4.4 Thema 18: Ziehen Sie Komposition der Vererbung vor

Vererbung ist ein mächtiges Tool zur Wiederverwendung von Code. Allerdings ist sie nicht immer die beste Alternative und führt bei inadäquater Verwendung zu anfälliger Software. Solange Sie Vererbung innerhalb eines Pakets einsetzen, werden Sie keine Probleme haben, da dann die Implementierungen von Subklasse und Superklasse unter der Kontrolle ein und desselben Programmierers stehen. Ebenfalls unproblematisch ist der Einsatz von Vererbung, wenn Klassen erweitert werden, die von Entwurf und Dokumentation her für Erweiterung konzipiert wurden (Thema 19). Werden jedoch normale konkrete Klassen über Paketgrenzen hinweg vererbt, kann es schnell gefährlich werden. Zur Erinnerung: Dieses Buch versteht unter dem Begriff »Vererbung« die *Implementierungsvererbung*, wenn eine Klasse eine andere erweitert. Die in diesem Thema behandelten Probleme gelten nicht für die *Schnittstellenvererbung*, wenn eine Klasse eine Schnittstelle implementiert oder eine Schnittstelle eine andere Schnittstelle erweitert.

Im Gegensatz zum Methodenaufruf verstößt die Vererbung gegen die Kapselung [Snyder86]. Mit anderen Worten, eine Subklasse hängt für ihr einwandfreies Funktionieren von den Implementierungsdetails ihrer Superklasse ab. Die Implementierung der Superklasse kann sich von einer Version zur nächsten ändern. Das kann für ihre Subklasse verheerende Folgen haben, auch wenn der Code der Subklasse nicht berührt wurde. Folglich muss eine Subklasse parallel zu ihrer Superklasse weiterentwickelt werden, es sei denn, die Autoren der Superklasse haben diese speziell für den Zweck der Erweiterung entworfen und dokumentiert.

Um dies an einem konkreten Beispiel zu veranschaulichen, nehmen wir an, wir haben ein Programm, das ein `HashSet` verwendet. Um die Leistung unseres Programms zu optimieren, müssen wir vom `HashSet` erfragen, wie viele Elemente seit seiner Erstellung hinzugefügt wurden. Verwechseln Sie das bitte nicht mit der aktuellen Größe, die sich verringert, wenn ein Element entfernt wird. Um diese Funktionalität zur Verfügung zu stellen, schreiben wir eine `HashSet`-Variante, die die Anzahl der versuchten Elementeinfügungen zählt und eine Accessor-Methode für diese Zählung exportiert. Die `HashSet`-Klasse enthält zwei Methoden (`add` und `addAll`), die Elemente hinzufügen können und die wir deshalb beide überschreiben:

```
// Fehlerhaft – Inadäquate Verwendung der Vererbung!
public class InstrumentedHashSet<E> extends HashSet<E> {
    // Zahl der versuchten Elementeinfügungen
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
```

```
        }
        @Override public boolean add(E e) {
            addCount++;
            return super.add(e);
        }
        @Override public boolean addAll(Collection<? extends E> c) {
            addCount += c.size();
            return super.addAll(c);
        }
        public int getAddCount() {
            return addCount;
        }
    }
```

Diese Klasse sieht auf den ersten Blick ordentlich aus, funktioniert aber leider nicht. Angenommen, wir erzeugen eine Instanz und fügen mit der Methode `addAll` drei Elemente hinzu. Übrigens erstellen wir dabei mit der seit Java 9 verfügbaren statischen Factory-Methode `List.of` eine Liste; wenn Sie mit einer früheren Java-Version arbeiten, verwenden Sie stattdessen:

```
Arrays.asList:
    InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
    s.addAll(List.of("Snap", "Crackle", "Pop"));
```

Wir würden erwarten, dass die `getAddCount`-Methode an dieser Stelle drei zurückgibt, aber ihr Rückgabewert lautet sechs. Was ist schiefgelaufen? Intern wurde die Methode `addAll` von `HashSet` auf der `add`-Methode implementiert, obwohl `HashSet` dieses Implementierungsdetail bewusst nicht dokumentiert. Die `addAll`-Methode in `InstrumentedHashSet` addierte drei zu `addCount` und rief dann über `super.addAll` die `addAll`-Implementierung von `HashSet` auf. Dadurch wiederum wurde die `add`-Methode, wie sie in `InstrumentedHashSet` überschrieben wurde, einmal für jedes Element aufgerufen, wobei `addCount` jeweils um eins erhöht wurde, was einer Gesamtzahl von insgesamt sechs entspricht: Jedes Element, das mit der `addAll`-Methode hinzugefügt wurde, wurde also doppelt gezählt.

Wir könnten die Subklasse reparieren, indem wir die überschriebene `addAll`-Methode löschen. Die resultierende Klasse würde zwar funktionieren, wäre dabei aber davon abhängig, dass die `addAll`-Methode von `HashSet` weiterhin auf der `add`-Methode implementiert bleibt. Diese Selbstnutzung ist ein Implementierungsdetail, das nicht in allen Implementierungen der Java-Plattform garantiert werden kann und sich unter Umständen von Version zu Version ändert. Deshalb wäre die resultierende `InstrumentedHashSet`-Klasse instabil.

Eine etwas bessere Lösung wäre, die `addAll`-Methode so zu überschreiben, dass sie über die angegebene Sammlung iteriert und die `add`-Methode einmal für jedes Element aufruft. Dies würde garantieren, dass das Ergebnis korrekt ist, unabhängig davon, ob die `addAll`-Methode von `HashSet` auf der `add`-Methode implementiert wurde oder nicht, da die `addAll`-Implementierung von `HashSet` nicht mehr aufgerufen würde. Diese Technik löst jedoch nicht alle unsere Pro-

bleme. Hierfür müssen wir nämlich die Methoden der Superklasse, die potenziell zur Selbstnutzung führen können, neu implementieren, was schwierig, zeitaufwendig sowie fehleranfällig ist und die Leistung senken kann. Außerdem ist es nicht immer möglich, da einige Methoden nicht ohne Zugriff auf private Felder implementiert werden können, auf die die Subklasse keinen Zugriff hat.

Was Subklassen ebenfalls instabil macht, ist die Tatsache, dass ihre Superklasse in späteren Versionen um neue Methoden erweitert werden kann. Angenommen, ein Programm hängt hinsichtlich seiner Sicherheit davon ab, dass alle Elemente, die in irgendeiner Sammlung eingefügt werden, irgendein Prädikat erfüllen. Dies kann gewährleistet werden, indem Sie die Sammlungsklasse ableiten und jede Methode dieser Subklasse überschreiben, die ein Element hinzufügen kann, um sicherzustellen, dass das Prädikat erfüllt ist, bevor das Element hinzugefügt wird. Dies funktioniert so lange, bis in einer späteren Version der Superklasse eine neue Methode hinzugefügt wird, die ein Element einfügen kann. Sobald dies geschieht, ist es möglich, allein durch Aufrufen der neuen Methode, die in der Subklasse nicht überschrieben wurde, ein illegales Element einzuschleusen. Und dies ist kein theoretisches Problem. Mehrere Sicherheitslücken dieser Art mussten bereits geschlossen werden, als `Hashtable` und `Vector` für das Collections-Framework nachgerüstet wurden.

Beide Probleme sind auf das Überschreiben von Methoden zurückzuführen. Möglicherweise denken Sie, dass eine Erweiterung einer Klasse sicher ist, wenn Sie lediglich neue Methoden hinzufügen und darauf verzichten, bestehende Methoden zu überschreiben. Diese Art der Erweiterung ist zwar viel sicherer, aber nicht ohne Risiko. Wenn die Superklasse in einer späteren Version eine neue Methode erhält und Sie dummerweise in der Subklasse eine Methode mit der gleichen Signatur, aber einem anderen Rückgabetyp deklariert haben, wird Ihre Subklasse nicht mehr kompilieren [JLS, 8.4.8.3]. Wenn Sie in der Subklasse eine Methode mit der gleichen Signatur und dem gleichen Rückgabetyp wie die neue Superklassenmethode deklariert haben, überschreiben Sie diese damit und haben die gleichen Probleme wie oben beschrieben. Außerdem darf bezweifelt werden, dass Ihre Methode den Vertrag der neuen Superklassenmethode erfüllt, da dieser Vertrag noch nicht geschrieben war, als Sie Ihre Subklassenmethode hinzugefügt haben.

Glücklicherweise gibt es eine Möglichkeit, alle oben beschriebenen Probleme zu umgehen. Anstatt eine bestehende Klasse zu erweitern, geben Sie Ihrer neuen Klasse ein privates Feld, das auf eine Instanz der bestehenden Klasse verweist. Dieses Design wird als *Komposition* bezeichnet, weil die bestehende Klasse zu einer Komponente der neuen wird. Jede Instanzmethode in der neuen Klasse ruft die entsprechende Methode auf der enthaltenen Instanz der bestehenden Klasse auf und gibt die Ergebnisse zurück. Dies wird als *Weiterleitung* (Forwarding) bezeichnet, und die Methoden der neuen Klasse *Weiterleitungsmethoden* genannt. Die resultierende Klasse ist absolut stabil und total unabhängig von den

Implementierungsdetails der bestehenden Klasse. Selbst das Hinzufügen neuer Methoden zur bestehenden Klasse hat keinen Einfluss auf die neue Klasse. Um dies an einem konkreten Beispiel zu veranschaulichen, betrachten wir eine neue Version von `InstrumentedHashSet`, die den Ansatz mit Komposition und Weiterleitung verfolgt. Beachten Sie, dass die Implementierung zweigeteilt ist, und zwar in die Klasse selbst und in eine wiederverwendbare *Weiterleitungsklasse*, die nur die Weiterleitungsmethoden enthält:

```
// Wrapper-Klasse – verwendet Komposition anstelle von Vererbung
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
     }
     @Override public boolean addAll(Collection<? extends E> c) {
         addCount += c.size();
         return super.addAll(c);
     }
     public int getAddCount() {
         return addCount;
     }
}

// Wiederverwendbare Weiterleitungsklasse
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }

    public void clear()               { s.clear();            }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()          { return s.isEmpty();   }
    public int size()                 { return s.size();      }
    public Iterator<E> iterator()     { return s.iterator();  }
    public boolean add(E e)           { return s.add(e);      }
    public boolean remove(Object o)   { return s.remove(o);   }
    public boolean containsAll(Collection<?> c)
                                      { return s.containsAll(c); }
    public boolean addAll(Collection<? extends E> c)
                                      { return s.addAll(c);      }
    public boolean removeAll(Collection<?> c)
                                      { return s.removeAll(c);   }
    public boolean retainAll(Collection<?> c)
                                      { return s.retainAll(c);   }
    public Object[] toArray()          { return s.toArray();  }
    public <T> T[] toArray(T[] a)      { return s.toArray(a); }
```

```
    @Override public boolean equals(Object o)
                                    { return s.equals(o);  }
    @Override public int hashCode()    { return s.hashCode(); }
    @Override public String toString() { return s.toString(); }
}
```

Das Design der `InstrumentedSet`-Klasse wird durch die `Set`-Schnittstelle ermöglicht, die die Funktionalität der `HashSet`-Klasse aufnimmt. Dieser Entwurf ist nicht nur robust, sondern auch äußerst flexibel. Die Klasse `InstrumentedSet` implementiert die `Set`-Schnittstelle und hat einen einzigen Konstruktor, dessen Argument ebenfalls vom Typ `Set` ist. Im Grunde transformiert die Klasse ein `Set` in ein anderes und fügt die Instrumentierungsfunktionalität hinzu. Im Gegensatz zum vererbungsbasierten Ansatz, der nur mit einer einzigen konkreten Klasse funktioniert und für jeden unterstützten Konstruktor in der Superklasse einen separaten Konstruktor erfordert, kann die Wrapper-Klasse zur Instrumentierung jeder `Set`-Implementierung verwendet werden und funktioniert mit jedem bereits vorhandenen Konstruktor:

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
```

Die `InstrumentedSet`-Klasse kann sogar verwendet werden, um vorübergehend eine `Set`-Instanz zu instrumentieren, die bereits ohne Instrumentierung verwendet wurde:

```
static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);
    ... // Verwende in dieser Methode iDogs statt dogs
}
```

Die `InstrumentedSet`-Klasse wird als *Wrapper*-Klasse oder auch *Hüllklasse* bezeichnet, da jede `InstrumentedSet`-Instanz eine andere `Set`-Instanz enthält (umhüllt). Dies wird auch als *Dekorator-Muster* bezeichnet [Gamma95], da die `InstrumentedSet`-Klasse ein Set durch Hinzufügen von Instrumentierung dekoriert. Manchmal wird die Kombination aus Komposition und Weiterleitung etwas frei als *Delegation* bezeichnet. Technisch gesehen ist es jedoch keine Delegation, es sei denn, das Hüllobjekt übergibt sich selbst an das umhüllte Objekt [Lieberman86; Gamma95].

Wrapper-Klassen haben nur wenige Nachteile. Eine Einschränkung ist, dass Wrapper-Klassen sich nicht für *Callback-Frameworks* eignen, in denen Objekte Referenzen auf sich selbst für spätere Aufrufe (Callbacks) an andere Objekte übergeben. Da ein umhülltes Objekt von seinem Wrapper nichts weiß, übergibt es eine Referenz auf sich selbst (`this`), und dem Wrapper entgehen die Callbacks. Dies nennt man auch das *SELF-Problem* [Lieberman86]. Einige machen sich Sorgen darüber, wie sich die Aufrufe von Weiterleitungsmethoden auf die Performance auswirken oder inwiefern Wrapper-Objekte den Speicherbedarf beeinträchtigen können. Beides spielt in der Praxis kaum eine Rolle. Es ist zwar mühsam, Weiterlei-

tungsmethoden zu schreiben, aber Sie müssen die wiederverwendbare Weiterleitungsklasse für jede Schnittstelle nur einmal schreiben, und manchmal stehen sogar schon Weiterleitungsklassen zur Verfügung. Beispielsweise stellt Guava Weiterleitungsklassen für alle Sammlungs-Schnittstellen [Guava] bereit.

Vererbung ist nur dann sinnvoll, wenn die Subklasse tatsächlich ein *Subtyp* der Superklasse ist. Mit anderen Worten, eine Klasse *B* sollte eine Klasse *A* nur dann erweitern, wenn eine »ist-eine«-Beziehung zwischen den beiden Klassen besteht. Wenn Sie überlegen, mit einer Klasse *B* eine Klasse *A* zu erweitern, stellen Sie sich zuerst folgende Frage: Ist wirklich jedes *B* ein *A*? Wenn Sie diese Frage nicht wahrheitsgemäß mit Ja beantworten können, sollte *B* nicht die Klasse *A* erweitern. Wenn die Antwort Nein lautet, ist es oft der Fall, dass *B* eine private Instanz von A enthält und eine andere API offenlegen sollte: *A* ist kein wesentlicher Bestandteil von *B*, sondern lediglich ein Detail seiner Implementierung.

In den Java-Plattform-Bibliotheken wird gegen dieses Prinzip wiederholt verstoßen. Zum Beispiel ist ein Stack kein Vektor, daher sollte die Klasse `Stack` nicht die Klasse `Vector` erweitern. Ebenso ist eine Eigenschaftsliste keine Hashtabelle, sodass `Properties` nicht `Hashtable` erweitern sollte. In beiden Fällen hätte man eine Komposition vorziehen sollen.

Wenn Sie Vererbung verwenden, wo eigentlich Komposition die bessere Wahl ist, werden unnötig Implementierungsdetails offengelegt. Die resultierende API bindet Sie an die ursprüngliche Implementierung und schränkt die Performance Ihrer Klasse für immer ein. Gravierender jedoch ist, dass Sie durch die Offenlegung der Interna den Clients den direkten Zugriff darauf ermöglichen. Im besten Fall ist die Folge eine verwirrende Semantik. Wenn zum Beispiel `p` auf eine `Properties`-Instanz verweist, dann kann `p.getProperty(key)` andere Ergebnisse liefern als `p.get(key)`: Die erste Methode berücksichtigt Standardwerte, die zweite von `Hashtable` geerbte Methode nicht. Am schlimmsten ist, dass der Client die Invarianten der Subklasse korrumpieren kann, indem er die Superklasse direkt modifiziert. Im Falle von `Properties` hatten die Designer vor, dass nur Strings als Schlüssel und Werte erlaubt sind, aber der direkte Zugriff auf die zugrunde liegende `Hashtable` erlaubt es, diese Invariante zu verletzen. Ist diese erst einmal verletzt, können andere Teile der `Properties`-API (`load` und `store`) nicht mehr verwendet werden. Als dieses Problem zutage trat, war es für eine Korrektur zu spät, da es bereits Clients gab, die von der Verwendung von Nicht-String-Schlüsseln und -Werten abhingen.

Es gibt noch ein paar letzte Fragen, die Sie sich stellen sollten, bevor Sie der Vererbung den Vorzug vor der Komposition geben. Hat die Klasse, die Sie zu erweitern gedenken, irgendwelche Fehler in ihrer API? Wenn ja, fühlen Sie sich wohl damit, diese Fehler in die API Ihrer Klasse zu übernehmen? Bei der Vererbung werden alle Fehler in der API der Superklasse weitergereicht, während Sie bei Komposition eine neue API entwerfen können, die diese Fehler verbirgt.

Zusammenfassend lässt sich sagen, dass Vererbung mächtig, aber problematisch ist, weil sie gegen die Kapselung verstößt. Sie ist nur dann sinnvoll, wenn eine echte Subtyp-Beziehung zwischen der Subklasse und der Superklasse besteht. Auch dann kann Vererbung die Stabilität gefährden, wenn die Subklasse in einem anderen Paket als die Superklasse liegt und die Superklasse nicht für Vererbung entworfen wurde. Um diese Instabilität zu vermeiden, verwenden Sie Komposition und Weiterleitung statt Vererbung, insbesondere wenn eine geeignete Schnittstelle zur Implementierung einer Wrapper-Klasse existiert. Wrapper-Klassen sind nicht nur robuster als Subklassen, sondern auch leistungsfähiger.

4.5 Thema 19: Entwerfen und dokumentieren Sie für Vererbung oder verbieten Sie sie

In Thema 18 wurden Sie auf die Gefahren hingewiesen, wenn Sie eine Klasse von einer fremden Klasse ableiten, die nicht für die Vererbung konzipiert und dokumentiert wurde. Was bedeutet es also, wenn eine Klasse für die Vererbung entworfen und dokumentiert ist?

Erstens muss die Klasse genau dokumentieren, welche Auswirkungen das jeweilige Überschreiben der einzelnen Methoden hat. Mit anderen Worten, **die Klasse muss dokumentieren, wie sie selbst die überschreibbaren Methoden verwendet.** Für jede öffentliche oder geschützte Methode muss in der Dokumentation angegeben werden, welche überschreibbaren Methoden diese Methode aufruft, in welcher Reihenfolge und wie sich die Ergebnisse der einzelnen Aufrufe auf die nachfolgende Verarbeitung auswirken. Mit *überschreibbar* meinen wir nicht-final und entweder öffentlich oder geschützt. Generell muss eine Klasse alle Umstände dokumentieren, unter denen sie eine überschreibbare Methode aufrufen darf. Beispielsweise könnten Aufrufe von Hintergrund-Threads oder statischen Initialisierern erfolgen.

Eine Methode, die überschreibbare Methoden aufruft, enthält am Ende ihres Dokumentationskommentars eine Beschreibung dieser Aufrufe. Die Beschreibung ist in einem speziellen Abschnitt der Spezifikation unter der Bezeichnung »Implementation Requirements« (Implemenierungsanforderungen) zu finden, der mit dem Javadoc-Tag `@implSpec` generiert wird. Dieser Abschnitt informiert Sie über die Interna der Methode. Das folgende Beispiel wurde aus der Spezifikation für `java.util.AbstractCollection` kopiert:

```
public boolean remove(Object o)
```

Entfernt, sofern vorhanden, eine einzelne Instanz des angegebenen Elements aus dieser Sammlung (optionale Operation). Formaler ausgedrückt, entfernt ein Element `e` *so, dass* `Objects.equals(o, e)`*, wenn diese Sammlung ein oder mehrere solcher Elemente enthält. Liefert* `true` *zurück, wenn diese Sammlung das angegebene Element enthalten hat (oder sich diese Sammlung infolge des Aufrufs geändert hat).*