PEARSON Education

100% Markt+Technik

Auf Vista getestet

Java 6

Praxis der objektorientierten Programmierung

DIRK LOUIS PETER MÜLLER

KOMPENDIUM

Einführung | Arbeitsbuch | Nachschlagewerk

12 Objektorientiert denken – objektorientiert programmieren

Letzten Endes basiert die gesamte objektorientierte Programmierung auf einem einzigen Grundelement: der Klasse. Über die Technik der Klassendefinition, der Erzeugung von Objekten aus Klassen sowie der Arbeit mit Objekten konnten Sie sich bereits im zurückliegenden Teil informieren. Weiterführende, an die Klassen geknüpfte Techniken und Konzepte werden Sie im vor Ihnen liegenden Teil kennenlernen.

Doch objektorientierte Programmierung besteht nicht nur aus der Anwendung objektorientierter Techniken. Mindestens ebenso wichtig ist es, ein Gefühl für die besondere Art der objektorientierten Programmierung zu bekommen. Halten wir daher einen Moment inne und überlegen wir, was es – bar jeder Technik – bedeutet, objektorientiert zu programmieren.

12.1 Objektorientiertes Programmieren

Objektorientierte Programmierung bedeutet, dass der Programmierer versucht, die ihm gestellten Aufgaben mithilfe von Objekten zu lösen. Der erste Schritt dazu besteht üblicherweise darin, die beteiligten Objekte zu identifizieren und sich zu überlegen, wie diese zusammenarbeiten müssen, um die gestellte Aufgabe zu lösen. Erstaunlicherweise fällt dies blutigen Programmieranfängern oft leichter als Umsteigern, die von der imperativen Programmierung kommen. Warum ist dem so?

12.1.1 Denken Sie in Objekten!

Wer noch nie programmiert haben, denkt von Natur aus »objektorientiert«. Ein Beispiel: Angenommen Sie möchten am nächsten Morgen um Punkt 7 Uhr geweckt werden. Was tun Sie? Sie holen einen Wecker (d.h., Sie suchen sich ein passendes Objekt) und stellen diesen so ein, dass er Sie um 7 Uhr weckt (d.h., Sie weisen das Objekt an, um 7 Uhr zu klingeln).

Ein Programmierer denkt dagegen automatisch auf der Abstraktionsebene, die seiner Programmiersprache entspricht.

- Die niedrigste Abstraktionsebene ist die Ebene der Maschinen- und Assemblersprachen, auf der der Programmierer wie eine Maschine denkt: Ich lade die Systemzeit in ein Register und errechne, ob es schon

7 Uhr ist. Wenn ja, löse ich einen Systembeep aus, wenn nein, springe ich zurück zu dem Ladebefehl für die Systemzeit[1].

- Die imperative/strukturierte Programmierung stellt die nächsthöhere Abstraktionsebene dar. Hier denkt der Programmierer in Daten, Operatoren und Funktionen: Ich speichere die Zeit, zu der geweckt werden soll, in einer Variablen. In einer Schleife rufe ich dann eine Funktion auf, die die Systemzeit abfragt. Diese Zeit vergleiche ich mit der abgespeicherten Weckzeit. Ist die Zeit zum Wecken gekommen, wird die Schleife verlassen und ich rufe die Funktion für den Systembeep auf.
- Die objektorientierte Programmierung abstrahiert noch weiter. Sie fordert uns auf, Objekte zu identifizieren und festzulegen, welche Aufgaben welches Objekt übernehmen soll: Ich erzeuge ein Objekt `wecker`, das zu einem beliebigen Zeitpunkt klingeln kann. Dieses Objekt weise ich an, um 7 Uhr zu klingeln.

Die Schwierigkeit für Umsteiger von strukturierten Programmiersprachen besteht also vor allem darin, die niedere, datenorientierte Abstraktionsebene zu verlassen und wieder »natürlich« zu denken.

12.1.2 Wie sind Objekte beschaffen?

Die Objekte, mit denen der Programmierer plant und arbeitet, verfügen über

- *Merkmale* und
- *Verhaltensweisen*.

  Die Merkmale beschreiben das Objekt und seinen Zustand. (Später, wenn die Klasse für das Objekt implementiert wird, werden die Merkmale durch Felder repäsentiert.)

  Die Verhaltensweisen sind die Aktionen, die das Objekt ausführen kann. (Die Verhaltensweisen entsprechen den Methoden der Klasse.)

Welche Merkmale und Verhaltensweisen hätte beispielsweise das `wecker`-Objekt aus dem vorangehenden Abschnitt, das um 7 Uhr klingeln soll?

Zuerst einmal benötigt das `wecker`-Objekt ein Merkmal »alarmZeit«, in dem abgespeichert wird, wann der Alarm ausgelöst werden soll. Ebenfalls sinnvoll wäre ein Merkmal »alarmAktiviert«, das festhält, ob der Alarm aktiviert ist oder nicht. Wenn das Weckerprogramm über eine grafische Benutzeroberfläche verfügt, sollte das `wecker`-Objekt darüber hinaus Merkmale enthalten, die sein Aussehen beschreiben, beispielsweise »digitalOderAnalog«, »Hintergrundfarbe« oder »Sekundenanzeige«.

Nun zu den Verhaltensweisen. Damit das `wecker`-Objekt, die ihm gestellte Aufgabe (»um 7 Uhr zu wecken«) erfüllen kann, muss es zumindest über die Verhaltensweisen »AlarmEinstellen« und »Klingeln« verfügen. Eine Verhaltensweise »AlarmAbstellen« wäre ebenfalls wünschenswert.

---

1 Ich muss gestehen, dass es schon einige Zeit her ist, dass ich in Assembler programmiert habe. Für Ungenauigkeiten in der obigen Darstellung möchte ich daher um Nachsicht bitten.

Mit »Verhaltensweisen« verbinden wir in der Regel Aktivitäten, die ein Objekt von selbst zeigt: das Wachsen einer Pflanze, das Wiehern eines Pferdes, das Umkippen einer Flasche oder eben das Klingeln des Weckers. Im objektorientierten Sinne zählen zu den Verhaltensweisen aber auch Aktivitäten, die mit der Bedienung oder Verwendung eines Objekts zu tun haben, also das Gießen einer Pflanze, das Pflegen eines Pferdes, das Öffnen einer Flasche oder eben das »Stellen« eines Weckers.

Arten von Objekten

Das oben beschriebene `wecker`-Objekt ist dem Modell eines echten Weckers nachempfunden. Für den Programmierer hat dies den Vorteil, dass ihm sein Wissen über Aufbau und Bedienung eines Weckers bei Entwurf und Schreiben seines Programms zugute kommt. Stellt er seinen Programmcode anderen Programmierern zur Verfügung, können sich diese dank ihrer alltäglichen Erfahrungen im Umgang mit echten Weckern, schneller in Aufbau des Programms und des `wecker`-Objekts eindenken.

Die Nachbildung realer Dinge durch Objekte ist daher ein wichtiges und mächtiges Instrument der objektorientierten Programmierung. Auf der anderen Seite würde die objektorientierte Programmierung schnell an ihre Grenzen stoßen, wenn ihre Objekte ausschließlich Dinge der realen Welt nachbilden könnten. Glücklicherweise kann ein Objekt aber nahezu alles sein, was als Einheit aus Merkmalen und zugehörigen Verhaltensweisen ausgedrückt werden kann. So gibt es

- Objekte, die Dinge der realen Welt nachbilden.
- Objekte, die virtuelle Dinge repräsentieren (beispielsweise Dinge, die auf dem Bildschirm zu sehen sind, wie Fenster oder Schaltflächen oder ein gezeichnetes Monster).
- Objekte, die Daten repräsentieren oder verwalten (ein Programm zur Verwaltung von Musik-CDs könnte die einzelnen CDs als Objekte darstellen und zusammen in einem Container-Objekt verwalten, das das Ablegen, Sortieren und Suchen bestimmter CDs erleichtert).
- Objekte, die einfach eine bestimmte Funktionalität zur Verfügung stellen (beispielsweise für einen gegebenen Satz von Daten statistische Berechnungen wie Mittelwert, Standardabweichung u. a. ausführen).
- u.v.a.

12.2 Wie findet man einen objektorientierten Lösungsansatz?

Kleinere Programme werden häufig aus dem Bauch heraus geschrieben. Der Programmierer weiß, welche Aufgabe das Programm erfüllen soll (meist ist es eh nur eine einzige), und hat im Kopf bereits eine Vorstellung von dem ungefähren Programmablauf. Vielleicht steht der gesamte Code des Pro-

gramms in der `main()`-Methode, vielleicht hat der Programmierer Teilprobleme in andere statische Methoden ausgelagert, die er von `main()` aus direkt aufruft, vielleicht hat er sogar eine eigene Klasse definiert, die er instanziiert und deren Methoden er nutzt.

Für etwas größere Programme sind dagegen meist schon mehrere Klassen zu definieren. Trotzdem werden auch diese Programme häufig aus dem Bauch heraus programmiert. Der Programmierer beginnt mit einer groben Vorstellung von den Mindestanforderungen, die das Programm erfüllen soll, und setzt diese in Code um. Während der Arbeit an dem Programm lernt er immer mehr über den Aufbau selbigens und passt seine Implementierung an – beispielsweise wenn er erkennt, dass diese oder jene Funktionalität, die er für sein Programm benötigt, am besten durch Objekte einer Klasse zur Verfügung gestellt würde, oder wenn er das Programm schrittweise ausbaut.

Werden die Programme noch umfangreicher, wird die angebotene Funktionalität immer komplexer, arbeitet gar ein ganzes Team von Programmierern an dem Projekt, verbietet sich die Programmierung »aus dem Bauch heraus« und es bedarf einer strengeren, methodischen Vorgehenweise.

12.2.1 Moderne Software-Entwicklung

Moderne Software-Entwicklung gliedert sich grundsätzlich in fünf Phasen:

1. **Anforderungsspezifikation** – In der ersten Phase wird festgelegt, welchem Zweck das Programm dienen und welche Aufgaben es erfüllen soll. Wird das Programm im Auftrag eines Kunden erstellt, ist es wichtig, die Anforderungen im Gespräch mit dem Kunden zu erarbeiten.
2. **Analyse** – In dieser Phase werden Bedienung und Arbeitsweise des Programms ausgearbeitet. Wichtig ist, dass die Analyse aus Sicht des Anwenders geschieht und sich auch dessen Terminologie bedient.

   Nach Beendigung der Analyse-Phase sollten Sie eine klare Vorstellung davon haben, wie die Anwender Ihr Programm nutzen und was sie von dem Programm erwarten. Zudem sollte die Analyse einen Eindruck von den Abläufen im Programm geben und erste Hinweise auf die benötigten Klassen liefern.
3. **Design** – In der Design-Phase werden die Ergebnisse der Analyse ausgewertet und in Klassen-Spezifikationen umgesetzt. Dabei ist zu erfassen, welche Klassen (Objekte) benötigt werden, welches die wichtigsten Elemente der Klassen sind und in welchen Beziehungen die Klassen zueinander stehen (Klasse A definiert Felder vom Typ der Klasse B, Klasse C ist von Klasse D abgeleitet, eine Methode der Klasse E übernimmt Objekte der Klasse A als Argumente und so weiter).
4. **Implementierung** – Das Klassen-Design wird in Code umgesetzt.
5. **Test** – Das Programm (oder einzelne Module des Programms) werden kompiliert, ausgeführt und getestet. Fehler werden analysiert und behoben.

Wenn Sie Glück haben, müssen Sie im Zuge der Programmentwicklung jede dieser Phasen nur ein einziges Mal durchlaufen (das sogenannte *Wasserfall-Modell*, siehe Abbildung 12.1).

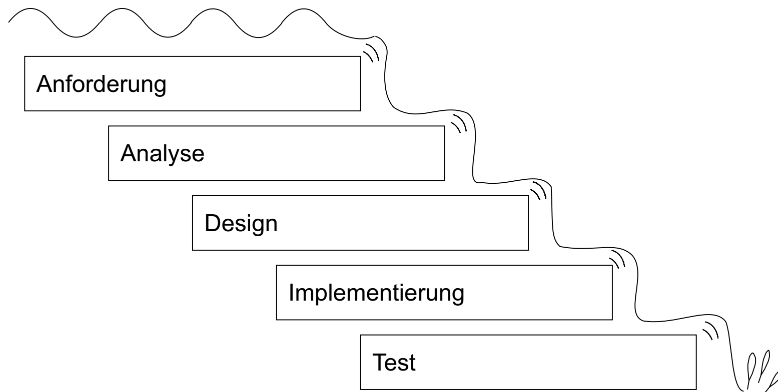


Abbildung 12.1: Das Wasserfall-Modell

Der Wasserfall ist jedoch eher eine Wunschvorstellung denn ein realistisches Modell. In der Praxis stellt sich die Software-Entwicklung vielmehr als iterativer Prozess dar, d.h., es muss immer wieder zu früheren Stufen zurückgesprungen werden.

Zeigen sich beispielsweise beim Testen Fehler in der Implementierung, muss der Code überarbeitet und die Fehler ausgemerzt werden.

Unstimmigkeiten und konzeptionelle Mängel können auch schon früher auftreten. Beispielsweise wenn die Programmierer in der Implementierungsphase entdecken, dass das Zusammenspiel mancher Klassen nicht stimmt oder dass zusätzliche Klassen benötigt werden. Dann müssen Design und gegebenenfalls Analyse korrigiert und verbessert werden.

Wenn Sie im Kundenauftrag arbeiten, empfiehlt es sich darüber hinaus, einen Prototyp zu entwerfen, der das Verhalten des angestrebten Produkts simuliert. Der Prototyp sollte die Benutzerschnittstelle und die Arbeit mit dem Programm demonstrieren. Ein Prototyp für eine Bankautomaten-Software würde beispielsweise alle benötigten Befehle anbieten (Kontostand abfragen, Geld abheben, Überweisung tätigen), diese jedoch nicht vollständig implementieren (Zugriff auf Konten-Datenbank der Bank etc.), sondern mit fest codierten Zahlen auf einfache Weise simulieren – allerdings so, dass der Kunde ein genaues Bild davon erhält, wie mit dem Programm später gearbeitet wird. Anhand eines solchen Prototyps lassen sich Schwachstellen im Programm und Missverständnisse zwischen Ihnen und Ihrem Kunden schnell aufdecken. Oftmals hilft der Prototyp auch dem Kunden, eine genauere Vorstellung von seinen Wünschen und Anforderungen an das Programm zu gewinnen. Unter Umständen muss danach zwar der gesamte Entwicklungsprozess, beginnend mit der Anforderungsspezifikation, erneut durchlaufen werden, doch je größer und komplexer das Projekt umso lohnenswerter ist es, Zeit und Mühe in die frühen Entwicklungsphasen zu investieren.

UML

In den Achtzigern und Neunzigern des letzten Jahrhunderts wurden eine Reihe von Ideen und Methoden entwickelt, mit denen sich der Software-Entwicklungsprozess effizienter gestalten lässt (Coad-Yourdon, Booch, Rumbaugh OMT, OOSE/Jacobson, ROOM etc.). Manche dieser Methoden konzentrierten sich auf die objektorientierte Analyse (OOA), andere auf das objektorientierte Design (OOD), nicht wenige verfolgten einen ganzheitlichen Ansatz. Im Mittelpunkt der meisten Methoden standen dabei zwei Kernfragen:

- »Wie findet man am besten heraus, welche Klassen/Objekte für ein Programm benötigt werden?« und
- »Wie lassen sich die Ergebnisse der Analyse- und Design-Phasen am übersichtlichsten und sinnvollsten darstellen?«

Die Beantwortung der letzten Frage führte dazu, dass eine Vielzahl von unterschiedlichen Notationen für die Beschreibung und Dokumentation von Analyse- und Design-Ergebnissen eingeführt wurde. Dabei arbeiteten die meisten Notationen mit durchaus vergleichbaren Mitteln und einem quasi universellen Satz von Elementen.

Mitte der Neunziger ergab es sich, dass Booch, Jacobson und Rumbaugh, die in den Achtziger noch jeder für sich ihre eigene Methodik entworfen hatten, ihren Weg zur Firma Rational Software, fanden. Dort arbeiteten sie zusammen eine universelle Notation für die objektorientierte Softwareentwicklung aus: UML, die »Unified Modelling Language« war geboren. Die Bemühungen der »drei Amigos«[2] wurden von der Industrie aufgegriffen und UML wurde zum De-facto-Standard für objektorientierte Analyse und Design.

Heute basieren nahezu alle wichtigen Programme für objektorientierte Analyse und Design auf UML.

Gute UML-basierte CASE-Tools für die objektorientierte Anwendungsentwicklung sind beispielsweise

– *Rational Rose (*`http://www.rational.com`*)*
– *ObjectiF (*`http://www.microtool.de`*)*
– *ArgoUML (*`http://argouml.tigris.org/`*) (Open Source)*

12.2.2 Fallbeispiel – Temperaturregelung

Auch für Programmierer, die nicht in einem Software-Entwicklungsteam arbeiten, die nicht mit CASE-Tools arbeiten und deren Programme noch von überschaubarer Komplexität sind, kann die Beschäftigung mit UML und den Prinzipien der objektorientierten Analyse durchaus von Vorteil sein und sich bei der Planung und Konzeption neuer Programme auszahlen.

2 So der offizielle Spitzname des produktiven Trios.

In diesem Sinne möchten wir Sie anhand eines kleinen Beispiels ein wenig in die Praxis objektorientierter Analyse und Designs einführen.

REF

Für eine ausführlichere Unterweisung in der Kunst des objektorientierten Designs mit UML lesen Sie bitte in der einschlägigen Fachliteratur nach.

Anforderungsspezifikation

Der erste Schritt besteht darin aufzuschreiben, wofür das Programm gedacht ist und was es leistet.

»Das Programm soll die Temperatur im Raum überwachen und auf einem konstanten Wert halten. Der Anwender kann die gewünschte Temperatur einstellen. Fällt oder steigt die Raumtemperatur aufgrund externer Einflüsse, stellt das Programm die Abweichung von der Solltemperatur fest und versucht sie auszugleichen.«

Die Analyse-Phase

Wenn Sie mit der Anforderungsanalyse beginnen, sollten Sie im Hinterkopf behalten, dass es dabei vorrangig *nicht* um die Identifizierung irgendwelcher Klassen, Objekte und Methoden geht. Es geht erst einmal nur darum, zu begreifen und zu analysieren, wie das Programm eingesetzt wird und was es leisten muss. Für eine gute Anforderungsanalyse ist daher viel wichtiger, dass man sich in dem Arbeitsfeld auskennt, in dem das Programm eingesetzt wird (der sogenannten **Domäne**), als dass man über Kenntnisse in objektorientierter Programmierung verfügt.

TIPP

Je komplexer das Einsatzfeld und die Aufgaben des Programms sind, umso wichtiger ist es, Fachleute aus dem jeweiligen Einsatzfeld in die Analyse einzubeziehen. Ebenfalls von Vorteil sind bereits gesammelte Erfahrungen in objektorientierter Analyse. In größeren Software-Teams wird die Analyse daher meist von Spezialisten übernommen.

Ein weit verbreitetes Verfahren zur objektorientierten Analyse ist die Formulierung von **Anwendungsfällen** (»use cases«).

Die Analyse mit Anwendungsfällen beginnt mit der Identifizierung der Akteure und Anwendungsfälle. In Anwendungsfalldiagrammen werden sodann die Beziehungen und Interaktionen zwischen den Akteuren und den Anwendungsfällen grafisch dargestellt. Zum Schluss setzt man detailliertere Beschreibungen der Anwendungsfälle – die sogenannten **Szenarien** – auf.

Anwendungsfälle identifizieren

Ein Anwendungsfall beschreibt immer, wer was mit dem Programm macht.

Der erste Anwendungsfall für das Temperaturregelungsprogramm lautet:

»Der Anwender stellt die Solltemperatur ein.«

Der Akteur ist in diesem Fall der »Anwender«, der eigentliche Anwendungsfall die Festlegung der Solltemperatur.

Der zweite Anwendungsfall besteht darin, dass sich die Außentemperatur ändert und das Programm die Temperaturdifferenz durch Regelung auszugleichen versucht.

Akteur ist hier die Außenwelt, den Anwendungsfall nennen wir »Temperatur regeln«.

Anwendungsfalldiagramme

Die Beziehung zwischen den Akteuren und den Anwendungsfällen wird in einem Anwendungsfalldiagramm dargestellt (siehe Abbildung 12.2).

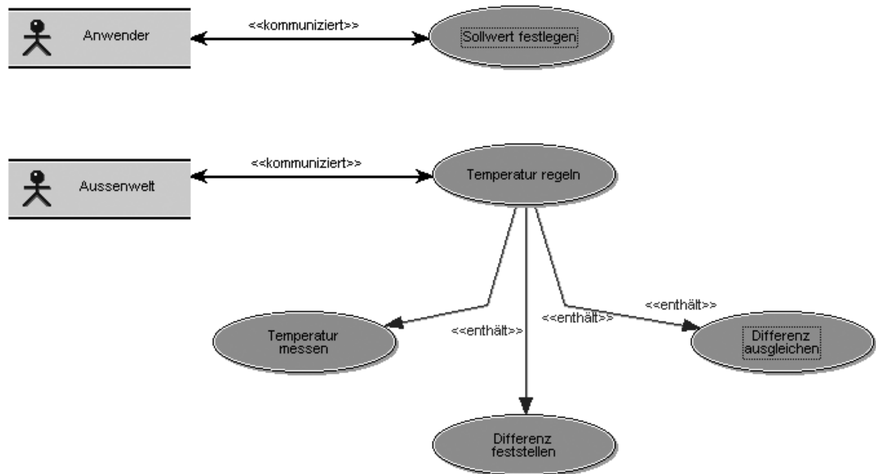


Abbildung 12.2: Anwendungsfälle für die Temperaturregelung

Wie Sie der Abbildung entnehmen können, werden die Beziehungen zwischen den Akteuren und Anwendungsfällen unterschiedlich klassifiziert. Typische Beziehungen sind:

- `<<kommuniziert>>`. Drückt die Beziehung zwischen einem Akteur und einem Anwendungsfall aus.
- `<<enthält>>`. Drückt aus, dass ein Anwendungsfall einen anderen »aufruft«.
- `<<erweitert>>`. Drückt aus, dass ein Anwendungsfall unter bestimmten Bedingungen durch die Einbindung eines anderen Anwendungsfalls erweitert wird.

Beschreibungen und Szenarien

Anwendungsfalldiagramme eignen sich hervorragend dazu, komplexe Abläufe zu visualisieren. Für eine gründliche Analyse ist es aber notwendig, weiter ins Detail zu gehen. Aus diesem Grunde setzt man Beschreibungen und **Szenarien**[3] für die einzelnen Anwendungsfälle auf.

3 Grundsätzlich gibt es zu jedem Anwendungsfall ein Hauptszenario, das den Ablauf des Anwendungsfalls beschreibt. Manche Anwendungsfälle können in Abhängigkeit von bestimmten Bedingungen unterschiedlich ablaufen. Diese abgewandelten Abläufe werden dann durch eigene Szenarien beschrieben.

Die Beschreibung des Anwendungsfalls »Sollwert festlegen« könnte wie folgt aussehen:

1. Akteur gibt die gewünschte Solltemperatur ein.
2. Das System speichert die Solltemperatur.

Der Anwendungsfall »Temperatur regeln« ließe sich beschreiben durch:

1. System fordert Messglied auf, die Außentemperatur festzustellen.
2. System gibt Messwert an Regelglied, um Differenz festzustellen.
3. System gibt Differenz an Stellglied, um Unterschied auszugleichen.

Messglied, Regelglied und Stellglied sind dabei die typischen Elemente eines Regelkreises.

Betrachten wir zum Schluss noch die Beschreibung des untergeordneten Anwendungsfalls »Temperatur messen«:

1. System fragt aktuelle Temperatur ab.
2. Messglied fragt Temperatur von Außenwelt ab.
3. Messglied liefert Temperatur an System zurück.

Besser als die Textbeschreibungen können Diagramme die Abläufe beschreiben. Abbildung 12.3 zeigt ein Aktivitätsdiagramm, das für den kompletten Anwendungsfall »Temperatur regeln« erstellt wurde. Es verdeutlicht die Chronologie der Abläufe und zeigt auch an, welche Akteure/Komponenten die einzelnen Schritte ausführen. (Die Komponenten wurden der Beschreibung des Anwendungsfalls entnommen. Die Komponente Regelkreis steht für das »System«.)

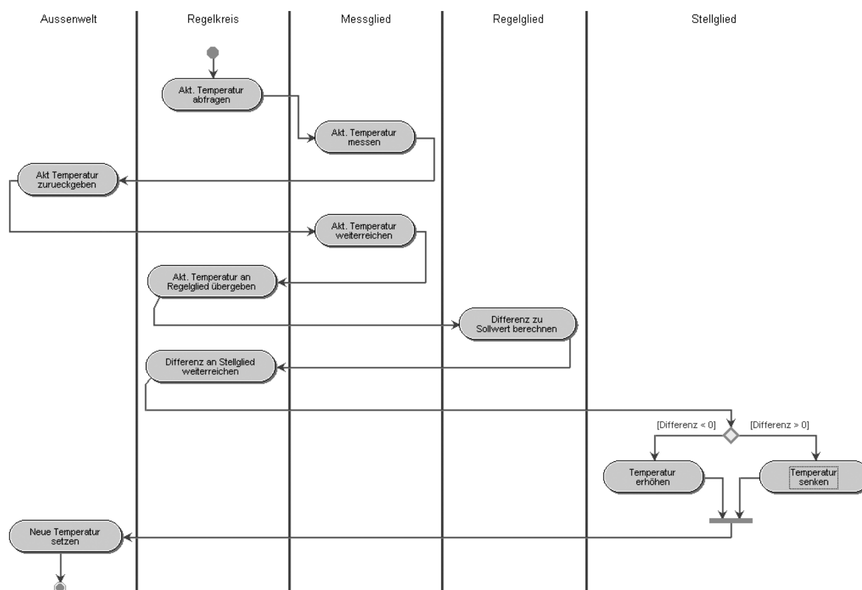


Abbildung 12.3: Aktivitätsdiagramm für den Anwendungsfall »Temperatur regeln«

Die Design-Phase

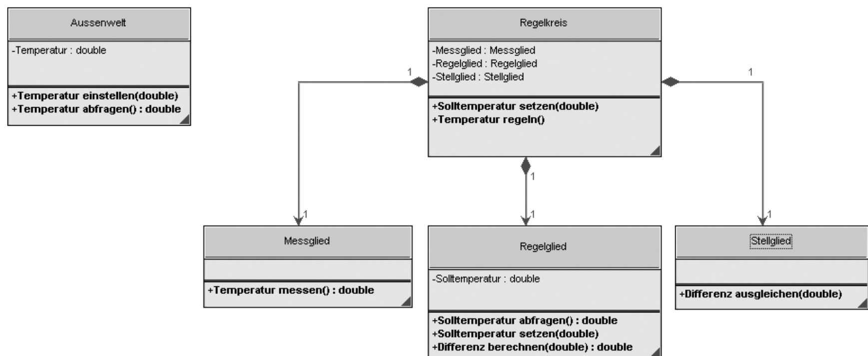
In der Design-Phase geht es darum, die Erkenntnisse aus der Analyse in ein funktionierendes Klassen-Design umzuwandeln. Der erste Schritt hierzu ist die Erstellung eines **Klassendiagramms**.

Hinweise auf die Klassen, die Sie benötigen, geben uns die Anwendungsfälle. Alle Substantive, die in den Beschreibungen der Anwendungsfälle auftauchen, sind potenzielle Kandidaten für zu implementierende Klassen. Wenn Sie bereits ein Aktivitätsdiagramm erstellt haben, übernehmen Sie die »Komponenten« als Klassen in das Klassendiagramm.

Für jede Klasse legen Sie fest, welche Operationen sie ausführen soll (Hinweise auf die benötigten Operationen geben wiederum die Anwendungsfall-Beschreibungen und das Aktivitätsdiagramm). Können Sie aus den Operationen bereits auf benötigte Datenelemente (Attribute) schließen, führen Sie diese ebenfalls in der Klassenspezifikation auf.

Selbstverständlich stehen die Operatoren für die Methoden und die Attribute für die Felder der Klasse, doch sollten Sie in der Design-Phase noch nicht zu sehr an Implementierungsdetails denken. Gute OOD-Software gestattet es, für Operatoren und Attribute je zwei Namen anzugeben: einen Namen, der im Diagramm verwendet wird, und einen zweiten, den das Element in der später zu erzeugenden Klassendefinition tragen wird.

Abbildung 12.4: Klassendiagramm für das Temperaturregelungsprogramm



Im Klassendiagramm werden nicht nur die einzelnen Klassen mit ihren Attributen und Operatoren dargestellt, sondern auch die Beziehungen zwischen den Klassen.

Die Linien mit den ausgefüllten Rauten (siehe Abbildung 12.4) dienen in UML beispielsweise dazu, eine Komposition auszudrücken, d.h., die Klasse, auf die die Raute weist, enthält ein eingebettetes Objekt (sprich ein Feld) vom Typ der Klasse am anderen Ende der Beziehung.

- **Assoziation**

Die Assoziation ist die häufigste und allgemeinste Beziehung zwischen zwei Klassen. Sie besagt, dass eine Klasse konzeptionell mit einer anderen Klasse verbunden ist. Beispielsweise könnte eine Klasse `Lehrer` mit einer Klasse `Schüler` assoziiert sein. (Die Beziehung könnte dann den Namen »unterrichtet« tragen.)

Die Assoziation ist gerichtet. Ein Pfeil zeigt an, welche Klasse eine andere (auf die der Pfeil weist) »benutzt«. Sind die beiden Klassen gegenseitig aufeinander angewiesen, wird die Assoziation durch eine Linie ohne Pfeile dargestellt.

Die Assoziation besitzt auch eine Multiplizität, die angibt, wie viele Objekte einer Klasse mit wie vielen Objekten einer anderen Klasse assoziiert sind. (Beispielsweise könnte ein Objekt der Klasse `Lehrer` mit beliebig vielen (*) Objekten der Klasse `Schüler` assoziiert sein.)

- **Abhängigkeit**

Die Abhängigkeit zeigt an, dass eine Klasse eine andere benötigt. Meist wird sie benutzt, wenn die Operation einer Klasse ein Objekt einer anderen Klasse als Parameter verwendet.

Die Abhängigkeit ist gerichtet.

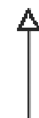
- **Aggregation**

Aggregation liegt vor, wenn eine Klasse andere Klassen als Bestandteil enthält Eine unausgefüllte Raute markiert das Ende der Klasse, die eine andere enthält.

Die normale Aggregation kann noch verstärkt werden. Man spricht dann von Komposition (angezeigt durch eine ausgefüllte Raute). Wenn eine Klasse `B` mit einer Klasse `A` durch Komposition verbunden ist, wird das Objekt der Klasse `B`, das in der Klasse `A` enthalten ist, zusammen mit den Objekten der Klasse `A` erzeugt und aufgelöst. Des Weiteren kann ein Objekt, das in einem Objekt der Klasse `A` enthalten ist, nicht mehr Teil eines anderen Objekts sein.

Die Aggregation ist gerichtet.

- **Generalisierung**



Die Generalisierung zeigt an, dass eine Klasse eine Erweiterung einer anderen Klasse ist. Sie entspricht dem objektorientierten Konzept der Vererbung (siehe nachfolgendes Kapitel).

Die Assoziation ist gerichtet. Der Pfeil weist von der erweiterten Klasse zur Basisklasse.

Gibt es viele Klassen, ist es am übersichtlichsten, wenn man für jedes Anwendungsfalldiagramm ein eigenes Klassendiagramm erstellt. Besonders komplexe Anwendungsfälle kann man in mehreren Klassendiagrammen nachbilden. Auf jeden Fall sollte man sich um eine Zuordnung von Anwendungsfall- und Klassendiagrammen bemühen, da dies die Orientierung erleichtert (denken Sie daran, dass objektorientierte Programmierung ein iterativer Prozess ist, in dem man meist mehrmals von der Analyse zum Design und wieder zurück springt).

Implementierung

Schließlich werden die Klassenspezifikationen aus der Design-Phase in echte Klassendefinitionen umgesetzt und die Klassenmethoden implementiert. Wenn Sie Ihre Klassendiagramme mit einer modernen UML-Software erstellt haben, können Sie die Klassendefinitionen mit höchster Wahrscheinlichkeit von der UML-Software generieren lassen. Sie brauchen dann nur noch die Methoden zu implementieren (falls Sie dies nicht ebenfalls schon in der UML-Software erledigt haben) und den erzeugten Code nachbearbeiten.

Wenn Sie die Klassendiagramme auf Papier entworfen haben, müssen Sie die zugehörigen Quelldateien selbst anlegen und mit Ihren Klassendefinitionen füllen.

Die Quelltextdateien zu dem Temperaturregelungsprojekt finden Sie auf der Buch-CD. Es handelt sich allerdings nur um die Implementierung eines Prototyps. Anstatt die Außentemperatur zu messen und über eine Air Condition-Einheit zu regeln, liest und verändert der Regelkreis den Wert des `temperatur`*-Feldes der Klasse* `Aussenwelt`*.*

Die Klassen des Regelkreises und der Simulationsklasse `Aussenwelt` *wurden in einem eigenen Paket* `regelkreis` *definiert. Zum Testen des Regelkreises wurde die Klasse* `Simulation` *geschrieben, die eine* `main()`*-Methode enthält und im unnamed-Paket definiert ist.*

Das Simulationsprogramm wird mit der gewünschten Solltemperatur als Argument aufgerufen. Es stellt die Außentemperatur auf 0.3 Grad kälter als die Solltemperatur ein und ruft dann in einer `for`*-Schleife den Regelkreis auf, der die Temperatur in 0.5 Grad-Schritten ausgleicht. Nach jeder 10. Iteration erzeugt ein Zufallsgenerator eine weitere Temperaturabweichung zwischen –1 und + 1 Grad.*

```
C:\Beispiele\Kapitel12>javac -d . Simulation.java
```

```
C:\Beispiele\Kapitel12>java Simulation 24
1. Iteration    Temperatur = 23,7
2. Iteration    Temperatur = 23,75
3. Iteration    Temperatur = 23,8
4. Iteration    Temperatur = 23,85
```

```
5. Iteration    Temperatur = 23,9
6. Iteration    Temperatur = 23,95
7. Iteration    Temperatur = 24
8. Iteration    Temperatur = 24
9. Iteration    Temperatur = 24

10. Iteration    Temperatur = 24,9
11. Iteration    Temperatur = 24,85
12. Iteration    Temperatur = 24,8
13. Iteration    Temperatur = 24,75
14. Iteration    Temperatur = 24,7
15. Iteration    Temperatur = 24,65
...
```

12.2.3 Programme aus mehreren Quelldateien

Wenn Sie größere Programme schreiben, die mehrere bis viele Klassen umfassen, sollten Sie die Klassendefinitionen unbedingt auf mehrere Quelldateien verteilen. Hierzu ein paar Tipps.

Aufteilung der Klassen

Die Grundregel zur Aufteilung von Klassen auf Quelltextdateien lautet:

»Eine Klasse – eine Quelldatei«

Dies ist natürlich nur eine Empfehlung, aber eine, die beherzigt werden sollte. Wenn Sie es in einem speziellen Fall dennoch vorziehen, von dieser Regel abzuweichen, dann sollten Sie dafür einen Grund haben – etwa:

- Eine Klasse A benötigt eine Hilfsklasse B. Darum definieren Sie B zusammen mit A in einer Datei. (Hier wäre überdies zu prüfen, ob B nicht sogar als (private?) innere Klassen von A definiert werden sollte, siehe Kapitel 10.4.)
- Sie haben mehrere kleine Klassen mit ähnlicher Funktionalität, die sich zusammen in einer Quelldatei einfach übersichtlicher verwalten und besser bearbeiten lassen (»Sieben Zwerge«-Prinzip).
- Sie schreiben ein Testprogramm mit ein bis drei Klassen begrenzten Umfangs, und Sie scheuen den Aufwand, die Klassen aufzuteilen.

Die Zusammenlegung von Klassendefinitionen in Quelltextdateien hat in Java allerdings ihre Grenzen, denn

»`public`-*Klassen müssen in Quelltextdateien abgespeichert werden, die den gleichen Namen tragen wie die Klasse.*«

Woraus folgt, dass in einer Quelltextdatei zwar mehrere Klassen, aber immer nur höchstens eine `public`-Klasse definiert werden kann.

Aber auch Quelltextdateien, die eine nicht-`public` Klasse enthalten, sollten Sie nach der Klasse benennen: Erstens müssen Sie dann nicht lange darüber nachgrübeln, wie Sie die Quelldatei nennen sollen. Zweitens können Sie

dann am Namen der Quelldatei ohne Mühe ablesen, welche Klasse in ihr definiert ist. Drittens unterstützen Sie den Compiler dadurch bei der Suche nach benötigten Klassendefinitionen.

TIPP

Trifft der Compiler bei der Übersetzung einer Quelldatei auf eine Klasse, die ihm noch nicht bekannt ist, sucht er im Quell- und Klassenpfad nach einer Class- oder Quelltextdatei (!), die den gleichen Namen trägt wie die Klasse. Findet er eine solche Datei, lädt er sie und entnimmt ihr die Klassendefinition. Eine Klasse, die in einer anders benannten Quelltextdatei definiert ist, wird nur gefunden, wenn es im Klassenpfad bereits eine Class-Datei der Klasse gibt oder Sie dem Compiler beim Aufruf explizit angeben, die betreffende Quelltextdatei zu laden und zu kompilieren. (Mehr zur Arbeitsweise des Compilers im Anhang zu den JDK-Tools.)

Pakete

Größere Programme (oder Bibliotheken), und ganz besonders solche, die Sie an andere Programmierer oder Kunden weitergeben, sollten Sie in Paketen organisieren.

Für Programme und Bibliotheksmodule, die für den privaten Gebrauch bestimmt sind, genügt meist ein einfacher Paketname, beispielsweise:

```
package regelkreis;
```

Programme oder Bibliotheken, die Sie verkaufen, sollten über eindeutige Paketnamen verfügen. Am einfachsten stellen Sie dies sicher, indem Sie den Namen Ihrer Webdomäne (sofern Sie über eine solche verfügen), in umgekehrter Reihenfolge als Präfix des Paketnamens verwenden:

```
package de.carpelibrum.software.regelkreis;
```

Wenn Sie alle Klassen eines Programms (oder einer Bibliothek) in einem Paket definieren, vereinfacht dies die Kompilierung. Wenn Sie allerdings mit mehreren Programmierern zusammen an einem großen Projekt arbeiten, sollten Sie sich überlegen, ob es nicht sinnvoller und übersichtlicher ist, das Projekt in Domänen mit eigenen Paketnamen aufzuteilen.

Verteilung der Quelldateien

Grundsätzlich sollten Sie für jedes Programm ein eigenes Verzeichnis anlegen und in diesem die Quelldateien des Programms speichern.

Wenn Ihr Programm jedoch aus mehreren Paketen besteht, empfiehlt es sich, unter dem Hauptverzeichnis des Programms für jedes der Pakete ein eigenes Unterverzeichnis anzulegen und in diesem Verzeichnis die zu dem Paket gehörenden Quelldateien abzuspeichern. Dies vereinfacht die spätere Kompilierung.

Beispiel

Ihr Programm definiert das Paket `demo`, verschiedene Klassen des Programms sind in den Paketen `demo.unterpaket1` und `demo.unterpaket2` definiert.

Dann legen Sie unter dem Hauptverzeichnis des Programms ein Verzeichnis `demo` mit zwei Unterverzeichnissen `unterpaket1` und `unterpaket2` an und verteilen, die Quelltextdateien entsprechend ihrer `package`-Deklaration auf diese Verzeichnisse.

Vergessen Sie nicht Klassen, die in anderen Paketen verwendet werden, als `public` *zu deklarieren, und denken Sie auch daran, dass Sie auf die* `public`*-Klassen anderer Pakete über den Paketnamen zugreifen müssen (oder zuvor den Namen importieren, siehe Kapitel 10.1).*

Kompilierung und Ausführung

Keine eigenen Pakete

Wenn Ihr Programm kein eigenes Paket definiert und alle Quelltextdateien in einem Verzeichnis stehen, wechseln Sie auf der Konsole in dieses Verzeichnis und kompilieren Sie die Quelltextdateien mit dem Befehl:

```
javac *.java
```

Um das Programm auszuführen, rufen Sie folgenden Befehl auf:

```
java Programm
```

(Wobei `Programm` für den Namen der Klasse steht, die die `main()`-Methode enthält.)

Ein eigenes Paket

Wenn Ihr Programm ein eigenes Paket `demo` definiert und alle Quelltextdateien in einem Verzeichnis stehen, wechseln Sie auf der Konsole in dieses Verzeichnis und kompilieren Sie die Quelltextdateien mit dem Befehl:

```
javac -d . *.java
```

Class-Dateien von Klassen, die in einem Paket definiert sind, speichert der Compiler immer in einem Verzeichnis, das den Namen des Pakets trägt.[4] Die Option –`d .` sorgt dafür, dass dieses Verzeichnis unter dem aktuellen Verzeichnis (.) angelegt wird.

Um das Programm auszuführen, rufen Sie folgenden Befehl auf:

```
java demo.Programm
```

(Wobei `Programm` für den Namen der Klasse steht, die die `main()`-Methode enthält.)

Wechseln Sie nicht aus dem Hauptverzeichnis in das vom Compiler angelegte Paketverzeichnis demo*, um von dort aus* `java Programm` *aufzurufen. Der Interpreter findet dort zwar eine Class-Datei namens* Programm.class*, erwartet aber in dieser eine Klasse* `Programm` *vorzufinden, die im »unnamed«-Paket definiert ist, während die gefundene* `Programm`*-Klasse im Paket* `demo` *definiert ist. Der Interpreter quittiert dies mit einer Fehlermeldung.*

4 Enthält die Paketangabe Unterpakete (beispielsweise `eigenesPaket.subPaket`) werden die Class-Dateien im entsprechenden Verzeichnispfad gespeichert (`/eigenesPaket/subPaket`).

Mehrere eigene Pakete

Wenn Ihr Programm mehrere eigene Pakete definiert und die Quelltextdateien entsprechend ihrer Paketzugehörigkeit auf Unterverzeichnisse des Programmverzeichnisses verteilt sind (vergleiche Beispiel aus Abschnitt »Verteilung der Quelldateien«), wechseln Sie auf der Konsole in das übergeordnete Programmverzeichnis und kompilieren Sie die Quelltextdateien mit dem Befehl:

```
javac -d . demo/Programm.java
```

(Wobei `Programm` für den Namen der Klasse steht, die die `main()`-Methode enthält und davon ausgegangen wird, dass die Quelltextdatei im Unterverzeichnis `demo` steht.)

Um das Programm auszuführen, rufen Sie folgenden Befehl auf:

```
java demo.Programm
```

Achtung! Im obigen *javac*-Aufruf nutzen Sie den Umstand, dass sich der Compiler während der Übersetzung des Programms alle an dem Programm beteiligten Klassen zusammensucht und kompiliert. Handelte es sich um eine Bibliothek, deren Klassen auf verschiedene Pakete (Verzeichnisse) verteilt sind, müssen Sie davon ausgehen, dass die Klassen sich nicht gegenseitig verwenden. Folglich müssen Sie für eine vollständige Kompilierung alle zu kompilierenden Quelldateien angeben:

```
javac -d . demo/*.java  demo/unterpaket1/*.java  demo/unterpaket2/*.java
```

TIPP

Wenn die Klasse mit der `main()`*-Methode im unnamed-Paket liegt, gehen Sie analog vor, speichern die Quelltextdateien ohne Paket-Deklaration im Hauptverzeichnis des Programms und rufen folgende Befehle auf:*

```
javac -d . Programm.java
   java Programm
```