F. Steimann
Mitarbeit: D. Keller

Moderne Programmiertechniken und -methoden

Kurseinheiten 1

mathematik
und
informatik

FernUniversität in Hagen

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

002 193 752 (04/09)

01853-5-01-S 1

Alle Rechte vorbehalten
© 2011 FernUniversität in Hagen
Fakultät für Mathematik und Informatik

Inhaltsverzeichnis

Vorwort ... **i**

Übersicht ... **ii**

1 Interfacebasierte Programmierung ... **1**

1.1 Der Begriff des Interfaces ... *1*

1.2 Interfaces als Typen ... *3*
1.2.1 Explizite Interfaceimplementierung ... 5
1.2.2 Nominale vs. strukturelle Typkonformität ... 7
1.2.3 Interfaces vs. abstrakte Klassen ... 7

1.3 Eigenschaften von Interfaces ... *8*
1.3.1 Aufrufen und aufgerufen werden: die zwei Seiten eines Interfaces ... 8
1.3.2 Totale und partielle Interfaces ... 10
1.3.3 Öffentliche vs. veröffentlichte Interfaces ... 11

1.4 Anzeichen interfacebasierter Programmierung ... *13*

1.5 Arten des Gebrauchs von Interfaces ... *17*
1.5.1 Übersicht ... 17
1.5.2 Anbietende Interfaces ... 17
1.5.3 Allgemeine Interfaces ... 18
1.5.4 Idiosynkratische Interfaces ... 18
1.5.5 Familieninterfaces ... 20
1.5.6 Kontextspezifische Interfaces ... 20
1.5.7 Client/Server-Interfaces ... 21
1.5.8 Ermöglichende Interfaces ... 22
1.5.9 Server/Client-Interfaces ... 22
1.5.10 Server/Item-Interfaces ... 25
1.5.11 Zusammenfassung ... 27

1.6 Dependency injection ... *27*
1.6.1 Constructor injection ... 28
1.6.2 Setter injection ... 29
1.6.3 Interface injection ... 29
1.6.4 Assembler ... 30
1.6.5 Einschränkungen ... 31
1.6.6 Alternativen ... 31
1.6.7 Fazit ... 32

1.7 Umkehrung von Abhängigkeiten mit Interfaces ... *32*

1.8 Interpretation von Interfaces als Rollen ... *36*

1.9 Werkzeugunterstützung für das interfacebasierte Programmieren ... *37*

1.10 Weiterführende Literatur ... *38*

1.11 Lösungen zu den Selbsttestaufgaben ... *39*

2 Design by contract **41**

2.1 Verhaltensspezifikation durch Zusicherungen: Vor- und Nachbedingungen, Invarianten *42*

2.2 Ein paar einfache Beispiele *45*

2.3 Design by contract in der Analysephase *47*

2.4 Design by contract in der Programmierung *48*

2.4.1 Zeitpunkt der Überprüfung von Zusicherungen 48

2.4.2 Beeinflussung der Programmierung 50

2.5 Zusicherungen und Vererbung *51*

2.6 Spezifikationssprachen für das Design by contract *54*

2.6.1 EIFFEL 55

2.6.2 JAVA 57

2.6.3 ICONTRACT 60

2.6.4 JML 62

2.6.5 Grenzen der Ausdrucksstärke 65

2.7 Design by contract als Form des Testens *65*

2.8 Vor- und Nachteile des Design by contract *66*

2.9 Zusammenfassung und Ausblick *68*

2.10 Weiterführende Literatur *68*

2.11 Lösungen der Selbsttestaufgaben *69*

3 Unit-Testen **71**

3.1 Der Begriff des Testfalls *71*

3.1.1 Was testet ein Testfall? 73

3.1.2 Organisation von Testfällen 74

3.2 JUNIT *74*

3.2.1 Interner Aufbau des JUNIT-3.8-Frameworks 75

3.2.2 Die Anwendung von JUNIT 85

3.2.3 Änderungen in JUNIT 4 88

3.2.4 Änderungen mit JUNIT 4.4 89

3.2.5 Probleme von JUNIT 91

3.3 Vererbung von Testfällen *92*

3.4 Das Testen von Interfaces *92*

3.5 Testen mit Mock-Objekten *95*

3.5.1 Ausprogrammierte Mock-Objekte 96

3.5.2 Mock-Frameworks 99

3.5.3 Grenzen der Einsetzbarkeit von Mock-Objekten 101

3.5.4 Mock-Objekte bei ermöglichenden Interfaces 102

3.6 Auswertung von Unit-Tests zur Fehlerlokalisierung *102*

3.6.1 Abdeckungsbasierte Fehlerlokalisierung 103

3.6.2 Modellbasierte Fehlerlokalisierung 104

3.6.3 Andere Arten von Fehlerlokatoren 105

3.6.4 Kombination von Fehlerlokatoren ... 105

3.7 Kontinuierliches Testen ... 106

3.8 Wer testet die Tests? ... 106

3.9 Unit-Testen, Design by contract, Typprüfung – drei Wege, ein Ziel ... 108

3.10 Weiterführende Literatur ... 110

3.11 Lösungen der Selbsttestaufgaben ... 111

4 Entwurfsmuster ... 113

4.1 Historisches ... 113

4.2 Übergeordnete objektorientierte Programmierprinzipien ... 114

4.2.1 Offene Rekursion und das Vererbungsinterface ... 115
4.2.2 Vererbung vs. Komposition ... 117
4.2.3 Forwarding vs. Delegation ... 118

4.3 Definition ... 119

4.4 Wichtige Entwurfsmuster ... 119

4.4.1 COMPOSITE Pattern ... 120
4.4.2 OBSERVER Pattern ... 127
4.4.3 TEMPLATE METHOD Pattern ... 131
4.4.4 STRATEGY Pattern ... 132
4.4.5 ROLE OBJECT Pattern ... 133
4.4.6 FACTORY METHOD Pattern ... 135
4.4.7 ADAPTER Pattern ... 140
4.4.8 FACADE Pattern ... 142
4.4.9 VISITOR Pattern ... 143

4.5 Bewertung ... 149

4.6 Ausblick ... 150

4.7 Weiterführende Literatur ... 151

4.8 Lösungen der Selbsttestaufgaben ... 151

5 Refactoring ... 153

5.1 Einordnung ... 154

5.1.1 Katalogisierung ... 154
5.1.2 Refaktorisierungen als Algorithmen ... 155
5.1.3 Refactoring to patterns ... 156
5.1.4 Werkzeugunterstützung ... 157
5.1.5 Ein Beispiel ... 158
5.1.5.1 Vorbedingungen ... 159
5.1.5.2 Durchführung ... 161
5.1.5.3 Nachbedingungen ... 162

5.2 Eine Auswahl von Refactorings ... 162

5.2.1 Bedingungen vereinfachen ... 163

5.2.1.1 Verschachtelte Bedingungen durch Wächter ersetzen (REPLACE NESTED CONDITIONAL WITH GUARD CLAUSES) ... 163
5.2.1.2 Bedingung zerlegen (DECOMPOSE CONDITIONAL) ... 165
5.2.1.3 Bedingung durch Polymorphismus ersetzen (REPLACE CONDITIONAL WITH POLYMORPHISM) ... 166
5.2.1.4 Einführung eines Nullobjekts (INTRODUCE NULL OBJECT) ... 168
5.2.1.5 Zusicherung einfügen (INTRODUCE ASSERTION) ... 170
5.2.2 Lesbarkeit verbessern ... 171
5.2.2.1 Methode oder Variable umbenennen ... 171
5.2.2.2 Parameterklasse einführen ... 172
5.2.2.3 Konstruktor durch eine Factory-Methode ersetzen (REPLACE CONSTRUCTOR WITH FACTORY METHOD) ... 172
5.2.2.4 Fehlercode durch Ausnahme ersetzen (REPLACE ERROR CODE WITH EXCEPTION) ... 173
5.2.2.5 Ausnahme durch Vorbedingung ersetzen (REPLACE EXCEPTION WITH PRECONDITION) ... 175
5.2.2.6 Ausnahme durch Test ersetzen (REPLACE EXCEPTION WITH TEST) ... 176
5.2.3 Daten organisieren ... 177
5.2.3.1 Feld kapseln (ENCAPSULATE FIELD) ... 177
5.2.3.2 Collections kapseln (ENCAPSULATE COLLECTION) ... 179
5.2.3.3 Attributwert durch Objekt ersetzen (REPLACE DATA VALUE WITH OBJECT) 180
5.2.3.4 Wert durch Referenz ersetzen (CHANGE VALUE TO REFERENCE) ... 182
5.2.3.5 Klasse extrahieren (EXTRACT CLASS) ... 183
5.2.3.6 Unidirektionale in bidirektionale Verknüpfung ändern (CHANGE UNIDIRECTIONAL ASSOCIATION TO BIDIRECTIONAL) ... 185
5.2.4 Generalisierung einsetzen ... 187
5.2.4.1 Superklasse extrahieren (EXTRACT SUPERCLASS) ... 188
5.2.4.2 Feld oder Methode nach oben verschieben (PULL UP FIELD/METHOD) ... 188
5.2.4.3 Feld oder Methode nach unten verschieben (PUSH DOWN FIELD/METHOD) 189
5.2.4.4 Subklasse extrahieren (EXTRACT SUBCLASS) ... 189
5.2.4.5 Interface extrahieren (EXTRACT INTERFACE) ... 191
5.2.4.6 Interface berechnen (INFER TYPE) ... 192
5.2.4.7 Subklassen durch Felder ersetzen (REPLACE SUBCLASS WITH FIELDS) ... 193
5.2.4.8 Vererbung durch Delegation ersetzen (REPLACE INHERITANCE WITH DELEGATION) ... 194
5.2.5 Methoden organisieren ... 196
5.2.5.1 Methode extrahieren (EXTRACT METHOD) ... 196
5.2.5.2 Methode in Methodenobjekt auslagern (REPLACE METHOD WITH METHOD OBJECT) 199
5.2.5.3 Feld oder Methode verlagern (MOVE FIELD/METHOD) ... 201
5.2.5.4 Delegat verbergen (HIDE DELEGATE): das Gesetz Demeters (Law of Demeter) 202
5.2.5.5 Mittelsmann entfernen (REMOVE MIDDLEMAN) ... 204
5.2.5.6 Klassenfremde Methode einführen (INTRODUCE FOREIGN METHOD) ... 204
5.2.5.7 Lokale Erweiterung einführen (INTRODUCE LOCAL EXTENSION) ... 205

5.3 Zusammenfassung und Ausblick ... 205

5.4 Weiterführende Literatur ... 206

5.5 Lösungen der Selbsttestaufgaben *207*

6 Metaprogrammierung **209**

6.1 Metaprogrammierung auf sich selbst: Reflektion *211*
6.1.1 Reflektieren ohne zu verändern: Introspektion 211
6.1.2 Introspektion in JAVA 212
6.1.3 Interzession 213
6.1.4 Modifikation 214
6.1.5 Bewertung der Reflektion 214

6.2 Programmieren mit Metadaten: Annotationen und Attribute *215*
6.2.1 Annotationstypen 216
6.2.2 Annotationsinstanzen und deren Verwendung 217
6.2.3 Annotationsverarbeitung zur Übersetzungszeit 218

6.3 Aspektorientierte Programmierung *219*
6.3.1 Entwicklungsgeschichtliche Einordnung 220
6.3.2 Inhalte von Aspekten 222
6.3.3 Charakterisierung der aspektorientierten Programmierung 224
6.3.4 Aspektorientierte Programmierung und Modularisierung 229
6.3.5 Aspektorientierte Programmierung und Lesbarkeit 231
6.3.6 ASPECTJ 232
6.3.6.1 Die wichtigsten Programmierkonstrukte von ASPECTJ 233
6.3.6.2 Ein größeres Beispiel: Das OBSERVER Pattern als Aspekt 236

6.4 Zusammenfassung und Ausblick *238*

6.5 Weiterführende Literatur *239*

6.6 Lösungen der Selbsttestaufgaben *240*

7 Extreme Programming **241**

7.1 Geschichte des Extreme Programming *242*

7.2 Ziele des Extreme Programming *244*

7.3 Der Test-first-Ansatz *244*

7.4 Das Programmieren in Paaren *246*

7.5 Keine Planung *248*

7.6 Die Kundin vor Ort *250*

7.7 Gemeinsame Verantwortung *251*

7.8 Der Prozeß des Extreme Programming *252*

7.9 Voraussetzungen für den Einsatz von Extreme Programming *254*

7.10 Werkzeuge des Extreme Programming *256*

7.11 Extreme Programming als risikogetriebene Methode *257*

7.12 Zusammenfassung *259*

7.13 Übergang zu agilen Prozessen *260*

7.14 Weiterführende Literatur 261

Vorwort

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

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

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

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

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

Übersicht

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

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

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

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

Auch wenn es viele nicht wahrhaben wollen: Die Programmierung ist ein evolutionärer Prozeß, bei dem einmal getroffene Entscheidungen ständig auf die Probe gestellt werden und gegebenenfalls wieder geändert werden müssen. Die Ände-

rung von Code ist aber in der Regel eine verzweigte und entsprechend verzwickte Angelegenheit: Nur selten ist es mit einer Änderung an einer Stelle getan. Das führt dann häufig zu der Erkenntnis, daß Software unter Änderung verdirbt. Dem sollen sog. *Refactorings*, standardisierte und teilweise auch automatisierte Änderungen von Code, die dessen Bedeutung nicht verändern, entgegenwirken. Ja noch viel mehr: Mit Hilfe von Refactorings soll sich der Entwurf (und damit die Qualität) existierender Software durch gezielte Änderungen nachträglich verbessern lassen.

Als Abschluß der Programmiertechniken wird noch ein anderes Thema aufgegriffen, das — unter einem neuen Deckmäntelchen namens *aspektorientierte Programmierung* — derzeit für Aufsehen sorgt: die *Metaprogrammierung*. Unter Metaprogrammierung versteht man zunächst die Erstellung von Programmen, die Programme erzeugen oder ändern. Besonders interessant wird die Metaprogrammierung dann, wenn ein Programm sich selbst zu verändern in der Lage ist. Dies ist in gewisser Weise bei der aspektorientierten Programmierung der Fall. Mit einer Behandlung dieser und den ebenfalls erst vor kurzem aktuell gewordenen Annotationen schließt diese Kurseinheit.

Unit-Tests und Refactorings sind zwei Programmiertechniken, die im Rahmen einer bestimmten Programmiermethode größere Bekanntheit erlangt haben, nämlich des *Extreme Programming*. Extreme Programming vereint eine Vielzahl relativ unorthodoxer Herangehensweisen zu einem Gesamtkunstwerk, dessen Praxistauglichkeit in der Vergangenheit viel diskutiert wurde. Ohne daß heute ein abschließendes Ergebnis vorläge, hat das Extreme Programming aber immerhin über das Wort von den „agilen Methoden“ oder Prozessen zu einer immer stärker werdenden Abkehr von schwergewichtigen Softwareentwicklungsansätzen und damit, zumindest aus Sicht der Programmiererinnen, zu einer Art Befreiung geführt. Auch wenn die agile Softwareentwicklung längst nicht auf alle Projekte und Organisationen paßt, so läßt sich doch sicher die eine oder andere Anregung daraus mitnehmen.

1 Interfacebasierte Programmierung

> Program to an interface, not an implementation.
>
> Erstes Prinzip wiederverwendbaren objektorientierten Designs, aus [1].

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

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

1.1 Der Begriff des Interfaces

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

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

Die bekanntesten frühen Arbeiten zu Modulen und Interfaces sowie dem damit verbundenen Geheimnisprinzip („Information Hiding“) stammen von Parnas. Parnas argumentiert, daß jede Entwurfsentscheidung in einem Modul gefaßt und damit hinter einer Schnittstelle (Interface) verborgen werden soll, die sich bei einer (späteren) Änderung der Entwurfsentscheidung nicht mit ändert. Die Entwurfsentscheidung wird somit zum Geheimnis des Moduls und die Auswirkungen der Entwurfsentscheidung bleiben lokal begrenzt. Die Vorteile liegen auf der Hand: Änderungen, die sich in der Praxis niemals ausschließen lassen, wirken sich, bei einer vorausschauenden Modularisierung des Systems, immer nur auf Teile dessen aus; der Änderungsaufwand bleibt damit beschränkt, die unerwünschten Nebeneffekte werden minimiert. Auch wenn es in der Praxis nicht immer ganz so einfach ist, so hat der Parnassche Modularisierungsansatz doch unbestreitbar Vorteile.

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

Beispiel

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

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

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

---

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

Klassen als Module

Mit dem Aufkommen der objektorientierten Programmierung wurde das Modulkonzept weitgehend durch den Klassenbegriff ersetzt.[2] Eine Klasse hat eine Schnittstelle, hinter der sie ihre Entwurfsentscheidung, im wesentlichen die Implementierung der Methoden, verbirgt (bzw. verbergen kann). Während jedoch in Sprachen wie ADA und MODULA die Schnittstellenspezifikation von der Implementierung getrennt (etwa in verschiedenen Abschnitten des Quelltextes oder sogar in verschiedenen Dateien) vorgenommen wurde, wird in Programmiersprachen wie JAVA oder C# die Schnittstelle einer Klasse gemeinsam mit ihrer Implementierung spezifiziert. Dazu werden lediglich den Attributen bzw. Methodennamen sog. *Access modifier* vorangestellt, die angeben, ob ein Element der Klasse öffentlich zugänglich und damit Teil ihrer Schnittstelle sein soll oder ob der Zugriff nur eingeschränkt (im Rahmen der Möglichkeiten der jeweiligen Programmiersprache) erfolgen können soll. Im folgenden nennen wir Schnittstellenspezifikationen, die durch öffentliche Access modifier (in JAVA `public`) an Ort und Stelle der Implementierung deklariert werden, **Klasseninterfaces**.

Schnittstellen von Klassen

Klasseninterface einer Klasse

Beispiel

Das Klasseninterface der JAVA-Klasse

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

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

1.2 Interfaces als Typen

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

---

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

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

Interfaces als Ersatz für fehlende Mehrfachvererbung?

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

Entwicklung der Wahrnehmung und des Gebrauchs von Interfaces

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

Eine weitere interessante Entwicklung ist ebenfalls Abbildung 1.1 zu entnehmen: das Verhältnis von interfacetypisierten zu klassentypisierten Variablen. Dieses hat sich nämlich nur einmal sprunghaft verändert, und zwar (wiederum) beim Wechsel von JAVA 1 auf JAVA 2. Neben SWING ist dafür diesmal auch das JAVA-2-Collection-Framework verantwortlich, dessen Interfaces (wie beispielsweise `List`) zwar nur selten implementiert werden, die jedoch sehr häufig in Variablendeklarationen auftauchen. Diese Beobachtung deutet bereits darauf hin, daß

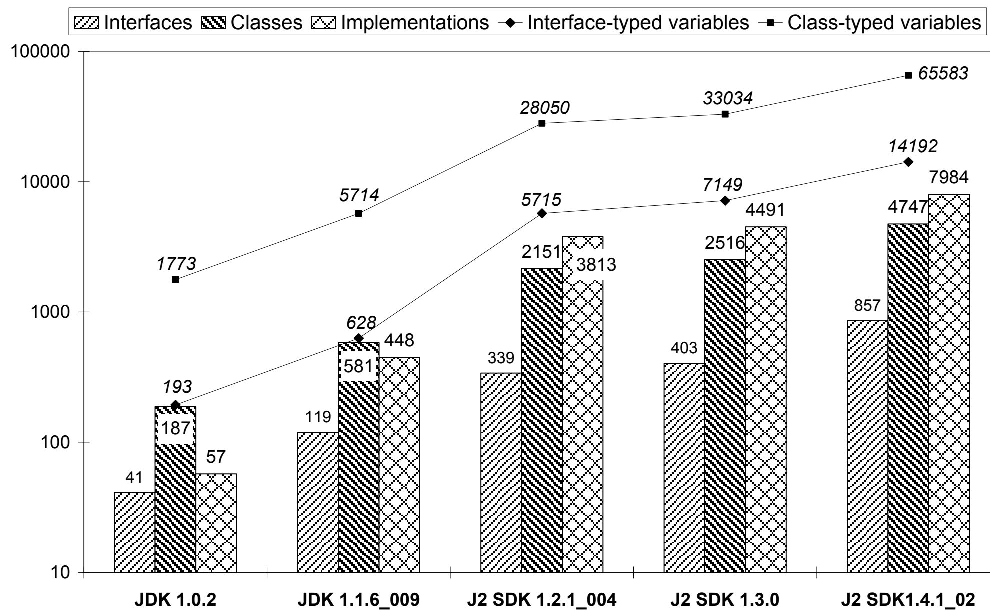

Abbildung 1.1: Entwicklung des Einsatzes von Interfaces im JDK über die Zeit. Man beachte die logarithmische Skala: gleiche Differenzen in der Höhe einzelner Säulen entsprechen gleichen Verhältnissen. So ist das Verhältnis von Klassen zu Interfaces über alle Versionen in etwa konstant geblieben (ca.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

es durchaus unterschiedliche Arten der Verwendung von Interfaces gibt, ein Umstand, der Gegenstand von Abschnitt 1.5 sein wird.

Selbsttestaufgabe 1.1

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

1.2.1 Explizite Interfaceimplementierung

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

Beispiel

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

```
14    interface I {
15          void m();
16    }
17    interface J {
18          void m();
19    }
20    class A : I, J {
21          …
22    }
```

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

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

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

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

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

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

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

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

Beide Eigenschaften von C# , die mehrfache Implementierung der gleichen Signatur, wenn verschiedene Interfaces einer Klasse dies verlangen, und der Zugriff auf Methoden ausschließlich über Interfacetypen, stehen für das Prinzip des interfacebasierten Programmierens, das von MICROSOFT schon früh (als *Interface inheritance*) über die objektorientierte Programmierung (hier insbesondere die *Vererbung zwischen Klassen*, die *Implementation inheritance*) gestellt wurde. So ba-

siert beispielsweise der COM-Standard auf Interfaces, ohne objektorientiert zu sein, und VISUAL BASIC kam — mit Interfaces — lange ohne Klassen aus.

Zusammenfassung

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

1.2.2 Nominale vs. strukturelle Typkonformität

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

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

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

1.2.3 Interfaces vs. abstrakte Klassen

Man kann darüber diskutieren, ob Interfaces-als-Typen zwingende Voraussetzung für die interfacebasierte Programmierung sind. Tatsächlich würde dies Programmiersprachen wie C++ von der interfacebasierten Programmierung ausschließen. Dies ist schon allein deswegen nicht sinnvoll, weil es in C++ abstrakte Klassen gibt, die — genau wie Interfaces in JAVA und C# — keine Implementierungen ihrer Methoden vorgeben müssen, und weil C++ Mehrfachvererbung er-

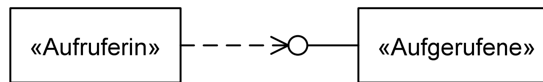

Abbildung 1.2: Aufruferin und Aufgerufene (engl. caller und called oder callee) eines Interfaces. Die Aufrufabhängigkeit wird hier (wie in UML) durch einen gestrichelten Pfeil dargestellt, die Implementierung eines Interfaces durch den Kreis mit Verbindungslinie (sog. Lollipop-Notation; das Interface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

laubt, eine Klasse also insbesondere mehrere abstrakte Klassen direkt erweitern kann (was dann ja der Implementierung mehrerer Interfaces entspräche). Der einzige Nachteil ist, daß hier nicht auf Sprachebene unterschieden werden kann, ob eine abstrakte Klasse ohne jede Implementation die Funktion einer Superklasse (*Generalisierung*) oder eines Interfaces hat.

gewichtiger Nachteil von Interfaces à la JAVA

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

1.3 Eigenschaften von Interfaces

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

1.3.1 Aufrufen und aufgerufen werden: die zwei Seiten eines Interfaces

Jedes Interface hat zwei Seiten: Es trennt die *Aufrufende* (oder *Aufruferin*) von der *Aufgerufenen* (Abbildung 1.2). Dabei sind Aufrufende und Aufgerufene zwei *Rollen* einer Beziehung, nämlich der Beziehung des Aufrufens (vgl. Abschnitt 1.8). Da die Aufruferin von der Aufgerufenen programmiertechnisch abhängig ist, spricht man auch von einer *Abhängigkeitsrelation* (engl. dependency), genauer von einer **Aufrufabhängigkeit** (engl. call dependency); wie wir allerdings noch sehen werden, kann die mit der Aufrufabhängigkeit einhergehende inhaltliche Abhängigkeit durchaus auch umgekehrt bestehen, weswegen wir den Begriff nachfolgend vermeiden wollen. Ähnlich ist es mit der Bezeichnung *Client* für die Aufruferin und *Server* für die Aufgerufene: Auch hier kann das inhaltliche Verhältnis umgekehrt, also die Aufrufende der Server und die Aufgerufene der Client sein (s. Abschnitt 1.5.8).

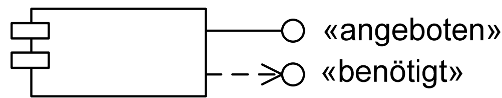

Abbildung 1.3: Angebotene (engl. provided) und benötigte (engl. required) Interfaces einer Komponente. Die Komponente ist einmal Aufruferin und einmal Aufgerufene.

Im Kontext der *komponentenbasierten Programmierung* spricht man häufig von *angebotenen* (engl. provided) und *benötigten* (engl. required) Interfaces (Abbildung 1.3). Über angebotene Interfaces wird eine Klasse oder Komponente aufgerufen, bei benötigten Interfaces ist sie selbst die Aufruferin. Aus der Sicht der Komponente bezeichnen angebotenes und benötigtes Interface also zwei verschiedene Interfaces; allerdings ist das benötigte Interface einer Komponente in der Regel das angebotene einer anderen (vgl. Abbildung 1.2).

In einem objektorientierten Programm finden sich die beiden Seiten eines Interfaces auf verschiedene Weise wieder. Die Aufgerufene (genauer: die Klasse eines aufgerufenen Objektes) implementiert ein Interface und bietet somit alle Dienste an, die in dem Interface spezifiziert sind. Die Aufruferin (der ja schließlich das Objekt benennen muß, das sie aufrufen möchte) deklariert eine Variable mit dem Interface als Typ. Dazu muß er (genauer: seine Klasse[3]) das Interface zunächst importieren. Das Objekt, das durch die Variable referenziert wird, bietet dann alle Services des Interfaces an; auf diesem Objekt können die benötigten Funktionen aufgerufen werden. Das folgende Beispiel veranschaulicht den Sachverhalt.

Beispiel

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

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

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

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

1.3.2 Totale und partielle Interfaces

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

Häufig möchte man den Zugriff auf ein Objekt für verschiedene Aufruferinnen differenzieren. In solchen Fällen benötigt eine Aufruferin nur einen Teil der angebotenen Funktionen. Ein entsprechendes Interface, das nicht den vollen Funktionsumfang der Aufgerufenen abdeckt, nennen wir im folgenden ein **partielles Interface**. Partielle Interfaces eines Objektes können disjunkt sein (sich nicht überlappen), müssen dies aber nicht.

Beispiel

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

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

Generalisierung vs. Fokussierung

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

das Interface Segregation Principle

Partielle Interfaces bedienen das sog. **Interface Segregation Principle** [9]. Es besagt, daß die Abhängigkeit einer Klasse von einer anderen auf die Teile des Interfaces beschränkt sein soll, die tatsächlich benötigt werden. Dadurch soll vermieden werden, daß alle Benutzerinnen einer Klasse über diese Klasse miteinander verkoppelt werden: Wenn eine Benutzerin eine Änderung des Interfaces erfordert, so die Befürchtung, müssen alle anderen Benutzerinnen entsprechend an-

gepaßt werden, auch wenn sie diese Änderung eigentlich gar nicht betrifft. Das Interface Segregation Principle liest sich wie folgt:

> Clients should not be forced to depend upon interfaces that they do not use.
>
> Interface Segregation Principle, Robert C. Martin, [9].

1.3.3 Öffentliche vs. veröffentlichte Interfaces

Die Implementierung eines Interfaces macht den durch das Interface abgedeckten Teil einer Klasse anderen Klassen, die das Interface verwenden, zugänglich. Unabhängig von separat definierten, implementierten Interfaces kann aber auch schon das *Klasseninterface*, das mittels der Zugriffsmodifizierer `public` etc. bei der Klassendefinition deklariert wird, einen solchen Zugriff ermöglichen. Beiden ist jedoch gemeinsam, daß die Gewährung des Zugriffs nicht dediziert erfolgen kann, daß also nicht angegeben werden kann, wer genau den Zugriff erhält. Aus Sicht der Klasse gibt es nur eine Öffentlichkeit.[4]

Berücksichtigung eines verteilten und asynchronen Softwareentwicklungsprozesses

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

Letzteres ist immer dann der Fall, wenn man für Wiederverwendung durch Dritte entwickelt, wenn also die betroffenen Klassen nicht aus einer Hand stammen. Die Öffentlichkeit besteht dann aus einer unbekannten Menge von Klassen, von denen man nichts weiß, außer daß sie mit den alten Schnittstellen funktionierten. Diese alten Schnittstellen zu ändern ist dann schlichtweg unmöglich (vgl. dazu die Ausführungen zu Interfaces vs. abstrakte Klassen in Abschnitt 1.2.3).[5] Gleichwohl können die Schnittstellen von Klassen, die *ausschließlich* voneinander abhängen, immer noch geändert werden. Die Frage ist nur, wie man feststellen kann, ob diese Ausschließlichkeit besteht.

---

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

[5] Damit verwandt ist die im MICROSOFT-Umfeld bekannte sog. *DLL hell*.

Differenzierung der Öffentlichkeit

Was man dazu benötigt, ist eine Differenzierung der Öffentlichkeit, und zwar in eine, unter die die Klassen fallen, die man selbst kontrolliert, und in eine, unter die die Klassen fallen, auf die man keinen Zugriff hat. Genau das ist mit der Unterscheidung von **öffentlichem** und **veröffentlichtem Interface** (engl. **public** bzw. **published interface**) [8] gemeint: Während die öffentliche Schnittstelle im wesentlichen dem entspricht, was man in JAVA und ähnlichen Sprachen mit dem Schlüsselwort `public` herstellt, deklariert die veröffentlichte Schnittstelle eine unveränderliche (die immer auch öffentlich ist, denn sonst hätte die Unveränderlichkeit kaum einen Nutzen). Ein spezielles Schlüsselwort gibt es dafür jedoch nicht.[6] Man ist aber frei, den Quellcode einer veröffentlichten Schnittstelle entsprechend zu kennzeichnen, z. B. mittels eines entsprechenden Kommentars oder einer *Annotation*.[7]

JAVA's Paket-Konstrukt zur Abgrenzung nur beschränkt geeignet

Nun mag man einwenden, JAVA besäße schon ein Sprachmittel, um verschiedene Grade von Öffentlichkeit herzustellen. So gibt es dort das Paket-Konstrukt, das es erlaubt, den Zugriff auf Elemente des gleichen Pakets zu beschränken, also paketlokale Öffentlichkeit zu vereinbaren (mittels des Default access modifiers, der aus dem Weglassen der anderen besteht). Da Pakete (bzw. die damit verbundenen Sichtbarkeiten) aber nicht schachtelbar sind, eröffnet das einem lediglich die Möglichkeit, alle Klassen, die untereinander Öffentlichkeit benötigen, aber nicht veröffentlicht werden sollen, in jeweils ein Paket zu verfrachten und darin mit Standardsichtbarkeit (eben paketlokal) zu versehen. Aufgrund der Transitivität der Sichtbarkeitsbeziehung (eine erste Klasse, die eine zweite sieht, die eine dritte sieht, sieht auch die dritte) ergibt sich daraus nicht selten ein großes Paket, das alles enthält, so daß der zweite Zweck von Paketen, ein Programm zu strukturieren, der Veröffentlichung von Schnittstellen geopfert werden muß. Das gegenwärtige Paketkonzept bietet also keine praktikable Unterstützung zur Unterscheidung von öffentlichen und veröffentlichten Schnittstellen.[8]

Es ist also wichtig, daß man sich im klaren darüber ist, welche der öffentlichen Schnittstellen eines Projekts man veröffentlichen will, und daß man dann darüber Klarheit herstellt. Dabei sollte man bei der Veröffentlichung von Schnittstellen eher zurückhaltend agieren, da diese schnell zu Altlasten werden können, die

---

6 Das liegt schon daran, daß die Entwicklungszeit außerhalb des Regelungsbereichs der Sprachdefinition liegt (sieht man einmal von den Möglichkeiten der Metaprogrammierung ab; s. Kurseinheit 6). So wird Quellcode in JAVA et al. immer noch in Dateien gespeichert, auf deren Änderung ein Compiler keinen Einfluß hat, so daß eine Überprüfung der Einhaltung der Unveränderlichkeit kaum stattfinden kann.

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

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

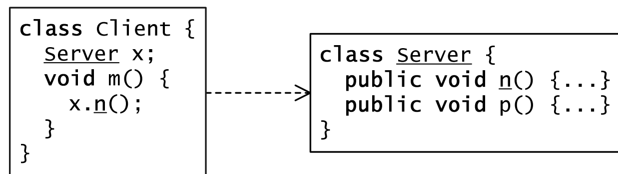

Abbildung 1.4: Die Klasse `Client` hängt unmittelbar von der Klasse `Server` ab, da die Variable `x` in `Client` als vom Typ `Server` deklariert ist. Zudem verwendet `Client` auch noch eine Methode, die in `Server` definiert ist. Der programmiersprachliche Niederschlag der Abhängigkeiten ist durch Unterstreichen hervorgehoben. S. a. Fußnote 9.

man nicht so leicht (im Extremfall gar nicht) wieder loswird, die sich bei der weiteren Entwicklung aber ziemlich einschränkend auswirken können.

1.4 Anzeichen interfacebasierter Programmierung

In der objektorientierten Programmierung rufen Objekte (Instanzen von Klassen) sich gegenseitig auf, um gemeinsam einen bestimmten Zweck zu erfüllen. Damit sich Objekte gegenseitig aufrufen können, müssen sie einander „kennen“. Dabei hat ein Objekt Kenntnis von einem anderen, wenn es eine Instanzvariable (ein Feld) besitzt, deren Inhalt eben dieses andere Objekt ist. Da (in Sprachen wie JAVA oder C#) die Variable das Objekt nicht selbst enthält, sondern lediglich einen Verweis (Pointer) darauf, spricht man auch davon, daß das andere Objekt referenziert wird. Alternativ kann ein Objekt (bzw. eine Methode dieses Objekts) auch vorübergehend von einem anderen Objekt Kenntnis erlangen, in dem es (den Verweis auf) dieses andere Objekt als aktuellen Methodenparameter oder als Ergebnis eines Methodenaufrufs übergeben bekommt. In allen Fällen, also sowohl bei der Instanzvariable als auch beim formalen Parameter als auch beim Rückgabewert, ist der Verweis typisiert, so daß nur Objekte des verwendeten Typs und seiner Subtypen referenziert werden können. Im nicht interfacebasierten Programmieren ist dieser Typ charakteristischerweise eine Klasse, im interfacebasierten ein Interface. Abbildung 1.4 gibt ein Beispiel für die nicht interfacebasierte Programmierung.

Man sieht hier, wie eine Klasse `Client` von einer Klasse `Server` abhängt[9] (dargestellt durch den gestrichelten Pfeil), weil Instanzen von `Client` eine Variable (namens `x`) haben, deren Inhalt ein Objekt vom Typ (der Klasse) `Server` ist. Wegen dieser Abhängigkeit spricht man von einer starken Kopplung zwischen Client und Server; der Client kann nicht funktionieren, solange keine Klasse mit dem Namen „Server“ zur Verfügung steht. Solche starken Kopplungen sind häufig unerwünscht, weil sie die Flexibilität von Software einschränken. Auch wenn sie sich nicht immer vermeiden lassen (im Gegenteil — eine starke Kopplung ist

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

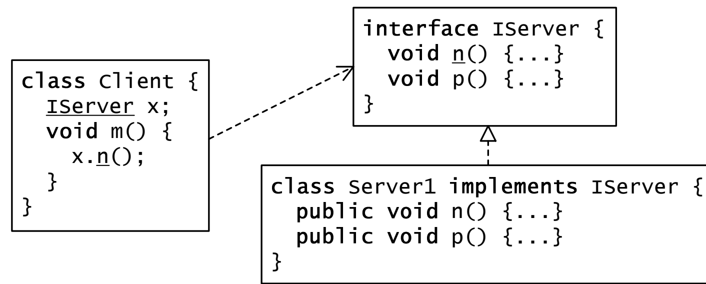

Abbildung 1.5: Entkopplung durch Verwendung eines Interfaces. Die Klasse `Client` ist jetzt direkt nur noch von Interface `IServer` abhängig. Die Klasse des Servers, hier in „Server1" umbenannt, kann durch eine andere ersetzt werden, ohne daß `Client` davon betroffen wäre.

oft natürlich und sie zu vermeiden wäre dann unsinnig; s. u.), geht es bei der interfacebasierten Programmierung vor allem darum, wie man sie umgeht.

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

Die Klasse `Client` ist jetzt nur noch vom Interface `IServer` und damit von keiner konkreten Implementierung mehr abhängig. Die Implementierung des Interfaces, hier durch die Klasse `Server1`, kann im Rahmen der Wartung nach Belieben ausgetauscht werden — es ist aber auch möglich, daß in einem Programm mehrere Implementierungen gleichzeitig zur Verfügung stehen. Das Diagramm täuscht allerdings darüber hinweg, daß eine gewisse Restabhängigkeit von der Klasse des Server-Objekts bestehen bleibt: Das Server-Objekt muß nämlich irgendwann einmal erzeugt und dem Client bekannt gemacht werden. Wenn dies auf Seiten des Clients selbst erfolgt (der Konstruktoraufruf `new Server1()` also beim Client durchgeführt wird), dann besteht die Abhängigkeit des Clients vom Server natürlich trotz des Interfaces weiter. Dies läßt sich aber mit der sog. *Dependency injection* (Abschnitt 1.6) oder durch den Einsatz von *Factory-Methoden* (Abschnitt 4.4.6) vermeiden.

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

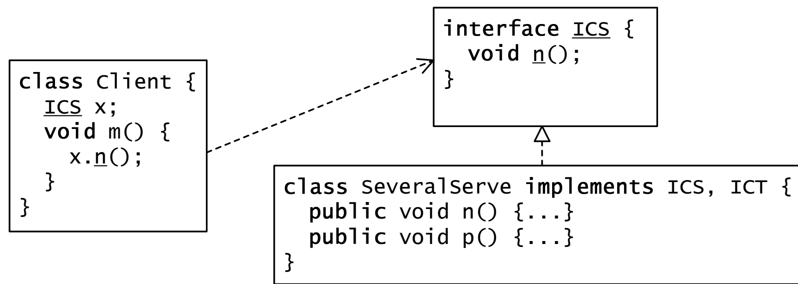

Abbildung 1.6: Zusätzliche Entkopplung des Clients von den Funktionen des Servers, die der Client nicht benötigt. Das Interface `ICS` (kurz für `InterfaceClientServer`) enthält jetzt nur noch die Methode `n()`; es kann deshalb leichter von mehreren Servern angeboten werden als das umfangreichere Interface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

`p()` von einem anderen Client benötigt werden, kann der Server dafür ein weiteres, separates Interface zur Verfügung stellen (`ICT` in Abbildung 1.6).

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

Selbsttestaufgabe 1.2

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

Anzeichen der interfacebasierten Programmierung

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

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

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

Tabelle 1.1: Die am häufigsten implementierten Interfaces im JDK 1.4

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

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

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

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

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

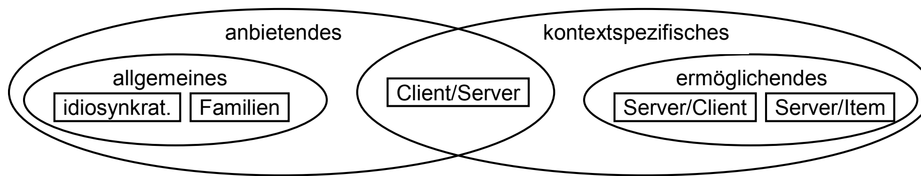

Abbildung 1.7: Klassifikation der verschiedenen Arten (des Gebrauchs von) von Interfaces: jedes fällt in eine der durch Rechtecke markierten Kategorien. Man beachte, daß *anbietend* und *ermöglichend* sowie *allgemein* und *kontextspezifisch* einander jeweils ausschließen.

1.5 Arten des Gebrauchs von Interfaces

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

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

1.5.1 Übersicht

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

1.5.2 Anbietende Interfaces

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

Gewöhnlich geht man davon aus, daß die Aufruferin sich über ein Interface an die Aufgerufene wendet, weil sie etwas von ihr will. Man könnte auch sagen: Die Aufruferin nimmt eine Dienstleistung der Aufgerufenen in Anspruch. Umgekehrt bietet die Aufgerufene über das Interface ihre Dienstleistung an. Das Interface (bzw. der damit verbundene *Vertrag*; vgl. Kurseinheit 2) hat somit den Charakter eines Dienstangebots; wir sprechen daher von einem **anbietenden Interface**.[10]

1.5.3 Allgemeine Interfaces

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

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

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

1.5.4 Idiosynkratische Interfaces

Ein allgemeines Interface, das lediglich von einer Klasse implementiert wird, wobei diese Implementierung über die Zeit geändert werden darf, aber nie zwei

---

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

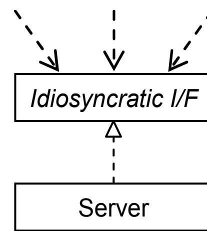

Abbildung 1.8: Ein idiosynkratisches Interface wird von genau einer Klasse des Programms implementiert. Da es sich um ein *anbietendes Interface* handelt, ist diese Klasse immer Dienstanbieterin (Server) für eine oder mehrere andere. Das idiosynkratische Interface umfaßt alle öffentlichen Funktionen der Klasse; es kann daher die Klasse in allen Variablendeklarationen vollständig ersetzen. Dies ist die klassische Form eines Interfaces; sie wird aber in Programmiersprachen wie JAVA oder C# kaum verwendet, da dort das *Klasseninterface* den gleichen Zweck erfüllt.

alternative Implementierungen gleichzeitig in einem Projekt existieren, nennen wir ein **idiosynkratisches Interface**. Die Situation ist in Abbildung 1.8 dargestellt. Idiosynkratische Interfaces tragen häufig die Namen der Klassen, die sie implementieren; die Klasse trägt dann einen Namenszusatz, der sie als Implementierung ausweist.

Beispiel

Das idiosynkratische Interface der Klasse

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

ist

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

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

Während idiosynkratische Interfaces dem ursprünglichen Sinn des Schnittstellenkonzepts entsprechen, wird man sie in aktuellen Sprachen wie JAVA oder C# kaum finden, da durch das *Klasseninterface* (spezifiziert durch die Verwendung des `public` Schlüsselworts in den Methodendefinitionen einer Klasse) bereits ein idiosynkratisches (eben das Klassen-) Interface zur Verfügung steht. Die zusätzliche Bereitstellung eines separaten Interfaces wäre in der Tat sinnlos, wollte man nicht mehrere alternative Implementierungen innerhalb eines Projektes vorsehen. Das Interface wäre damit aber nicht mehr idiosynkratisch. Wir können da-

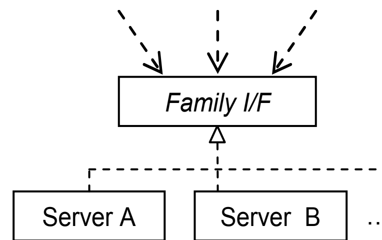

Abbildung 1.9: Ein Familieninterface ist gleichzeitig (*totales*) *Interface* mehrerer Klassen, die alle dasselbe Protokoll anbieten. Klientinnen können so mit unterschiedlichen Dienstanbietern arbeiten, ohne daß sie etwas davon wissen müssen.

her nicht erwarten, viele idiosynkratische Interfaces in JAVA- oder C#-Programmen zu finden. Dies steht im Gegensatz zu vielen Lehrbüchern, in denen man öfter beispielhaft Interfaces wie `IDog` oder `IPerson` findet, deren Natur idiosynkratisch ist.

1.5.5 Familieninterfaces

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

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

1.5.6 Kontextspezifische Interfaces

Interfaces, die speziell für bestimmte Verwendungen von Objekten einer Klasse entworfen wurden und die deswegen nicht das gesamte *Klasseninterface* abdecken, die also nicht allgemein (und nicht *total*, sondern *partiell*) sind, nennen wir **kontextspezifisch**. Ein kontextspezifisches Interface faßt all die Eigenschaften einer Aufgerufenen zusammen, die von einer oder mehreren Aufruferinnen aus einem bestimmten Kontext heraus benötigt werden. Dabei ist ein kontextspezifisches Interface in der Praxis nicht einer speziellen Aufruferin zugeordnet — diese Zuordnung, die bedeuten würde, daß ein Interface-als-Typ nur bestimmten Klassen zugänglich gemacht würde, ist in Sprachen wie JAVA oder C# auch gar nicht vorgesehen —; vielmehr ist es die Funktion (oder *Rolle*; vgl. Abschnitt 1.8),

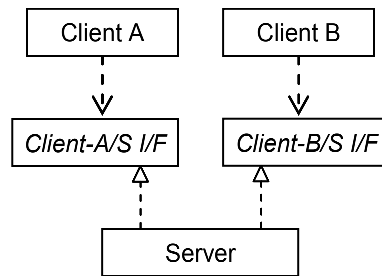

Abbildung 1.10: Ein Client/Server-Interface ist ein *kontextspezifisches, anbietendes Interface*, das speziell für eine oder mehrere Klientinnen (bzw. Kontexte, in denen diese Klientinnen den Server benötigen) entworfen wurde. Auf ein Serverobjekt kann vom gleichen oder von verschiedenen Klientinnen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

die die Aufgerufene in dem gegebenen Kontext spielt, die darüber entscheidet, was in das Interface gehört. Es ist somit möglich, daß dieselbe Aufruferin dieselbe Aufgerufene über verschiedene kontextspezifische Interfaces anspricht, nämlich genau dann, wenn sie dies aus verschiedenen Kontexten heraus tut.

Beispiel

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

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

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

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

Erzwingung der Verwendung von Interfaces in C#

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

1.5.7 Client/Server-Interfaces

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

Vorkommen von Client/Server-Interfaces

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

1.5.8 Ermöglichende Interfaces

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

Vorkommen von ermöglichenden Interfaces in Frameworks

Ermöglichende Interfaces findet man häufig in Frameworks, bei denen eine *Umkehrung der Ausführungskontrolle* (engl. inversion of control) stattfindet: Anstatt wie bei der Benutzung von Programmbibliotheken üblich selbst die Ausführungssteuerung des Programms in die Hand zu nehmen und andere Klassen bei Bedarf aufzurufen, werden die Anwendungsklassen bei der Verwendung eines Frameworks in das Framework eingeklinkt (über sog. *Hooks* oder *Plug points*; s. Abschnitt 4.4.3) und durch das Framework aufgerufen. Man nennt dies gelegentlich aus das *Hollywood-Prinzip*, wegen der dort üblichen Ansage „don't call us, we call you". Allerdings finden ermöglichende Interfaces vornehmlich bei sog. *Black-box-Frameworks* Verwendung, bei denen das Aufrufen über Komposition und Delegation bzw. Forwarding erfolgt; die besser bekannten (und weiter verbreiteten) *White-box-Frameworks* werden im Gegensatz dazu über *Vererbung* und *offene Rekursion* erweitert (s. a. Abschnitt 4.2.1).

1.5.9 Server/Client-Interfaces

Wir kommen nun zu den beiden zu unterscheidenden Fällen. Im ersten Fall ist die Aufgerufene selbst die Nutznießerin der Aufruferin und die Aufruferin die Erbringerin der Dienstleistung; wir nennen daher ein solches Interface ein **Ser-**

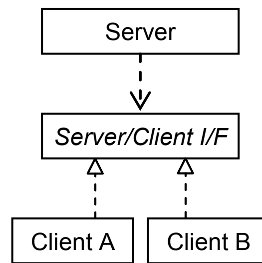

Abbildung 1.11: Bei einem Server/Client-Interface ist das Verhältnis von Aufruferin zu Aufgerufener umgekehrt: Die Aufruferin bietet einen Dienst, von dem die Aufgerufene profitiert. Man findet solche Interfaces häufig in Frameworks, in denen die Umkehrung des Ausführungskontrolle (engl. inversion of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

ver/Client-Interface (Abbildung 1.11). Ein typisches Beispiel für einen Server/Client-Interface ist `java.lang.Runnable`: hier ist die Klasse `Thread` der Server, der es einem Client ermöglicht, Methoden in einem eigenen Thread laufen zu lassen. Der Client muß dazu lediglich die Methode `void run()` implementieren, die der Server aufruft; das Vorhandensein dieser Methode wird über die deklarierte Implementierung des Interfaces `Runnable` sichergestellt:

```
64    public interface Runnable {
65      void run();
66    }

67    class Client implements Runnable {
68          …
69          public void run() {…}
70    }

71    Client client = new Client();
72    new Thread(client).start();
```

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

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

Die symmetrische Benennung der Verwendungsarten Server/Client- und Client/Server-Interfaces legt bereits nahe, daß sie paarweise auftreten können. So etwas kommt zum Beispiel vor, wenn in einer Client/Server-Konstellation der Server für die Ausführung seiner Dienste Information benötigt, die ihm nicht beim Aufruf übergeben wurde. In solchen Fällen wird der Server zur gegebenen Zeit beim Client rückfragen, um diese Information zu erhalten. Es ist dazu allerdings notwendig, daß der Server den Client kennt — in der Regel wird das da-

durch erreicht, daß der Client sich selbst (in JAVA durch die Variable `this` repräsentiert) beim Aufruf an den Server übergibt.[11] Je nachdem, wie langfristig die Beziehung zwischen Client und Server ist, kann der Server auch einen Verweis auf den Client (ggf. auf alle seine Clients) halten, was einer Registrierung des (der) Clients beim Server entspricht.

Beispiel

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

90          …
91          (new Server()).serve(this);
92          …
93  }
```

Ein weiteres Beispiel für paarig auftretende Client/Server-Server/Client-Interfaces ist die asynchrone Kommunikation, bei der das Ergebnis (der Rückgabewert) eines Methodenaufrufs durch einen speziellen Rückruf (engl. *callback*[12]) übergeben werden muß .

Obwohl Server/Client-Interfaces häufig mit Client/Server-Interfaces gepaart vorkommen, gibt es auch zahlreiche sinnvolle Verwendungen für erstere allein. Ein neben dem oben bereits diskutierten `Runnable` weiteres gutes Beispiel ist der Event-listener-Mechanismus, der in JAVA besonders in den (Framework-)Klassen zum grafischen Benutzer-Interface (AWT und SWING) sattsam Verwendung findet (auch als *OBSERVER Pattern* bekannt; s. Abschnitt 4.4.2). Hierbei registrieren sich all die Objekte bei einem Server-Objekt, die von dessen *Zustandsänderungen*

[11] In Sprachen wie JAVA ist der Absender einer Nachricht dem Empfänger sonst nicht bekannt.

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

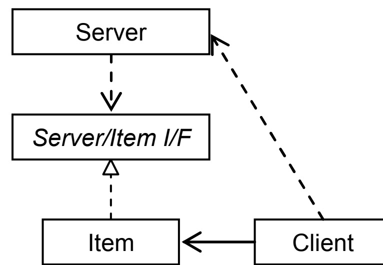

Abbildung 1.12: Bei einem Server/Item-Interface hält der Client Objekte (engl. items), die ein Server für ihn bearbeiten soll. Damit die Bearbeitung unabhängig von der Art der Objekte durchgeführt werden kann, müssen diese das Server/Item-Interface implementieren. Der Server ruft dann die Items über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

in Kenntnis gesetzt werden wollen.[13] Die zu informierenden Objekte (Clients) implementieren dazu ein Listener-Interface, über das sie vom Server aufgerufen werden können. Gleichzeitig bezeichnet dieses Interface den Typ der Objekte, die der Server zur Notifikation registrieren kann. Nur wenn die Clients vom Server zusätzliche Informationen benötigen (die bei der Notifikation nicht übergeben wurden), wird auch ein Client/Server-Interface benötigt.[14]

1.5.10 Server/Item-Interfaces

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

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

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

101  class Server {
102      …
```

13 Dieser Mechanismus wird in Abschnitt 4.4.2 noch ausführlicher behandelt.

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

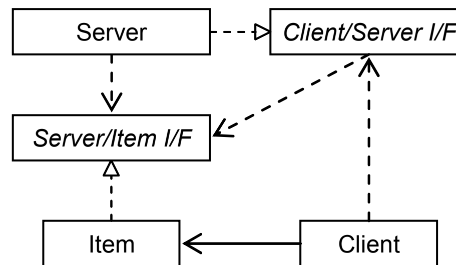

Abbildung 1.13: Zusätzliche Entkopplung des Clients vom Server durch ein Client/Server-Interface. Da der Client seine Items an den Server übergeben muß, damit dieser seine Dienste darauf verrichten kann, ist das Client/Server-Interface vom Server/Item-Interface abhängig. Diese Abhängigkeit äußert sich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

```
103         void sort(List<? extends Comparable> items) {
104            …
105            if (a.compareTo(b) > 0) {…}
106            …
107         }
108     }
109     class Client {
110         Server server;
111         List<Comparable> items;
112         …
113            server.sort(items);
114         …
115     }
```

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

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

Tagging- oder Marker-Interfaces

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

Beispiel

Das wohl bekannteste Tagging-Interface ist `java.io.Serializable`: Es enthält keinerlei Funktionen, die von den implementierenden Klassen anzubieten wären. Statt dessen erlaubt es dem Serialisierungsframework von JAVA, zur Laufzeit

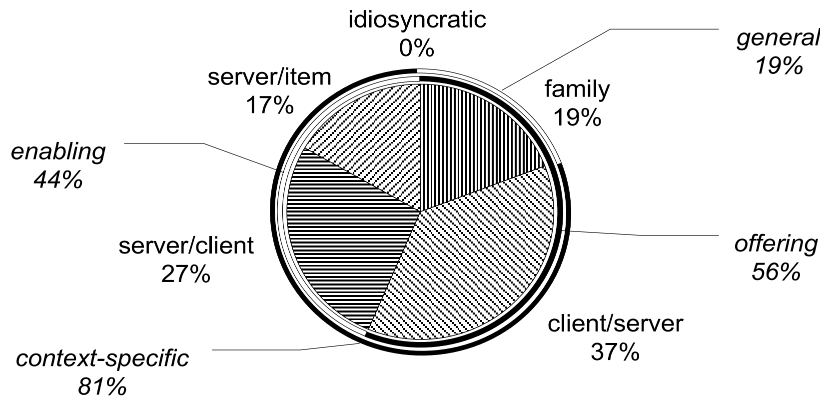

Abbildung 1.14: Relative Verteilung der verschiedenen Arten (des Gebrauchs) von Interfaces, ermittelt anhand der 100 am häufigsten implementierten und 100 am häufigsten referenzierten Interfaces (Überlapp von 43 Interfaces) innerhalb des JDK 1.4.

(mittels des Operators `instanceof`) für ein Objekt zu bestimmen, ob seine Klasse als serialisierbar gekennzeichnet („getagt“) wurde. In C# ist `Serializable` übrigens ein Attribut (s. Abschnitt 6.2).

1.5.11 Zusammenfassung

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

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

1.6 Dependency injection

Die konsequente Verwendung von Interfaces in Variablen- und Methodendeklarationen erlaubt es, die Anzahl der Referenzierungen anderer Klassen und damit die Abhängigkeit von diesen (bzw. die damit verbundene Kopplung) zu verrin-

gern. Es bleibt jedoch die Abhängigkeit von einer Klasse, die durch den Aufruf des Konstruktors zum Zwecke der Erzeugung einer Instanz dieser Klasse entsteht. Diese läßt sich durch die sog. **Dependency injection** eliminieren.

Codebeispiel

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

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

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

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

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

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

unglückliche Namenswahl

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

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

1.6.1 Constructor injection

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

```
118   class Client {
119         IServer server;
120         Client(IServer aServer) {
121            server = aServer;
```

```
122             …
123         }
124         …
125     }
```

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

Der offensichtliche Nachteil der Constructor injection ist, daß der Konstruktoraufruf pro Abhängigkeit, die injiziert werden soll, um ein Argument länger wird. Wenn zudem schon so mehrere alternative Konstruktoren zur Verfügung gestellt werden sollen und sich durch die zusätzliche Dependency injection auch noch mehrere Varianten ergeben, kann die Zahl der benötigten Konstruktoren exponentiell ansteigen. In diesem Fall ist die sog. Setter injection vorzuziehen.

1.6.2 Setter injection

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

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

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

1.6.3 Interface injection

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

```
133     interface ServerInjected {
134         void injectServer(IServer aServer);
135     }

136     class Client implements ServerInjected {
137         IServer server;
138         void injectServer(IServer aServer) {
139             server = aServer;
```

```
140         }
141         …
142     }
```

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

Selbsttestaufgabe 1.3

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

1.6.4 Assembler

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

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

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

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

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

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

Nun sprechen zwei Gründe gegen Assemblerklassen der obigen Art:

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

Aus beiden Gründen erscheint es sinnvoll, die Konfiguration nicht auszuprogrammieren, sondern in einer Konfigurationsdatei (gern auch im XML-Format) zu hinterlegen und die Konfiguration von einem universellen Assembler, der eine solche Datei lesen und interpretieren kann, durchführen zu lassen. Da hierfür aber wieder Konzepte der *Metaprogrammierung* (s. Kurseinheit 6) erforderlich sind (unter anderem müssen Instanzen von Klassen erzeugt werden, deren Namen lediglich in einer Datei hinterlegt sind), gehen wir hier nicht weiter darauf ein.

1.6.5 Einschränkungen

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

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

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

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

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

1.6.6 Alternativen

Dependency injection ist nicht die einzige Möglichkeit, Abhängigkeiten durch Konstruktoraufrufe zu beseitigen — die Verwendung von *Factories* (Abschnitt 4.4.6) und (den im Zusammenhang mit Dependency injection häufig angeführten) sog. *Service locators* sind gebräuchliche Alternativen. Dabei wird zur Erzeugung einer Instanz der Serverklasse kein Konstruktor aufgerufen, sondern eine standardisierte Methode einer Factory oder eines Service locators, die eine ent-

sprechende Instanz zurückgibt. In der einfachsten Form könnte das wie folgt aussehen:

```
157  class Client {
158       IServer server = ServiceLocator.newServer();
159       …
160  }

161  class ServiceLocator {
162       static IServer newServer() {
163          return new Server();
164       }
165  }
```

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

1.6.7 Fazit

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

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

1.7 Umkehrung von Abhängigkeiten mit Interfaces

a. High-level modules should not depend on low-level modules. Both should depend on abstractions.

b. Abstractions should not depend on details. Details should depend on abstractions.

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

In traditioneller, prozeduraler Programmierung gilt es als gutes Design, wenn Prozeduren und Funktionen nach Abstraktionsgrad hierarchisch angeordnet sind. Eine solche Ordnung entsteht im allgemeinen automatisch aus der sog. *funktionalen Dekomposition*, bei der ein Gesamtproblem rekursiv so lange in Teil-

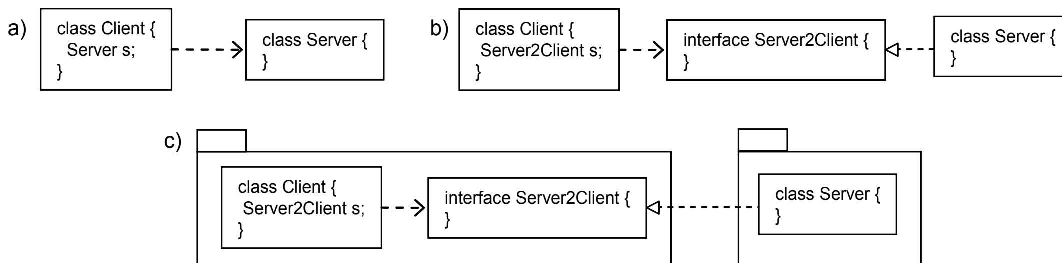

Abbildung 1.15: Schrittweise Herleitung der Abhängigkeitsumkehrung. a) konventionelle Aufrufabhängigkeit des Clients vom Server b) Aufrufabhängigkeit des Clients von einem Interface und Vererbungsabhängigkeit des Servers davon c) Umkehrung der Abhängigkeit gegenüber a) durch Betrachtung der Pakete, in denen Client, Server und Interface liegen

probleme zerlegt wird, bis diese nicht weiter sinnvoll zerlegbar sind oder deren Lösungen bereits existierenden Programmbibliotheken entnommen werden können. Für ein solches Design sind *Aufrufabhängigkeiten* von oben nach unten charakteristisch.

In der objektorientierten Programmierung gibt es diese Abhängigkeit natürlich auch. Sie äußert sich aber nicht nur im Aufruf von primitiveren Methoden, sondern auch in der Referenzierung von Klassen, deren Instanzen die Methoden zugeordnet sind und über die sie aufgerufen werden (s. Abbildung 1.15 a)). Da die Methoden einer Klasse nicht unbedingt alle das gleiche Abstraktionsniveau haben, läßt sich die hierarchische Ordnung der Methoden (die sich aus der funktionalen Dekomposition ergibt) nicht auch auf die Klassen übertragen. Dies wäre in einer geschichteten objektorientierten Architektur der Fall, auf die hier jedoch, da sie zur Erläuterung des Sachverhalts nicht notwendig ist, nicht eingegangen werden soll (die interessierte Leserin schaue bei [9] nach).

Wenn man nun die Klassen mit den aufgerufenen Methoden gegen andere austauschen möchte, muß man die Referenzen in den aufrufenden Klassen ändern. Die aufrufenden Klassen sind damit, im Gegensatz zu den aufgerufenen, nicht ohne Modifikationen wiederverwendbar. Dies ist beispielsweise für den Entwurf von Frameworks, bei denen die darin codierte Ausführungskontrolle wiederverwendet werden soll (die *Umkehrung der Ausführungskontrolle*; vgl. Abschnitt 1.5.8), hinderlich (aber nicht nur dort), weswegen man diese Form der Abhängigkeit gern vermeiden würde.

Vererbungsabhängigkeit

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

Blickwechsel durch Übergang zu größeren Organisationseinheiten

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

das Dependency Inversion Principle

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

Schönheitsfehler

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

Das Problem wird durch das folgende Beispiel verdeutlicht:

```
166  package a;

167  import b.B;
168  import c.C;

169  class A {
170       B b;
171       C c;
172       …
173          b.m(c);
174       c.n();
175       …
176  }
```

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

```
177   package a;

178   interface IB { void m(C c); }
179   interface IC { void n(); }

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

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

Inferenz des transitiv benötigten Interfaces

Man könnte nun auf die Idee kommen, `C` in `IB` und in `B` durch `IC` zu ersetzen. Das würde das obige Typproblem und die Abhängigkeit von `C` gleichermaßen beheben, setzt jedoch voraus, daß der Klasse `B` das Interface `IC` als Typ von `c` genügt. Diese Frage läßt sich jedoch i. a. nicht durch eine Analyse von `B` allein beantworten, da die Klasse `B` den Parameter `c` an andere Klassen weiterreichen kann, so daß sich das Problem erneut stellt. Tatsächlich ist eine Verfolgung aller möglichen Zuweisungen von `c` notwendig, um zu bestimmen, welchen Typ `c` mindestens haben muß (vgl. das INFER TYPE Refactoring in Abschnitt 5.2.4.6). Dieser Typ wäre dann auch in `A` zu verwenden, selbst wenn er nicht nur das Interface, das `A` von `C` benötigt, sondern das aller Klassen, die Zugriff auf den Parameter `c` erlangen, repräsentiert. Die Abhängigkeit von `C` wird damit durch ziemlich undurchsichtige Abhängigkeiten von `B` und ggf. weiteren Klassen ersetzt, was kaum der Sinn der Sache sein kann.

Cross casts an den Modulgrenzen

Eine einfache, aber brutale Lösung dieses Problems ist, `IC` für `A` tatsächlich wie oben (in Zeile 179) beschrieben zu implementieren und für `B` und alle anderen Klassen, die `C` referenzieren, ein eigenes Interface für `C` jeweils so, wie sie es brauchen. Es hätte dann jeder Client von `C` genau sein benötigtes Interface und an den Stellen, an denen eine Referenz auf eine Instanz der Klasse `C` an eine andere Klasse übertragen wird (per Parameter eines Methodenaufrufs wie in Zeile 184 oder per Zuweisung an ein Feld), müßte eine Typumwandlung, ein sog. **Cross cast**[15], stattfinden. Ein solcher Cross cast entzieht sich zwar der *statischen Typprüfung*

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

(und kann deswegen grundsätzlich zu *Laufzeittypfehlern* führen, weswegen er i. a. nicht gern gesehen wird), aber solange dieser Cross cast nur auf Variablen, die vorher mit `C` typisiert waren, durchgeführt wird, können dabei keine *Typumwandlungsfehler* (Type cast errors) auftreten.[16] Ob diese Lösung aber praktikabel ist (oder praktiziert wird), darüber liegen mir keine Erkenntnisse vor.

1.8 Interpretation von Interfaces als Rollen

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

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

- Verwendet man eine Klasse, dann ist damit die Art der Objekte, zu denen die Beziehung aufgenommen werden kann, festgelegt. Das gilt sogar, wenn auch mit Einschränkungen, für abstrakte Klassen, denn auch diese können (Teile der) Implementierung vorgeben, und es ist schließlich die Implementierung, die die Art (das Genus) der Objekte ausmacht.
- Verwendet man hingegen ein Interface, so ist damit nicht die Art, sondern lediglich die Rolle der Objekte, die den Platz der Variable einnehmen, festgelegt. Um eine Rolle spielen zu können, müssen die Objekte das mit der Rolle verbundene Verhalten mitbringen — sie müssen die Rolle (das Interface) implementieren.

Die Interpretation von Interfaces als Rollen wird auch vielfach durch die Namensgebung unterstützt: Viele der ermöglichenden Interfaces, die auf „able“ oder „ible“ enden, drücken eine Rolle aus, so z. B. die des Serialisierbaren (engl.

---

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

serializable) oder die des Vergleichbaren (engl. comparable). Aber auch typische Client/Server-Interfaces tragen Rollennamen: `MenuContainer` beispielsweise drückt eine Rolle von Objekten so unterschiedlicher Klassen wie `Button`, `CheckBox` etc. aus. Lediglich die allgemeinen Interfaces drücken in der Regel keine Rollen aus; sie stehen in Konkurrenz zu abstrakten Klassen, die die inhaltliche Verwandtheit in den Vordergrund rücken.

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

1.9 Werkzeugunterstützung für das interfacebasierte Programmieren

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

Glücklicherweise gibt es mittlerweile einige Werkzeuge, mit deren Hilfe das Erstellen und Verwenden von Interfaces bis zu einem gewissen Grad automatisiert erfolgen kann. Diese Werkzeuge, die auch unter dem Namen *Refactorings* bekannt sind und die in Kurseinheit 5 ausführlicher behandelt werden, sind heute bereits Bestandteil populärer Entwicklungsumgebungen wie ECLIPSE, NETBEANS oder INTELLIJ IDEA. Speziell für das interfacebasierte Programmieren können beispielsweise die ECLIPSE-Refactorings EXTRACT INTERFACE (bzw. das intelligentere INFER TYPE, das die Methoden eines Interfaces automatisch bestimmt; s. Abschnitt 5.2.4.6), GENERALIZE TYPE und USE SUPERTYPE WHERE POSSIBLE verwendet werden. Das Umbenennen von Methoden bzw. das Ändern von Signaturen

eines Interfaces und aller es implementierenden Klassen wird einer durch das Refactoring RENAME METHOD abgenommen.

Das Bewußtsein, daß man Interfaces in Variablendeklarationen verwenden sollte, ist eine Sache, beim Programmieren immer daran zu denken eine andere. Mit dem DECLARED TYPE GENERALIZATION CHECKER (DTGC) existiert[17] ein Plugin für ECLIPSE, das für jede Variable im Programm überprüft, ob nicht vielleicht ein Interface existiert, mit dem sie besser deklariert werden könnte. Der DTGC greift dazu auf das ECLIPSE-interne Refactoring (s. Kurseinheit 5) GENERALIZE DECLARED TYPE zurück und erbt damit dessen Unzulänglichkeiten (insbesondere die mangelnde transitive Verfolgung von Zuweisungen). Alternativ kann zur Prüfung möglicher *Generalisierungen* auch eine Typinferenz (die von INFER TYPE; s. o.) verwendet werden, doch das dauert ziemlich lange und schlägt zudem eine kaum zu überblickende Vielzahl von verschiedenen Interfaces vor.

Die Darstellung und Umkehrung von Paketabhängigkeiten wie in Abschnitt 1.7 (Umkehrung von Abhängigkeiten mit Interfaces) beschrieben wird komfortabel durch den PACKAGE DEPENDENCY INVERTER[18] erledigt. Er erlaubt es, für alle Verwendungen einer Klasse eines Pakets aus einem anderen Paket heraus ein gemeinsames benötigtes Interface des anderen Pakets von der Klasse zu berechnen und einzusetzen. Dabei werden auch die Abhängigkeiten berücksichtigt, die sich transitiv, aus der Weiterreichung von Instanzen der benutzten Klasse an andere Pakete, ergeben. Die praktische Verwendung des PACKAGE DEPENDENCY INVERTER zeigt aber auch, daß sich längst nicht alle Abhängigkeiten zwischen Paketen umkehren lassen: Die Verwendung eines Typs, zu dem eine Aufrufabhängigkeit besteht, als formaler Parametertyp kann dazu führen, daß das benötigte Interface eine neue Abhängigkeit zu ihm einführt, die sich zwar auch umkehren läßt, aber nur um den Preis einer neuen in die andere Richtung. Wie so oft widersetzt sich die Realität einer schönen Idee.

1.10 Weiterführende Literatur

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

---

17 http://www.fernuni-hagen.de/ps/prjs/DTGC/

18 http://www.fernuni-hagen.de/ps/prjs/PDI/

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

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

[1] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns — Elements of Reusable Software* (Addison-Wesley, 1995).

[2] EW Dijkstra „The structure of "THE"-multiprogramming system“ *CACM* 11:5 (1968) 341–346.

[3] DL Parnas „On the criteria to be used in decomposing systems into modules“ *CACM* 15:12 (1972) 1053–1058.

[4] *IEEE Standard Computer Dictionary* (IEEE, 1991).

[5] http://www.acm.org/class/

[6] B Liskov, A Snyder, R Atkinson, C Schaffert „Abstraction mechanisms in CLU“ *CACM* 20:8 (1977) 564–576.

[7] PS Canning, WR Cook, WL Hill, WG Olthoff „Interfaces for strongly-typed object-oriented programming“ in: *Proc. of OOPSLA* (1989) 457–467.

[8] M Fowler „Public vs. published interfaces“ *IEEE Software* 19:2 (2002) 18–19.

[9] RC Martin *Agile Software Development. Principles, Patterns, and Practices* (Prentice Hall International, 2003).

[10] L Mikhajlov, E Sekerinski „A study of the fragile base class problem“ in: *12th European Conference on Object-Oriented Programming (ECOOP'98)* Springer LNCS 1445 (1998) 355–382.

[11] F Steimann, P Mayer „Patterns of interface-based programming“ *Journal of Object Technology* 4:5 (2005) 75–94.

1.11 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 1.1 (Seite 5)

s. Tabelle 1.1 und Tabelle 1.2

Selbsttestaufgabe 1.2 (Seite 15)

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

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

Selbsttestaufgabe 1.3 (Seite 30)

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