen meist nicht mehrere Threads zugreifen. Durch Methoden wie synchronizedMap und synchronizedList in der Klasse Collections können diese Datenstrukturen aber mit Synchronisation ausgestattet werden, sodass sie auch in einer nebeneinläufigen Umgebung einsetzbar sind. Allerdings sind diese Datenstrukturen bei gleichzeitigen Zugriffen durch viele Threads nur wenig effizient. Vor allem sind Iteratoren über diesen Datenstrukturen bei Nebenläufigkeit nicht robust, das heißt, nach Änderungen der Datenstrukturen funktionieren sie nicht mehr vernünftig. In java.util.concurrent findet man Varianten dieser Datenstrukturen, die auch bei gleichzeitigen Zugriffen durch viele Threads noch effizient sind und etwas robustere Iteratoren bieten. Allerdings unterscheiden sich die nebeneinläufigen Datenstrukturen in vielen Details von den Varianten aus java.util. Darin spiegelt sich die Tatsache wider, dass man in der nebeneinläufigen und parallelen Programmierung zum Teil andere Datenstrukturen und Algorithmen braucht als in der sequentiellen Programmierung. Wer effiziente nebeneinläufige Programme schreiben will, muss also auf jeden Fall umlernen, auch wenn nur vorgefertigte Programmteile zum Einsatz kommen.

Oft lässt sich dieses Ziel nicht ganz erreichen. Dann muss man dafür sorgen, dass es möglichst wenige gleichzeitige Zugriffe auf gemeinsame Daten gibt, vor allem möglichst wenige gleichzeitige Schreibzugriffe. Klassen wie ConcurrentHashMap können in diesem Fall einen Ausweg bieten. Allerdings muss dabei sichergestellt sein, dass die gemeinsamen Daten keine Einschränkungen in der Ausführungsreihenfolge der Teilaufgaben bedingen, die Daten also nicht auf komplexe Weise voneinander abhängen. Man weiß ja nicht, wann genau welche Teilaufgabe ausgeführt wird.

Sind Abhängigkeiten in der Ausführungsreihenfolge unvermeidlich, wird es schwierig. Man kann versuchen, Struktur in die Einschränkungen bezüglich der Ausführungsreihenfolge zu bringen, sodass beispielsweise Klassen wie Phaser einsetzbar werden, die es erlauben, Teilaufgaben in mehreren Phasen auszuführen. Sobald man jedoch auf solche Formen der Synchronisation angewiesen ist, kann man nicht mehr so einfach die Zuteilung der Teilaufgaben an Threads einem Executor überlassen, sondern muss sich selbst darum kümmern. Jeder Versuch der Kontrolle der Ausführungsreihenfolge lässt den Schwierigkeitsgrad der Lösung rasant ansteigen. Auch wenn man die zahlreichen Synchronisationsmöglichkeiten auf höherer Ebene nützt, erreicht man bald eine Komplexität, die nahe an der Komplexität liegt, die man bei reiner Nutzung der primitiven Sprachkonstrukte der Nebenläufigkeit hätte. Ganz kann man heute auf die primitiven Sprachkonstrukte trotz zahlreicher und gut durchdachter vorgefertigter Lösungen noch nicht verzichten.

4.2.3 Synchronisation und Objektorientiertheit


 Auch zu viel Synchronisation macht sich negativ bemerkbar: Die gleichzeitige Ausführung von Threads wird verhindert und die Laufzeit des Programms möglicherweise verlängert. In Extremfällen wird die Ausführung so stark verzögert, dass überhaupt kein Fortschritt mehr möglich ist. Gefürchtet sind Deadlocks, das sind zyklische Abhängigkeiten zwischen zwei oder mehr Threads: Beispielsweise möchte ein Thread \( p \), der bereits den Lock auf ein Object \( x \) hält, auch den Lock auf ein anderes Objekt \( y \) und wartet darauf, dass ein anderer Thread \( q \) den Lock auf \( y \) freigibt. Wenn \( q \) zufällig auf das Freiwerden des Locks auf \( x \) wartet bevor jener auf \( y \) freigegeben werden kann, befinden sich \( p \) und \( q \) in einem Deadlock und warten ewig aufeinander. Übliche Techniken zur Vermeidung von Deadlocks bestehen in der Verhinderung solcher Zyklen durch eine lineare Anordnung aller Objekte im System: Locks dürfen nur in dieser Reihenfolge angefordert werden. Wenn \( x \) in der Anordnung vor \( y \) steht, ist zwar Thread \( p \) erlaubt, nicht jedoch \( q \), da ein Thread, der bereits einen Lock auf \( y \) hält, keinen Lock auf \( x \) mehr anfordern darf. Leider ist eine lineare Anordnung in der Praxis viel einschränkender als man oft glaubt: Dadurch werden alle Arten von zyklischen Strukturen verhindert, bei deren Abarbeitung Synchronisation nötig sein könnte. Oft nimmt man für solche Strukturen die Gefahr von Deadlocks in Kauf. Neben Deadlocks gibt es eine Reihe weiterer Ursachen für unerwünschte (vorübergehende oder dauerhafte) gegenseitige Behinderungen von Threads durch Synchronisation. Eigenschaften, die die Abwesenheit solcher unerwünschter gegenseitiger Behinderungen betreffen (neben Deadlocks beispielsweise auch Livelocks und Starvation), nennt man zusammengefasst Liveness-Properties.

 Manchmal wird empfohlen, Liveness-Properties wie geforderte nicht-funktionale Eigenschaften eines Programms zu behandeln. Das bedeutet, dass man sich beim Schreiben des Programms zunächst nicht darum küm-
mert, sondern erst durch ausgiebiges Testen Verletzungen dieser Eigenschaften zu finden versucht. Treten beim Testen keine Probleme (mehr) auf, nimmt man die Eigenschaften als erfüllt an.


Um Synchronisation in Untertypbeziehungen einzubeziehen muss man vor allem Client-kontrollierte History-Constraints berücksichtigen. Synchronisation bewirkt ja Einschränkungen auf der Reihenfolge, in der Methoden abgearbeitet werden, und genau solche Einschränkungen werden durch Client-kontrollierte History-Constraints dargestellt. Das impliziert,

Abhängigkeiten, die durch die notwendige Synchronisation in die Software eingeführt werden, stehen auch der direkten Wiederverwendung von Code, also der Vererbung oft im Weg \[26\]. Dafür gibt es zwar einige Lösungsansätze, die aber allesamt nicht überzeugen können, vor allem weil zumindest einige davon in Widerspruch zur Ersetzbarkeit stehen.


4.3 Annotationen und Reflexion


4.3.1 Annotationen und Reflexion in Java

Syntax. Zur klaren Unterscheidung von anderen Sprachkonstrukten beginnt jede Annotation in Java mit dem Zeichen „@“. Sie steht unmittelbar
vordem Programmteil, auf den sie sich bezieht. Beispielsweise können wir die Annotation \texttt{@Override} vor eine Methodendefinition schreiben. Der Compiler prüft, ob die Methodendefinition mit dieser Annotation versehen ist und verlangt nur in diesem Fall, dass die Methode eine andere Methode überschreibt. Ein Compiler, der \texttt{@Override} nicht versteht, könnte diese Überprüfung theoretisch auch weglassen. Aber praktisch jeder Compiler versteht \texttt{@Override}, da es sich dabei um eine vom System vorgegebene Annotation handelt. Wir können auch eigene Annotationen erfinden. Außerdem können Annotationen Argumente enthalten. Eigene Annotationen müssen deklariert werden, bevor sie verwendbar sind. Dafür hat man die Syntax von Interface-Definitionen übernommen und leicht abgewandelt. Hier ist ein Beispiel für die Definition einer etwas komplexeren Annotation:

\begin{verbatim}
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
    String who(); // author of bug fix
    String date(); // when was bug fixed
    int level(); // importance level 1-5
    String bug(); // description of bug
    String fix(); // description of fix
}
\end{verbatim}

Solche Annotationen kann man zu Klassen, Interfaces und Enums hinzufügen um auf Korrekturen hinzuweisen. Das könnte so aussehen:

\begin{verbatim}
@BugFix(who="Kaspar", date="1.10.2013", level=3,
    bug="class unnecessary and maybe harmful",
    fix="content of class body removed")
public class Buggy { }
\end{verbatim}

Die einzelnen Einträge in der Definition der Annotation beschreiben also Datenfelder, die in der Annotation gesetzt werden. Auch wenn diese Einträge syntaktisch wie Methodendeklarationen aussehen, gibt es doch deutliche Unterschiede zu normalen Methoden. Die Parameterlisten müssen immer leer sein, und die erlaubten Ergebnistypen sind stark eingeschränkt: Nur elementare Typen (wie int), Enum-Typen, String, Class und andere Annotationen sowie eindimensionale Arrays dieser Typen sind erlaubt. Wie im Beispiel wird häufig String verwendet.
Die Definition von BugFix ist selbst mit zwei Annotationen versehen. An ihnen fällt auf, dass die Argumente in den runden Klammern nicht die Form name=wert haben, sondern nur einfache Werte darstellen. Eine syntaktische Vereinfachung erlaubt das Weglassen des Namens wenn die Annotation nur ein Argument namens value hat. Ebenso können die runden Klammern weggelassen werden wenn die Annotation keine Argumente hat, etwa bei @Override. Das Argument von @Target ist ein Array, die geschwungenen Klammern stellen also ein simples Aggregat zur Initialisierung eines (in diesem Fall einelementigen) Arrays dar. Bei einelementigen Arrays kann man die geschwungenen Klammern weglassen.

Annotationen auf der Definition von Annotationen haben folgende Bedeutungen: Das Argument von @Target legt fest, was annotiert werden kann. Es ist ein Array von Werten des Enums ElementType mit Werten wie METHOD, TYPE, PARAMETER, CONSTRUCTOR und so weiter; siehe das API von ElementType. Die gerade definierte Annotation kann an alle Sprachelemente angeheftet werden, die im Array vorkommen. Ohne @Target ist die gerade definierte Annotation überall anheftbar.

@Retention legt fest, wie weit die gerade definierte Annotation sichtbar bleiben soll. Mit dem Wert SOURCE der Enum RetentionPolicy wird die Annotation vom Compiler genauso verworfen wie Kommentare. Solche Annotationen sind nur für Werkzeuge, die auf dem Source-Code operieren, von Interesse. Der Wert CLASS sorgt dafür, dass die Annotation in der übersetzten Klasse vorhanden bleibt, aber während der Programm­ausführung nicht mehr sichtbar ist. Das ist nützlich für Werkzeuge, die auf dem Byte-Code operieren. Schließlich sorgt der Wert RUNTIME dafür, dass die Annotation auch zur Laufzeit zugreifbar ist.

Weiters sind in der Definition einer Annotation noch die parameterlosen Annotationen @Documented und @Inherited verwendbar. Erstere sorgt dafür, dass die Annotation in der generierten Dokumentation vorkommt, letztere dafür, dass das annotierte Element auch in einer Unter­klasse als annotiert gilt.

Man kann Default-Belegungen für Parameter von Annotationen angeben. Beispielsweise wird die Definition von BugFix um folgende Zeile (innerhalb der geschwungenen Klammern) erweitert:

```java
String comment() default "";
```

Aufgrund der Default-Belegung braucht man bei der Verwendung kein Argument für comment angeben. Das ist besonders dann sinnvoll, wenn
4 Kreuz und quer

die Zeile erst später hinzugefügt wird, das heißt, wenn @BugFix schon vorher verwendet wurde. Alle diese Stellen der Verwendung braucht man durch die Default-Belegung nicht unbedingt zu ändern.

Verwendung zur Laufzeit. Wurde über @Retention(RUNTIME) festgelegt, dass eine Art von Annotationen auch zur Laufzeit zugreifbar ist, dann generiert der Compiler ein entsprechendes Interface. Dieses sieht für obiges Beispiel so aus:

```java
public interface BugFix extends Annotation {
    String who();
    String date();
    int level();
    String bug();
    String fix();
}
```

Man kann wie in folgendem Code-Stück auf die Annotation zugreifen:

```java
String s = "";
BugFix a = Buggy.class.getAnnotation(BugFix.class);
if (a != null) { // null if no such Annotation
    s += a.who() + " fixed a level " + a.level() + " bug";
}
```


Obiges Code-Stück ist nur sinnvoll verwendbar, wenn man genau weiß, auf welche Annotation man zugreifen möchte. Durch getAnnotations kann man alle Annotationen der Klasse gleichzeitig auslesen, ohne deren genaue Typen zu kennen:
4.3 Annotationen und Reflexion

Annotation[] as = Buggy.class.getAnnotations();
for (Annotation a : as) {
    if (a instanceof BugFix)
        String s = ((BugFix)a).who; ...
}

Aber mit dem so erzeugten Array von Annotationen kann man meist auch nur dann weiterarbeiten, wenn man weiß, welche Annotationen man haben möchte. Auf die Datenfelder kann man ja nur zugreifen, wenn man vorher einen Cast auf den richtigen Typ der Annotation macht.


Ähnliches gilt für Informationen über Variablen in Objekten des Typs Field. Man kann über Reflection die Werte sichtbarer Variablen lesen und schreiben, und natürlich kann man auch Annotationen abfragen.

Reflexion ist in der Programmierung eine äußerst mächtige Technik. Man kann sehr flexibel fast alles zur Laufzeit entscheiden, was üblicher-
weise schon beim Programmieren vor dem Compilieren festgelegt werden muss; nur das Ändern von Klassen zur Laufzeit ist in Java verboten. Genau in dieser Flexibilität liegt jedoch die größte Gefahr der Reflexion. Das Programm wird gänzlich undurchschaubar und kaum wartbar, wenn man diese Freiheit auf unkontrollierte Weise nutzt.

4.3.2 Anwendungen von Annotationen und Reflexion


Häufig verwendet man @Override. Statt dieser Annotation wäre auch ein Modifier sinnvoll gewesen, aber aufgrund der geschichtlichen Entwicklung hat sich eine Annotation angeboten. Man kann jede Annotation als Modifier sehen, den ein (Pre-)Compiler oder das Laufzeitsystem versteht.

Manchmal stolpert man über eine @Deprecated-Annotation, mit der Programmelemente, die man nicht mehr verwenden sollte, gekennzeichnet werden. Eigentlich stellt sie nur eine Form von Kommentar dar. Die Annotation ermöglicht jedoch, dass ein Compiler bei Verwendung dieser Programmelemente eine Warnung ausgibt.

Eine gefährliche Rolle spielt @SuppressWarnings. Diese Annotation weist den Compiler an, alle Warnungen zu unterdrücken, die im Argument durch Zeichenketten beschrieben sind. Auch wenn manche Warnung lästig ist, sollte man von der Verwendung solcher Annotationen Abstand nehmen. Warnungen haben ja einen Grund, den man nicht vernachlässigen sollte. Es kommt gar nicht so selten vor, dass man eine solche Annotation „vorübergehend“ in den Code schreibt, weil man sich erst später um ein Problem kümmern möchte, und dann darauf vergisst. Auch wenn


Hier ist ein einfaches Beispiel für den Einsatz von Reflexion:

```java
static void execAll(String n, Object... objs) {
    for (Object o : objs) {
        try { o.getClass().getMethod(n).invoke(o); }
        catch (Exception ex) {...}
    }
}
```

In den als Parametern übergebenen Objekten wird jeweils eine parameterlose Methode aufgerufen, die einen ebenfalls als Parameter übergebenen Namen hat. Dieser Name kann genauso wie die Objekte zur Laufzeit bestimmt werden. Auf den ersten Blick geht das ganz einfach. Bei genauem Hinsehen fällt auf, dass bei den Aufrufen einiges passieren kann, mit dem man vielleicht nicht rechnet, ganz zu schweigen von der Gefahr, dass wir nicht wissen, was die mit `invoke` aufgerufenen Methoden machen. Möglicherweise ist die Methode nicht `public`, verlangt (andere) Argumente, oder existiert gar nicht. In diesen Fällen werden Ausnahmen geworfen, die wir irgendwie abfangen müssen.

Wie das Beispiel zeigt, ist die Verwendung der Reflexion im Grunde sehr einfach. Schwierigkeiten verursacht nur das ganze Rundherum, z.B. der notwendige Umgang mit vielen Sonderfällen, die in der normalen Programmierung vom Compiler ausgeschlossen werden. Die Gefahr kommt hauptsächlich daher, dass wir keinerlei Verhaltensbeschreibungen der mit
invoke aufgerufenen Methoden haben. Wir haben nicht einmal intuiti-
ve Vorstellungen davon. Bei entsprechender Organisation könnte man alle
diese Probleme lösen. Aber die Erfahrung zeigt, dass die reflexive Pro-
grammierung dennoch gefährlich ist und zu Wartungsproblemen führt.

Ein Spezialbereich. Wir betrachten nun JavaBeans-Komponenten als Bei-
pispiel für den Einsatz von Reflexion und Annotationen. JavaBeans ist ein
Werkzeug, mit dem grafische Oberflächen ganz einfach aus Komponen-
ten aufgebaut werden. Der Großteil der Arbeit wird von Werkzeugen bzw.
fertigen Klassen erledigt. JavaBeans-Komponenten sind gewöhnliche Klas-
sen, die bestimmte Namenskonventionen einhalten.

Ein JavaBeans-Konzept sind „Properties“, deren Werte von außen zu-
greifbar sind. Solche Properties führt man ein, indem man entsprechende

```java
public void setProp(int x) { ... }
public int getProp() { ... }
```

nehmen die Werkzeuge automatisch an, dass prop eine Property des Typs
int ist. Existiert nur eine dieser Methoden, ist die Property nur les- oder
schreibbar. Lesbare Properties des Typs boolean können statt mit get
auch mit is beginnen. Die grafische Oberfläche eines Werkzeugs erlaubt
über Menüs simple Zugriffe auf Properties. Dabei findet und verwendet es
Properties über Reflexion. Fast alle nötige Information steckt in den Na-
men, Ergebnistypen und Parametertypen der Methoden. Zum Auffinden
komplexerer Konzepte wie „Events“ geht man ähnlich vor.

In seltenen Fällen benötigt man Information, die nicht leicht über Refle-
xion verfügbar ist. Dafür gibt es z.B. die @ConstructorProperties-
Annotation: In übersetzten Klassen kann man kaum feststellen, welche
Parameter eines Konstruktors welcher Property entsprechen. Die Argu-
mente einer solchen Annotation zählen einfach die Properties entsprechend
der Parameterreihenfolge auf und machen diese Information dadurch über
Reflexion zugänglich. Diese Information ist im Zusammenhang mit Java-
Beans sinnvoll. Wird die Klasse nicht als JavaBean verwendet, stört die
Annotation nicht; sie wird einfach ignoriert.

In der Java-EE (Enterprise-Edition) verwendet man EJB (Enterprise-
JavaBeans) als Komponentenmodell mit vielen Möglichkeiten zur Dar-
stellung von Geschäftslogiken, hauptsächlich in Web-Anwendungen. Da-
bei kommen Annotationen in großem Stil zum Einsatz (mehr als 30 ver-
schiedene). Beispielsweise verwendet man @TransactionAttribute

4.4 Aspektorientierte Programmierung

Ein wesentliches Designziel bei der Programmierung ist die Aufteilung der Funktionalität eines Programms auf einzelne Module (\textit{separation of concerns}). In der objektorientierten Programmierung funktioniert die Aufteilung gut für die Kernfunktionalitäten (\textit{core concerns}), die sich leicht in eine Klasse packen lassen. Oft gibt es aber Anforderungen, die alle Bereiche eines Programms betreffen (\textit{cross-cutting concerns}, Querschnittsfunktionalitäten). Die aspektorientierte Programmierung kapselt Verhalten, das mehrere Klassen betrifft, in Aspekten. Ein Aspekt beschreibt dabei an einer Stelle sowohl eine Funktionalität, als auch alle Stellen im Programm, an denen diese Funktionalität angewendet werden soll.


Die aspektorientierte Programmierung bietet eine Lösung. Wir verwenden AspectJ (\url{www.eclipse.org/aspectj}), es gibt aber für fast alle anderen Programmiersprachen ebenso aspektorientierte Erweiterungen (z.B. AspectL für Lisp und LOOM.NET für C#). In AspectJ kapselt ein
4 Kreuz und quer

Aspekt vollständig alle Teile von Cross-Cutting-Concerns, die Implementierung der Funktionalität und Spezifikationen der Stellen, wo diese angewendet werden soll. Dazu werden in AspectJ folgende Begriffe verwendet:

**Join-Point** Ein Join-Point ist eine identifizierbare Stelle während einer Programmausführung wie z.B. der Aufruf einer Methode oder der Zugriff auf ein Objekt.

**Pointcut** Ein Pointcut ist ein Programmkonstrukt, das einen Join-Point auswählt und kontextabhängige Information dazu sammelt, z.B. die Argumente eines Methodenaufrufs oder das Zielobjekt.

**Advice** Ein Advice ist jener Programmc ode, der vor (before()), um (around()) oder nach (after()) dem Join-Point ausgeführt wird.

**Aspect** Ein Aspekt ist wie eine Klasse das zentrale Element in AspectJ. Ein Aspekt enthält alle Deklarationen, Methoden, Pointcuts und Advices.

Anhand des folgenden Programmstückes soll gezeigt werden, welche Stellen in einem Programm Join-Points sein können.

```java
01 public class Test{
02 Methodenausf. public static void main(String[]a) {
03 Konstruktorauf. Point pt1 = new Point(0,0);
04 Methodenauf. pt1.incrXY(3,6);
05 }
06 }
07
08 Klasseninit public class Point {
09 Objektinit private int x;
10 private int y;
11 public Point(int x, int y) {
12 Feldzugr.(write) this.x = x;
13 this.y = y;
14 }
15 public void incrXY(int dx, int dy){
16 Feldzugr.(read) x = this.x + dx;
17 y += dy;
18 }
19 }
```

194
4.4 Aspektorientierte Programmierung

In Zeile 02 ist ein möglicher Join-Point die Ausführung des Beginns der Methode `main`, in Zeile 03 der Aufruf des Konstruktors `Point` und in Zeile 12 der schreibende Zugriff auf des Feld `this.x`.

In einem Pointcut werden Join-Points definiert. In einem Advice können anonyme Pointcuts definiert werden, meistens werden aber benannte Join-Points verwendet. Die Syntax sieht folgendermaßen aus:

\[
[Sichtbarkeit] \text{pointcut Name ([Argumente]) : Pointcuttyp(Signatur)}
\]


```java
public pointcut AddListener() :
    call(* javax..*.add*Listener(EventListener+))
```

Bei der Signatur steht ein

- `*` für eine beliebige Anzahl von Zeichen außer einem `,`
- `..` für eine beliebige Anzahl jedes beliebigen Zeichens,
- `+` für jeden Untertyp eines Typs,
- `!` (Negation) für alle Join-Points außer dem Spezifizierten,
- `|` für die Vereinigungsmenge von Join-Points und
- `&&` für die Durchschnittsmenge von Join-Points.

Es gibt mehrere Arten von Pointcut-Typen:

- **execution(MethodSignature)** Ausführung einer Methode
- **call(MethodSignature)** Aufruf einer Methode
- **execution(ConstructorSignature)** Ausführung eines Konstruktors
- **call(ConstructorSignature)** Aufruf eines Konstruktors
- **get(FieldSignature)** lesender Feldzugriff
- **set(FieldSignature)** schreibender Feldzugriff
4 Kreuz und quer

**staticinitialization**(*TypeSignature*) Initialisierung einer Klasse

**preinitialization**(*ConstructorSignature*) erster Schritt der Initialisierung eines Objekts

**initialization**(*ConstructorSignature*) Initialisierung eines Objekts

**handler**(*TypeSignature*) Ausführung einer Ausnahmenbehandlung

Folgende Pointcuts sind kontrolflussbasiert:

**cflow**(*Pointcut*) alle dem Kontrollfluss entsprechenden Join-Points eines Pointcuts inklusive des äußersten

**cflowbelow**(*Pointcut*) alle Join-Points innerhalb eines Pointcuts, die dem Kontrollfluss entsprechen

Folgende Pointcuts sind sichtbarkeitsbasiert:

**within**(*Typepattern*) alle Join-Points innerhalb des lexikalischen Sichtbereichs einer Klasse oder eines Aspekts

**withincode**(*Method/ConstructorSignature*) alle Join-Points im lexikalischen Sichtbereich der Methode oder des Konstruktors

Der Pointcut `set private float Account._balance` beschreibt alle schreibenden Zugriffe auf ein privates Feld vom Typ `float` der Klasse `Account` mit dem Namen `_balance`.

Der folgende Pointcut

```
call(* java.io.PrintStream.print*(..)) && !within(TraceAspect)
```

schließt alle Pointcuts innerhalb von `TraceAspect` von der spezifizierten Menge der Pointcuts (alle Aufrufe von Methoden aus `PrintStream` die mit `print` beginnen) aus.

In einem *Advice* wird angegeben, welche Anweisungen an den ausgewählten Join-Points ausgeführt werden können. Ein Advice hat die Form

```
before() : Pointcut {Programmcode}
```

In diesem Fall wird der Programmcode vor dem Pointcut ausgeführt. Wird das Schlüsselwort `after()` verwendet, wird der Programmcode nach dem
4.4 Aspektorierte Programmierung

Pointcut ausgeführt. Mit dem Schlüsselwort about() kann Programm-
kode um den Pointcut ausgeführt werden. Dabei kann der originale Pro-
grammcode z.B. komplett umgangen werden oder mit anderen Argumenten
ausgeführt werden.

Im folgenden Programmstück wird ein Advice mit einem anonymen
Pointcut und einem mit Namen versehenen Pointcut gezeigt:

```java
before() : call(* Account.*(..)) {Benutzer überprüfen}

pointcut connectionOperation(Connection connection) :
    call(* Connection.*(..) throws SQLException);
before(Connection connection) :
    connectionOperation(connection) {
        System.out.println("Operation auf " + connection);
    }
```

Der Advice {Benutzer überprüfen} wird vor jedem Aufruf einer
Methode mit beliebigem Ergebnistyp und beliebiger Signatur der Klasse
Account ausgeführt. Der zweite Advice zeigt wie Informationen über ein
Argument eines Pointcuts und eines Advices weitergereicht werden kann.
Der Pointcut connectionOperation beschreibt dabei alle Aufrufe von
Methoden der Klasse Connection, die eine SQLException Ausnahme
werfen können.

Ein Aspekt fasst Deklarationen, Pointcuts und Advices zusammen. Er
sieht genau so wie ein Klasse aus, nur das Schlüsselwort class ist durch
das Schlüsselwort aspect ersetzt:

```java
public aspect JoinPointTraceAspect {
    private int _callDepth = -1;

    pointcut tracePoints(): !within(JoinPointTraceAspect);

    before() : tracePoints() {
        _callDepth++;
        print("Before", thisJoinPoint);
    }

    after() : tracePoints() {
        _callDepth--;
        print("After", thisJoinPoint);
    }
}
```
private void print(String prefix, Object message) {
    for(int i=0, spaces=_callDepth*2; i<spaces; i++) {
        System.out.print(" ");
    }
    System.out.println(prefix + ": " + message);
}

Der Aspekt JoinPointTraceAspect gibt alle Join-Points eines Programms aus. Die Ausgabe wird der Schachtelungstiefe entsprechend eingeraubt. Dabei wird in einem before advice die Einrücktiefe erhöht, in einem after advice die Einrücktiefe wieder erniedrigt und die Art des Join-Points ausgegeben. thisJoinPoint ist dabei ein Zeiger auf das Join-Point-Objekt mit allen Informationen über einen Join-Point. Wird dieser Aspekt auf das vorher beschriebene Testprogramm Point angewendet, wird folgende Ausgabe erzeugt.

Before: staticinitialization(Test.<clinit>)
After: staticinitialization(Test.<clinit>)
Before: execution(void Test.main(String[]))
    Before: staticinitialization(Point.<clinit>)
After: staticinitialization(Point.<clinit>)
    Before: preinitialization(Point<int, int>)
After: preinitialization(Point<int, int>)
    Before: initialization(Point<int, int>)
        Before: execution(Point<int, int>)
    After: set(int Point.x)
    Before: set(int Point.x)
    After: set(int Point.y)
    Before: set(int Point.y)
    After: execution(Point<int, int>)
After: initialization(Point<int, int>)
After: call(Point<int, int>)
    Before: call(void Point.incrXY(int, int))
    Before: execution(void Point.incrXY(int, int))
        Before: get(int Point.x)
    After: get(int Point.x)
    After: set(int Point.x)


4.5 Wiederholungsfragen

1. Wie werden Ausnahmebehandlungen in Java unterstützt?
2. Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?
3. Wozu kann man Ausnahmen verwenden? Wozu soll man sie verwenden, wozu nicht?
4. Durch welche Sprachkonzepte unterstützt Java die nebenläufige Programmierung? Wozu dienen diese Sprachkonzepte?
5. Wozu brauchen wir Synchronisation? Welche Granularität sollen wir dafür wählen?
6. Zu welchen Problemen kann Synchronisation führen, und was kann man dagegen tun?
8. Was bedeutet separation of concerns?
9. Was sind core concerns, was cross cutting concerns?
10. Was sind Join-Points, Pointcuts, Advices und Aspekte?
11. An welchen Programmpunkten können sich Join-Points befinden?
5 Software-Entwurfsmuster

Entwurfsmuster (Design-Patterns) dienen der Wiederverwendung kollektiver Erfahrung. Wir wollen exemplarisch einige häufig verwendete Entwurfsmuster betrachten. Da das Thema der Lehrveranstaltung objektorientierte Programmierungstechniken sind, konzentrieren wir uns dabei auf Implementierungsaspekte und erwähnen andere in der Praxis wichtige Aspekte nur am Rande. Die Idee der Entwurfsmuster gründet sich im Wesentlichen auf ein weiters bekanntes Buch (Gang-of-Four-Buch), das allen Interessierten empfohlen wird [11]:

E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1994.

Es gibt eine Reihe neuerer Ausgaben und Übersetzungen in andere Sprachen, die ebenso empfehlenswert sind. Wir betrachten im Skriptum und in der Lehrveranstaltung nur einen kleinen Teil der im Buch beschriebenen und in der Praxis häufig eingesetzten Entwurfsmuster.

5.1 Grundsätzliches


5 Software-Entwurfsmuster


5.1.1 Bestandteile von Entwurfsmustern

Jedes Entwurfsmuster besteht hauptsächlich aus diesen vier Elementen:

**Name:** Der Name ist wichtig, damit man in einem einzigen Begriff ein Problem, dessen Lösung und Konsequenzen daraus ausdrücken kann. Damit wird der Softwareentwurf auf eine höhere Ebene verlagert; man braucht nicht mehr jedes Detail einzeln anzusprechen. Es ist gar nicht leicht, geeignete Namen für Entwurfsmuster zu finden, die jede(r) Entwickler(in) mit dem Entwurfsmuster assoziert. Solche Namen müssen sich im Laufe der Zeit gegenüber anderen durchsetzen. Beispielsweise haben wir in Abschnitt 3.4.2 ein Entwurfsmuster namens *Visitor-Pattern* kennengelernt. Ohne lange darüber nachdenken zu müssen, sollten wir diesen Begriff gleich mit mehrfachem dynamischem Binden, Visitor- und Element-Klassen, einer grundlegenden Implementierungstechnik dahinter, der Möglichkeit zur Vermeidung dynamischer Typabfragen und Typumwandlungen, aber auch dem Problem der hohen Anzahl an Methoden verbinden können. Erst dadurch, dass uns so viel dazu einfällt, wird der Begriff zu einem geeigneten Namen für ein Entwurfsmuster. Typisch ist auch die Abstraktion über eine konkrete Problemstellung: Es geht nicht nur um fressende Tiere, sondern um einen breiten Anwendungsbereich, der auch kovariante Probleme einschließt. Wegen der Breite und Wichtigkeit des Anwendungsbereichs wurde das Visitor-Pattern (wie auch alle anderen wichtigen Entwurfsmuster) in der Fachliteratur eingehend analysiert, mit ähnlichen Techniken verglichen und schließlich auch häufig angezweifelt. Heute kennt man mehrere ganz unterschiedliche Varianten, auf die wir hier nicht näher eingehen. Gerade diese
umfangreiche Diskussion hat den Begriff erst wirklich etabliert. Genaugenommen verbindet man mit dem Begriff nicht mehr nur eine einzige, eng umrisse Technik, sondern eine ganze Fülle ähnlicher Lösungsansätze in einem Anwendungsbereich, deren Eigenschaften man sehr gut kennt.


Lösung: Das ist die Beschreibung einer bestimmten Lösung der Problemstellung. Diese Beschreibung ist allgemein gehalten, damit sie leicht an unterschiedliche Situationen angepasst werden kann. Sie soll jene Einzelheiten enthalten, die zu den beschriebenen Konsequenzen führen, aber nicht mehr.

Im Beispiel des Visitor-Patterns enthält die Beschreibung Erklärungen dafür, wie die Klassenstrukturen aussehen, welche Abhängigkeiten zwischen den Klassen bestehen und wie sich bestimmte Methoden darin verhalten. Meist gibt es nicht nur eine einzige „empfohlene“ Struktur, sondern mehrere, einander ähnliche Varianten. Man kann jene Variante wählen, die am ehesten zur konkreten Aufgabenstellung passt. Je nach Entwurfmuster kann aus einer mehr oder

Konsequenzen: Das ist eine Liste von Eigenschaften der Lösung. Man kann sie als eine Liste von Vor- und Nachteilen der Lösung betrachten, muss dabei aber aufpassen, da ein und dieselbe Eigenschaft in manchen Situationen einen Vorteil darstellt, in anderen einen Nachteil und in wieder anderen irrelevant ist.


5.1 Grundsätzliches

Fautstregel: Entwurfs muster sollen zur Abschätzung der Konsequenzen von Designentscheidungen eingesetzt werden, können aber nur in begrenztem Ausmaß und mit Vorsicht als Bausteine zur Erzielung bestimmter Eigenschaften dienen.

5.1.2 Iterator als Beispiel


Name und Problemstellung lassen sich ganz kurz so umreißen: Ein Iterator, auch Cursor genannt, ermöglicht den sequentiellen Zugriff auf die Elemente eines Aggregats (das ist eine Sammlung von Elementen, beispielsweise eine Collection), ohne die innere Darstellung des Aggregats offenzulegen.

Dieses Entwurfsmuster ist verwendbar, um

- auf den Inhalt eines Aggregats zugreifen zu können, ohne die innere Darstellung offen legen zu müssen;
- mehrere (gleichzeitige bzw. überlappende) Abarbeitungen der Elemente in einem Aggregat zu ermöglichen;
- eine einheitliche Schnittstelle für die Abarbeitung verschiedener Aggregatstrukturen zu haben, das heißt, um polymorphe Iterationen zu unterstützen.

Bei der Beschreibung der meisten Entwurfsmustern folgt hier ein typisches kleines Anwendungsbeispiel. Da wir mit solchen Beispielen schon vertraut sind, verzichten wir ausnahmsweise darauf.


Iteratoren haben drei wichtige Eigenschaften:

- Sie unterstützen unterschiedliche Varianten in der Abarbeitung von Aggregaten. Für komplexe Aggregate wie beispielsweise Bäume gibt
5.1 Grundsätzliches

es zahlreiche Möglichkeiten, in welcher Reihenfolge die Elemente abgearbeitet werden. Es ist leicht, mehrere Iteratoren für unterschiedliche Reihenfolgen auf demselben Aggregat zu implementieren.


Es gibt zahlreiche Möglichkeiten zur Implementierung von Iteratoren. Hier sind einige Anmerkungen zu Implementierungsvarianten:

• Man kann zwischen internen und externen Iteratoren unterscheiden. Interne Iteratoren kontrollieren selbst, wann die nächste Iteration erfolgt, bei externen Iteratoren bestimmen die Anwender, wann sie das nächste Element abarbeiten möchten. Alle Beispiele zu Iteratoren, die wir bis jetzt betrachtet haben, sind externe Iteratoren, bei denen Anwender in einer Schleife nach dem jeweils nächsten Element fragen. Ein interner Iterator enthält die Schleife selbst. Der Anwender übergibt dem Iterator eine Routine, die vom Iterator auf allen Elementen ausgeführt wird.

Externe Iteratoren sind flexibler als interne Iteratoren. Zum Beispiel ist es mit externen Iteratoren leicht, zwei Aggregate miteinander zu vergleichen. Mit internen Iteratoren ist das schwierig. Andererseits sind interne Iteratoren oft einfacher zu verwenden, da eine Anwendung die Logik für die Iterationen (also die Schleife) nicht braucht.
Interne Iterationen spielen vor allem in der funktionalen Programmierung eine große Rolle, da es dort gute Unterstützung für die dynamische Erzeugung und Übergabe von Routinen (in diesem Fall Funktionen) an Iteratoren gibt, andererseits aber externe Schleifen nur umständlich zu realisieren sind. In der objektorientierten Programmierung werden hauptsächlich externe Iteratoren eingesetzt.

- Oft ist es schwierig, externe Iteratoren auf Sammlungen von Elementen zu verwenden, wenn diese Elemente zueinander in komplexen Beziehungen stehen. Durch die sequentielle Abarbeitung geht die Struktur dieser Beziehungen verloren. Beispielsweise erkennt man an einem vom Iterator zurückgegebenen Element nicht mehr, an welcher Stelle in einem Baum das Element steht. Wenn die Beziehungen zwischen den Elementen bei der Abarbeitung benötigt werden, ist es meist einfacher, interne statt externer Iteratoren zu verwenden. Beispielsweise können wir die Methode \texttt{max} in \texttt{CollectionOps2} (siehe Abschnitt 3.1.3) als internen Iterator betrachten, der eine durch das Argument spezifizierte Methode – in diesem Fall einen Vergleich – auf die Elemente des Aggregats anwendet.


- Es kann gefährlich sein, ein Aggregat zu verändern, während es von einem Iterator durchwandert wird. Wenn Elemente dazugefügt oder entfernt werden, passiert es leicht, dass Elemente nicht oder doppelt...


\section*{5.2 Erzeugende Entwurfsmuster}


\subsection*{5.2.1 Factory Method}


In diesem Beispiel ist der Typ des zu erzeugenden Objekts unter Zuhilfenahme eines Objekts von DocCreator in einer Variablen als zentraler Ablage festgelegt. Aus Gründen der Einfachheit haben die Konstruktoren der Dokumente hier keine Parameter. Wenn Sie welche hätten, dann könnten entsprechende Argumente auf zahlreiche Arten festgelegt werden:

- Argumente von allgemeinem Interesse (nicht spezifisch für bestimmte Dokumente) können an newDoc übergeben und über create an den (nicht näher bekannten) Konstruktor weitergeleitet werden.


- Am einfachsten ist es, für bestimmte Dokumente spezifische aber unveränderliche Argumente fix in die Methode create einzucodieren.

Generell ist das Entwurfsmuster anwendbar wenn

- eine Klasse neue Objekte erzeugen soll, deren Klasse aber nicht selbst kennt;

- eine Klasse möchte, dass ihre Unterklassen die Art der Objekte bestimmen, welche die Klasse erzeugt;
5.2 Erzeugende Entwurfsmuster

- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren, und man das Wissen, an welche Unterklasse delegiert wird, lokal halten möchte;

- die Allokation und Freigabe von Objekten zentral in einer Klasse verwaltet werden soll.

Das Entwurfsmuster hat folgende Struktur:


Factory-Methods haben unter anderem folgende Eigenschaften:


- Sie verknüpfen parallele Klassenhierarchien, die Creator-Hierarchie mit der Product-Hierarchie. Beispielsweise ist die Klassenstruktur


Es ist manchmal sinnvoll, der Factory-Method Parameter mitzugeben, die bestimmen, welche Art von Produkt erzeugt werden soll. In diesem Fall bietet die Möglichkeit des Überschreibens noch mehr Flexibilität.

Hier ist eine Anwendung von Factory-Methods mit lazy initialization:

```java
class Creator {
    private Product product = null;
    protected abstract Product createProduct();
    public Product getProduct() {
        if (product == null)
            product = createProduct();
        return product;
    }
}
```

Ein neues Objekt wird nur einmal erzeugt. Die Methode getProduct gibt bei jedem Aufruf dasselbe Objekt zurück.

5.2 Erzeugende Entwurfsmuster

5.2.2 Prototype

Das Entwurfsmuster *Prototype* dient dazu, die Art eines neu zu erzeugenden Objekts durch ein Prototyp-Objekt zu spezifizieren. Neue Objekte werden durch Kopieren dieses Prototyps erzeugt.

Zum Beispiel kann man in einem System, in dem verschiedene Arten von Polygonen wie Dreiecke und Rechtecke vorkommen, ein neues Polygon durch Kopieren eines bestehenden Polygons erzeugen. Das neue Polygon hat dieselbe Klasse wie das Polygon, von dem die Kopie erstellt wurde. An der Stelle im Programm, an der der Kopiervorgang aufgerufen wird (sagen wir in einem Zeichenprogramm), braucht diese Klasse nicht bekannt zu sein. Das neue Polygon kann etwa durch Ändern seiner Größe oder Position einen vom kopierten Polygon verschiedenen Zustand erhalten:

```java
public Polygon duplicate(Polygon orig) {
    Polygon copy = orig.clone();
    copy.move(X_OFFSET, Y_OFFSET);
    return copy;
}
```

Generell ist dieses Entwurfsmuster anwendbar, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden, und wenn

- die Klassen, von denen Objekte erzeugt werden sollen, erst zur Laufzeit bekannt sind, oder

- vermieden werden soll, eine Hierarchie von „Creator“-Klassen zu erzeugen, die einer parallelen Hierarchie von „Product“-Klassen entspricht (also Factory-Method vermieden werden soll), oder

- jedes Objekt einer Klasse nur wenige unterschiedliche Zustände haben kann; es ist oft einfacher, für jeden möglichen Zustand einen Prototyp zu erzeugen und diese Prototypen zu kopieren, als Objekte durch `new` zu erzeugen und dabei passende Zustände anzugeben.


Prototypes haben unter anderem folgende Eigenschaften:

- Sie verstecken die konkreten Produktklassen vor den Anwendern (Client) und reduzieren damit die Anzahl der Klassen, die Anwender kennen müssen. Die Anwender brauchen nicht geändert zu werden, wenn neue Produktklassen dazukommen oder geändert werden.

- Prototypen können auch zur Laufzeit jederzeit dazugegeben und weggenommen werden. Im Gegensatz dazu darf die Klassenstruktur zur Laufzeit in der Regel nicht verändert werden.

5.2 Erzeugende Entwurfsmuster


Es ist schwer den Überblick über ein System zu behalten, das viele Prototypen enthält. Das gilt vor allem für Prototypen, die zur Laufzeit dazukommen. Zur Lösung dieses Problems haben sich Prototyp-Manager bewährt, das sind assoziative Datenstrukturen (kleine Datenbanken), in denen nach geeigneten Prototypen gesucht wird.


5.2.3 Singleton

Das Entwurfsmuster Singleton sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf dieses Objekt.


Dieses Entwurfsmuster ist anwendbar wenn

- es genau ein Objekt einer Klasse geben soll, und dieses global zugreifbar sein soll;
- die Klasse durch Vererbung erweiterbar sein soll, und Anwender die erweiterte Klasse ohne Änderungen verwenden können sollen.


Singletons haben unter anderem folgende Eigenschaften:

- Sie erlauben den kontrollierten Zugriff auf das einzige Objekt.
5.2 Erzeugende Entwurfsmuster

• Sie vermeiden durch Verzicht auf globale Variablen unnötige Namen und weitere unangenehme Eigenschaften globaler Variablen.

• Sie unterstützen Vererbung.

• Sie verhindern, dass irgendwo Instanzen außerhalb der Kontrolle der Klasse erzeugt werden.

• Sie erlauben auch mehrere Instanzen. Man kann die Entscheidung zugunsten nur eines Objekts im System jederzeit ändern und auch die Erzeugung mehrerer Objekte ermöglichen. Die Klasse hat weiterhin vollständige Kontrolle darüber, wie viele Objekte erzeugt werden.

• Sie sind flexibler als statische Methoden, da statische Methoden kaum Änderungen erlauben und dynamisches Binden nicht unterstützen.

Einfache Implementierungen dieses Entwurfsmusters bereiten keinerlei Schwierigkeiten, wie folgendes Beispiel zeigt:

```java
public class Singleton {
    private static Singleton singleton;
    protected Singleton() {
        // soll nicht von außen aufgerufen werden
        singleton = null;
    }
    public static Singleton instance() {
        if (singleton == null)
            singleton = new Singleton();
        return singleton;
    }
}
```


Überraschenderweise ist die Implementierung eines solchen Singletons gar nicht mehr einfach. Die folgende Lösung ist noch am einfachsten, wenn auch vielleicht nicht ganz zufriedenstellend:
5 Software-Entwurfsmuster

public class Singleton {
    private static Singleton singleton = null;
    protected Singleton() { ... }
    public static Singleton instance(int kind) {
        if (singleton == null)
            switch (kind) {
            case 1: singleton = new SingletonA(); break
            case 2: singleton = new SingletonB(); break
            default: singleton = new Singleton();
            }
        return singleton;
    }
}
public class SingletonA extends Singleton {
    protected SingletonA() { ... }
}
public class SingletonB extends Singleton {
    protected SingletonB() { ... }
}

Nur der erste Aufruf von instance wählt die zu verwendende Alternative. Nach Erzeugung des Objekts hat kind keinerlei Bedeutung. Um die feste Verdrahtung der Alternativen in Singleton zu vermeiden kann man instance in den Untertypen implementieren:

public class Singleton {
    protected static Singleton singleton = null;
    protected Singleton() { ... }
    public static Singleton instance() {
        if(singleton==null) singleton = new Singleton();
        return singleton;
    }
}
public class SingletonA extends Singleton {
    protected SingletonA() { ... }
    public static Singleton instance() {
        if(singleton==null) singleton = new SingletonA();
        return singleton;
    }
}

218
5.3 Entwurfsmuster für Struktur und Verhalten


5.3 Entwurfsmuster für Struktur und Verhalten


5.3.1 Decorator

Das Entwurfsmuster Decorator, auch Wrapper genannt, gibt Objekten dynamisch zusätzliche Verantwortlichkeiten (siehe Abschnitt 1.4.3). Decorators stellen eine flexible Alternative zur Vererbung bereit.

Manchmal möchte man einzelnen Objekten zusätzliche Verantwortlichkeiten geben, nicht aber der ganzen Klasse. Zum Beispiel möchte man einem Fenster am Bildschirm Bestandteile wie einen Scroll-Bar geben, anderen Fenstern aber nicht. Es ist sogar üblich, dass der Scroll-Bar dynamisch während der Verwendung eines Fensters nach Bedarf dazukommt und auch wieder weggenommen wird:


5 Software-Entwurfsmuster

public interface Window {
    void show(String text);
}

public class WindowImpl implements Window {
    public void show(String text) { ... }
}

public abstract class WinDecorator implements Window {
    protected Window win;
    public void show(String text) { win.show(text); }
}

public class ScrollBar extends WinDecorator {
    public ScrollBar(Window w) { win = w; }
    public void scroll(int lines) { ... }
    public Window noScrollBar() {
        Window w = win;
        win = null; // no longer usable
        return w;
    }
}

Window w = new WindowImpl(); // no scroll bar
ScrollBar s = new ScrollBar(w); // add scroll bar
w = s; // s aware of scroll bar, w not
w.show("Text"); // no matter if scroll bar or not
s.scroll(3); // works only with scroll bar
w = s.noScrollBar(); // remove scroll bar

Im Allgemeinen ist dieses Entwurfsmuster anwendbar

- um dynamisch Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte dadurch zu beeinflussen;

- für Verantwortlichkeiten, die wieder entzogen werden können;

- wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind, beispielsweise um eine sehr große Zahl an Unterklassen zu vermeiden, oder weil die Programmiersprache in einem speziellen Fall keine Vererbung unterstützt (beispielsweise bei final Klassen).

Das Entwurfsmuster hat folgende Struktur, wobei der Pfeil mit einem Kästchen für Aggregation (also eine Referenz auf ein Objekt, dessen Bestandteil das die Referenz enthaltende Objekt ist) steht:

Decorators haben folgende Eigenschaften:

- Sie bieten mehr Flexibilität als statische Vererbung. Wie bei stati-
5 Software-Entwurfsmuster


- Sie vermeiden Klassen, die bereits weit oben in der Klassenhierarchie mit Methoden und Variablen überladen sind. Es ist nicht notwendig, dass „ConcreteComponent“ die volle gewünschte Funktionalität enthält, da durch das Hinzufügen von Dekoratoren gezielt neue Funktionalität verfügbar gemacht werden kann.


- Sie führen zu vielen kleinen Objekten. Ein Design, das Dekoratoren häufig verwendet, führt nicht selten zu einem System, in dem es viele kleine Objekte gibt, die einander ähneln. Solche Systeme sind zwar einfach konfigurierbar, aber schwer zu verstehen und zu warten.


Dekoratoren eignen sich gut dazu, die Oberfläche beziehungsweise das Erscheinungsbild eines Objekts zu erweitern. Sie sind nicht gut für inhärente Erweiterungen geeignet. Auch für Objekte, die von Grund auf
5.3 Entwurfsmuster für Struktur und Verhalten

umfangreich sind, eignen sich Dekoratoren kaum. Für solche Objekte sind
andere Entwurfsmuster, beispielsweise Strategy, besser geeignet. Auf diese
Entwurfsmuster wollen wir hier aber nicht eingehen.

5.3.2 Proxy

Ein Proxy, auch Surrogate genannt, stellt einen Platzhalter für ein anderes
Objekt dar und kontrolliert Zugriffe darauf.

Es gibt zahlreiche, sehr unterschiedliche Anwendungsmöglichkeiten für
Platzhalterobjekte. Ein Beispiel ist ein Objekt, dessen Erzeugung teuer
ist, weil umfangreiche Daten geladen werden. Man erzeugt das eigentli-
che Objekt erst, wenn es wirklich gebraucht wird. Stattdessen verwendet
man in der Zwischenzeit einen Platzhalter, der erst bei Bedarf durch das
eigentliche Objekt ersetzt wird. Falls nie auf die Daten zugegriffen wird,
erspart man sich den Aufwand der Objekterzeugung:

```java
public interface Something {
    void doSomething();
}

public class ExpensiveSomething implements Something {
    public void doSomething() { ... }
}

public class VirtualSomething implements Something {
    private ExpensiveSomething real = null;
    public void doSomething() {
        if (real == null)
            real = new ExpensiveSomething();
        real.doSomething();
    }
}
```

Jedes Platzhalterobjekt enthält im Wesentlichen einen Zeiger auf das ei-
gentliche Objekt (sofern dieses existiert) und leitet in der Regel Nach-
richten an das eigentliche Objekt weiter, möglicherweise nachdem weitere
Aktionen gesetzt wurden. Einige Nachrichten werden manchmal auch di-
rekt vom Proxy behandelt.

Das Entwurfsmuster ist anwendbar, wenn eine intelligente Referenz
auf ein Objekt als ein simpler Zeiger nötig ist. Hier sind einige übliche
Situationen, in denen ein Proxy eingesetzt werden kann (keine vollständige
Aufzählung):
Remote-Proxies sind Platzhalter für Objekte, die in anderen Namensräumen (zum Beispiel auf Festplatten oder auf anderen Rechnern) existieren. Nachrichten an die Objekte werden von den Proxies über komplexere Kommunikationskanäle weitergeleitet.

Virtual-Proxies erzeugen Objekte bei Bedarf. Da die Erzeugung eines Objekts aufwendig sein kann, wird sie so lange verzögert, bis es wirklich einen Bedarf dafür gibt.

Protection-Proxies kontrollieren Zugriffe auf Objekte. Derartige Proxies sind sinnvoll, wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.

Smart-References ersetzen einfache Zeiger. Sie können bei Zugriffen zusätzliche Aktionen ausführen. Typische Verwendungen sind

- das Mitzählen der Referenzen auf das eigentliche Objekt, damit das Objekt entfernt werden kann, wenn es keine Referenz mehr darauf gibt (Reference-Counting);
- das Laden von persistenten Objekten in den Speicher, wenn das erste Mal darauf zugegriffen wird (wobei die Unterscheidung zu Virtual-Proxies manchmal unklar ist);
- das Zuschern, dass während des Zugriffs auf das Objekt kein gleichzeitiger Zugriff durch einen anderen Thread erfolgt (beispielsweise durch Setzen eines „Locks“).

Es gibt zahlreiche weitere Einsatzmöglichkeiten. Der Phantasie sind hier kaum Grenzen gesetzt.

Die Struktur dieses Entwurfsmusters ist recht einfach:
Die abstrakte Klasse oder das Interface „Subject“ definiert die gemeinsame Schnittstelle für Objekte von „RealSubject“ und „Proxy“. Objekte von „RealSubject“ und „Proxy“ können gleichermaßen verwendet werden, wo ein Objekt von „Subject“ erwartet wird. Die Klasse „RealSubject“ definiert die eigentlichen Objekte, die durch die Proxies (Platzhalter) repräsentiert werden. Die Klasse „Proxy“ definiert schließlich die Proxies. Diese Klasse

- verwaltet eine Referenz „realSubject“, über die ein Proxy auf Objekte von „RealSubject“ (oder auch andere Objekte von „Subject“) zugreifen kann;

- stellt eine Schnittstelle bereit, die der von „Subject“ entspricht, damit ein Proxy als Ersatz des eigentlichen Objekts verwendet werden kann;

- kontrolliert Zugriffe auf das eigentliche Objekt und kann für dessen Erzeugung oder Entfernung verantwortlich sein;

- hat weitere Verantwortlichkeiten, die von der Art abhängen.

Es kann mehrere unterschiedliche Klassen für Proxies geben. Zugriffe auf Objekte von „RealSubject“ können durch mehrere Proxies (möglicherweise unterschiedlicher Typen) kontrolliert werden, die in Form einer Kette miteinander verbunden sind.

In obiger Grafik zur Struktur des Entwurfsmusters zeigt ein Pfeil von „Proxy“ auf „RealSubject“. Das bedeutet, „Proxy“ muss „RealSubject“ kennen. Dies ist notwendig, wenn ein Proxy Objekte von „RealSubject“ erzeugen soll. In anderen Fällen reicht es, wenn „Proxy“ nur „Subject“ kennt, der Pfeil also auf „Subject“ zeigt.

In der Implementierung muss man beachten, wie man auf ein Objekt zeigt, das in einem anderen Namensraum liegt oder noch gar nicht existiert. Für nicht existierende Objekte könnte man zum Beispiel `null` verwenden und für Objekte in einer Datei den Dateinamen.

5.3.3 Template-Method

Eine Template-Method definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse. Template-Methods erlauben einer Unterklasse, bestimmte Schritte zu überschreiben, ohne die Struktur des Algorithmus zu ändern. Dieses Entwurfsmuster ist anwendbar

- um den unveränderlichen Teil eines Algorithmus nur einmal zu implementieren und es Unterklassen zu überlassen, den veränderbaren Teil des Verhaltens festzulegen;

- wenn gemeinsames Verhalten mehrerer Unterklassen (zum Beispiel im Zuge einer Refaktorisierung) in einer einzigen Klasse lokal zusammengefasst werden soll, um Duplikate im Code zu vermeiden;

- um mögliche Erweiterungen in Unterklassen zu kontrollieren, beispielsweise durch Template-Methods, die Hooks aufrufen und nur das Überschreiben dieser Hooks in Unterklassen ermöglichen.

Die Struktur dieses Entwurfsmusters ist recht einfach:

\[
\begin{align*}
\text{AbstractClass} \\
\quad \text{templateMethod()} \\
\quad primitiveOperation1() \\
\quad primitiveOperation2() \\
\end{align*}
\]

\[
\begin{align*}
\text{ConcreteClass} \\
\quad primitiveOperation1() \\
\quad primitiveOperation2() \\
\end{align*}
\]

Die (meist abstrakte) Klasse „AbstractClass“ definiert (abstrakte) primitive Operationen und implementiert das Grundgerüst des Algorithmus, das die primitiven Operationen aufruft. Die Klasse „ConcreteClass“ implementiert die primitiven Operationen.

Template-Methods haben unter anderem folgende Eigenschaften:
5.3 Entwurfsmuster für Struktur und Verhalten

- Sie stellen eine fundamentale Technik zur direkten Wiederverwendung von Programmc ode dar (siehe Beispiele in Abschnitt 2.3.2). Sie sind vor allem in Klassenbibliotheken und Frameworks sinnvoll, weil sie ein Mittel sind, um gemeinsames Verhalten zu faktorisieren.

- Sie führen zu einer umgekehrten Kontrollstruktur, die manchmal als Hollywood-Prinzip bezeichnet wird („Don’t call us, we’ll call you“). Die Oberklasse ruft die Methoden der Unterklassen auf – nicht wie in den meisten Fällen umgekehrt.

- Sie rufen meist nur eine von mehreren Arten von Operationen auf:
  - konkrete Operationen in „ConcreteClass“;
  - konkrete Operationen in „AbstractClass“, also Operationen, die ganz allgemein auch für Unterklassen sinnvoll sind;
  - abstrakte primitive Operationen, die einzelne Schritte im Algorithmus ausführen;
  - Factory-Methods;


Die primitiven Operationen, die von der Template-Methode aufgerufen werden, sind in der Regel protected Methoden, damit sie nicht in unerwünschten Zusammenhängen aufrufbar sind. Primitive Operationen, die überschrieben werden müssen, sind als abstract deklariert. Die Template-Methode selbst, also die Methode, die den Algorithmus implementiert, soll nicht überschrieben werden. Sie kann final sein.

Ein Ziel bei der Entwicklung einer Template-Methode sollte sein, die Anzahl der primitiven Operationen möglichst klein zu halten. Je mehr Operationen überschrieben werden müssen, desto komplizierter wird die direkte Wiederverwendung von „AbstractClass“.
5.4 Wiederholungsfragen

1. Erkläre Sie folgende Entwurfsmuster und beschreiben Sie jeweils das Anwendungsgebiet, die Struktur, die Eigenschaften und wichtige Details der Implementierung:
   - Decorator
   - Factory-Method
   - Iterator
   - Prototype
   - Proxy
   - Singleton
   - Template-Method
   - Visitor (siehe Abschnitt 3.4.2)

2. Welche Arten von Iteratoren gibt es, und wofür sind sie geeignet?

3. Wie wirkt sich die Verwendung eines Iterators auf die Schnittstelle des entsprechenden Aggregats aus?

4. Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?

5. Was ist ein robuster Iterator? Wozu braucht man Robustheit?

6. Wird die Anzahl der benötigten Klassen im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?

7. Wird die Anzahl der benötigten Objekte im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?


9. Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton?

10. Welche Unterschiede und Ähnlichkeiten gibt es zwischen Decorator und Proxy?

12. Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)

13. Kann man mehrere Decorators bzw. Proxies hintereinander verketten? Wozu kann so etwas gut sein?

14. Was unterscheidet Hooks von abstrakten Methoden?
5 Software-Entwurfsmuster
Literaturverzeichnis


Literaturverzeichnis


