PDF-Anhang zum Buch:

Testen von Software und Embedded Systems

Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen

Uwe Vigenschow
uwe@vigenschow.com

2. überarbeitete und aktualisierte Auflage
Erscheinungjahr: 2010
Verlag: dpunkt.verlag, Heidelberg
Website zum Buch: www.oo-testen.de
Stand: 3. September 2010

Lektorat: Christa Preisendanz
Copy Editing: Ursula Zimpfer
Satz: Uwe Vigenschow

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne schriftliche Genehmigung des Verfassers urheberrechtswidrig und damit strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die in diesem Dokument verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichen Schutz unterliegen.

Alle Angaben und Programme in diesem Dokument wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die im Zusammenhang mit der Verwendung dieses Dokuments entstehen.

Inhaltsverzeichnis

A Unified Modeling Language 2

In diesem Buch verwende ich die UML 2 zur Visualisierung. Vielleicht sind nicht jedem Leser die zum vollen Verständnis der Diagramme notwendigen Aspekte geläufig. Daher werden in diesem Abschnitt kurz die wichtigsten Elemente erläutert, um jedem Leser das Nachvollziehen der Beispiele zu ermöglichen [2, 4].

Die insgesamt dreizehn UML-Diagramme werden in zwei grundlegende Arten unterteilt: Struktur- und Verhaltensdiagramme. Zu den Strukturdiagrammen zählt das Klassendiagramm, während Aktivitäts-, Zustands- und die Interaktionsdiagramme zu den Verhaltensdiagrammen gehören. Es gibt vier Arten von Interaktionsdiagrammen, wovon wir das Sequenz- und das Timing-Diagramm verwendet haben (Abb. A.1). In Anhang A.2 kommt noch das Interaktionsübersichtsdiagramm als dritte Form hinzu. In den Beispielen sind nur die sechs, in der Abbildung schwarz gedruckten UML-Diagrammarten verwendet worden.

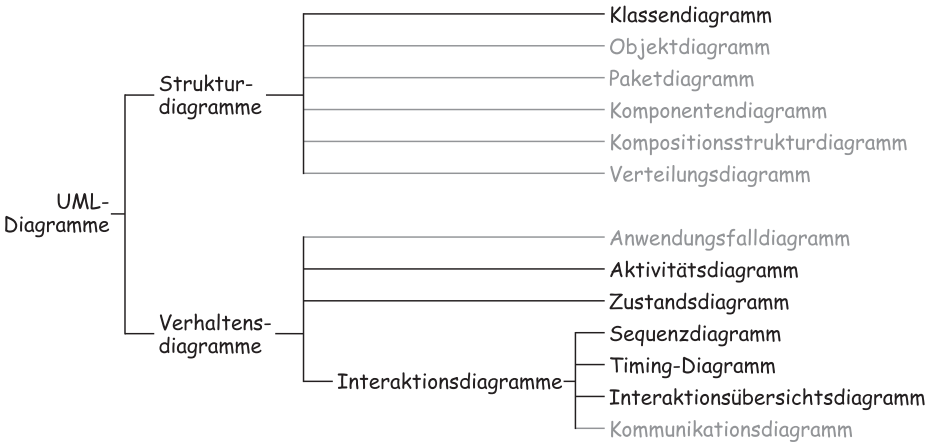


Abbildung A.1: Übersicht der UML-Diagramme

Mit Strukturdiagrammen wie einem Klassendiagramm können wir dauerhafte, also statische, strukturelle Aspekte und Regeln modellieren. Über die Modellierung mit Verhaltensdiagrammen definieren wir die Dynamik auf

der Struktur. Diese Art der Trennung ist sehr praktisch und wir finden sie z. B. auch in jedem U-Bahnhof wieder. Das dauerhafte Streckennetz mit seinen Bahnhöfen, Gleisverbindungen und Umsteigemöglichkeiten wird in einem Netzplan, einer Art *Strukturdiagramm*, dargestellt. Die Dynamik auf dem Schienennetz entnehmen wir einem *Verhaltensdiagramm*, dem Fahrplan, mit den genauen Ankunfts- und Abfahrzeiten.

A.1 Klassen, Vererbung und Assoziationen

A.1.1 Klassen, Attribute und Operationen

Eine Klasse ist eine Art Bauplan für gleichartige Objekte, in dem Eigenschaften und Verhalten festgelegt werden können. Die Eigenschaften werden über Attribute definiert und das Verhalten über Operationen modelliert. Sowohl für Attribute wie auch für Operationen können Regeln in Form von Zusicherungen gegeben werden (Abb. A.2).

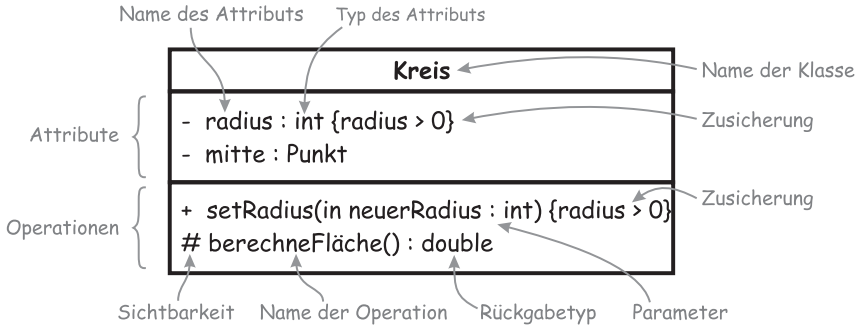


Abbildung A.2: Eine Klasse in UML mit Attributen, Operationen und Zusicherungen

Damit wir sowohl die Attribute einer Klasse nach außen hin verstecken können als auch exakt definieren können, welche Operationen von anderen Klassen aufgerufen werden dürfen, gibt es das Konzept der Sichtbarkeit. Darüber stellen wir die Kapselung von klasseninternen Aspekten sicher. In Tabelle A.1 sind die vier Möglichkeiten dargestellt.

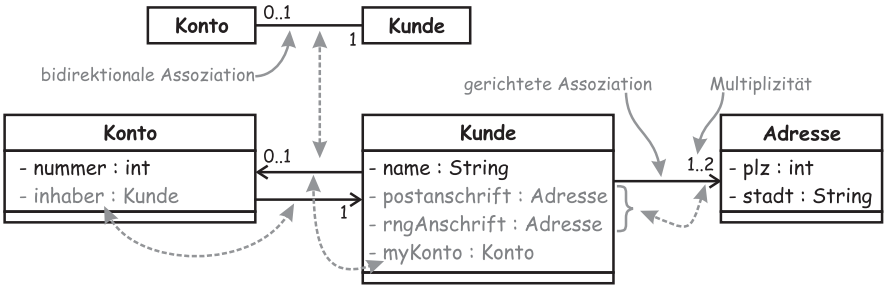
Bei der Parameterliste von Operationen können wir über die Präfixe `in`, `out` bzw. `inout` modellieren, ob ein Parameter nur hineingeht, herauskommt oder beides. Darüber wird also ein call-by-value (Kopie des Parameterobjekts) bzw. call-by-reference (Zeiger bzw. Referenz auf das Parameterobjekt) modelliert.

Symbol	Bedeutung
-	private: nur innerhalb der Klasse sichtbar
+	public: uneingeschränkt sichtbar
#	protected: nur innerhalb der Vererbungshierarchie von der Klasse an abwärts sichtbar
~	package: nur innerhalb des Pakets sichtbar

Tabelle A.1: Die vier möglichen Sichtbarkeiten von Attributen und Operationen in der UML

A.1.2 Assoziationen zwischen Klassen

Der Bauplan einer Klasse beschreibt die statische Struktur der Objekte. Ein Attribut vom Typ einer anderen Klasse verbindet zwei Klassen dauerhaft und eng miteinander. Über dieses Attribut können die public-Methoden der anderen Klasse aufgerufen werden. Diese Verbindung heißt *Assoziation* (Abb. A.3).




```
public class Konto {
  int nummer;
  Kunde inhaber;
...
}
```

```
public class Kunde {
  String name;
  Adresse postanschrift;
  Adresse rngAnschrift;
  Konto myKonto;
...
}
```

```
public class Adresse{
  int plz;
  String stadt;
...
}
```

Abbildung A.3: Assoziationen zwischen Klassen in UML (oben) und im Code (unten)

Im Code gibt es kein Element *Assoziation*. Wir sehen nur das Ziel der Assoziation als Attribut. Eine Assoziation ist daher stets unidirektional. Eine bidirektionale Assoziation besteht aus zwei voneinander unabhängigen unidirektionalen Assoziationen (Abb. A.3 links oben). In der Modellierung sind

daher die Assoziation und das entsprechende Attribut äquivalent zueinander und es wird in der Regel nur die Assoziation im Klassendiagramm dargestellt und nicht zusätzlich das Attribut. Daher sind die entsprechenden Attribute, die mit Assoziationen modelliert sind, in Abb. A.3 grau dargestellt. Auf die zugeordnete Assoziation weist jeweils ein gestrichelter Doppelpfeil hin.

Über die Multiplizität an einer Assoziation können die Mengengerüste modelliert werden. Dies ist entweder eine konstante Anzahl wie z. B. `1`, eine Bereich wie z. B. `0..5` oder `*`, womit 0 bis beliebig viele, also eine Collection, modelliert werden kann.

A.1.3 Generalisierung und Vererbung

Neben der Assoziation gibt es noch die Generalisierung von Klassen. Gemeinsamkeiten werden dabei in eine Oberklasse geschoben, von der Unterklassen erben. Wie wir im Buch [5] gesehen haben, birgt dieses Konzept seine Risiken. Unproblematisch ist hingegen die Erweiterung von Interfaces über die Generalisierung. In der UML wird die Realisierung eines Interface von der Generalisierung unterschieden, obwohl beides in Sprachen wie C++ über die Generalisierung erfolgt. Die Darstellung ist auch ähnlich (Abb. A.4).

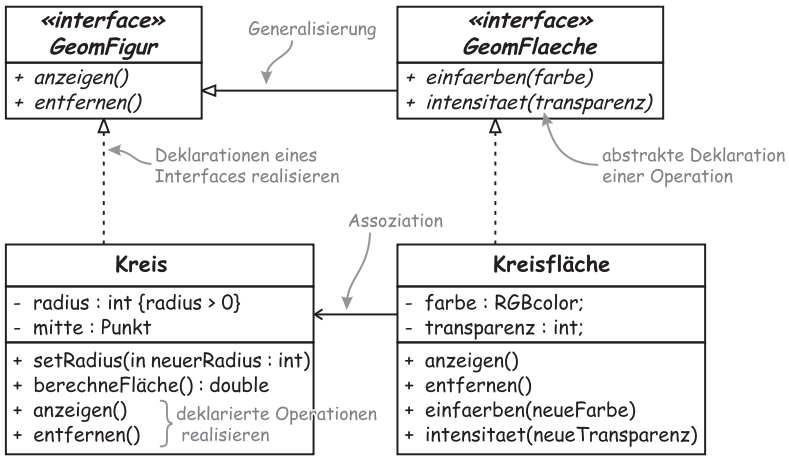


Abbildung A.4: Generalisierung von Klassen und Interfaces und die Realisierung von Interfaces in UML

A.1.4 Interfaces, Komponenten und Ports

Interfaces sind rein abstrakte Klassen, in denen nur Operationen deklariert und Konstanten definiert werden können. Wie in Abb. A.4 zu sehen, wird Abstraktion durch kursive Schrift angezeigt. Dies ist nur schwer erkennbar, weshalb im Buch öfter zusätzlich das Merkmal {`abstract`} verwendet wurde.

Wenn wir eine Gruppe von Klassen zu einer Komponente mit gemeinsamer Verantwortlichkeit zusammenfassen, so sollten die Zugriffe in und aus einer Komponente über Interfaces entkoppelt sein. Um dabei die Übersicht zu behalten, gibt es eine Kurznotation dafür: die Ball-and-Socket-Notation (Abb. A.5).

Abbildung A.5: Komponenten, Interfaces und Ports in UML

Über die Ball-and-Socket-Notation kann sofort zwischen einem bereitgestellten (Ball) und einem benötigten (Socket) Interface unterschieden werden. Der Name des Interface wird neben das Symbol geschrieben. *Classifier* aus dem Metamodell der UML wie Klassen oder Komponenten können mit solchen Interfaces versehen werden.

Läuft die Kommunikation mit dem Interface über einen gekapselten Kommunikationspunkt, so wird dies durch einen Port ausgedrückt. Das Symbol dafür ist ein kleines Quadrat am Rand des Rechtecks des Classifiers, von dem ein oder mehrere Interfaces ausgehen (Abb. A.5, links). Der Classifer, also z. B. eine Komponente, ist dann darüber gekapselt oder wie es im UML-Metamodell heißt: Aus einem Classifier wird ein Encapsulated-Classifier.

Technisch werden Ports in der Regel durch einzelne Klassen, sogenannte Portklassen, realisiert. Darin können wir zusätzliche Funktionalität implementieren, um z. B. Filterungen oder ein Caching zu ermöglichen oder ein definiertes Protokoll zu überwachen.

A.2 Aktivitätsdiagramme und Interaktionsübersicht

Mit Aktivitätsdiagrammen können wir Abläufe modellieren. Dies kann ein Prozess sein, ein Anwendungsfall oder ein Schritt innerhalb eines An-

wendungsfalls. Über die den Aktivitätsdiagrammen hinterlegte Petrinetz-Semantik ergeben sich breite Einsatzmöglichkeiten über das objektorientierte Paradigma hinaus. Der Ablauffluss wird dabei durch sog. *Token* gesteuert. Hier wird die Nähe zu den Petrinetzen mit ihren Marken deutlich. In den Abb. A.6 bis A.10 sind die im Buch verwendeten Elemente der Aktivitätsdiagramme in grau benannt.

Ein einzelner Schritt in einer Aktivität heißt *Action*. Wie wir gesehen haben, können Aktionen über Kontroll- und Objektflüsse miteinander verbunden werden und so einen Ablauf beschreiben. Mit der zusätzlichen Möglichkeit des Objektflusses neben dem Kontrollfluss[1] kann auch der dynamische Aspekt einer objektorientierten Zerlegung bis in das Softwaredesign hinein modelliert werden (Abb. A.6).

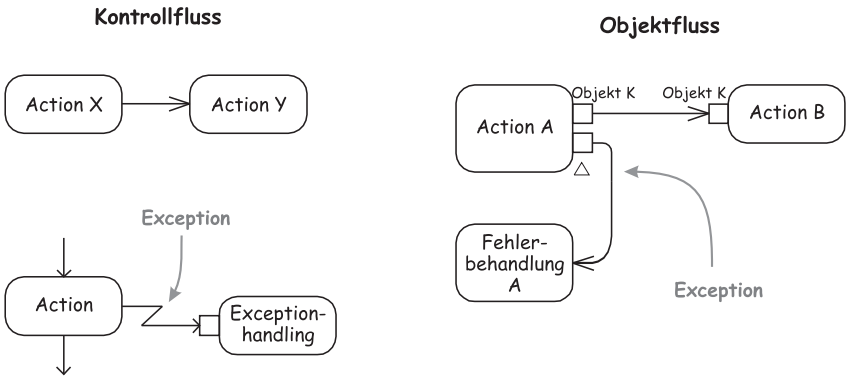


Abbildung A.6: In der UML 2 wird eine Action innerhalb einer Aktivität als Rechteck mit abgerundeten Ecken dargestellt. Für den Objektfluss (rechts) werden kleine Quadrate, die sog. Pins, an die Action angesetzt. Eine Exception kann im Objektfluss durch ein kleines Dreieck am Pin dargestellt werden (rechts) oder als blitzförmiger Pfeil im Kontrollfluss (links unten).

Wenn bei einem Kontrollfluss mehrere Kanten aus einer Aktion herauskommen bzw. hineingehen, bedeutet dies eine implizite Synchronisation bzw. ein implizites Aufspalten (Splitting) des Kontrollflusses (Abb. A.7 unten). Wenn diese Semantik beim Modellieren nicht bedacht wird, können schnell ungewollte Effekte im Modell entstehen wie z. B. Deadlocks.

In der Semantik mehrerer ein- bzw. ausgehender Kanten liegt eine Inkompatibilität zwischen der UML 1.x und der UML 2.0. Diese ist für UML-erfahrene Teams oder lang laufende Projekte eine potenzielle Fehlerquelle,

[1] Die korrekte Übersetzung des englischen *control flow* lautet natürlich Steuerungsfluss. Leider hat sich in der deutschsprachigen Fachliteratur der Begriff *Kontrollfluss* etabliert [2, 4, 6].

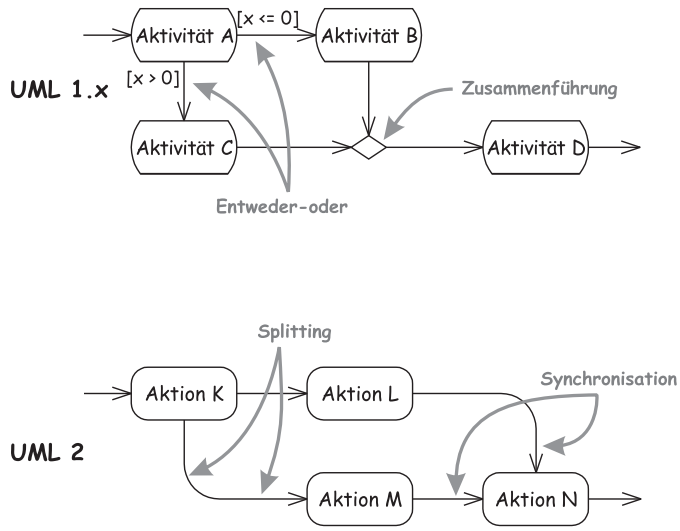


Abbildung A.7: Kontrollfluss im Vergleich UML 1.x (oben) zur UML 2 (unten). Die Semantik des Aufspaltens des Kontrollflusses hat sich geändert!

für die eine einfache Lösung existiert: Es gibt nur eine eingehende und eine ausgehende Kante. Entscheidungen bzw. Splittings und deren Synchronisation werden explizit modelliert (Abb. A.8).

Mit den Aktivitätsdiagrammen ist noch wesentlich mehr möglich. Ein Aktivitätsschritt kann als Analogie zu einer Klassenmethode interpretiert werden. Die Ein- und Ausgänge beschreiben dann die In- bzw. Out-Parameter der Methode. Besonders deutlich wird dies bei der Betrachtung des Objektflusses. Der Objektfluss wird durch die sog. *Pins* an einer Aktion gekennzeichnet, an denen der Name des Objekts bzw. Parameters steht (Abb. A.9).

In Abb. A.9 ist auch die Modellierung von Signalen und Ausnahmen dargestellt. Signale können gesendet oder empfangen werden. Das Empfangen eines Signals kann dabei ähnlich wie ein Startpunkt als Quelle für die imaginären Tokens sein, an denen wir ähnlich einem Cursor die aktuelle Position ausmachen, an der wir uns gerade im Ablauf einer Aktivität befinden.

Für den Kontroll- und Objektfluss gibt es eigene Arten, eine Ausnahme bzw. Exception zu modellieren. Im Kontrollfluss erkennen wir einen solchen exklusiven Ausgang an dem Blitzknick in der Transition (Abb. A.6 links) bzw. als Blitzknicksymbol oberhalb einer Transition (Abb. A.9 rechts oben).

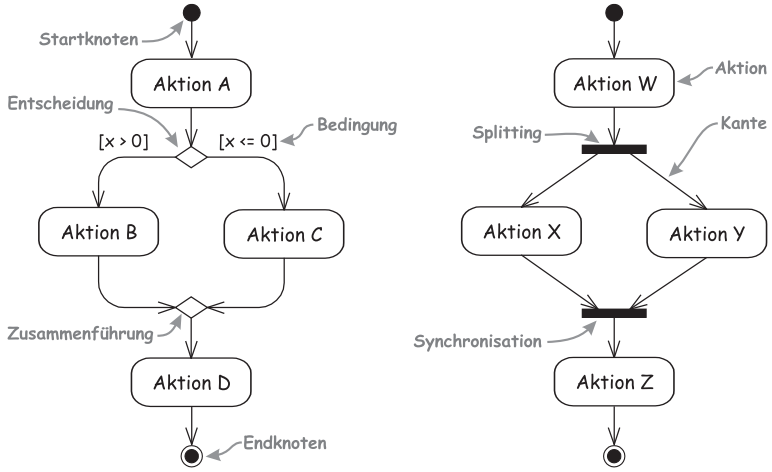


Abbildung A.8: Durch das explizite Herausziehen wird eine kompatible Darstellung von Entscheidungen bzw. Splitting und Synchronisation in der UML 1.x und UML 2.0 erreicht. Wenn wir so wie in diesem Beispiel in UML 2 modellieren, sind wir UML 1.x-kompatibel.

Im Objektfluss wird ein Ausnahmeparameter als Dreieck am Pin dargestellt (Abb. A.6 rechts und A.9 links in der Mitte).

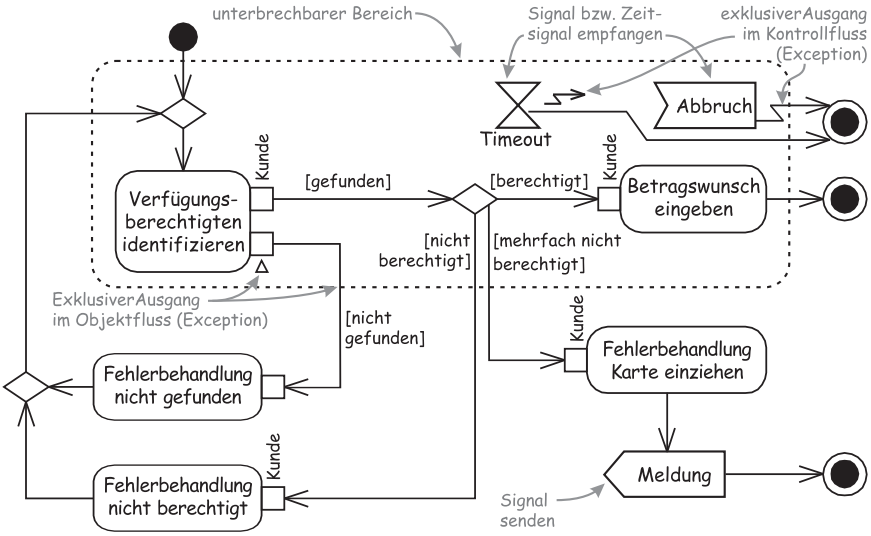


Abbildung A.9: Für das Umfeld der Action *Verfügungsberechtigten identifizieren* wurde der Objektfluss modelliert.

Zur Illustration der Möglichkeiten von Aktivitätsdiagrammen sind in den Abb. A.9 und A.10 Detaillierungen der Action *Verfügungsberechtigten identifizieren* aus dem Geldautomatenbeispiel aus [5] zu sehen. Über die hierarchische Aufteilung, also die detaillierte Modellierung einer Aktivität in einem eigenen Diagramm, können wir sowohl den Überblick behalten wie auch wichtige Detailinformation dokumentieren.

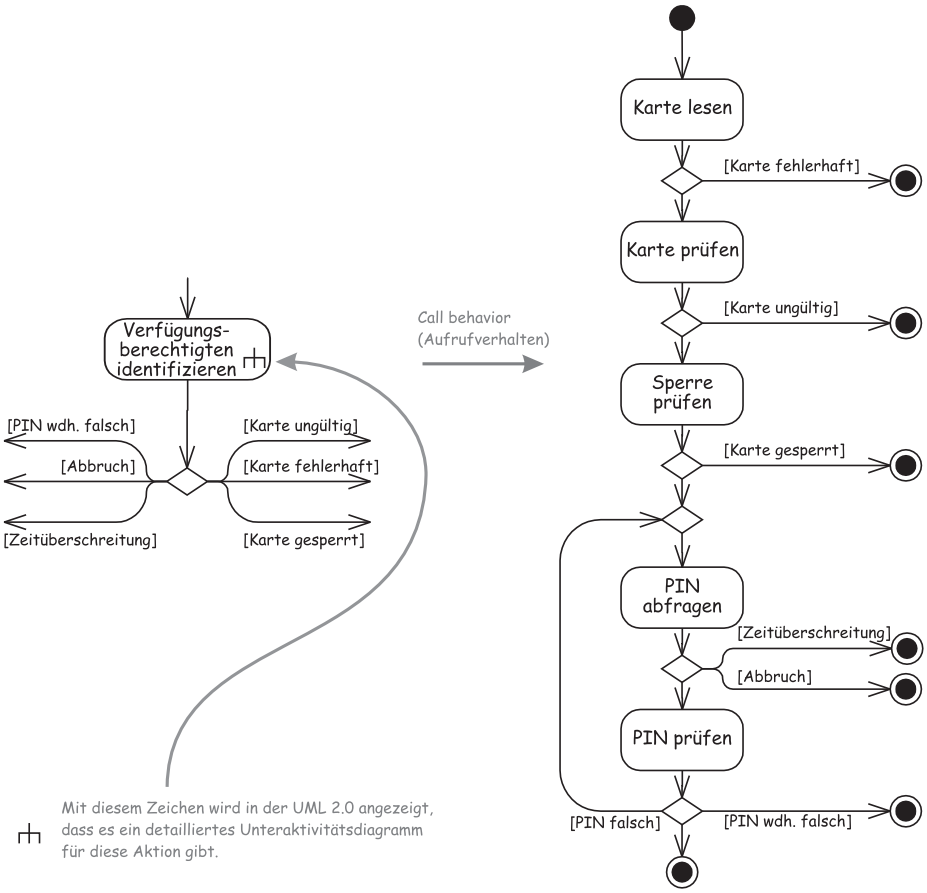


Abbildung A.10: Was kann alles innerhalb der Aktion *Verfügungsberechtigten identifizieren* schiefgehen? Hier wurde der Kontrollfluss detailliert heruntergebrochen. Die Testfälle zu finden ist nun quasi ein Kinderspiel.

Wir können so wunderbar bis auf Klassenmethodenebene die Abläufe und Objektflüsse modellieren. Der Wert für die Testfallfindung ist offensichtlich. Ausnahmen, Varianten und Fehler lassen sich detailliert modellieren. Selbst Exceptions und ihre Fehlerbehandlung sind darstellbar, wie in Abb. A.9 exemplarisch gezeigt. Der Ausnahmeparameter einer Exception wird

durch ein kleines Dreieck am Pin symbolisiert (Abb. A.6). Genau wie bei programmierten Exceptions handelt es sich hierbei um einen exklusiven Ausgang, der im Fehlerfall durchlaufen wird.

Gerade in der Beschreibung von Testfällen kann es hilfreich sein, Aktivitätsdiagramme und Sequenzdiagramme zu kombinieren. Eine Interaktionsübersicht ist ein Aktivitätsdiagramm, in dem Teilabläufe durch referenzierte oder eingebettete Sequenzdiagramme repräsentiert werden. So können wir konkrete Detailabläufe als Sequenzdiagramm modellieren und in einen übersichtlichen Gesamtzusammenhang und gemeinsamen Ablauf bringen (Abb. A.11).

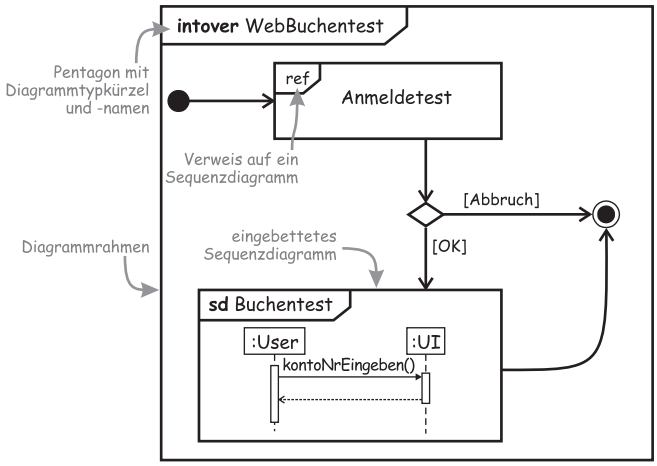


Abbildung A.11: Zentrale Elemente einer Interaktionsübersicht

In Abb. A.11 sehen wir auch die Möglichkeit, einem Diagramm einen Rahmen mit Diagramminformation oben links in einem Pentagon zu geben. Die Art eines Diagramms wird dort über ein Kürzel benannt. Diese lauten für die in diesem Buch verwendeten Diagrammarten `cd` für Klassendiagramm, `ad` für Aktivitätsdiagramm, `sd` für Sequenzdiagramm und `intover` oder `interaction` für Aktivitätsübersichten. Auf dieses Kürzel folgt direkt der Name des Diagramms.

Wir werden früher oder später nicht darum herumkommen, uns intensiver mit der UML 2 zu beschäftigen. Die OMG bietet dazu eine dreistufige Zertifizierung für UML 2 an [6]. Die entsprechenden Begriffe und Elemente sind dafür wie Vokabeln zu lernen und das Metamodell bzw. für die höchste Stufe auch das *Meta Object Facility* (MOF) müssen verstanden sein. Damit möchte die OMG sicherstellen, dass alle UML 2-Anwender ein gemeinsames Verständnis der UML haben. Für unsere Arbeit als Entwickler reicht sicherlich die unterste Stufe (Fundamental) aus.

A.3 Sequenz- und Timing-Diagramm

Neben den Aktivitätsdiagrammen gibt es noch weitere Diagramme, mit denen wir Interaktionen modellieren können. Wir haben davon die Sequenz- und Timing-Diagramme kennengelernt. Die wichtigsten Elemente der Sequenzdiagramme sind in Abb. A.12 dargestellt.

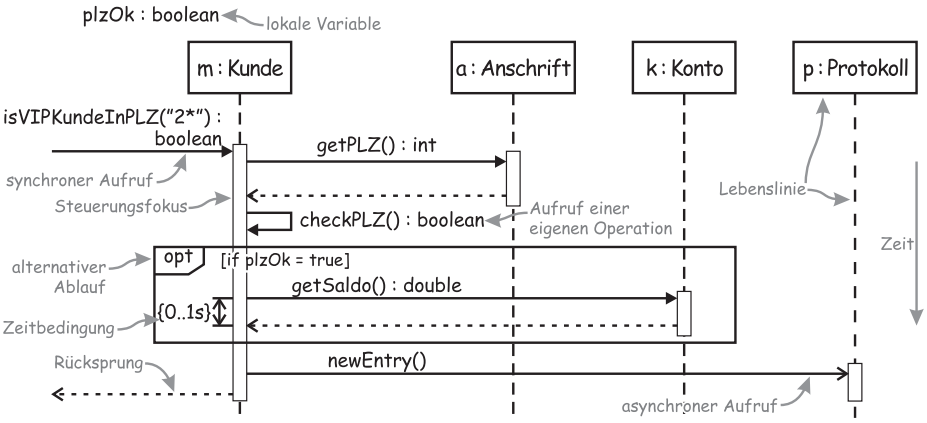


Abbildung A.12: Zentrale Elemente eines Sequenzdiagramms

Mit Sequenzdiagrammen modellieren wir konkrete Abläufe zwischen Objekten und deren zeitliches Zusammenspiel. Die Zeit läuft implizit von oben nach unten. Jedes Objekt hat eine eigene Lebenslinie. Die Zeit ist zwischen den Lebenslinien nicht synchronisiert, sodass jede Lebenslinie ihre eigene Zeitlinie hat. Reihenfolgen können daher nur über das Ausgehen und Eintreffen von Nachrichten exakt bestimmt werden. Mit *Nachrichten* sind die Aufrufe von Operationen gemeint.[2] Ein solcher Aufruf kann synchron oder asynchron erfolgen, was über die unterschiedliche Spitze des Pfeils dargestellt wird. Wenn ein Objekt aktiv ist, wird die Lebenslinie zum Steuerungsfokus verbreitert (Abb. A.12).

Bei synchronen Aufrufen kann der Rücksprung über einen gestrichelten Pfeil modelliert werden. Dies ist optional und wird nur empfohlen, wenn diese Information für die Modellierung des Timings wichtig ist. In Abb. A.12 ist dies nur innerhalb des optionalen Blocks notwendig, da hier eine Zeitbedingung für die Dauer des synchronen Aufrufs gegeben ist. Um bei den ganzen Pfeilen eine bessere Übersicht zu erhalten, können Sie die anderen Rücksprünge gerne weglassen.

Für eine stärkere Flexibilität in der Modellierung können alternative Abläufe, Verzweigungen und Verweise in einem Diagramm in einem eige-

[2] Diese Metapher geht noch auf die objektorientierte *Ursprache* Smalltalk zurück.

nen Rahmen dargestellt werden. Im Beispiel finden wir einen optionalen Block im Ablauf vor. In Tabelle A.2 sind die gebräuchlichsten Möglichkeiten für solche Blöcke aufgelistet.

Operator	Bedingung, Parameter	Bedeutung
alt	[bedingung 1] [bedingung 2] [else]	Verzweigung von einer zu mehreren Möglichkeiten
loop	minint maxint [bedingung]	Der Block wird als Schleife mindestens `minint` Mal, aber maximal `maxint` Mal wiederholt, falls die angegebene Bedingung erfüllt ist. Der Einfachheit halber kann auch `loop while [bedingung]` oder `loop until [bedingung]` verwendet werden.
break	[bedingung]	Innerhalb einer Schleife wird diese sofort beendet, wenn dieser Block erreicht wird.
opt	[bedingung]	Diese optionale Teilsequenz wird nur ausgeführt, wenn die Bedingung erfüllt ist.
par		Die Teilsequenz wir nebenläufig in einem eigenen Thread ausgeführt.
ref		Verweis auf ein weiteres Sequenzdiagramm, das an dieser Stelle eingebunden ist.

Tabelle A.2: Die gebräuchlichen Möglichkeiten für verschachtelte Blöcke in Sequenzdiagrammen in der UML [4]

Eine Variante des Sequenzdiagramms ist das Timing-Diagramm. Obwohl hier die gleichen Elemente wie im Sequenzdiagramm möglich sind, bietet es dennoch durch die andere Darstellungsweise gänzlich andere Vorteile. Die Diagrammrichtung ist um 90 Grad gedreht und es gibt eine feste, lineare und für alle Lebenslinien gleiche Zeitskala (Abb. A.13).

Der Fokus der Darstellung liegt im Timing-Diagramm auf den Zustandswechseln der beteiligten Objekte. Dafür wird jede Lebenslinie als eigener Kasten und die möglichen Zustände darin übereinander dargestellt. Über eine Zustandslinie für jede Lebenslinie modellieren wir Zustandswechsel. Als Auslöser für solche Zustandswechsel können Operationen aufgerufen werden, die über Pfeile innerhalb der Zeitskala exakt positioniert werden können. Wie im Sequenzdiagramm sind auch Zeitbedingungen möglich.

Diese Diagrammform ist z. B. in der Steuerungstechnik schon lange gebräuchlich und auf diese Weise in die UML aufgenommen worden. So

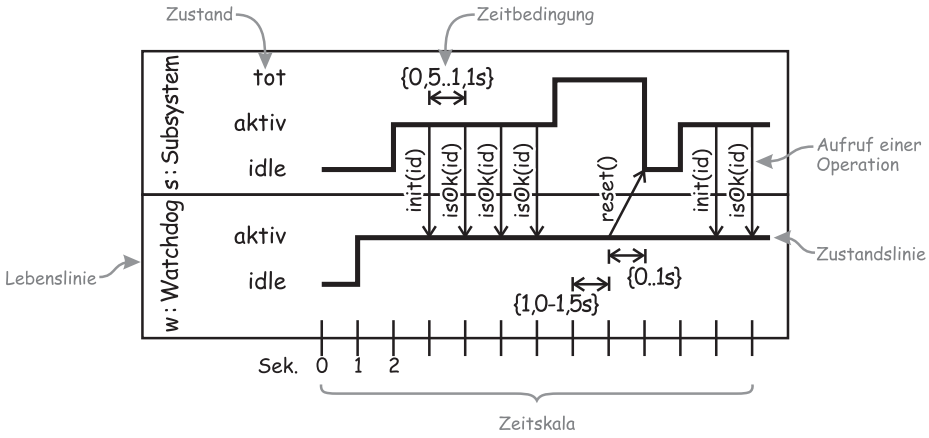


Abbildung A.13: Das Timing-Diagramm als alternative Darstellung von sequenziellen Abläufen am Beispiel des Watchdog-Pattern aus [5]

können wir Echtzeitsysteme modellieren bzw. die jeweils zeitkritischen Teile solcher Systeme.

A.4 Zustandsdiagramme

Bei zustandsbehafteten Systemen kann es hilfreich sein, die Zustandsübergänge und andere Regeln in einem Zustandsdiagramm zu modellieren. Ein Zustand wird dabei über ein Rechteck mit abgerundeten Ecken symbolisiert. Die Übergänge werden mit Pfeilen als Kanten bzw. Transitionen dargestellt (Abb. A.14).

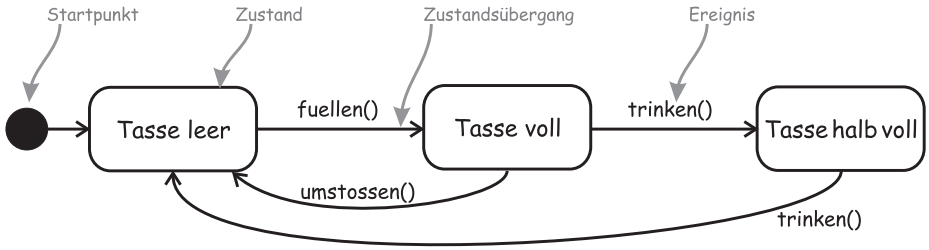


Abbildung A.14: Zustandsmodellierung in der UML am Beispiel einer Espressotasse

Auch bei endlos zirkulierenden Zustandsmodellen wie in Abb. A.14 kann es einen Startpunkt geben. Damit wird der initiale Startzustand angezeigt:

Wenn eine Espressotasse *auf die Welt kommt*, ist sie leer und kann gefüllt werden.

Das Ereignis `trinken()` kann für mehrere Zustände Relevanz haben und ggf. zu einem Zustandswechsel führen. Beim ersten Trinken wechselt die kleine Tasse auf halb voll, nach dem zweiten Schluck ist sie wieder leer. Aus einem Zustand können auch mehrere Transitionen herausführen. So kann aus einer vollen Tasse sowohl getrunken werden als auch diese umgestoßen werden. Nach dem Umstoßen ist sie schlagartig wieder leer. Interessant ist an diesem Beispiel, dass eine halb volle Tasse nicht umgestoßen werden kann, sondern nur eine volle. Somit ist in dieses Beispiel auch *Murphys Gesetz* eingeflossen.

Übergänge können an Bedingungen geknüpft sein und eine Aktion auslösen. Ebenfalls kann ein Zustand eigene Aktionen bereit stellen, die bei bestimmten Ereignissen erfolgen. Die drei Ereignisse `entry` (Aktion beim Eintritt in den Zustand ausführen), `exit` (Aktion beim Verlassen des Zustands ausführen) und `do` (Aktion ausführen, solange der Zustand aktiv ist) sind vordefiniert und können bei Bedarf genutzt werden (Abb. A.15).

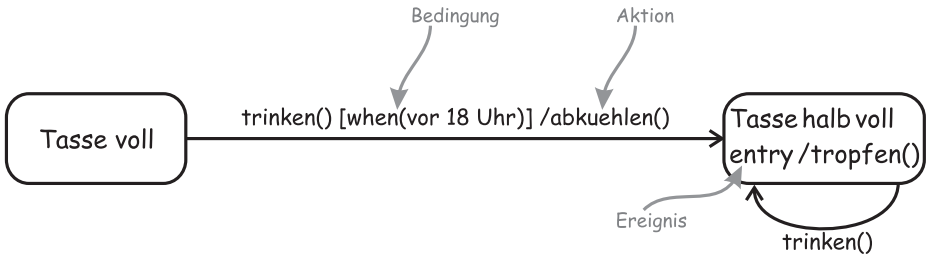


Abbildung A.15: Bedingungen und Aktionen in der Zustandsmodellierung mit der UML

Eine Transition kann auch wieder auf denselben Zustand führen, von dem sie ausgegangen ist. Wenn dann für das Ereignis `entry` eine Aktion definiert ist, wird diese erneut ausgeführt (Abb. A.15 rechts). Immer wenn ein Eintritt in den Zustand `Tasse halb voll` erfolgt, rinnt ein Tropfen von Rand der Tasse auf den Tisch.

Zustände können hierarchisch gestaffelt sein. Ein Zustand kann also Unterzustände beinhalten. Die Konfiguration dieser Unterzustände kann beim Verlassen des übergeordneten Zustands festgehalten werden, damit diese Konfiguration beim Wiedereintritt in den Oberzustand erneut hergestellt werden kann. Dieser Vorgang heißt Historisierung und wird durch ein `H` in einem Kreis symbolisiert (Abb. A.16).

Diese Historisierung kann sich nur auf die oberste Ebene der Unterzustände beziehen oder auch weitere Verschachtelungen mit berücksichti-

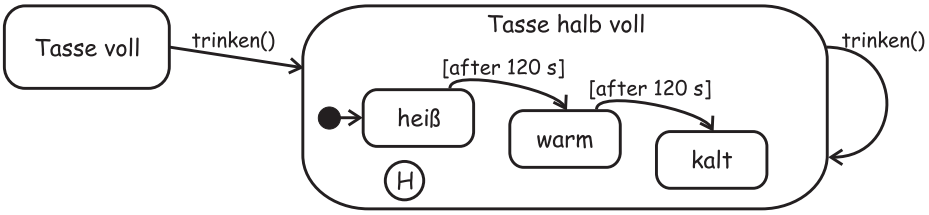


Abbildung A.16: Historisierung in der Zustandsmodellierung mit der UML

gen. Die letztere, sog. *tiefe Historisierung* wird durch ein H* in einem Kreis ausgedrückt.

Im Beispiel aus Abb. A.16 kann der Espresso in der halb vollen Tasse abkühlen, wobei zwischen den Unterzuständen `heiss`, `warm` und `kalt` unterschieden wird. Zu Beginn ist der Espresso heiß und nach jeweils zwei Minuten kühlt er um eine Stufe ab. Wenn er kalt ist, bleibt es dabei. Beim Verlassen des Zustands `Tasse halb voll` wird der Temperatur-Unterzustand festgehalten und der letzte Unterzustand beim Wiedereintritt in den Oberzustand reaktiviert. Es wird nicht wieder mit dem Initialzustand `heiss` begonnen.

Wie auch bei den anderen Diagrammen gibt es noch eine Menge mehr Möglichkeiten der Modellierung. Für das Verständnis der Abbildungen und Beispiele im Buch [5] sollten die hier genannten Elemente der UML aber ausreichen.

B Übersicht aller 37 objektorientierten Testmuster

Bereich	Testdesign-Muster	Kurzbeschreibung
Testmethodik	Kategorie-Partition	Design einer Testsuite basierend auf einer Input/Output-Analyse
	Kombinierte Funktion	Design einer Testsuite für ausgewähltes Verhalten nach kombinatorischer Logik
	Rekursive Funktion	Design einer Testsuite für rekursive Funktionen
	Polymorphische Nachrichten	Design einer Testsuite für einen Client eines polymorphen Servers
Klassen	Invariante Grenzen	Indentifizieren eines Testvektors für komplexe Bereiche
	Nicht modale Klasse	Design einer Testsuite für eine Klasse ohne sequenzielle Zusicherungen
	Modale Klasse	Design einer Testsuite für eine Klasse mit sequenziellen Zusicherungen
	Quasi-modale Klasse	Design einer Testsuite für eine Klasse mit inhaltsabhängigen sequenziellen Zusicherungen

Tabelle B.1: Übersicht der 37 Testmuster nach Binder [1] (Teil 1). Die im Buch [5] näher erläuterten Testmuster sind kursiv dargestellt.

Bereich	Testdesign-Muster	Kurzbeschreibung
Klassen-interne Integration	Small Pop	Reihenfolge von Codierung und Test auf Methoden- bzw. Klassenebene
	Alpha-Omega-Zyklus	Reihenfolge von Codierung und Test auf Methoden- bzw. Klassenebene
Flattened Klassen	*Polymorpher Server*	Design einer Testsuite konform zum Substitutionsprinzip einer Polymorphen Server-Hierarchie
	Modale Hierarchie	Design einer Testsuite für eine Hierarchie modaler Klassen
Wiederverwendbare Komponenten	Abstrakte Klasse	Entwicklung und Test einer Interface-Implementierung
	Generische Klasse	Entwicklung und Test einer parametrierbaren Klasse
	Neues Framework	Entwicklung und Test einer Demoapplikation eines neuen Frameworks
	Genutztes Framework	Test von Änderungen eines breit eingesetzten Frameworks
Subsystem	Klassen-Assoziationen	Design einer Testsuite für die Implementierung von Assoziationen
	Round-trip-Szenario	Design einer Testsuite für aggregiertes zustandsbasiertes Verhalten
	Gesteuerte Ausnahmen	Design einer Testsuite zur Prüfung des Exception-Handlings
	Mode Machine	Design einer Testsuite zur Implementierung eines Stimulus-Antwort-Szenarios

Tabelle B.2: Übersicht der 37 Testmuster nach Binder [1] (Teil 2). Die im Buch [5] näher erläuterten Testmuster sind kursiv dargestellt.

Bereich	Testdesign-Muster	Kurzbeschreibung
Integration	Big Bang	nicht-inkrementelle Integration
	Bottom-up	Integration nach Abhängigkeiten
	Top-down	Integration nach Steuerungshierarchien
	Zusammenarbeit	Integration nach Cluster-Szenarien
	Backbone	Hybridintegration von Subsystemen
	Schicht	Integration von n-tier-Architekturen
	Client/Server	Integration von Client/Server-Architekturen
	Verteilte Dienste	Integration verteilter Architekturen
	Hochfrequenz	Systembau und Test in regelmäßig wiederkehrenden Intervallen
Applikation	Erweiterte Use Cases	Entwicklung testbarer Use Cases, Design einer Testsuite zur Abdeckung der Input/Output-Beziehungen
	CRUD	Ausführen aller Basisoperationen Create-Read-Update-Delete
	Zuteilen nach Frequenz	Zuteilung der Systemtestarbeiten zur Maximierung der operativen Zuverlässigkeit
Regressionstest	Alles nachtesten	Alle Tests durchführen
	Riskante Use Cases nachtesten	Nur Tests für riskanten Code durchführen
	Profile nachtesten	Nur Tests für oft benutzte Teile durchführen
	Geänderten Code nachtesten	Nur Tests für den Code durchführen, der von Änderungen abhängt
	Firewall nachtesten	Nur Tests für geänderten Code durchführen

Tabelle B.3: Übersicht der 37 Testmuster nach Binder [1] (Teil 3)

C Das komplette JUnit 3-Beispiel

C.1 Download

Alle Codebeispiele aus dem Anhang und noch einige mehr stehen unter `www.oo-testen.de` zum Download bereit.

Die Beispiele dort und das folgende Beispiel im Anhang sind zu JUnit 3 kompatibel. Damit sollten sie auch in den Testrunnern der verschiedenen Entwicklungsumgebungen unter JUnit 4 laufen oder nach geringen Anpassungen lauffähig sein.

C.2 Modale Klasse mit Mock testen

Das in [5] beschriebene Beispiel für den Test einer modalen Vertragsklasse finden Sie hier in seiner vollständigen Form. Daneben sind die Vertragsklasse selbst, die notwendigen Dummys, die Mock-Klasse und die Interfaces abgebildet.

Die Vertragstestklasse

```
package vertrag;

import junit.framework.*;

public class TestVertrag extends TestCase {
  private static final double delta = 0.001;

  private Vertrag vertrag = null;
  private KundenDummy kunde = null;
  private MitarbeiterDummy vermittler = null;
  private ProviRechnerMock proviRechner = null;

  protected void setUp() throws Exception {
    super.setUp();
    kunde = new KundenDummy("Paul Meyer");
    vermittler = new MitarbeiterDummy("Heinz", "Schulze");
    proviRechner = new ProviRechnerMock();
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    kunde = null;
```

```
    vermittler = null;
    proviRechner = null;
  }

  public void testAngelegtPositiv() {
    // positiver Grenzwert beim Pruefen
    vertrag = new Vertrag(1, kunde, vermittler, proviRechner);
    assertEquals(1, vertrag.getNummer());
    assertEquals("Paul Meyer",vertrag.getKunden());
    assertEquals("Heinz Schulze", vertrag.getVermittler());
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());
    vertrag = null;

    // negativer Grenzwert beim Pruefen
    vertrag = new Vertrag(0, kunde, vermittler, proviRechner);
    assertEquals(0, vertrag.getNummer());
    assertEquals("Paul Meyer",vertrag.getKunden());
    assertEquals("Heinz Schulze", vertrag.getVermittler());
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());
    vertrag = null;
  }

  public void testAngelegtNegativ() {
    // beliebiger Zwischenwert
    vertrag = new Vertrag(7, kunde, vermittler, proviRechner);
    assertEquals(7, vertrag.getNummer());
    assertEquals("Paul Meyer",vertrag.getKunden());
    assertEquals("Heinz Schulze", vertrag.getVermittler());
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag.abrechnen();  //nicht erlaubt
    assertEquals(7, vertrag.getNummer());
    assertEquals("Paul Meyer",vertrag.getKunden());
    assertEquals("Heinz Schulze", vertrag.getVermittler());
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag.aktivieren();  //nicht erlaubt
    assertEquals(7, vertrag.getNummer());
    assertEquals("Paul Meyer",vertrag.getKunden());
    assertEquals("Heinz Schulze", vertrag.getVermittler());
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag.ruhen();  //nicht erlaubt
    assertEquals(7, vertrag.getNummer());
    assertEquals("Paul Meyer",vertrag.getKunden());
    assertEquals("Heinz Schulze", vertrag.getVermittler());
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag.beenden();  //nicht erlaubt
    assertEquals(7, vertrag.getNummer());
    assertEquals("Paul Meyer",vertrag.getKunden());
    assertEquals("Heinz Schulze", vertrag.getVermittler());
```

```
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag = null;
  }

  public void testPruefenAbgelehntPositiv() {
    vertrag = new Vertrag(1, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertTrue(vertrag.isAktiv());
    assertEquals(0.0, vertrag.getProvision(), delta);
    vertrag = null;

    vertrag = new Vertrag(0, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_ABGELEHNT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());
    vertrag = null;
  }

  public void testPruefenNegativ() {
    vertrag = new Vertrag(7, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);

    vertrag.pruefen();  //nicht nochmal erlaubt
    assertEquals(7, vertrag.getNummer());
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag.aktivieren();  //nicht erlaubt
    assertEquals(7, vertrag.getNummer());
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag = null;
  }

  public void testAbgelehntNegativ() {
    vertrag = new Vertrag(0, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_ABGELEHNT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag.abrechnen();  //nicht erlaubt
    assertEquals(0, vertrag.getNummer());
    assertEquals(vertrag.VZ_ABGELEHNT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag.aktivieren();  //nicht erlaubt
    assertEquals(0, vertrag.getNummer());
    assertEquals(vertrag.VZ_ABGELEHNT, vertrag.getZustand());
```

```
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag.pruefen();  //nicht erlaubt
    assertEquals(0, vertrag.getNummer());
    assertEquals(vertrag.VZ_ABGELEHNT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag.ruhen();  //nicht erlaubt
    assertEquals(0, vertrag.getNummer());
    assertEquals(vertrag.VZ_ABGELEHNT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag.beenden();  //nicht erlaubt
    assertEquals(0, vertrag.getNummer());
    assertEquals(vertrag.VZ_ABGELEHNT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag = null;
  }

  public void testAbrechnenPositiv() {
    vertrag = new Vertrag(1, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    vertrag.abrechnen();
    assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag = null;
  }

  public void testAbrechnenNegativ() {
    vertrag = new Vertrag(2, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    vertrag.abrechnen();
    assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag.abrechnen();  //nicht nochmal erlaubt
    assertEquals(2, vertrag.getNummer());
    assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());

    vertrag.pruefen();  //nicht erlaubt
    assertEquals(2, vertrag.getNummer());
    assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());
```

```
215
      vertrag.aktivieren();  //nicht erlaubt
      assertEquals(2, vertrag.getNummer());
      assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
      assertEquals(proviRechner.getLetztenWert(),
220     vertrag.getProvision(), delta);
      assertTrue(vertrag.isAktiv());

      vertrag = null;
    }
225
    public void testRuhenPositiv() {

      vertrag = new Vertrag(1, kunde, vermittler, proviRechner);
      assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
230   vertrag.pruefen();
      assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
      assertEquals(0.0, vertrag.getProvision(), delta);
      assertTrue(vertrag.isAktiv());
      vertrag.ruhen();
235   assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
      assertEquals(0.0, vertrag.getProvision(), delta);
      assertTrue(!vertrag.isAktiv());

      vertrag = null;
240
      vertrag = new Vertrag(3, kunde, vermittler, proviRechner);
      assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
      vertrag.pruefen();
      assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
245   vertrag.abrechnen();
      assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
      assertEquals(proviRechner.getLetztenWert(),
        vertrag.getProvision(), delta);
      assertTrue(vertrag.isAktiv());
250   vertrag.ruhen();
      assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
      assertEquals(proviRechner.getLetztenWert(),
        vertrag.getProvision(), delta);
      assertTrue(!vertrag.isAktiv());
255
      vertrag = null;
    }

    public void testRuhenNegativ() {
260   vertrag = new Vertrag(4, kunde, vermittler, proviRechner);
      assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
      vertrag.pruefen();
      assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
      vertrag.ruhen();
265   assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
      assertEquals(0.0, vertrag.getProvision(), delta);
      assertTrue(!vertrag.isAktiv());

      vertrag.pruefen();  //nicht erlaubt
270   assertEquals(4, vertrag.getNummer());
      assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
      assertEquals(0.0, vertrag.getProvision(), delta);
      assertTrue(!vertrag.isAktiv());

275   vertrag.abrechnen();  //nicht erlaubt
      assertEquals(4, vertrag.getNummer());
      assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
      assertEquals(0.0, vertrag.getProvision(), delta);
```

```
    assertTrue(!vertrag.isAktiv());

    vertrag.ruhen();  //nicht nochmal erlaubt
    assertEquals(4, vertrag.getNummer());
    assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag = null;
  }

  public void testBeendenPositiv() {
    vertrag = new Vertrag(1, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertTrue(vertrag.isAktiv());
    vertrag.beenden();
    assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag = null;

    vertrag = new Vertrag(5, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    vertrag.abrechnen();
    assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());
    vertrag.beenden();
    assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag = null;
  }

  public void testBeendenNegativ() {
    vertrag = new Vertrag(32000, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    vertrag.abrechnen();
    assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(vertrag.isAktiv());
    vertrag.beenden();
    assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag.beenden();  //nicht nochmal erlaubt
    assertEquals(32000, vertrag.getNummer());
    assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());
```

```
    vertrag.pruefen();  //nicht erlaubt
345 assertEquals(32000, vertrag.getNummer());
    assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());
350
    vertrag.abrechnen();  //nicht erlaubt
    assertEquals(32000, vertrag.getNummer());
    assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
355   vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

    vertrag.ruhen();  //nicht erlaubt
    assertEquals(32000, vertrag.getNummer());
360 assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
    assertTrue(!vertrag.isAktiv());

365 vertrag.aktivieren();  //nicht erlaubt
    assertEquals(32000, vertrag.getNummer());
    assertEquals(vertrag.VZ_BEENDET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
370 assertTrue(!vertrag.isAktiv());

    vertrag = null;
  }

375 public void testAktivieren() {
    vertrag = new Vertrag(1, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
    assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
380 assertTrue(vertrag.isAktiv());
    vertrag.ruhen();
    assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
    assertTrue(!vertrag.isAktiv());
    vertrag.aktivieren();
385 assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    assertEquals(0.0, vertrag.getProvision(), delta);

    assertTrue(vertrag.isAktiv());

390 vertrag = null;

    vertrag = new Vertrag(3, kunde, vermittler, proviRechner);
    assertEquals(vertrag.VZ_ANGELEGT, vertrag.getZustand());
    vertrag.pruefen();
395 assertEquals(vertrag.VZ_GEPRUEFT, vertrag.getZustand());
    vertrag.abrechnen();
    assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
400 assertTrue(vertrag.isAktiv());
    vertrag.ruhen();
    assertEquals(vertrag.VZ_RUHEND, vertrag.getZustand());
    assertEquals(proviRechner.getLetztenWert(),
      vertrag.getProvision(), delta);
405 assertTrue(!vertrag.isAktiv());
    vertrag.aktivieren();
```

```
        assertEquals(vertrag.VZ_ABGERECHNET, vertrag.getZustand());
        assertEquals(proviRechner.getLetztenWert(),
          vertrag.getProvision(), delta);
410     assertTrue(vertrag.isAktiv());

        vertrag = null;
      }
    }
```

Codebeispiel C.1: Test der modalen Klasse Vertrag aus [5]

Die Vertragsklasse

Die hier dargestellte Form der Vertragsklasse basiert auf Enumerations für die Zustände. In Java können Enumerations durch Konstanten simuliert werden. Die hier gezeigte Lösung ist weitgehend sprachunabhängig. Unter Java bietet sich eine andere Implementierung des Zustandsmusters an, die weiter unten beschrieben wird.

```
package vertrag;
 /*
 * Einfache Implementierung ohne Java-Spezialitaeten
 * Nutzung von Dummy-Implementierungen der Interfaces Mandant und Vermittler
 */

public class Vertrag {
  public static final int VZ_UNGUELTIG = 0;
  public static final int VZ_ANGELEGT = 1;
  public static final int VZ_GEPRUEFT = 2;
  public static final int VZ_ABGERECHNET = 3;
  public static final int VZ_ABGELEHNT = 4;
  public static final int VZ_RUHEND = 5;
  public static final int VZ_BEENDET = 6;

  // zum Historisieren aktiver Zustaende fuer ruhen() und aktivieren()
  private int historyState;

  private interface Zustand {
    abstract public int pruefen();
    abstract public int provisionieren();
    abstract public int ruhen();
    abstract public int aktivieren();
    abstract public int beenden();
    abstract public int getZustand();
  }

  private class Angelegt implements Zustand {
    public int pruefen() {
      if (getNummer() > 0) {
        return VZ_GEPRUEFT;
      } else{
        return VZ_ABGELEHNT;
      }
    }

    public int provisionieren() {
      return VZ_UNGUELTIG;
    }
    public int ruhen() {
      return VZ_UNGUELTIG;
    }
    public int aktivieren() {
      return VZ_UNGUELTIG;
    }
    public int beenden() {
      return VZ_UNGUELTIG;
    }
    public int getZustand() {
      return VZ_ANGELEGT;
    }
  }

  private class Geprueft implements Zustand {
```

```
55      public int pruefen() {
          return VZ_UNGUELTIG;
        }
        public int provisionieren() {
          myProvi = myProviRechner.berechneProvi(getNummer());
60        return VZ_ABGERECHNET;
        }
        public int ruhen() {
          historyState = getZustand();
          return VZ_RUHEND;
65      }
        public int aktivieren() {
          return VZ_UNGUELTIG;
        }
        public int beenden() {
70        return VZ_BEENDET;
        }
        public int getZustand() {
          return VZ_GEPRUEFT;
        }
75    }

      private class Abgerechnet implements Zustand {
        public int pruefen() {
          return VZ_UNGUELTIG;
80      }
        public int provisionieren() {
          return VZ_UNGUELTIG;
        }
        public int ruhen() {
85        historyState = getZustand();
          return VZ_RUHEND;
        }
        public int aktivieren() {
          return VZ_UNGUELTIG;
90      }
        public int beenden() {
          return VZ_BEENDET;
        }
        public int getZustand() {
95        return VZ_ABGERECHNET;
        }
      }

      private class Abgelehnt implements Zustand {
100     public int pruefen() {
          return VZ_UNGUELTIG;
        }
        public int provisionieren() {
          return VZ_UNGUELTIG;
105     }
        public int ruhen() {
          return VZ_UNGUELTIG;
        }
        public int aktivieren() {
110       return VZ_UNGUELTIG;
        }
        public int beenden() {
          return VZ_UNGUELTIG;
        }
115     public int getZustand() {
          return VZ_ABGELEHNT;
        }
      }
```

```
  private class Ruhend implements Zustand {
120   public int pruefen() {
        return VZ_UNGUELTIG;
      }
      public int provisionieren() {
        return VZ_UNGUELTIG;
125   }
      public int ruhen() {
        return VZ_UNGUELTIG;
      }
      public int aktivieren() {
130     return historyState;   // historisierten Zustand wieder einnehmen
      }
      public int beenden() {
        return VZ_UNGUELTIG;
      }
135   public int getZustand() {
        return VZ_RUHEND;
      }
  }
  private class Beendet implements Zustand {
140   public int pruefen() {
        return VZ_UNGUELTIG;
      }
      public int provisionieren() {
        return VZ_UNGUELTIG;
145   }
      public int ruhen() {
        return VZ_UNGUELTIG;
      }
      public int aktivieren() {
150     return VZ_UNGUELTIG;
      }
      public int beenden() {
        return VZ_UNGUELTIG;
      }
155   public int getZustand() {
        return VZ_BEENDET;
      }
  }

160

  private long nummer;
  private Mandant myMandant;
  private Vermittler myVermittler;
  private ProviRechner myProviRechner;
165 private Zustand myZustand;
  private double myProvi;


  public Vertrag(long nr, Mandant neuerMandant, Vermittler abschlussVermittler,
170              ProviRechner provisionsRechner) {
    myZustand = new Angelegt();
    nummer = nr;
    setStatus(VZ_ANGELEGT);
    historyState = VZ_UNGUELTIG;
175 myMandant = neuerMandant;
    myVermittler = abschlussVermittler;
    myProviRechner = provisionsRechner;
    myProvi = 0.0;
  }
180

  public long getNummer(){
    return nummer;
```

```
  }

  public String getVermittler() {
    return myVermittler.getVornamen() + " " + myVermittler.getNachnamen();
  }

  public String getKunden() {
    return myMandant.getNamen();
  }

  public double getProvision() {
    return myProvi;
  }

  public boolean pruefen() {
    setStatus(myZustand.pruefen());
    if (VZ_GEPRUEFT == myZustand.getZustand()) {
      return true;
    }
    return false;
  }

  public void abrechnen() {
    setStatus(myZustand.provisionieren());
  }

  public void ruhen() {
    setStatus(myZustand.ruhen());
  }

  public void aktivieren() {
    setStatus(myZustand.aktivieren());
  }

  public void beenden() {
    setStatus(myZustand.beenden());
  }

  public int getZustand() {
    if (null != myZustand) {
      return myZustand.getZustand();
    } else {
      return VZ_UNGUELTIG;
    }
  }

  public boolean isAktiv() {
    if ((VZ_ANGELEGT == getZustand()) ||
        (VZ_GEPRUEFT == getZustand()) ||
        (VZ_ABGERECHNET == getZustand())) {
      return true;
    } else {
      return false;
    }
  }

  private void setStatus(int neuerZustand) {
    if (VZ_UNGUELTIG != neuerZustand) {
      if (neuerZustand != myZustand.getZustand()) {  // ein neuer Zustand?
        myZustand = null;           // alten Zustand frei geben
        switch (neuerZustand) {     // Zielzustand anlegen
          case VZ_ANGELEGT: myZustand = new Angelegt();
            break;
          case VZ_GEPRUEFT: myZustand = new Geprueft();
```

```
            break;
          case VZ_ABGERECHNET: myZustand = new Abgerechnet();
            break;
250       case VZ_ABGELEHNT: myZustand = new Abgelehnt();
            break;
          case VZ_RUHEND: myZustand = new Ruhend();
            break;
          case VZ_BEENDET: myZustand = new Beendet();
255         break;
          default: myZustand = null;
            break;
        }
      }
260   }
  }
}
```

Codebeispiel C.2: Die modale Klasse Vertrag aus [5]

Unter Java bietet es sich bei der Realisierung des Zustandsmusters an, die Konstanten durch Klassen zu ersetzen [3]. Die Änderungen am Code sind nur marginal. Es wird eine zusätzliche Klasse `Ungueltig` benötigt, und die Implementierungen der Zustände erfordern einen leeren Default-Konstruktor.

```
package vertrag;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
 5
public class Vertrag {
  public static final Class VZ_UNGUELTIG = Ungueltig.class;
  public static final Class VZ_ANGELEGT = Angelegt.class;
  public static final Class VZ_GEPRUEFT = Geprueft.class;
10 public static final Class VZ_ABGERECHNET = Abgerechnet.class;
  public static final Class VZ_ABGELEHNT = Abgelehnt.class;
  public static final Class VZ_RUHEND = Ruhend.class;
  public static final Class VZ_BEENDET = Beendet.class;

15 // zum Historisieren aktiver Zustaende fuer ruhen() und aktivieren()
  private Class historyState;

  private interface Zustand {
    Class pruefen();
20  Class provisionieren();
    Class ruhen();
    Class aktivieren();
    Class beenden();
    Class getZustand();
25 }

  private class Ungueltig implements Zustand {
    public Ungueltig() {
      // damit es laeuft
30  }
    public Class pruefen() {
      return VZ_UNGUELTIG;
    }
    public Class provisionieren() {
```

```
35      return VZ_UNGUELTIG;
      }
      public Class ruhen() {
        return VZ_UNGUELTIG;
      }
40    public Class aktivieren() {
        return VZ_UNGUELTIG;
      }
      public Class beenden() {
        return VZ_UNGUELTIG;
45    }
      public Class getZustand() {
        return VZ_UNGUELTIG;
      }
    }
50  ...
```

Codebeispiel C.3: Realisierung der Zustände als Klassen

Die zentrale Anpassung erfolgt in `setZustand()`.

```
    private void setStatus(Class neuerZustand) {
      try {
        Constructor cnstrctr = neuerZustand.getDeclaredConstructor
          (new Class[] {Vertrag.class});
5       cnstrctr.setAccessible(true);
        myZustand = (Zustand) cnstrctr.newInstance(new Object[]{this});
      }
      catch (IllegalAccessException ex) {
        myZustand = (Zustand) new Ungueltig();
10    }
      catch (InstantiationException ex) {
        myZustand = (Zustand) new Ungueltig();
      }
      catch (SecurityException ex1) {
15      myZustand = (Zustand) new Ungueltig();
      }
      catch (NoSuchMethodException ex1) {
        myZustand = (Zustand) new Ungueltig();
      }
20    catch (InvocationTargetException ex2) {
        myZustand = (Zustand) new Ungueltig();
      }
      catch (IllegalArgumentException ex2) {
        myZustand = (Zustand) new Ungueltig();
25    }
    }
```

Codebeispiel C.4: Die Methode setStatus()

Die Interface-, Dummy- und Mock-Klassen

```
package vertrag;

public interface Mandant {
  public String getNamen();
}
```

Codebeispiel C.5: Das Interface des Mandanten

```
package vertrag;

public interface Vermittler {
  public String getVornamen();
  public String getNachnamen();
}
```

Codebeispiel C.6: Das Interface des Vermittlers

```
package vertrag;

public interface ProviRechner {
  public double berechneProvi(long nr);
}
```

Codebeispiel C.7: Das Interface der Provisionsrechners

```
package vertrag;

public class KundenDummy implements Mandant{
  private String myName;

  public KundenDummy(String kundenName) {
    myName = kundenName;
  }

  public String getNamen() {
    return myName;
  }
}
```

Codebeispiel C.8: Dummy-Realisierung des Mandanten-Interface durch die Klasse KundenDummy

```
package vertrag;

public class MitarbeiterDummy implements Vermittler {
  private String myVorname;
  private String myNachname;

  public MitarbeiterDummy(String vorname, String nachname) {
    myVorname = vorname;
    myNachname = nachname;
  }

  public String getVornamen() {
    return myVorname;
  }

  public String getNachnamen() {
    return myNachname;
  }
}
```

Codebeispiel C.9: Dummy-Realisierung des Vermittler-Interface durch die Klasse `MitarbeiterDummy`

```
package vertrag;

public class ProviRechnerMock implements ProviRechner{
  private double letzterWert;
  public ProviRechnerMock() {
    letzterWert = 0.0;
  }

  public double berechneProvi(long nr) {
    double provi = nr * 2.5;
    letzterWert = provi;
    return provi;
  }

  public double getLetztenWert() {
    return letzterWert;
  }
}
```

Codebeispiel C.10: Die Mock-Realisierung des Provisionsrechner-Interface. Die Methode `getLetztenwert()` wird nur für die Prüfungen innerhalb der Testklasse benötigt.

Literatur

[1] Robert V. Binder. *Testing Object-Oriented Systems – Models, Patterns, and Tools*. Addison-Wesley, 2000.

[2] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer und Werner Retschitzegger. *UML@Work – Objektorientierte Modellierung mit UML 2*. dpunkt.verlag, 3. Auflage, 2005.

[3] Steven John Metsker. *Design Patterns Java Workbook*. Addison-Wesley, 2002.

[4] Bernd Oestereich. *Analyse und Design mit der UML 2.3 – Objektorientierte Softwareentwicklung*. Oldenbourg, 9. Auflage, 2009. Unter Mitarbeit von Stefan Bremer.

[5] Uwe Vigenschow. *Testen von Software und Embedded Systems – Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*. dpunkt.verlag, 2. Auflage, 2010.

[6] Tim Weilkiens und Bernd Oestereich. *UML 2 - Zertifizierung: Fundamental, Intermediate und Advanced*. dpunkt.verlag, 2006.

Index