KAPITEL 4

Module sollten tief sein

Eine der wichtigsten Techniken im Umgang mit Softwarekomplexität besteht darin, Systeme so zu designen, dass man sich beim Entwickeln jederzeit immer nur einem kleinen Teil der Gesamtkomplexität gegenübersehen muss. Dieser Ansatz wird als *modulares Design* bezeichnet, und dieses Kapitel stellt seine grundlegenden Prinzipien vor.

Modulares Design

Beim modularen Design wird ein Softwaresystem in eine Sammlung von *Modulen* unterteilt, die verhältnismäßig unabhängig voneinander sind. Module können viele Formen annehmen, zum Beispiel Klassen, Subsysteme oder Services. In einer idealen Welt wäre jedes Modul vollständig unabhängig von den anderen: Beim Entwickeln könnte man in jedem der Module arbeiten, ohne irgendetwas über eines der anderen Module wissen zu müssen. In dieser Welt wäre die Komplexität eines Systems die Komplexität seines schlimmsten Moduls.

Leider ist dieses Ideal nicht erreichbar. Module müssen zusammenarbeiten, indem sie Funktionen oder Methoden anderer Module aufrufen. Daher müssen Module etwas über andere wissen. Es wird *Abhängigkeiten* zwischen den Modulen geben: Ändert sich ein Modul, müssen sich eventuell auch andere Module ändern, damit wieder alles passt. So sorgen beispielsweise die Argumente einer Methode für eine Abhängigkeit zwischen der Methode und jeglichem Code, der diese Methode aufruft. Ändern sich die erforderlichen Argumente, müssen alle Aufrufe der Methode angepasst werden, damit diese der neuen Signatur entsprechen. Abhängigkeiten können viele andere Formen annehmen, und sie können ziemlich subtil sein. Als Beispiel sei eine Methode erwähnt, die nur dann korrekt funktioniert, wenn eine andere Methode zuvor aufgerufen wurde. Das Ziel des modularen Designs ist es, die Abhängigkeiten zwischen Modulen zu minimieren.

Um Abhängigkeiten erkennen und managen zu können, stellen wir uns vor, dass jedes Modul aus zwei Teilen besteht: einer *Schnittstelle* (*Interface*) und einer *Implementierung*. Die Schnittstelle setzt sich zusammen aus allem, was man beim Entwickeln

in einem anderen Modul wissen muss, um das gegebene Modul zu verwenden. Typischerweise beschreibt die Schnittstelle, *was* das Modul tut, aber nicht, *wie* es das tut. Die Implementierung dagegen besteht aus dem Code, der die durch die Schnittstelle gemachten Versprechungen umsetzt. Arbeitet man an einem bestimmten Modul, muss man die Schnittstelle und die Implementierung dieses Moduls verstehen, dazu aber auch die Schnittstellen aller anderen Module, die vom gegebenen Modul aufgerufen werden. Beim Entwickeln sollte man die Implementierungen von anderen Modulen mit Ausnahme des aktuell bearbeiteten nicht kennen müssen.

Stellen Sie sich ein Modul vor, das balancierte Bäume implementiert. Es enthält vermutlich ausgeklügelten Code, um sicherzustellen, dass der Baum balanciert bleibt. Aber diese Komplexität ist beim Einsatz des Moduls nicht sichtbar. Beim Anwenden sieht man nur eine recht einfache Schnittstelle, um Operationen zum Einfügen, Entfernen und Auslesen von Knoten aufrufen zu können. Um eine Einfügeoperation aufzurufen, müssen lediglich Schlüssel und Wert für den neuen Knoten geliefert werden – die Mechanismen zum Durchlaufen des Baums und zum Aufteilen der Knoten sind in der Schnittstelle nicht sichtbar.

Im Rahmen dieses Buchs verstehen wir ein Modul als eine Codeeinheit, die eine Schnittstelle und eine Implementierung besitzt. Jede Klasse in einer objektorientierten Programmiersprache ist ein Modul. Methoden in einer Klasse oder Funktionen in einer nicht objektorientierten Sprache können auch als Module angesehen werden – jede hat eine Schnittstelle und eine Implementierung, und auch auf sie können modulare Designtechniken angewendet werden. Subsysteme und Services auf höherer Ebene sind ebenfalls Module – ihre Schnittstellen mögen allerdings anders aussehen, wie zum Beispiel Kernelaufrufe oder HTTP-Requests. Ein Großteil der Diskussion von modularem Design wird sich in diesem Buch auf das Designen von Klassen konzentrieren, aber die Techniken und Konzepte lassen sich auch auf andere Arten von Modulen anwenden.

Die besten Module sind diejenigen, deren Schnittstellen viel einfacher als ihre Implementierungen sind. Solche Module haben zwei Vorteile. Zum einen minimiert eine einfache Schnittstelle die Komplexität, die ein Modul für den Rest des Systems freisetzt, und zum anderen sorgt eine Veränderung eines Moduls, die dessen Schnittstelle nicht beeinflusst, dafür, dass kein anderes Modul davon beeinflusst wird. Ist die Schnittstelle eines Moduls viel einfacher als dessen Implementierung, wird es viele Aspekte des Moduls geben, die geändert werden können, ohne dass andere Module beeinflusst werden.

Was ist eine Schnittstelle?

Die Schnittstelle eines Moduls enthält zwei Arten von Informationen: formale und informelle. Die formalen Teile einer Schnittstelle werden explizit im Code spezifiziert und und einige davon können von der Programmiersprache auf Korrektheit überprüft werden. So ist beispielsweise die formale Schnittstelle einer Methode deren Signatur, zu der die Namen und Typen der Parameter gehören, der Typ des

Rückgabewerts und Informationen über Exceptions, die von der Methode geworfen werden. Die meisten Programmiersprachen stellen sicher, dass jeder Aufruf einer Methode die richtige Anzahl und den richtigen Typ von Argumenten bereitstellt, die zu ihrer Signatur passen. Die formale Schnittstelle einer Klasse besteht aus den Signaturen aller öffentlichen Methoden, dazu aus den Namen und Typen aller öffentlichen Variablen.

Jede Schnittstelle enthält auch informelle Elemente. Diese werden nicht so spezifiziert, dass sie von der Programmiersprache verstanden oder durch sie sichergestellt werden können. Zu den informellen Teilen einer Schnittstelle gehört ihr High-Level-Verhalten, etwa dass eine Funktion die Datei löscht, deren Name durch eines der Argumente angegeben wurde. Gibt es Einschränkungen bezüglich des Einsatzes einer Klasse (zum Beispiel muss eine Methode aufgerufen werden, bevor eine andere aufgerufen werden kann), sind diese ebenfalls Teil der Klassenschnittstelle. Ganz allgemein gilt: Muss man eine bestimmte Information kennen, um ein Modul nutzen zu können, ist diese Information Teil der Modulschnittstelle. Die informellen Aspekte einer Schnittstelle können nur durch Kommentare beschrieben werden, und die Programmiersprache kann nicht sicherstellen, dass die Beschreibung vollständig oder exakt ist.[1] Bei den meisten Schnittstellen sind die informellen Aspekte größer und komplexer als die formellen Anteile.

Einer der Vorteile einer sauber spezifizierten Schnittstelle ist, dass sie genau angibt, was die Entwicklung wissen muss, um das entsprechende Modul nutzen zu können. Das hilft dabei, das im Abschnitt »Symptome der Komplexität« auf Seite 21 beschriebene Problem der »unbekannten Unbekannten« auszumerzen.

Abstraktionen

Der Begriff *Abstraktion* steht in einem engen Zusammenhang mit der Idee des modularen Designs. **Eine Abstraktion ist eine vereinfachte Sicht einer Entität, die unwichtige Details auslässt.** Abstraktionen sind nützlich, weil sie es uns erleichtern, über komplexe Dinge nachzudenken und sie zu verändern.

Bei der modularen Programmierung stellt jedes Modul eine Abstraktion in Form seiner Schnittstelle bereit. Diese dient als vereinfachte Sicht auf die Funktionalität des Moduls – die Details der Implementierung sind aus Sicht der Abstraktion unwichtig, daher werden sie in der Schnittstelle weggelassen.

1 Es gibt Sprachen – vor allem innerhalb der Forschungsgemeinde –, bei denen das vollständige Verhalten einer Methode oder Funktion formal durch eine Spezifikationssprache beschrieben werden kann. Die Spezifikation kann automatisch geprüft werden, um sicherzustellen, dass sie der Implementierung entspricht. Eine interessante Frage ist, ob solch eine formale Spezifikation die informellen Teile einer Schnittstelle ersetzen könnte. Aktuell bin ich der Meinung, dass eine in natürlicher Sprache beschriebene Schnittstelle vermutlich intuitiver und für die Entwicklung verständlicher ist als eine, die in einer formalen Spezifikationssprache geschrieben ist.

Bei der Definition der Abstraktion ist das Wort »unwichtig« entscheidend. Je mehr unwichtige Details weggelassen werden, desto besser ist es. Aber ein Detail kann nur dann bei einer Abstraktion weggelassen werden, wenn es unwichtig ist. Eine Abstraktion kann auf zwei Arten fehlerhaft sein. Zum einen kann sie Details enthalten, die nicht wirklich wichtig sind – wenn das geschieht, wird die Abstraktion komplizierter als nötig, was die kognitive Last bei ihrem Einsatz erhöht. Der zweite Fehler ist, dass eine Abstraktion Details weglässt, die eigentlich wichtig sind. Das führt zu Unklarheiten: Schaut man sich nur die Abstraktion an, besitzt man nicht alle Informationen, die zum korrekten Einsatz notwendig sind. Eine Abstraktion, die wichtige Details weglässt, ist eine *falsche Abstraktion*: Sie scheint einfach zu sein, ist es aber in Wirklichkeit nicht. Der Schlüssel beim Designen von Abstraktionen ist, zu verstehen, was wichtig ist, und nach Designs Ausschau zu halten, die die Menge an wichtigen Informationen minimiert.

Stellen Sie sich zum Beispiel ein Dateisystem vor. Die von einem Dateisystem bereitgestellte Abstraktion lässt viele Details weg, zum Beispiel die Mechanismen zum Auswählen der Blöcke auf einem Storage-Device. Diese Details sind für das Einsetzen des Dateisystems unwichtig (solange das System eine ausreichende Leistung liefert). Aber manche der Implementierungsdetails eines Dateisystems sind wichtig. Die meisten Dateisysteme puffern Daten im Hauptspeicher und verzögern das Schreiben neuer Daten auf das Storage-Device, um die Performance zu verbessern. Manche Anwendungen – wie zum Beispiel Datenbanken – müssen genau wissen, wann Daten auf den Storage geschrieben werden, um sicherstellen zu können, dass diese auch nach einem Systemabsturz vorhanden sind. Daher müssen die Regeln zum Wegschreiben der Daten auf den Secondary Storage in der Schnittstelle des Dateisystems sichtbar sein.

Wir sind nicht nur beim Programmieren auf Abstraktionen angewiesen, um die Komplexität meistern zu können, sondern überall in unserem Alltag. Eine Mikrowelle enthält komplexe Elektronik, um Wechselstrom in Mikrowellenstrahlung umzuwandeln und diese dann im ganzen Garraum zu verteilen. Zum Glück findet man beim Kochen eine deutlich einfachere Abstraktion vor, die nur aus ein paar Knöpfen zum Steuern von Dauer und Intensität der Mikrowellen besteht. Autos bieten eine einfache Abstraktion, die es uns erlaubt, sie zu fahren, ohne die Mechanismen von Elektromotoren, Batteriemanagement, Antiblockiersystemen, Tempomat und so weiter verstehen zu müssen.

Tiefe Module

Die besten Module sind die, die leistungsfähige Funktionalität bieten, dabei aber trotzdem eine einfache Schnittstelle haben. Ich nutze den Begriff *tief* (*deep*), um solche Module zu beschreiben. Um sich die Tiefe zu verdeutlichen, stellen Sie sich vor, dass jedes Modul durch ein Rechteck dargestellt wird (siehe Abbildung 4-1). Die Fläche jedes Rechtecks ist proportional zur durch das Modul implementierten Funktionalität. Die obere Kante eines Rechtecks repräsentiert die Schnittstelle des

Moduls – die Länge dieser Kante steht für deren Komplexität. Die besten Module sind tief: Sie besitzen viel Funktionalität, die hinter einer einfachen Schnittstelle verborgen ist. Ein tiefes Modul ist eine gute Abstraktion, weil nur ein kleiner Teil seiner internen Komplexität für seine Nutzerinnen und Nutzer sichtbar ist.

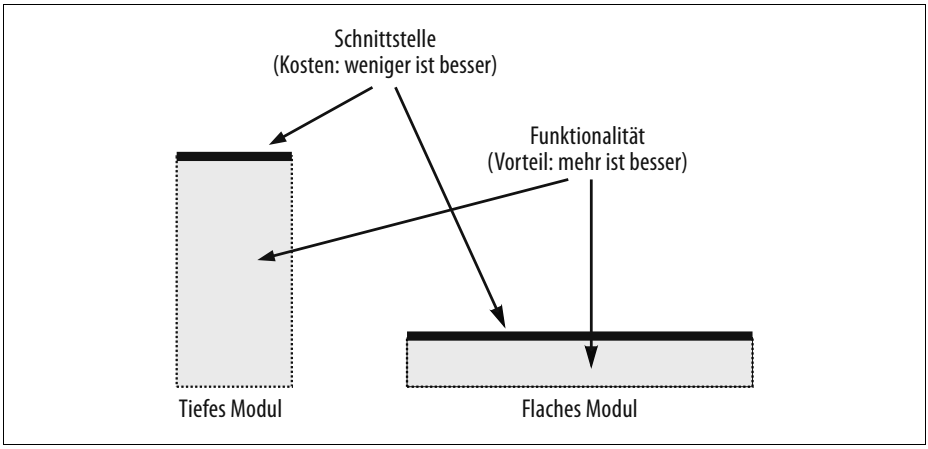

Abbildung 4-1: Tiefe und flache Module. Die besten Module sind tief: Sie erlauben, viel Funktionalität über eine einfache Schnittstelle anzusprechen. Ein flaches Modul ist eines mit einer recht komplexen Schnittstelle, aber nicht viel Funktionalität – so wird nicht viel Komplexität verborgen.

Die Modultiefe ist ein Ansatzpunkt, um über den Konflikt zwischen Kosten und Vorteilen nachzudenken. Die Vorteile eines Moduls bestehen in dessen Funktionalität, die Kosten eines Moduls (in Bezug auf die Systemkomplexität) in dessen Schnittstelle. Die Schnittstelle eines Moduls repräsentiert die Komplexität, die das Modul in den Rest des Systems trägt: Je kleiner und einfacher die Schnittstelle ist, desto weniger Komplexität bringt sie mit sich. Die besten Module sind solche mit den größten Vorteilen und den geringsten Kosten. Schnittstellen sind gut, aber mehr oder größere Schnittstellen sind nicht unbedingt besser!

Der Mechanismus für Datei-I/O von Unix und seinen Nachfolgern wie Linux ist ein wunderschönes Beispiel für eine tiefe Schnittstelle. Es gibt nur fünf grundlegende Systemaufrufe mit einfachen Signaturen:

```
int open(const char* path, int flags, mode_t permissions);
ssize_t read(int fd, void* buffer, size_t count);
ssize_t write(int fd, const void* buffer, size_t count);
off_t lseek(int fd, off_t offset, int referencePosition);
int close(int fd);
```

Der Systemaufruf `open` übernimmt einen hierarchischen Dateinamen wie `/a/b/c` und gibt einen *Dateideskriptor* als Integer-Wert zurück, der dann als Referenz für die geöffnete Datei dient. Die anderen Argumente beim Öffnen stellen zusätzliche Informationen bereit, wie zum Beispiel, ob die Datei zum Lesen oder Schreiben geöffnet wird, ob eine neue Datei erzeugt werden soll, wenn es sie noch nicht gibt, und wie die Zugriffsberechtigungen für die Datei aussehen, wenn eine neue Datei

angelegt wird. Die Systemaufrufe `read` und `write` übertragen Informationen zwischen Pufferbereichen im Speicher der Anwendung und der Datei, `close` beendet den Zugriff auf die Datei. Auf die meisten Dateien wird sequenziell zugegriffen, daher ist das die Standardvorgehensweise – ein wahlfreier Zugriff wird über den Systemaufruf `lseek` erreicht, durch den man die aktuelle Zugriffsposition anpasst.

Für eine moderne Implementierung der I/O-Schnittstelle von Unix sind Hunderttausende Zeilen Code erforderlich, die komplexe Themen wie die folgenden angehen:

- Wie werden Dateien auf der Festplatte repräsentiert, um einen effizienten Zugriff zu ermöglichen?
- Wie werden Verzeichnisse abgelegt, und wie werden hierarchische Pfadnamen verarbeitet, um die Datei zu finden, die dadurch angesprochen wird?
- Wie werden die Berechtigungen sichergestellt, sodass der eine Anwender nicht die Dateien einer anderen Anwenderin verändern oder löschen kann?
- Wie wird der Dateizugriff implementiert? Wie wird beispielsweise die Funktionalität zwischen Interrupt-Handlern und Code im Hintergrund aufgeteilt, und wie kommunizieren diese beiden Elemente zuverlässig miteinander?
- Welche Scheduling-Richtlinien werden genutzt, wenn es konkurrierende Zugriffe auf mehrere Dateien gibt?
- Wie können die Daten von kürzlich angesprochenen Dateien im Speicher gehalten werden, um die Anzahl der Festplattenzugriffe zu verringern?
- Wie kann eine Vielzahl von unterschiedlichen Secondary-Storage-Devices – wie beispielsweise Festplatten oder Flash Drives – in ein einzelnes Dateisystem integriert werden?

Alle diese Probleme – und viele mehr – werden von der Implementierung des Unix-Dateisystems behandelt. Sie sind für Programmiererinnen und Programmierer, die die Systemaufrufe nutzen, unsichtbar. Die Implementierungen der Unix-I/O-Schnittstelle haben sich im Laufe der Jahre massiv weiterentwickelt, aber die fünf grundlegenden Kernelaufrufe haben sich nicht geändert.

Ein weiteres Beispiel eines tiefen Moduls ist der Garbage Collector in einer Sprache wie Go oder Java. Dieses Modul besitzt überhaupt keine Schnittstelle – es arbeitet unsichtbar hinter den Kulissen, um ungenutzten Speicher aufzuräumen. Indem ein System um eine Garbage Collection erweitert wird, schrumpft tatsächlich die Gesamtschnittstelle, da die Schnittstelle zum Freigeben von Objekten wegfällt. Das Implementieren eines Garbage Collector ist ziemlich komplex, aber diese Komplexität wird gegenüber den Programmierern verborgen.

Tiefe Module wie Unix-I/O und Garbage Collectors liefern leistungsfähige Abstraktionen, weil sie sich leicht einsetzen lassen, dabei aber signifikante Implementierungskomplexität verbergen.

Flache Module

Im Gegensatz dazu ist bei einem flachen Modul dessen Schnittstelle im Vergleich zur bereitgestellten Funktionalität recht komplex. Beispielsweise ist eine Klasse flach, die eine verkettete Liste implementiert. Es braucht nicht viel Code, um eine verkettete Liste zu implementieren (zum Einfügen oder Löschen eines Elements sind nur ein paar Zeilen Code notwendig), daher verbirgt die Abstraktion der verketteten Liste nicht sehr viele Details. Die Komplexität der Schnittstelle einer verketteten Liste ist fast so groß wie die Komplexität ihrer Implementierung. Flache Klassen lassen sich manchmal nicht vermeiden, und sie können auch immer noch ihre Berechtigung haben, aber sie sind beim Managen von Komplexität nicht sehr hilfreich.

Hier ein extremes Beispiel einer flachen Methode aus einem Projekt in einem Kurs zu Softwaredesign:

```
private void addNullValueForAttribute(String attribute) {
    data.put(attribute, null);
}
```

Vom Aspekt der Komplexität aus betrachtet macht diese Methode die Dinge nur schlimmer. Sie bietet keine Abstraktion, da die gesamte Funktionalität durch ihre Schnittstelle sichtbar ist. Aufrufender Code muss beispielsweise vermutlich wissen, dass das Attribut in der Variablen `data` gespeichert wird. Es ist nicht einfacher, über die Schnittstelle nachzudenken als über die vollständige Implementierung. Ist die Methode sauber dokumentiert, wird die Dokumentation länger sein als der Code der Methode. Es erfordert sogar mehr Tippaufwand, um die Methode aufzurufen, also die Variable `data` direkt zu verändern. Die Methode sorgt für zusätzliche Komplexität (in Form einer neuen Schnittstelle, die Entwicklerinnen und Entwickler lernen müssen), bietet aber keinen Vorteil, der dies ausgleichen würde.

Warnzeichen: Flache Module

Ein flaches Modul ist eines, dessen Schnittstelle im Verhältnis zur bereitgestellten Funktionalität relativ kompliziert ist. Flache Module helfen nicht sehr beim Kampf gegen Komplexität, weil ihr Vorteil (nicht lernen zu müssen, wie sie intern funktionieren) durch die Kosten für das Erlernen und Einsetzen ihrer Schnittstellen aufgezehrt wird. Kleine Module tendieren dazu, flach zu sein.

Klassizitis

Leider wird der Wert tiefer Klassen heutzutage nicht allzu sehr anerkannt. Üblicherweise ist man in der Programmierung der Ansicht, dass Klassen *klein* und nicht tief sein sollten. Häufig wird Studierenden beigebracht, dass es beim Design von Klassen am wichtigsten sei, größere Klassen in kleinere aufzuteilen. Den gleichen Rat gibt es auch oft bei Methoden: »Jede Methode, die größer als N Zeilen ist, sollte in mehrere Methoden aufgeteilt werden.« (N kann auch einen Wert von nur

10 haben.) Dieser Ansatz führt zu vielen flachen Klassen und Methoden, die die Gesamtkomplexität des Systems vergrößern.

Das Extrem beim »Klassen sollten klein sein«-Ansatz ist ein Syndrom, das ich als *Klassizitis* bezeichne und das von der irrtümlichen Ansicht herrührt, dass »Klassen gut sind und mehr Klassen also besser sind«. In Systemen, die an Klassizitis leiden, werden Entwicklerinnen und Entwickler darin bestärkt, die Menge an Funktionalität in jeder neuen Klasse zu minimieren: Wollen Sie mehr Funktionalität haben, bringen Sie mehr Klassen ins Spiel. Klassizitis kann zu Klassen führen, die für sich betrachtet einfach sind, aber die Komplexität des Gesamtsystems vergrößern. Kleine Klassen tragen nicht viel Funktionalität bei, daher muss es viele von ihnen geben, die jeweils eine eigene Schnittstelle besitzen. Diese Schnittstellen summieren sich auf Systemebene zu einer unglaublichen Komplexität auf. Kleine Klassen führen zudem zu einem weitschweifigen Programmierstil, weil jede Klasse immer wieder neuen Code erfordert.

Beispiele: Java und Unix-I/O

Eines der bekanntesten Beispiele für Klassizitis ist heutzutage die Klassenbibliothek von Java. Die Programmiersprache Java erfordert nicht viele kleine Klassen, aber in der Programmier-Community von Java scheint eine Kultur der Klassizitis Fuß gefasst zu haben. Um zum Beispiel eine Datei zum Einlesen serialisierter Objekte zu öffnen, musste man in der Java-Entwicklung viele Jahre lang drei verschiedene Objekte erstellen:

```
FileInputStream fileStream =
        new FileInputStream(fileName);
BufferedInputStream bufferedStream =
        new BufferedInputStream(fileStream);
ObjectInputStream objectStream =
        new ObjectInputStream(bufferedStream);
```

Ein `FileInputStream`-Objekt stellt nur rudimentäre I/O zur Verfügung – es kennt kein gepuffertes I/O und kann auch keine serialisierten Objekte lesen oder schreiben. Das `BufferedInputStream`-Objekt ermöglicht das Puffern eines `FileInputStream`, und der `ObjectInputStream` bietet dann zusätzlich das Lesen und Schreiben serialisierter Objekte. Die ersten beiden Objekte im Code (`fileStream` und `bufferedStream`) werden nie mehr verwendet, nachdem die Datei erst einmal geöffnet wurde – alle weiteren Operationen nutzen `objectStream`.

Es ist besonders nervig (und fehleranfällig), dass das Puffern explizit angefordert werden muss, indem ein eigenes `BufferedInputStream`-Objekt erstellt wird – vergisst ein Entwickler, dieses Objekt zu erzeugen, gibt es kein Puffern, und die I/O wird langsam sein. Vielleicht würde man bei der Java-Entwicklung argumentieren, dass nicht jeder die Datei-I/O puffern will, daher sollte es nicht in den zugrunde liegenden Mechanismus eingebaut sein. Eventuell wäre es ja besser, das Puffern separat zu halten, damit die Entwickler selbst entscheiden können, ob sie es nutzen

wollen oder nicht. Es ist zwar gut, die Wahl zu haben, aber **Schnittstellen sollten so designt sein, dass sie den häufigsten Fall so einfach wie möglich machen** (siehe die Formel auf Seite 20). So gut wie jede Anwendung der Datei-I/O wird einen Buffer nutzen wollen, daher sollte es standardmäßig angeboten werden. Für die wenigen Situationen, in denen Puffern nicht erwünscht ist, kann die Bibliothek einen Mechanismus anbieten, um es zu deaktivieren. Jeder Mechanismus, der das Puffern abschaltet, sollte in der Schnittstelle ganz klar getrennt sein (zum Beispiel durch einen eigenen Konstruktor für `FileInputStream` oder mithilfe einer Methode, die den Puffermechanismus abschaltet oder ersetzt), sodass sich die meisten Personen dessen Existenz beim Entwickeln gar nicht bewusst sind.

Im Gegensatz dazu haben die Designer der Unix-Systemaufrufe den häufigsten Fall einfach gestaltet. Sie haben beispielsweise erkannt, dass sequenzielles I/O am gebräuchlichsten ist, daher haben sie das zum Standardverhalten gemacht. Der wahlfreie Zugriff ist auch noch recht leicht mit dem Systemaufruf `lseek` zu nutzen, aber braucht man beim Entwickeln nur sequenziellen Zugriff, muss man diesen Mechanismus nicht kennen. Besitzt eine Schnittstelle viele Features, von denen aber fast immer nur wenige benötigt werden, ist die effektive Komplexität dieser Schnittstelle einfach die Komplexität der am häufigsten eingesetzten Features.

Zusammenfassung

Durch das Abtrennen der Schnittstelle eines Moduls von dessen Implementierung können wir die Komplexität der Implementierung vor dem Rest des Systems verbergen. Nutzt man ein Modul, muss man nur die durch die Schnittstelle bereitgestellte Abstraktion verstehen. Beim Design von Klassen und anderen Modulen ist es am wichtigsten, diese tief zu gestalten, sodass sie einfache Schnittstellen für die häufigsten Anwendungsfälle besitzen, aber trotzdem ausreichend Funktionalität bereitstellen. Das maximiert die Menge an Komplexität, die verborgen wird.