4 Klassen und Schnittstellen

Klassen und Schnittstellen bilden die grundlegenden Abstraktionseinheiten von Java und damit das Herzstück dieser Programmiersprache. Für den Entwurf Ihrer Klassen und Schnittstellen steht Ihnen in Java eine Vielzahl mächtiger Elemente zur Verfügung. Die Richtlinien in diesem Kapitel sollen Ihnen helfen, diese Elemente optimal zu nutzen, damit Ihre Klassen und Schnittstellen nutzbar, robust und flexibel sind.

4.1 Thema 15: Minimieren Sie den Zugriff auf Klassen und Member

Der wichtigste Faktor, der eine gute Komponente von einer schlechten unterscheidet, ist das Ausmaß, in dem die Komponente ihre internen Daten und andere Implementierungsdetails vor anderen Komponenten verbirgt. Eine gut durchdachte Komponente verbirgt alle ihre Implementierungsdetails und trennt die API sauber von der Implementierung. Die Komponenten kommunizieren dann nur noch über ihre APIs und kennen die Interna und inneren Abläufe der jeweils anderen Komponenten nicht. Dieses *Verbergen von Information*, auch *Datenkapselung* (encapsulation) genannt, ist ein grundlegendes Konzept des Softwaredesigns [Parnas72].

Das Verbergen von Informationen ist aus vielen Gründen wichtig, vor allem, weil es die Komponenten, die ein System bilden, *entkoppelt* und es so möglich ist, diese getrennt voneinander zu entwickeln, zu testen, zu optimieren, einzusetzen, nachzuvollziehen und zu modifizieren. Und es beschleunigt die Systementwicklung, da Komponenten parallel entwickelt werden können. Es erleichtert die Wartung, da die Komponenten schneller verstanden werden und sich leichter debuggen oder ersetzen lassen, ohne dass man befürchten muss, andere Komponenten zu beschädigen. Während das Verbergen von Informationen an sich nicht ausschlaggebend für gute Performance ist, schafft es die Voraussetzungen für ein effektives Performance-Tuning. Sobald ein System fertiggestellt ist und durch Profiling ermittelt wurde, welche Komponenten Performance-Probleme verursachen (Thema 67), können diese Komponenten optimiert werden, ohne die Funk-

tionsweise der anderen Komponenten zu beeinträchtigen. Das Verbergen von Informationen erhöht außerdem die Wiederverwendung von Software, da sich die entkoppelten Komponenten oft nicht nur in den Kontexten als nützlich erweisen, für die sie entwickelt wurden, sondern sich auch anderweitig einsetzen lassen. Und schließlich werden durch das Verbergen von Informationen die Risiken beim Erstellen großer Systeme reduziert, da sich einzelne Komponenten auch dann als gut erweisen können, wenn das System als Ganzes schlecht abschneidet.

Java bietet viel Unterstützung beim Verbergen von Informationen. Der *Zugriffskontrollmechanismus* [JLS, 6.6] spezifiziert die *Zugriffsrechte* auf Klassen, Schnittstellen und Member. Dabei bestimmt der Ort der Deklaration einer Entität und der in der Deklaration verwendeten Zugriffsmodifizierer (`private`, `protected` und `public`), sofern einer verwendet wird, wer wann wo auf die Entität zugreifen darf. Die korrekte Verwendung dieser Modifizierer ist für das Verbergen von Informationen unerlässlich.

Als Faustregel gilt: **Schränken Sie den Zugriff auf jede Klasse oder jeden Member so weit wie möglich ein.** Mit anderen Worten: Wählen Sie die niedrigstmögliche Zugriffsebene, bei der Ihre Software noch funktioniert.

Für Toplevel-Klassen und -Schnittstellen, das heißt für nicht geschachtelte Klassen und Schnittstellen, gibt es nur zwei mögliche Zugriffsebenen: *package private* und *öffentlich*. Wenn Sie eine Toplevel-Klasse oder -Schnittstelle mit dem Modifizierer `public` deklarieren, ist sie öffentlich, andernfalls ist sie package private. Wenn die Möglichkeit besteht, eine Toplevel-Klasse oder -Schnittstelle package private zu halten, sollten Sie davon Gebrauch machen. Indem Sie sie package private machen, wird sie Teil der Implementierung und nicht der exportierten API, das heißt, Sie können sie ändern, ersetzen oder in einer späteren Version entfernen, ohne Angst haben zu müssen, bestehenden Clients zu schaden. Wenn Sie diese Elemente als `public` deklarieren, sind Sie verpflichtet, sie aus Kompatibilitätsgründen dauerhaft zu unterstützen.

Wenn eine package private-Toplevel-Klasse oder -Schnittstelle nur von einer einzigen Klasse verwendet wird, sollten Sie in Erwägung ziehen, diese Toplevel-Klasse zu einer privaten statischen geschachtelten Klasse der sie verwendenden Klasse zu machen (Thema 24). Dann kann nur noch diese Klasse und keine der anderen Klassen im Paket darauf zugreifen. Viel wichtiger aber, als den Zugriff auf eine package private-Toplevel-Klasse zu begrenzen, ist es, den Zugriff auf eine grundlos als öffentlich deklarierte Klasse einzuschränken: Im Gegensatz zu der package private Toplevel-Klasse, die bereits Teil der Implementierung ist, gehört die öffentliche Klasse zur API des Pakets.

Für Member (Felder, Methoden, geschachtelte Klassen und geschachtelte Schnittstellen) gibt es vier verschiedene Zugriffsebenen, die hier geordnet aufgelistet sind:

- **private:** Der Zugriff auf den Member ist nur innerhalb der Toplevel-Klasse möglich, in der der Member deklariert ist.

- **package private:** Der Zugriff auf den Member ist innerhalb des Pakets, in dem der Member deklariert ist, von jeder Klasse aus möglich. Diese Zugriffsebene ist der *Standardzugriff*, den Sie erhalten, wenn Sie keinen Zugriffsmodifizierer angeben außer für Schnittstellen, die standardmäßig öffentlich sind.
- **protected**: Der Zugriff auf den Member ist von Subklassen der Klasse aus möglich, in der der Member deklariert ist (vorbehaltlich einiger Einschränkungen [JLS, 6.6.2]), sowie von jeder Klasse aus im Paket, in der der Member deklariert ist.
- **public**: Der Zugriff auf den Member kann von überall her erfolgen.

Nachdem Sie die öffentliche API Ihrer Klasse sorgfältig entworfen haben, sollten Sie erstmal alle anderen Member privat machen. Nur wenn eine andere Klasse des gleichen Pakets wirklich auf einen Member zugreifen muss, sollten Sie den `private`-Modifizierer entfernen und den Member package private machen. Wenn dies bei Ihnen häufiger vorkommt, sollten Sie das Design Ihres Systems überdenken und überlegen, ob eine andere Zerlegung zu Klassen führt, die besser voneinander getrennt gehalten werden können. Abgesehen davon sind sowohl private als auch package private Member Teil der Implementierung einer Klasse und haben normalerweise keinen Einfluss auf ihre exportierte API. Wenn die Klasse allerdings `Serializable` implementiert (Thema 86 und 87), können diese Felder in die exportierte API durchsickern.

Für die Member öffentlicher Klassen bedeutet es einen enormen Abbau der Zugriffsbeschränkungen, wenn die Zugriffsstufe von package private zu geschützt wechselt. Ein geschützter Member ist Teil der exportierten API der Klasse und muss für immer unterstützt werden. Außerdem repräsentiert ein geschützter Member einer exportierten Klasse ein öffentliches Versprechen für ein Implementierungsdetail (Thema 19). Der Bedarf an geschützten Membern sollte relativ gering sein.

Es gibt eine wichtige Regel, die Ihre Möglichkeiten reduziert, den Zugriff auf Methoden einzuschränken. Wenn eine Methode eine Methode der Superklasse überschreibt, kann der Zugriff darauf in der Subklasse nicht restriktiver sein als in der Superklasse [JLS, 8.4.8.3]. Dies ist notwendig, um sicherzustellen, dass eine Instanz der Subklasse überall dort verwendbar ist, wo eine Instanz der Superklasse verwendbar ist (das *Liskovsche Substitutionsprinzip*, siehe Thema 15). Wenn Sie gegen diese Regel verstoßen, erzeugt der Compiler bei dem Versuch, die Subklasse zu kompilieren, eine Fehlermeldung. Ein Sonderfall dieser Regel ist: Wenn eine Klasse eine Schnittstelle implementiert, müssen alle Methoden der Klasse, die auch in der Schnittstelle vorhanden sind, in der Klasse als öffentlich deklariert werden.

Um das Testen Ihres Codes zu erleichtern, könnten Sie versucht sein, den Zugriff auf eine Klasse, eine Schnittstelle oder einen Member leichter zu machen als notwendig. Das ist bis zu einem gewissen Grad in Ordnung. Es ist vertretbar, einen privaten Member einer öffentlichen Klasse zu Testzwecken package private

zu machen, aber es ist nicht vertretbar, die Zugriffsbeschränkungen noch weiter aufzuweichen. Mit anderen Worten, es ist nicht akzeptabel, eine Klasse, eine Schnittstelle oder einen Member zu einem Teil der exportierten API eines Pakets zu machen, um das Testen zu erleichtern. Glücklicherweise ist dies auch nicht notwendig, weil Tests als Teil des zu testenden Pakets ausgeführt werden können und so Zugriff auf seine package private Elemente erhalten.

Die Instanzfelder öffentlicher Klassen sollten möglichst nicht öffentlich sein (Thema 16). Wenn Sie ein Instanzfeld, das nicht-final ist oder eine Referenz auf ein veränderliches Objekt enthält, als `public` deklarieren, verzichten Sie auf die Möglichkeit, die Werte einzuschränken, die in diesem Feld gespeichert werden können. Das bedeutet, dass Sie die Möglichkeit aufgeben, Invarianten zu erzwingen, die das Feld betreffen. Außerdem verzichten Sie auf die Möglichkeit, Maßnahmen zu ergreifen, wenn das Feld geändert wird. Das bedeutet, **dass Klassen mit öffentlichen veränderlichen Feldern im Allgemeinen nicht threadsicher sind.** Selbst wenn ein Feld final ist und sich auf ein unveränderliches Objekt bezieht, verzichten Sie dadurch, dass sie es als `public` deklarieren, auf die Flexibilität, zu einer neuen internen Datenrepräsentation zu wechseln, in der das Feld nicht existiert.

Dasselbe gilt für statische Felder, mit einer Ausnahme. Sie können Konstanten über Felder offenlegen, die als `public static final` deklariert wurden, unter der Annahme, dass die Konstanten integraler Bestandteil der Abstraktion sind, die von der Klasse bereitgestellt wird. Per Konvention bestehen die Namen solcher Felder aus Großbuchstaben, wobei die einzelnen Wörter bei mehrteiligen Konstantennamen durch Unterstriche getrennt werden (Thema 68). Es ist wichtig, dass diese Felder entweder elementare Werte oder Referenzen auf unveränderliche Objekte enthalten (Thema 17). Ein Feld, das eine Referenz auf ein veränderliches Objekt enthält, weist alle Nachteile eines nicht-finalen Felds auf. Denn im Gegensatz zu den Referenzen, die nicht geändert werden können, sind Änderungen am referenzierten Objekt möglich – mit verheerenden Folgen.

Beachten Sie, dass ein Array mit einer Länge ungleich null immer veränderbar ist, sodass eine Klasse weder ein `public static final` Array-Feld aufweisen sollte noch einen Accessor, der ein solches Feld zurückgibt. Wenn eine Klasse ein solches Feld oder einen solchen Accessor hat, können Clients den Inhalt des Arrays ändern. Dies ist eine häufige Ursache von Sicherheitslücken:

```
// Potenzielle Sicherheitslücke!
public static final Thing[] VALUES =  { ... };
```

Beachten Sie, dass einige IDEs Accessoren generieren, die Referenzen auf private Array-Felder zurückliefern, was genau zu solchen Problemen führt. Es gibt zwei Möglichkeiten, das Problem zu beheben. Sie können das öffentliche Array privat machen und eine öffentliche unveränderliche Liste hinzufügen:

```
private static final Thing[] PRIVATE_VALUES = { ... };
```

```
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Oder Sie können das Array privat machen und eine öffentliche Methode hinzufügen, die eine Kopie eines privaten Arrays zurückliefert:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final Thing[] values() {
return PRIVATE_VALUES.clone();
}
```

Überlegen Sie bei Ihrer Entscheidung für eine der beiden Alternativen, was der Kunde eventuell mit dem Ergebnis zu tun gedenkt, welcher Rückgabetyp geeigneter ist und was hinsichtlich der Performance günstiger ist.

Ab Java 9 gibt es zwei zusätzliche, implizite Zugriffsebenen, die als Teil des *Modulsystems* eingeführt wurden. Ein Modul ist eine Bündelung von Paketen, so wie ein Paket eine Bündelung von Klassen ist. Ein Modul kann über *Export-Deklarationen* in seiner *Moduldeklaration*, die per Konvention in einer Quelldatei namens `module-info.java` enthalten ist, einige seiner Pakete explizit exportieren. Ein Zugriff auf öffentliche und geschützte Member von nicht exportierten Paketen in einem Modul ist von außerhalb des Moduls nicht möglich. Innerhalb des Moduls haben die Export-Deklarationen keinen Einfluss auf die Zugriffsrechte. Die Verwendung eines Modulsystems ermöglicht es den Paketen innerhalb eines Moduls, Klassen gemeinsam zu nutzen, ohne dass die Klassen für die ganze Welt sichtbar gemacht werden müssen. Die öffentlichen und geschützten Member von öffentlichen Klassen in nicht exportierten Paketen sind die Ursache zweier impliziter Zugriffsebenen, die intramodulare Analogien der normalen öffentlichen und geschützten Ebenen sind. Diese Art der gemeinsamen Nutzung auf Modulebene wird relativ selten benötigt und kann oft durch eine Reorganisation der Klassen innerhalb der Pakete eliminiert werden.

Im Gegensatz zu den vier Hauptzugriffsebenen sind die beiden modulbasierten Zugriffsebenen weitestgehend als Empfehlung zu verstehen. Wenn Sie die JAR-Datei eines Moduls in den Klassenpfad Ihrer Anwendung und nicht in den Modulpfad eintragen, weisen die Pakete im Modul wieder ihr nichtmodulares Verhalten auf: Alle öffentlichen und geschützten Member der öffentlichen Klassen der Pakete haben ihre normalen Zugriffsbeschränkungen, unabhängig davon, ob die Pakete vom Modul exportiert werden oder nicht [Reinhold, 1.2]. Der einzige Ort, an dem die neu eingeführten Zugriffsebenen strikt eingehalten werden, ist das JDK selbst: Auf die nicht exportierten Pakete in den Java-Bibliotheken kann außerhalb ihrer Module nicht zugegriffen werden.

Überlegen Sie genau, ob Sie Module benötigen. Der von den Modulen gebotene Zugriffsschutz ist für den typischen Java-Programmierer nur von begrenztem Nutzen und eigentlich nur eine Empfehlung. Bevor Sie in den Genuss der damit verbundenen Vorteile kommen, müssen Sie Ihre Pakete zu Modulen gruppieren, alle Abhängigkeiten explizit in Moduldeklarationen ausdrücken, Ihren

Verzeichnisbaum umorganisieren und spezielle Maßnahmen ergreifen, um den Zugriff auf nicht-modularisierte Pakete aus Ihren Modulen heraus zu ermöglichen [Reinhold, 3]. Es ist noch zu früh, um zu sagen, ob Module außerhalb des JDK eine breite Anwendung finden werden. In der Zwischenzeit sollten Sie sie besser vermeiden, es sei denn, es gibt einen zwingenden Grund für ihren Einsatz.

Zusammenfassend lässt sich sagen, dass Sie den Zugriff auf Programmelemente weitestgehend in angemessenem Rahmen einschränken sollten. Nachdem Sie sorgfältig eine minimale öffentliche API entwickelt haben, sollten Sie verhindern, dass einzelne Klassen, Schnittstellen oder Member Teil der API werden. Mit Ausnahme von `public static final`-Feldern, die als Konstanten dienen, sollten öffentliche Klassen keine öffentlichen Felder haben. Stellen Sie sicher, dass Objekte, die von `public static final`-Feldern referenziert werden, unveränderlich sind.

4.2 Thema 16: Verwenden Sie in öffentlichen Klassen Accessor-Methoden und keine öffentlichen Felder

Gelegentlich könnten Sie versucht sein, eine degenerierte Klasse zu schreiben, die nur dazu dient, Instanzfelder aufzunehmen:

```
// Degenerierte Klassen wie diese sollten nicht public sein!
class Point {
   public double x;
   public double y;
}
```

Da auf die Datenfelder solcher Klassen direkt zugegriffen wird, bieten diese Klassen nicht den Vorteil der *Datenkapselung* (Thema 15). Sie können die Repräsentation nicht ändern, ohne die API zu ändern, Sie können keine Invarianten erzwingen und Sie können keine Hilfsmaßnahmen ergreifen, wenn auf ein Feld zugegriffen wird. Den Hardlinern unter den objektorientierten Programmierern sind solche Klassen ein Gräuel und sollten ihrer Meinung nach immer durch Klassen mit privaten Feldern und öffentlichen *Accessoren* (Get-Methoden) und, bei veränderlichen Klassen, mit *Mutatoren* (Set-Methoden) ersetzt werden:

```
// Datenkapselung durch Accessoren und Mutatoren
class Point {
  private double x;
  private double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() { return x; }
```