Ferienkurs Design Patterns

© Goll

1 Das Konzept der Design Patterns 3
1.1 Beschreibung von Entwurfsmustern 4
1.2 Arten von Entwurfsmustern 5
1.3 Zukunft der Entwurfsmuster 6
2 Wichtige Entwurfsmuster 8
2.1 Aggregation 8
2.2 Delegations 13
2.3 Interfaces zur Zugriffseinschränkung 19
2.4 Schnittstellen und Varianten 22
2.5 Immutable (Unveränderlichkeit) 25
2.6 Schablonenmethode (Template Method) 28
2.7 Dekorierer 30
2.8 Kompositum 31
2.9 Proxy (Stellvertreter) 32
2.10 Observer 60
2.11 Producer/Consumer 72
2.12 Facade 75
2.13 Singleton 78
2.14 Abstrakte Fabrik 81
3 Architectural and design patterns 89
3.1 Architectural pattern MVC 89
3.2 Design patterns inherent in the MVC pattern 90
4 Ein zusammenhängendes Anwendungsbeispiel 96
4.1 Anforderungen 96
4.2 Analyse 96
4.3 Entwurf 96
4.4 Implementierung 96

© Goll

1 Das Konzept der Design Patterns

Design Patterns zogen in den letzten Jahren die Aufmerksamkeit vieler Softwareentwickler auf sich. Internetseiten wurden eingerichtet, Workshops durchgeführt, sogar Konferenzen wurden eigenes für Design Patterns ins Leben gerufen. Doch was sind Design Pattern überhaupt ?

Design Patterns - oder auch Entwurfsmuster genannt - spiegeln lediglich die bereits gewonnenen Erfahrungen vieler Entwickler wieder. In diesem Zusammenhang betrifft dies natürlich speziell die Erfahrungen der Softwareentwickler. Jedoch ist an dieser Stelle zu sagen, dass der Ursprung der Designpatterns nicht, wie man unter Umständen vermutet, in der Softwareentwicklung liegt. Muster kommen vielmehr aus dem Bereich der Architektur. Christopher Alexander[1] war es, der zu seiner Zeit den folgenden Satz formulierte: "Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen." Dieser Satz wird in nahezu jedem Buch über Entwurfsmuster zitiert und darf somit auch hier nicht fehlen. Diese Definition trifft sowohl auf Muster von Gebäuden und Städten, als auch auf objektorientierte Entwurfsmuster zu. Einfacher gesprochen kann man auch sagen: "Design Patterns sind wiederverwendbare Lösungen für immer wiederkehrende Entwurfsprobleme". In der Softwareentwicklung haben jedoch vier Personen den Begriff des Design Patterns wieder aufgegriffen und erst richtig bekannt gemacht. Im Herbst 1994 wurde das Buch "Design Patterns – Elements of Reusable Object-Oriented Software" von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides veröffentlicht. Es stellte zwar zu dieser Zeit keine neuen Erkenntnisse dar, war jedoch die erste Sammlung von gut beschriebenen Design Patterns. Das "**GoF**"-Buch[2] zählt heute zu den am meisten verkauften Büchern der modernen Softwareentwicklung. Um Entwurfsmuster anderen Entwicklern zugänglich zu machen, müssen diese dokumentiert werden. Entwurfsmuster sind dabei nicht etwa wiederverwendbarer Quellcode, der in einer Anwendung einfach übernommen werden kann, sondern vielmehr weitergereichte Erfahrungen vieler Jahre der objektorientierten Softwareentwicklung. Die konkrete Implementierung von Entwurfsmustern bleibt jedoch den Entwicklern selbst überlassen.

Mit Hilfe von Design Patterns kann also Entwurfswissen an andere Entwickler weiter gereicht werden. Im Gespräch mehrerer Entwickler kann außerdem auf bekannte Design Patterns Bezug genommen werden. Das erlaubt Diskussionen auf einem hohen Abstraktionsniveau [GAM1995],[GRA1998].

---

[1] Christopher Alexander ist praktizierender Architekt und Städteplaner. Er hat eine Professur für Architektur an der University of California in Berkley und ist Direktor des Center for Environment Structure.

[2] Die "Gang of Four" sind Erich Gamma, Richard Helm Ralph Johnson und John Vlissides. Diese Bezeichnung wurde den vier Autoren scherzhaft zugesagt und ist ohne großen Widerspruch von ihnen angenommen worden. Im Folgenden [GoF95] genannt.

Komplexe Aufgaben erfordern komplexe Beziehungen zwischen Objekten. Ein Design Pattern beschreibt ein Beziehungsgeflecht, mit dem die geforderte Aufgabe gelöst werden kann. In der Regel werden dabei die Abhängigkeiten zwischen den beteiligten Objekten minimiert. Eine Software, die auf einer Architektur mit wenigen Abhängigkeiten basiert, ist leicht zu pflegen und zu erweitern.

1.1 Beschreibung von Entwurfsmustern

Betrachtet man darüber hinaus etwas detaillierter die Entwurfsmuster der Softwaretechnik, lassen sie sich auf vier grundlegende Elemente reduzieren:

- Kontext
- Problem
- Lösung
- Konsequenzen

Diese vier Elemente sind eng miteinander verknüpft und sollten zunächst einmal erläutert werden.

Der **Kontext** beschreibt das jeweilige Umfeld, in dem erst das Problem auftreten kann. In diesem Umfeld kann das Muster zum Einen für sich alleine stehen, zum Anderen aber auch in Kombination mit anderen Entwurfsmustern gebracht werden. Im Kontext wird in aller Regel anhand eines Beispiels auf die Gegebenheiten eingegangen. Dieses Beispiel wird dann im weiteren Verlauf der Musterbeschreibung immer wieder zur Hilfe genommen.

Das **Problem** beschreibt, wann ein bestimmtes Muster anzuwenden ist, also wann das Problem in dem spezifizierten Kontext auftritt. Hier stellt sich auch zum ersten Mal die Frage, ob das jeweilige Pattern überhaupt sinnvoll und anwendbar für die gegebene Situation ist. In der Regel wird dies anhand einer Liste von Bedingungen, welche zutreffen müssen, abgearbeitet.

Der **Lösungsabschnitt** nimmt im Allgemeinen den meisten Platz für sich in Anspruch. Hier werden alle Elemente, aus denen der resultierende Entwurf besteht, vorgestellt. Zudem werden an dieser Stelle auch die Beziehungen zwischen den Elementen, Interaktionen und Zuständigkeiten erläutert. In der Lösung wird jedoch noch keine konkrete Implementierung beschreiben. Die Lösung beschreibt vielmehr eine abstrakte Sicht auf die Elemente der Architektur. Durch diese abstrakte Sicht bilden Patterns eine ideale Kommunikationsbasis für Entwickler.

Die **Konsequenzen** beschreiben die Vor- und Nachteile des aus der Anwendung des Musters resultierenden Entwurfs. Die Konsequenzen sind nicht unerheblich beim Einsatz von Patterns, da diese oft den Speicherverbrauch oder die Ausführungszeit betreffen. Sie beziehen sich ebenfalls auf die Flexibilität und Wiederverwendbarkeit des entstehenden Entwurfs.

Anhand dieser vier Elemente wurde nun verdeutlicht, aus was ein Entwurfsmuster besteht. Jedoch ist noch die Frage zu klären, wie ein Entwurfsmuster überhaupt entsteht. Wer entscheidet, wann die von ihm erdachte Architektur richtig oder falsch ist? Grundsätzlich kann man sagen, dass Design Patterns nicht erfunden, sondern gefunden werden. Da dieselben Entwurfsprobleme in der einen oder anderen Form immer wieder vorkommen, können die Erfahrungen, die im letzten Lösungsversuch gewonnen wurden, in einen erneuten Entwurf mit einfließen. Dadurch wird ein Entwurf im Laufe der Zeit immer wieder verbessert und reift somit allmählich zu einem Design Pattern heran. Die Erfahrungen, die man im Laufe der Zeit gewonnen hat, werden letztendlich dokumentiert und somit anderen Entwicklern zugänglich gemacht. Entwurfsmuster werden zudem nicht von einem einzigen Entwickler gefunden. Beim Finden von Entwurfsmustern spielen auch immer die Erfahrungen mehrerer Entwickler eine Rolle. Die Entwicklung erfolgt dabei stets unabhängig voneinander. Was sie jedoch miteinander verbindet ist, dass alle Entwickler vor demselben Architekturproblem stehen und eine nahezu identische Architektur zur Lösung des Problems vorschlagen.

Von Anwendersicht aus gesehen bieten Entwurfsmuster eine gewisse Sicherheit. Eine Sicherheit insofern, dass der Entwurf, den man verwenden will, sich in der Vergangenheit schon bewährt hat. Dies führt im Allgemeinen jedoch zu dem Trugschluß, dass das Anwenden von Entwurfsmustern in jedem Fall die richtige Entscheidung ist. Diese Annahme ist jedoch schlichtweg falsch. Das reine Anwenden von Entwurfsmustern garantiert einem Entwickler nicht, dass das Muster an dieser Stelle und für sein konkretes Entwurfsproblem überhaupt sinnvoll ist. Patterns sollten immer nur dann angewandt werden, wenn sie auch sinnvoll sind und Vorteile bringen. Aus diesem Grund müssen die drei Elemente Kontext, Problem und Konsequenzen schon vor dem Anwenden gut überdacht werden und auch andere Patterns zur Lösung des Problems beachtet werden. Unter Umständen führt sogar keines der 23 Patterns aus [GoF95] zum gewünschten Ziel. Ein guter Entwurf entsteht erst dann, wenn mehrere Entwurfsmuster untereinander und mit der Anwendung verwoben werden, um einen Synergie-Effekt zu erzielen.

1.2 Arten von Entwurfsmustern

Um eine Auswahl der Muster zu erleichtern, werden sie in bestimmte Kategorien eingeteilt. Diese Einteilung wird aber in jedem Buch oder Katalog über Entwurfsmuster anders gehandhabt. In [GoF95] wird eine Einteilung in die drei Kategorien **Erzeugungsmuster**, **Strukturmuster** und **Verhaltensmuster** vorgeschlagen. In [BU98] wird eine Einteilung in Architekturmuster, Entwurfsmuster und Idiome vorgeschlagen. Die Einteilung geschieht weitgehend frei und ist den Autoren selbst überlassen. Im Folgenden werden die Kategorien von [GoF95] als Grundlage genommen und näher erläutert:

- **Erzeugungsmuster** beschäftigen sich mit der Aufgabe, wann und wie bestimmte Objekte erzeugt werden. Darüber hinaus verbergen sie den eigentlichen Erzeugungsprozeß nach außen hin. Sie machen ein System unabhängig davon, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden.

Erzeugungsmuster bestimmen, **was** erzeugt wird, **wer** es erzeugt, **wie** es erzeugt und **wann** es erzeugt wird. Ein einfaches Beispiel für Erzeugungsmuster ist das Singleton-Muster. Der Zweck dieses Musters wird folgendermaßen beschrieben: "Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit." Auf die Erzeugungsmuster wird jedoch im Folgenden nicht näher eingegangen.

- **Strukturmuster** befassen sich mit der Zusammensetzung und der Granularität von Klassen und Objekten. Diese Kategorie wird noch in zwei weitere Kategorien aufgeteilt. Zum einen in die **klassenbasierten Strukturmuster**, welche sich die Eigenschaft der Vererbung zu Nutzen machen, um Schnittstellen oder Implementierungen zusammenzuführen. Die Mehrfachvererbung ist dafür eine einfaches Beispiel. Durch Mehrfachvererbung werden mehrere Klassen zu einer einzigen zusammengeführt. Die zweite Unterteilung bilden die **objektbasierten Strukturmuster**. Sie beschreiben die Möglichkeiten, Objekte zusammenzuführen, um neue Funktionalität zu gewinnen. Das Proxy-Muster ist ein Beispiel für die objektbasierten Strukturmuster. Der Proxy dient als Stellvertreter für ein anderes Objekt. Auf das Proxy-Muster wird im späteren Verlauf noch genauer eingegangen.

- **Verhaltensmuster** befassen sich mit den Zuständigkeiten und der Zusammenarbeit zwischen Klassen bzw Objekten. Sie beschreiben komplexe Interaktionen zwischen Objekten, die zur Laufzeit nur schwer nachzuvollziehen sind. Auch hier wird unterschieden zwischen den klassenbasierten und den objektbasierten Verhaltensmustern. **Klassenbasierte Verhaltensmuster** verwenden Vererbung, um das Verhalten unter den Klassen zu verteilen. **Objektbasierte Verhaltensmuster** dagegen verwenden die Objektkomposition. Das Beobachtermuster stellt ein Beispiel für Verhaltensmuster dar. Es definiert Abhängigkeiten zwischen Objekten, so dass bei Änderungen des Zustands eines Objektes alle abhängigen Objekte automatisch benachrichtigt werden.

1.3 Zukunft der Entwurfsmuster

Einen umfassenden Katalog von Entwurfsmustern wird es wohl nie geben, denn es entstehen ständig neue Entwurfsmuster und viele bereits unbewusst angewandte Entwurfsmuster sind noch unbenannt. Auch gehen von Zeit zu Zeit Entwurfsmuster in die objektorientierte Denkweise selbst über und sind keine Entwurfsmuster im eigentlichen Sinne mehr. Die sogenannte "Pattern Community", die Gemeinde von Entwurfsmuster-Anhängern, wächst stetig und bringt ständig neue Erkenntnisse im Zusammenhang mit Entwurfsmuster hervor. Es finden ständig Konferenzen zum Thema Entwurfsmuster auf der ganzen Welt statt. Entwurfsmuster stellen so einen wesentlichen Beitrag dar, die Softwareentwicklung auf ihrem Weg zur ausgereiften Ingenieurwissenschaft ein gutes Stück weiterzubringen.

© Goll

2 Wichtige Entwurfsmuster

Im Folgenden sollen einige Entwurfsmuster vorgestellt werden. Auf eine schematisierte Darstellung wird bewusst verzichtet. Eine solche ist für alle hier vorgestellten Entwurfsmuster in [GoF95] und [Gra98] zu finden. Stattdessen sollen die Entwurfsmuster detailliert und verständlich beschrieben werden. Varianten und interessante Aspekte der Entwurfsmuster sollen herausgestellt werden und ein Beispiel soll die Anwendung des Entwurfmusters verdeutlichen.

2.1 Aggregation

Das Aggregationspattern beschreibt die leicht erweiterbare Zusammensetzung eines Objekts aus mehreren Teilobjekten.

2.1.1 Ein Einführungsbeispiel

[GRA1998] Zur Konstruktion eines Flugzeugs stehen bereits die Komponenten `Motor`, `Landeklappen` und `Fahrgestell` zur Verfügung. Ein typischer Anfängerfehler wäre der Versuch, eine neue Klasse `Flugzeug` von den drei Komponentenklassen abzuleiten. Obwohl dem `Flugzeug`-Objekt dann sofort alle Methoden der Komponenten zur Verfügung stehen würden, ist diese Architektur nicht empfehlenswert. Wenn das Flugzeug von den `Landeklappen` und dem `Fahrgestell` abgeleitet wäre, so würde das zu einem Konflikt mit den namensgleichen Methoden `einfahren()` und `ausfahren()` führen, die beide Komponenten besitzen. Abgesehen davon wären solche Methodenaufrufe im Bezug auf das ganze Flugzeug unverständlich. Auch bei späteren Erweiterungen der Komponenten müsste man darauf achten, dass keine neue Methode angelegt wird, deren Namen bereits für die Methode einer anderen Komponente eingesetzt wird. Die Erweiterung um einen zweiten Motor der gleichen Bauart lässt kein Compiler über Mehrfachvererbung zu. Sie würde naturgemäß auch zu Namenskonflikten führen.

Es gibt keinen Fall, bei dem ein Entwickler auf Mehrfachvererbung zurückgreifen muss. Er kann immer den wesentlich eleganteren und flexibleren Weg über das Aggregationspattern gehen.

Das Objekt `Flugzeug` erhält dabei Referenzen auf seine Teilobjekte. Wenn nun Methoden der Teilobjekte aufgerufen werden sollen, so delegiert das `Flugzeug` den Aufruf an seine Teilobjekte.

© Goll

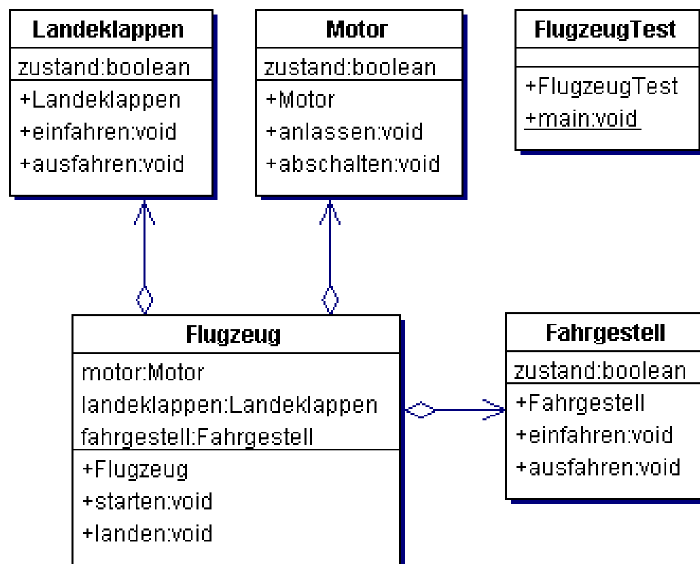


Abbildung 1: Zusammensetzen durch Aggregation

Das Klassendiagramm veranschaulicht, dass das Flugzeugobjekt `Flugzeug` spezielle Methodenaufrufe an seine Teilobjekte `Motor`, `Landeklappen` oder `Fahrgestell` weiterleiten kann.

Quellcode

```
class Flugzeug {
        Motor           motor;
        Landeklappen    landeklappen;
        Fahrgestell     fahrgestell;

        public Flugzeug () {
                motor        = new Motor();
                landeklappen = new Landeklappen();
                fahrgestell  = new Fahrgestell();
        }

        public void starten () {
                landeklappen.einfahren ();
                motor.anlassen ();
                fahrgestell.einfahren ();
                System.out.println ("Flugzeug: gestartet \n");
        }

        public void landen() {
                fahrgestell.ausfahren ();
                landeklappen.ausfahren ();
                motor.abschalten ();
                System.out.println ("Flugzeug: gelandet \n");
        }
}
```

 © Goll

```
class Motor {
        boolean zustand;

        public Motor (){
                zustand = false;
        }

        public void anlassen(){
                zustand = true;
                System.out.println("Motor: an");
        }

        public void abschalten(){
                zustand = false;
                System.out.println("Motor: aus");
        }
}
```

```
class Landeklappen{
        boolean zustand;

        public Landeklappen(){
                zustand = false;
        }

        public void einfahren (){
                zustand = false;
                System.out.println("Landeklappen: eingefahren");
        }

        public void ausfahren (){
                zustand = true;
                System.out.println("Landeklappen: ausgefahren");
        }
}
```

```
class Fahrgestell{
        boolean zustand;

        public Fahrgestell (){
                zustand = true;
        }

        public void einfahren (){
                zustand = false;
                System.out.println("Fahrgestell: eingefahren");
        }
```

```
        public void ausgefahren (){
                zustand = true;
                System.out.println("Fahrgestell: ausgefahren");
        }
}
```

```
class FlugzeugTest{

        public FlugzeugTest(){
        }

        public static void main(String[] args){

                Flugzeug flugzeug = new Flugzeug ();

                flugzeug.starten ();
                flugzeug.landen ();
        }
}
```

Ausgabe des Programms:

```
Landeklappen: eingefahren
Motor: an
Fahrgestell: eingefahren
Flugzeug: gestartet

Fahrgestell: ausgefahren
Landeklappen: ausgefahren
Motor: aus
Flugzeug: gelandet
```

Bemerkungen

Die Zusammensetzung eines Objekts aus Teilobjekten sollte immer über Aggregation erfolgen. Wenn das Objekt die Funktionalität der Teilobjekte über Mehrfachvererbung erhält, wird die Architektur unflexibel und fehleranfällig. Im Gegensatz zu C++ verbietet Java die Mehrfachvererbung generell. Diese Einschränkung ist ein Hinweis darauf, dass Java die modernere Sprache ist. Sie unterstützt einen flexiblen objektorientierten Entwurf vorbildlich.

2.2 Delegations

Überblick

Das Delegationspattern beschreibt, wie die Funktionalität einer Klasse flexibel erweitert werden kann.

2.2.1 Beispiel für die Realisierung von Rollen mit dem Delegations-Muster

[GRA1998] Auf einem Flughafen können Personen vereinfachend die Rolle eines Passagiers, eines Schalterangestellten und eines Flugbegleiters übernehmen. Auf den ersten Blick scheint die folgende, auf Ableitung basierende Architektur objektorientiert vernünftig zu sein.

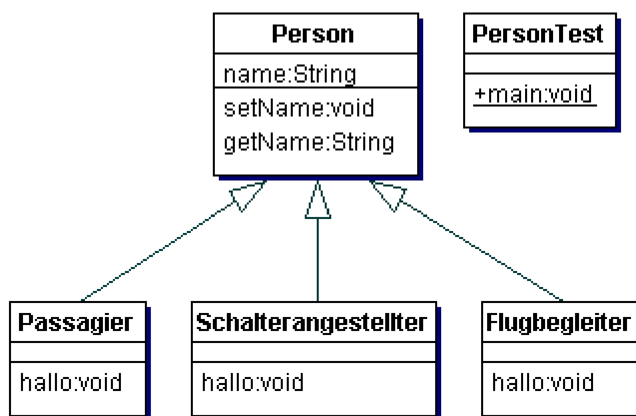


Abbildung 2: Negativbeispiel für übernommene Rollen

Das Klassendiagramm verdeutlicht, dass Objekte vom Typ `Passagier`, `Schalterangestellter` und `Flugbegleiter` alle Eigenschaften ihrer Vaterklasse `Person` erben. Bei dieser Architektur tritt allerdings ein Problem auf, wenn ein `Schalterangestellter` selbst fliegen will. In diesem Fall müsste er eine kombinierte Rolle `PassagierUndSchalterangestellter` einnehmen. Um alle möglichen Kombinationen der drei Rollen abzudecken, sind vier weitere Klassen nötig. Bei vier Rollen wären bereits 11 weitere Klassen nötig.

Ein weitaus ernsteres Problem ergibt sich dadurch, dass eine Person zu verschiedenen Zeiten unterschiedliche Rollen übernehmen und wieder abgeben kann. Dazu wäre die Erzeugung eines Objekts in der neuen Rolle nötig, das den mikroskopischen und den makroskopischen Zustand[3] des bisherigen Objekts übernimmt. Diese Aktion ist in jedem Fall aufwendig, fehlerträchtig und macht das Programm schwerer zu warten. Mit dem Delegations-Pattern lässt sich das Problem elegant umgehen.

[3] Der mikroskopische Zustand eines Objekts wird durch die Gesamtheit seiner Datenfeldwerte beschrieben. Der makroskopische Zustand resultiert aus der Wechselwirkung des Objekts mit seiner Umgebung.

© Goll

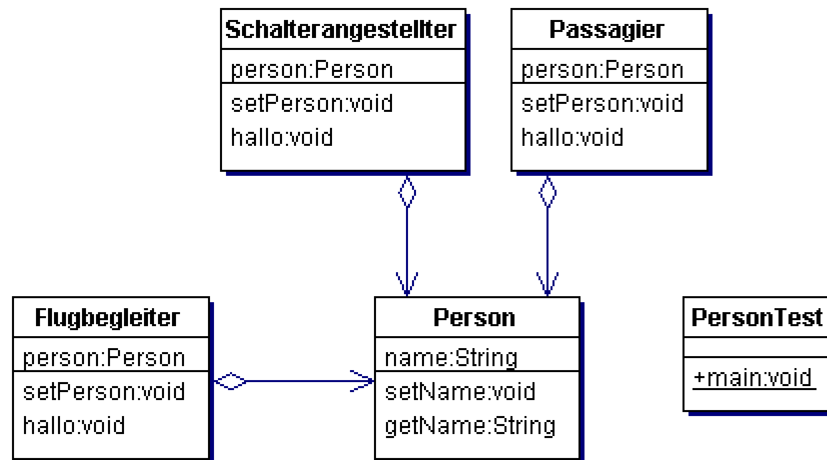


Abbildung 3: Rollen werden mit Hilfe des Delegationsprinzips realisiert

Das Klassendiagramm verdeutlicht, dass das Objekt `Person` in verschiedenen Rollen auftreten kann. Die delegierenden Objekte benötigen lediglich eine Referenz darauf.

Quellcode

```
class Person{
        String name;

        void setName(String name){
                this.name = name;
        }
        String getName(){
                return name;
        }
}
```

```
class Passagier{
        Person person;

        void setPerson(Person person){
                this.person = person;
        }
        void hallo(){
                System.out.println("Ich bin der Passagier " + person.getName());
        }
}
```

```
class Schalterangestellter{
        Person person;

        void setPerson(Person person){
                this.person = person;
```

```
        }
        void hallo(){
                System.out.println("Ich bin der Schalterangestellte " +
person.getName());
        }
}
```

```
class Flugbegleiter{
        Person person;

        void setPerson(Person person){
                this.person = person;
        }
        void hallo(){
                System.out.println("Ich bin der Flugbegleiter " + person.getName());
        }
}
```

```
class PersonTest{

        public static void main(String[] args){

                Person myPerson = new Person();
                myPerson.setName("Frank");

                Passagier myPassagier = new Passagier();
                myPassagier.setPerson(myPerson);
                myPassagier.hallo();

                Flugbegleiter myFlugbegleiter = new Flugbegleiter();
                myFlugbegleiter.setPerson(myPerson);
                myFlugbegleiter.hallo();
        }
}
```

Ausgabe des Programms:

```
Ich bin der Passagier Frank
Ich bin der Flugbegleiter Frank
```

Bemerkungen

Wenn ein Objekt zu verschiedenen Zeiten unterschiedliche Rollen übernehmen kann, ist die Spezialisierung in abgeleiteten Objekten die falsche Vorgehensweise. Durch Delegation kann das ”in Rollen schlüpfen” elegant nachmodelliert werden.
Spezialisierung durch Ableitung ist sinnvoll, wenn ein Objekt unveränderliche Ausprägungen besitzt. So können die Klassen `Mann` oder `Frau` gefahrlos von `Person` abgeleitet werden.

2.2.2 Beispiel für die Vorteile einer Delegation statt Vererbung

Auch Delegation ist eigentlich kein Entwurfsmuster mehr, sondern vielmehr guter objektorientierter Programmierstil. Mit Delegation kann die Funktionalität einer Klasse auf flexiblere Weise wiederverwendet werden als durch Vererbung. Delegation wird ebenso wie Schnittstellen und Varianten in fast jedem Entwurfmuster eingesetzt. Um das Prinzip der Delegation zu verdeutlichen, soll ein Beispiel gezeigt werden.

In einem Informationssystem sollen verschiedene Objekte in mehreren Warteschlangen verwaltet werden. Auf diese Warteschlangen soll von mehreren Threads aus zugegriffen werden. Um das Rad nicht vollständig neu erfinden zu müssen, soll die Klasse `LinkedList` der Java Collection-API wiederverwendet werden. Diese Klasse implementiert eine verkettete Liste aus Objekten der Klasse `Object`. Beim Zugriff auf eine `LinkedList` muss also ein dynamischer Typecast von der Klasse `Object` auf die eigentliche Klasse der enthaltenen Objekte durchgeführt werden. Da diese Vorgehensweise in Java unumgänglich ist[4], sollen die dynamischen Typecasts möglichst alle an einer zentralen Stelle erfolgen, um diese nicht in der gesamten Anwendung zu verstreuen. Des Weiteren stellt sich das Problem der Synchronisierung der gleichzeitigen Zugriffe der verschiedenen Threads ("thread-safety"). Da die Klasse `LinkedList` nicht "thread-safe" ist, muss diese Funktionalität hinzugefügt werden. Es gilt also zwei Probleme zu lösen, die entweder durch Vererbung oder durch Delegation gelöst werden können.

Zuerst soll eine **Lösung mit Vererbung** vorgestellt und deren Vor- und Nachteile erläutert werden. Zuerst soll dabei das Problem der Synchronisierung gelöst werden. Ausgehend von der Klasse `LinkedList` wird dafür eine Subklasse eingeführt, die von `LinkedList` abgeleitet ist. Diese fügt der `LinkedList` die Fähigkeit hinzu, die Zugriffe auf die Liste zu synchronisieren. Folgender Programmcode zeigt die Klasse `SynchronizedObjectQueue`.

```
import java.util.LinkedList;

public class SynchronizedObjectQueue extends LinkedList {

  public synchronized void enqueueObject(Object o) {
    this.addLast(o);
   }

  public synchronized Object dequeueObject() {
    return this.removeFirst();
  }

}
```

Um das Problem der dynamischen Typecasts zu lösen, soll nun eine weitere Klasse eingeführt werden, die eine Warteschlange für Objekte der Klasse `Message` darstellt. Die Zugriffsmethoden dieser Klasse erledigen den dynamischen Typecast

[4] Es existieren keine typisierbaren Klassen wie in C++ (Templates)

an einer zentralen Stelle und erlauben der Anwendung einen typisierten Zugriff auf die Warteschlange. Folgender Programmcode zeigt die Klasse `MessagQueue`.

```
public class MessageQueue extends SynchronizedObjectQueue {

  public void enqueue(Message m) {
     THIS.ENQUEUEOBJECT(M); // IMPLIZITER TYPECAST
    }

  public Message dequeue() {
    return (Message)this.dequeueObject(); // expliziter Typecast
  }

}
```

Folgender Programmcode zeigt, wie eine Anwendung typisiert und synchronisiert auf die Warteschlange zugreifen kann.

```
public class Application {

  public static void main(String[] argv) {
    new Application();
  }

  public Application() {

    MessageQueue mq = new MessageQueue();
    Message m = new Message("It's only gravity...");
    // ...
    mq.enqueue(m);
    // ...
    m = mq.dequeue();
    // ...

  }

}
```

Der Nachteil der Lösung liegt klar auf der Hand. Dadurch, dass die Spezialisierung durch Vererbung durchgeführt wurde, liegen die Schnittstellen aller Superklassen offen und können von der Applikation ebenfalls benutzt werden. Damit könnte die Anwendung z.B. auf die Methode `addLast()` zugreifen, die von der Klasse `LinkedList` geerbt wurde. Somit könnte die Anwendung die Synchronisierung der Zugriffe umgehen.

Im Folgenden soll nun die **Lösung mit Delegation** vorgestellt werden. Die Klasse `SynchronizedObjectQueue` wird nun nicht mehr von der Klasse `LinkedList` abgeleitet, sondern aggregiert lediglich ein Objekt dieser Klasse. Zugriffe auf die Warteschlange werden synchronisiert und an das aggregierte `LinkedList`-Objekt delegiert. Folgender Programmcode zeigt die veränderte Klasse.

© Goll

```
import java.util.LinkedList;

public class SynchronizedObjectQueue {

  private LinkedList queue = new LinkedList();

  public synchronized void enqueueObject(Object o) {
    this.queue.addLast(o); // Delegation
  }

  public synchronized Object dequeueObject() {
    return this.queue.removeFirst(); // Delegation
  }

}
```

Auch die Klasse `MessageQueue` wird angepasst, so dass sie ein Objekt der Klasse `SynchronizedObjectQueue` aggregiert und Zugriffe auf die Warteschlange an dieses Objekt delegiert werden.

```
public class MessageQueue {

  SynchronizedObjectQueue queue = new SynchronizedObjectQueue();

  public void enqueue(Message m) {
    this.queue.enqueueObject(m); // Delegation
  }

  public Message dequeue() {
    return (Message)this.queue.dequeueObject(); // Delegation
  }

}
```

Der Vorteil dieser Lösung ist nun, dass die Klasse `SynchronizedObjectQueue` und die Klasse `MessageQueue` lediglich die von den Klassen selbst implementierte Schnittstelle bereitstellen. Da nicht geerbt wird, können die Methoden der Superklassen nicht aufgerufen werden und die Kapselung ist gewährleistet.

2.3 Interfaces zur Zugriffseinschränkung

Das Interfacepattern beschreibt, wie der Zugriff einer Klasse auf die Instanz einer anderen Klasse eingeschränkt werden kann.

2.3.1 Ein Einführungsbeispiel

In einem Datenobjekt der Klasse `Dokumente` werden Prüfungsdokumente gespeichert. Objekte der Klasse `Student` sollen in der Lage sein, die alte Prüfung zu lesen. Sie dürfen jedoch nicht auf die neue Prüfung zugreifen, die erst geschrieben werden soll. Objekte der Klasse `Professor` können dagegen auf beide Dokumente zugreifen.

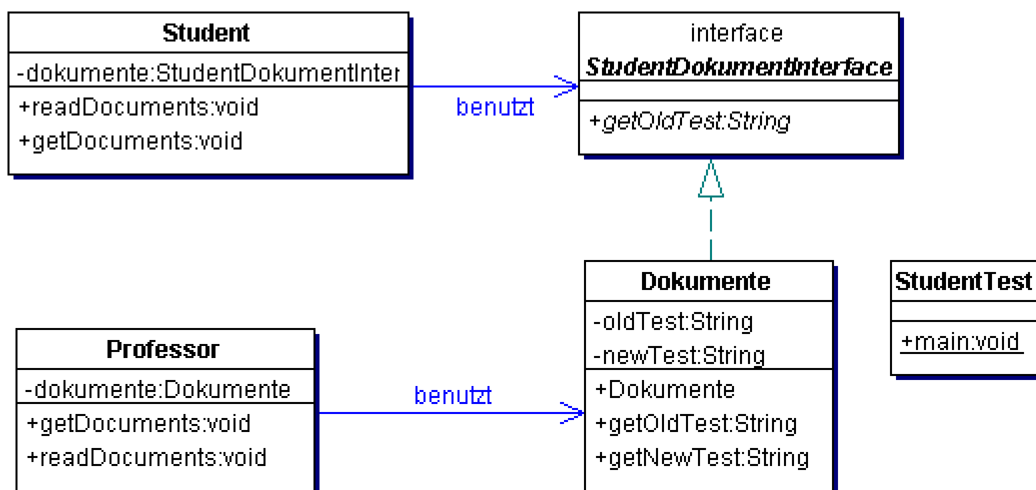


Abbildung 4: Indirekter Zugriff über ein Interface

Das Klassendiagramm veranschaulicht, dass das Objekt der Klasse `Professor` das Objekt `Dokumente` direkt aggregiert[5] und damit auf die Methoden `getOldTest()` und `getNewTest()` zugreifen kann. Ein Objekt der Klasse `Student` besitzt dagegen nur eine Referenz auf das Interface `StudentDokumentInterface`. Da die Dokumentenklasse dieses Interface implementiert, kann sie auch in der Gestalt des Interfaces auftreten[6]. In diesem Fall werden alle Methoden, die nicht von dem Interface "ererbt" wurden, ausgeblendet.

Es ist beachtlich, dass man mit der Referenz auf ein Interface arbeiten kann, obwohl man von dem Interface selbst keine Instanzen erzeugen kann.

[5] Aggregieren = eine Referenz auf ein Objekt als Membervariable enthalten.
[6] Diese Eigenschaft folgt aus dem "Liskov Substitution Principle".

© Goll

Quellcode

```
public interface StudentDokumentInterface {

   public String getOldTest();
} // interface StudentDokumentInterface

class Dokumente implements StudentDokumentInterface {
        private String oldTest;
        private String newTest;

        public Dokumente(){
                oldTest = "4 + 3 * 2 = ?";
                newTest = "2 * 5 + 2 = ?";
        }

        public String getOldTest () {
                return oldTest;
        }

        public String getNewTest () {
                return newTest;
        }
} // class Dokumente
```

```
class Professor{
        private Dokumente dokumente;

        public void setDocuments(Dokumente dokumente){
                this.dokumente = dokumente;
        }

        public void readDocuments(){

                if (dokumente != null){
                        System.out.println ("Professor liest alten Test: " +
dokumente.getOldTest());
                        System.out.println ("Professor liest neuen Test: " +
dokumente.getNewTest());
                }
        }
} // class Professor
```

```
class Student{
        private StudentDokumentInterface dokumente;

        public void setDocuments(StudentDokumentInterface sdi){
                dokumente = sdi;
        }

        public void readDocuments(){

                if (dokumente != null){
                        System.out.println ("Student liest alten Test:   " +
dokumente.getOldTest());
                        // dokumente.getNewTest() führt zu Compilierfehler
        }
} // class Student
```

```
class StudentTest{

        public static void main(String[] args){
                Student student = new Student();
                Professor professor = new Professor();
                Dokumente dokumente = new Dokumente();

                professor.setDocuments(dokumente);
                professor.readDocuments();
                student.setDocuments(dokumente);
                student.readDocuments();
        }
} // class StudentTest
```

Ausgabe des Programms:

```
Professor liest alten Test: 4 * 3 + 2 = ?
Professor liest neuen Test: 2 * 5 + 2 = ?
Student liest alten Test:   4 * 3 + 2 = ?
```

Bemerkungen

Über das Interfacepattern kann gewährleistet werden, dass nur die Objekte auf sensible Methoden eines Serviceobjekts zugreifen dürfen, die dafür vorgesehen sind.

© Goll

2.4 Schnittstellen und Varianten

Dieses Entwurfsmuster ist mittlerweile in die objektorientierte Denkweise übergegangen und kann eigentlich nicht mehr als Entwurfsmuster bezeichnet werden. Es handelt sich hierbei eher schon um guten objektorientierten Stil als um ein Entwurfsmuster. Es wird erwartet, dass auch andere Entwurfsmuster ihren Status verlieren und in die objektorientierte Denkweise oder sogar in objektorientierte Programmiersprachen übergehen.

Die Verwendung von Schnittstellen erlaubt es einer Anwendung, sich nicht direkt auf die konkrete Klasse eines Objekts selbst, sondern auf dessen Schnittstelle zu beziehen. Dies erlaubt es der Anwendung, später ohne Änderungen ein Objekt einer anderen Klasse zu verwenden, die die gleiche Schnittstelle implementiert. Die Anwendung kann also durch die Verwendung von **Schnittstellen** aus mehreren Objekt-**Varianten** auswählen, ohne dabei größere Änderungen vornehmen zu müssen.

Als Beispiel sei ein Messumformer vorgestellt, der mit Hilfe eines Digitalfilters Werte umformt. Es besteht dabei die Möglichkeit, einen Software-Digitalfilter oder einen Hardware-Digitalfilter zu benutzen. Um sich diese Entscheidung offen zu halten, wird ein Interface `DigitalFilter` eingeführt, dass die Schnittstelle zu einem Digitalfilter festlegt. Der Messumformer verwendet im Programmcode nur Referenzen auf das Interface `DigitalFilter` und die beiden Klassen `SoftwareFilter` und `HardwareFilter` implementieren dieses Interface. Folgender Programmcode zeigt die genannten Klassen und Interfaces.

```
public interface DigitalFilter {

  public void putRaw(float value);
  public float getFiltered();

}

public class HardwareFilter implements DigitalFilter {

  public HardwareFilter() {
    }

  public void putRaw(float value) {
    // ...
  }

  public float getFiltered() {
    // ...
  }

}
```

```
public class SoftwareFilter implements DigitalFilter {

  public SoftwareFilter() {
  }

  public void putRaw(float value) {
    // ...
  }

  public float getFiltered() {
    // ...
  }

}

public class Messumformer extends Thread {

  private DigitalFilter filter; // Referenz auf Interface!

  public static void main(String argv[]) {
    new Messumformer();
  }

  public Messumformer() {
    this.filter = new SoftwareFilter();
    this.start();
  }

  public void run() {

    float raw, filtered;

    while(true) {
      // ...
      filter.putRaw(raw);
      filtered = filter.getFiltered();
      // ...
     }

  }

}
```

Im Beispiel ist zu sehen, dass die Klassen `SoftwareFilter` und `HardwareFilter` beide das Interface `DigitalFilter` implementieren. Die Klasse `Messumformer` bezieht sich dabei lediglich auf dieses Interface. Bei der Erzeugung eines konkreten Objekts wird nun entschieden, ob ein Objekt der Klasse `SoftwareFilter` oder ein Objekt der Klasse `HardwareFilter` benutzt wird. Der Rest des Programms muss bei einer Änderung nicht angepasst werden. Die Anwendung wird somit effektiv von der konkreten Klasse der Digitalfilters entkoppelt und erlaubt es, verschiedene Varianten ohne weiteren Aufwand zu verwenden.

© Goll

Schnittstellen und Varianten werden in fast allen anderen Entwurfsmustern in der einen oder anderen Form verwendet. Lediglich die Rollen und Verantwortlichkeiten der Klassen unterschiedet von Entwurfsmuster zu Entwurfsmuster. **Fehler! Verweisquelle konnte nicht gefunden werden.** zeigt das Entwurfsmuster Strategy. Auch hier ist das Schnittstellen und Varianten Muster sehr gut zu erkennen.

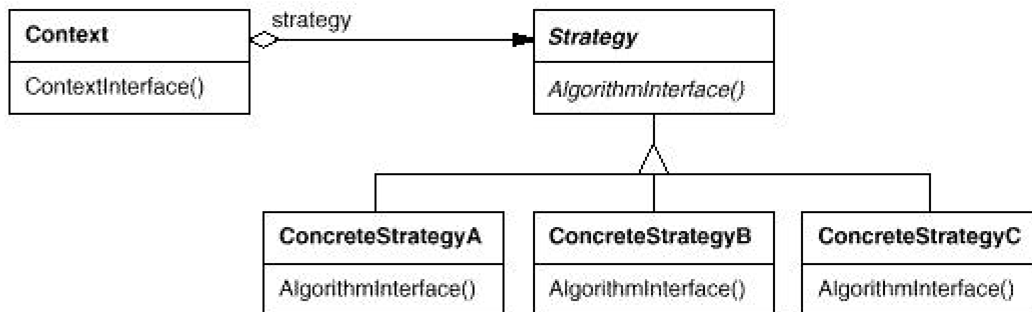


Abbildung 5: Schnittstellen und Varianten im Entwurfsmuster Strategy

Die Klasse `Context` benutzt ein Objekt, dessen Klasse das Interface `Strategy` implementiert. Welche konkrete Klasse dabei benutzt wird, ist in `Context` nicht festgelegt. Die verwendete Klasse kann somit beliebig variiert werden.

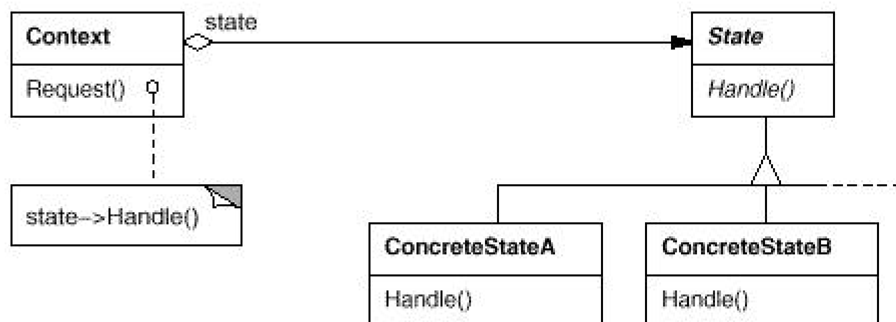


Abbildung 6: Schnittstellen und Varianten im Entwurfsmuster State

Fehler! Verweisquelle konnte nicht gefunden werden. zeigt das Entwurfsmuster State. Die Ähnlichkeit des Klassendiagramms zu **Fehler! Verweisquelle konnte nicht gefunden werden.** ist gut zu erkennen. Lediglich die Rollen und Verantwortlichkeiten unterschieden sich.

© Goll

2.5 Immutable (Unveränderlichkeit)

Das Immutablepattern beschreibt, wie ein Datenobjekt verwendet werden kann, damit seine Datenfeldwerte immer konsistent sind.

Ein Einführungsbeispiel

Die Daten eines Positionsobjekts sind nur als Momentaufnahme sinnvoll. Nachdem ein Thread den x-Wert eines Positionsobjektes gelesen hat, muss er auch noch den zugehörigen y-Wert lesen können. Dabei darf kein neuerer y-Wert ermittelt werden, auch wenn sich die Position des Positionsobjekts während der Abfrage geändert hat. Eine Momentaufnahme kann in einem Positionsobjekt gespeichert werden, das über seinen Konstruktor initialisiert wird, aber keine Methoden besitzt, um die Werte nachträglich zu ändern. Wenn ein Thread eine Referenz auf dieses Positionsobjekt besitzt, so kann er gefahrlos die Daten des gespeicherten Wertepaares nacheinander lesen. Falls sich eine Positionsänderung ergibt, wird ein neues Positionsobjekt erzeugt und entsprechend neu initialisiert. Wenn der Thread das nächste Wertepaar lesen möchte, so erhält er eine Referenz auf das neue Positionsobjekt.

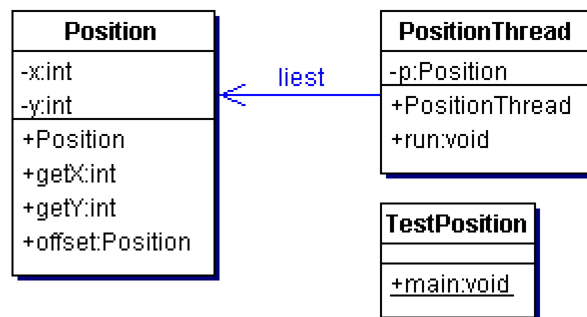


Abbildung 7: Der Thread greift auf eine unveränderbare Instanz zu

Das Klassendiagramm veranschaulicht, dass der Thread `PositionThread` in jedem Fall konsistente Daten von dem Objekt der Klasse `Position` lesen kann, da die Werte von `Position` nicht verändert werden können. Wenn der Thread zu einem späteren Zeitpunkt eine neue Positionsangabe benötigt, muss er sich eine neue Referenz auf das aktuell gültige Positionsobjekt holen.

© Goll

Quellcode

```
class Position {
   private int x;
   private int y;

   public Position(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public int getX() { return x; }

   public int getY() { return y; }

   public Position offset(int xOffset, int yOffset) {
      return new Position(x+xOffset, y+yOffset);
   }
}
```

```
import java.util.*;

class PositionThread implements Runnable {

   private Position p;

   public PositionThread(Position p) {
      this.p = p;
   }

   public void run() {
      System.out.println("Thread: x = "+p.getX());

      try {
         Thread.sleep(100);
      }
      catch (InterruptedException e) {
         System.out.println (e.getMessage());
      }

      System.out.println("Thread: y = "+p.getY());
   }
}
```


© Goll

```
class TestPosition {

   public static void main(String[] args){
      Position p = new Position(7,9);

      System.out.println("Main: x = "+p.getX());
      System.out.println("Main: y = "+p.getY());

      Thread t = new Thread (new PositionThread(p));
      t.start();

      p = p.offset(10,10);     // Referenz auf das neue Positionsobjekt

      try {
         Thread.sleep(100);
      }
      catch (InterruptedException e) {
         System.out.println (e.getMessage());
      }

      System.out.println("Main: x = "+p.getX());
      System.out.println("Main: y = "+p.getY());
   }
}
```

Ausgabe des Programms:

```
Main: x = 7
Main: y = 9
Thread: x = 7
Main: x = 17
Main: y = 19
Thread: y = 9
```

Bemerkungen

Unter Java kann ein konsistenter Zugriff auch über das Monitorkonzept [GOL1999] erreicht werden. Während ein Thread die Daten eines Objekts liest, kann verhindert werden, dass das Objekt von anderer Seite verändert wird. Die Datenänderung wird über das Monitorkonzept blockiert und so lange verzögert, bis der lesende Zugriff beendet wird.

2.6 Schablonenmethode (Template Method)

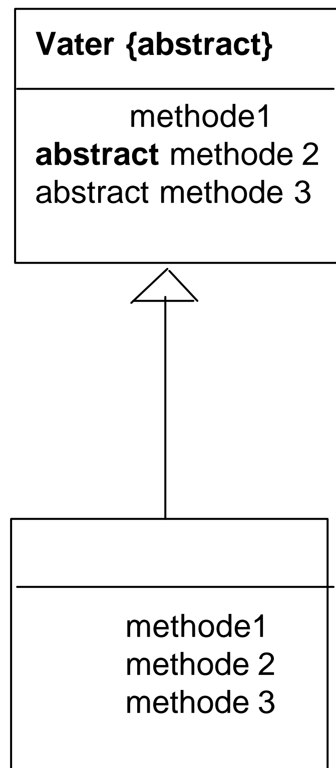


Bild 1

Die Implementierung der methode1 benutzt die abstrakten Methoden der eigenen Klassen

```
methode 1()
{
 methode2()
 methode3()
}
```

In der Sohnklasse werden die abstrakten Methoden der Vaterklasse implementiert.

Die Schablonenmethode versucht, soviel Code wie möglich in einer abstrakten Basisklasse zu spezifizieren und die Verträge dieser Methoden festzulegen!

Die Implementierung einer abstrakten Methode ist in der Vaterklasse noch nicht bekannt, nur der Vertrag! Die Implementierung wird bewußt an die noch nicht existierenden Kinder delegiert.

Diese Methode ist bei allen objektorientierten Frameworks vorzufinden, weil damit ein hohes Maß an Wiederverwendung der Spezifikation gewährleistet wird.

© Goll

+++++
Noch zu integrieren:

Mit Hilfe des Entwurfsmusters Template Method (Schablonenmethode) lässt sich das Gerüst eines Algorithmus in einer Methode implementieren, wobei die einzelnen Schritte des Algorithmus in Einschubmethoden ausgelagert werden. Dadurch wird es Subklassen ermöglicht, bestimmte Schritte des Algorithmus zu überschreiben ohne dessen Struktur zu verändern.

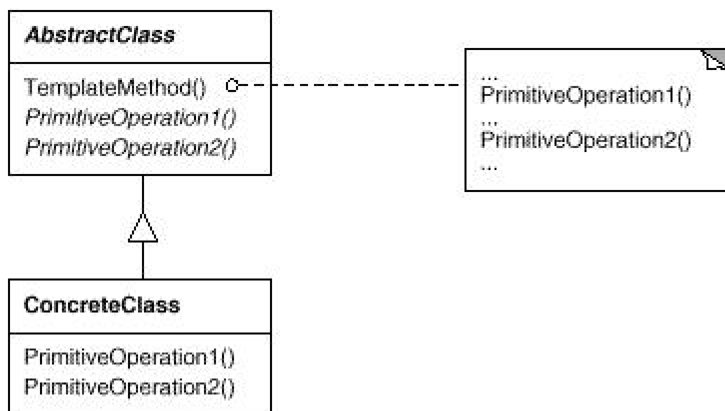


Abbildung 8: Klassendiagramm des Entwurfsmusters Template Method

Fehler! Verweisquelle konnte nicht gefunden werden. zeigt die Klasse `AbstractClass`, deren Methode `TemplateMethod()` die beiden Einschubmethoden `PrimitiveOpeation1()` und `PrimitiveOpeation2()` aufruft. Die beiden Einschubmethoden sind dabei abstrakte Methoden und werden von der Subklasse `ConcreteClass` implementiert.

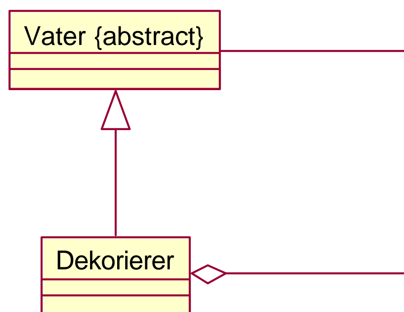
Im Folgenden soll ein Beispiel betrachtet werden. Es sollen verschiedene Klassen zur Sammlung von Objekten implementiert werden. Unabhängig von der verwendeten Datenstruktur (verkettete Liste, Baum, ...) soll festgestellt werden können, ob sich ein bestimmtes Objekt in der Sammlung befindet. Der dazu benötigte Algorithmus kann als Schablonenmethode in einer gemeinsamen Basisklasse definiert werden:

```
public abstract class Collection {

  public boolean contains(Object o) { // Schablonenmethode

    while(hasMoreElements()) {
      if(o.equals(nextElement())) { return true; }
    }

    return false;

  }

  public abstract boolean hasMoreElements(); // Einschubmethode
```

```
    PUBLIC ABSTRACT OBJECT NEXTELEMENT();        // EINSCHUBMETHODE
}
```

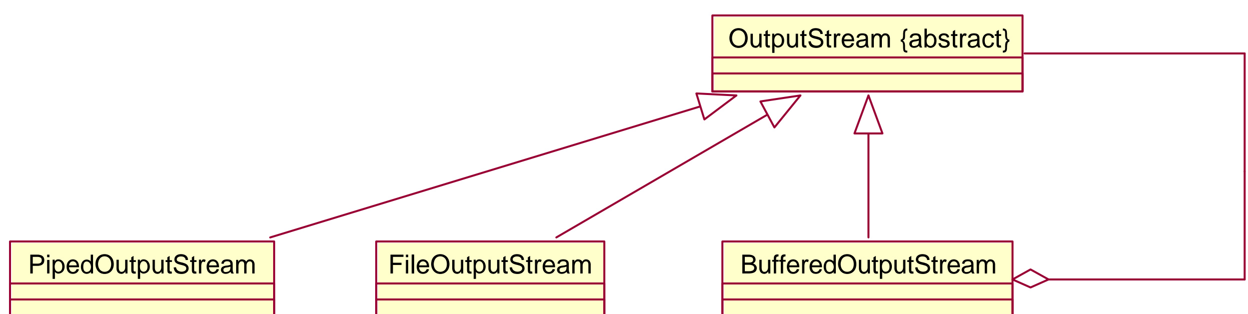
Die Sammlungs-Klassen leiten von dieser abstrakten Basisklasse ab und implementieren die abstrakten Einschubmethoden in Abhängigkeit von der verwendeten Datenstruktur.
++++++

2.7 Dekorierer



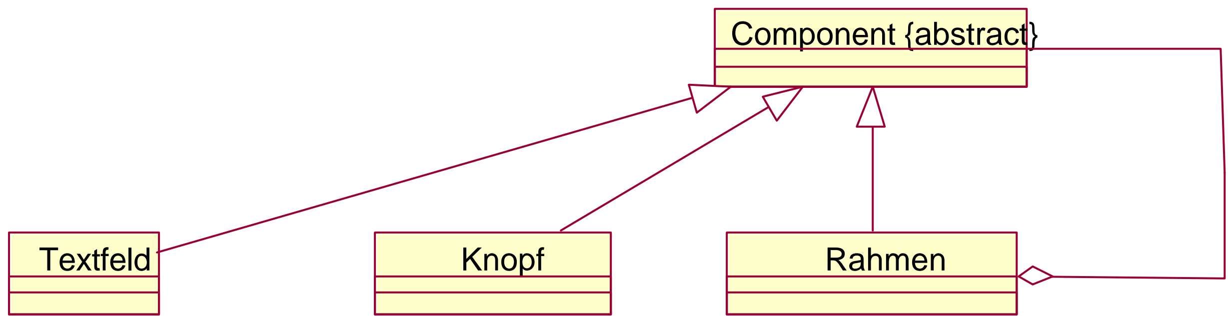
Die Klasse Dekorierer ist von der Klasse Vater abgeleitet und aggregiert gleichzeitig ein Objekt vom Typ Vater. Der Dekorierer kann durch diese Aggregationsbeziehung ein Objekt jeder beliebigen Subklasse von Vater aggregieren.

Beispiel1: Streams



Dekorierer mit der zusätzlichen Funktionalität der Bufferung kann sowohl für einen PipedOutputStream als auch für einen FileOutputStream verwendet werden.

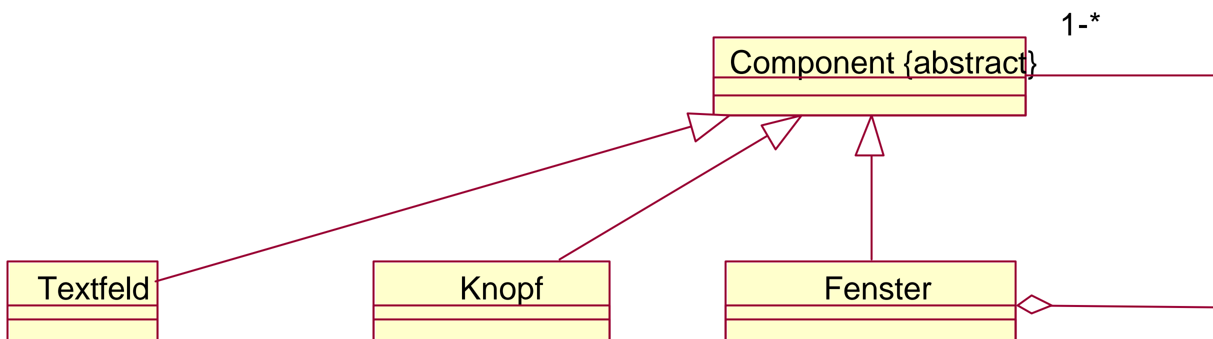
© Goll



Dekorierer dekoriert andere Komponenten mit einem Rahmen.

2.8 Kompositum

Beispiel mit den Klassen `Component`, `Textfeld`, `Knopf`, und `Fenster`



Das Kompositum-Muster verschafft die Möglichkeit, eine Gruppe von Objekten genauso wie ein einzelnes Objekt zu behandeln.

Ein Fenster repräsentiert eine Objektgruppe, die sich aus beliebig vielen Component-Objekten zusammensetzen kann. Wird an ein ein Fenster die Nachricht "vergrößern" geschickt, so vergrößert es sich selbst und schickt die Nachricht an alle aggregierten Compnent-Objekte weiter.

© Goll

2.9 Proxy (Stellvertreter)

Das Proxypattern beschreibt, wie der Zugriff auf ein Objekt kontrolliert werden kann.

2.9.1 Ein Einführungsbeispiel

Auf die Methode `bruchteil()` eines Serviceobjekts der Klasse `Quotient` darf nur zugegriffen werden, wenn der mitgegebene Parameter positiv ist. Um die Komplexität von `Quotient` nicht zu erhöhen, soll die Vorbedingung in einem vorgeschalteten Stellvertreterobjekt der Klasse `Proxy` überprüft werden.

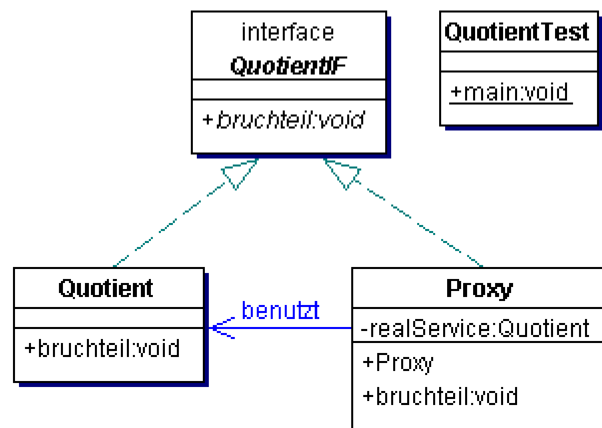


Abbildung 9: Zugriff über ein Proxyobjekt

Das Klassendiagramm veranschaulicht, dass alle Methoden des Interfaces `QuotientIF` sowohl in `Quotient` als auch in `Proxy` implementiert werden müssen. Wenn eine dieser Methoden bei `Proxy` aufgerufen wird, so kann eine Vorbedingung überprüft werden, bevor der Aufruf an das `Quotient`-Objekt delegiert wird.

Die Klasse `QuotientTest` kennt in dem vereinfachten Beispiel sowohl `Proxy` als auch `Quotient`. Damit könnte sie natürlich auch direkt auf `Quotient` zugreifen, ohne die Vorbedingung von `Proxy` überprüfen zu lassen. Normalerweise kennt das Objekt, das den Dienst in Anspruch nehmen darf, nur das Proxyobjekt. Das Proxyobjekt selbst erhält die Referenz auf das Serviceobjekt von anderer Seite.

© Goll

Quellcode

```
public interface QuotientIF {

    void bruchteil(int Zahl);
}
```

```
public class Quotient implements QuotientIF {

    public void bruchteil(int Zahl){
        System.out.println("Quotient: 1 durch " + Zahl + " ist " +          (float)1/Zahl);
      }
}
```

```
public class Proxy implements QuotientIF {

    private Quotient realService;

    public Proxy(Quotient s){
        realService = s;
    }
    public void bruchteil(int Zahl){
        if(Zahl > 0)
            System.out.println("Proxy: Dienst wird genehmigt ...");
            realService.bruchteil(Zahl);
    }

}
```

```
public class QuotientTest {

        public static void main(String[] args){
                Quotient      s = new Quotient();
                Proxy         p = new Proxy(s);

                p.bruchteil(8);
        }
}
```

Ausgabe des Programms:

```
Proxy: Dienst wird genehmigt ...
Quotient: 1 durch 8 ist 0.125
```

Bemerkungen

Das Proxypattern fließt in verschiedenen Ausprägungen in andere Design Patterns ein. Beim Facadepattern[7] genügt der Zugriff auf ein Proxyobjekt, um mit einer komplexen Objektstruktur zu kommunizieren. Ein Proxyobjekt, das auf dem lokalen Rechner des anfordernden Objekts liegt, kann verbergen, dass das eigentliche Serviceobjekt auf einem entfernten Rechner liegt. Dieses Vorgehen vereinfacht die Arbeit in verteilten Systemen. Es wird unter anderem bei RMI (Remote Method Invokation) angewendet.

2.9.2 Das Proxy Muster - ein Strukturmuster

"Das Proxy-Muster führt einen Stellvertreter (engl. Proxy) für eine Komponente ein. Klienten kommunizieren mit dem Stellvertreter statt mit der Komponente selbst. Es gibt verschiedene Gründe, einem Stellvertreter für eine Komponente einzuführen, unter anderen, die Effizienz zu erhöhen, den Zugriff zu vereinfachen und vor unberechtigtem Zugriff zu schützen."

So wird das Proxy-Pattern in [BU98] beschrieben. Aber was verbirgt sich wirklich dahinter? Zunächst einmal läßt sich sagen, dass das Proxy-Pattern in der Kategorie der objektbasierten Strukturmuster angesiedelt ist. Es ist sehr vielseitig einsetzbar und wird auch in nahezu jedem verteilten System verwendet. Das Proxy-Muster bietet eine Vielzahl von möglichen Varianten. Es kann zum Beispiel als ein Stellvertreter für ein Objekt eingesetzt werden, welches sich auf einem entfernten Rechner befindet, um dieses **Objekt lokal** zu **repräsentieren**. Es kann aber auch an Stelle einer Komponente eingesetzt werden, deren volle Funktionalität der Anwendung verborgen bleiben soll, wobei das Proxy-Objekt nur eine **eingeschränkte Schnittstelle** zu dieser Komponente **anbieten** soll. Eine weitere Variante des Proxies ist die **Zugriffskontrolle** der Klienten. Bestimmte Klienten sollen beispielsweise auch nur bestimmte Funktionen einer Komponente anwenden dürfen, während andere Klienten alle Funktionen benutzen dürfen. Hier dient ein Proxy als Stellvertreterobjekt zur Zugriffskontrolle. Die verschiedenen Varianten des Proxy-Patterns werden jedoch im Folgenden noch genauer vorgestellt.

2.9.2.1 Das Proxy-Pattern als Zugriffskontrolle

Im Folgenden wird auf die Variante Proxy als Zugriffskontolle näher eingegangen. Hierzu ein einleitendes Beispiel:

In einem Unternehmen kommt ein Warenwirtschaftssystem zum Einsatz, welches von mehreren Benutzern gleichzeitig bedient wird. Dieses System soll die gängigen Komponenten wie zum Beispiel Kundenverwaltung, Lagerverwaltung, Produktverwaltung, Einkauf und Verkauf zur Verfügung stellen. Die hierfür benötigten Daten werden in einer Datenbank gespeichert. Das System ist als ein verteiltes System realisiert, wobei die Datenbank an einem zentralen Punkt im Netzwerk verfügbar sein soll. Jeder der Anwender greift somit auf diese Datenbank zu, um die von ihm benötigten Informationen abzufragen. Aus der Erfahrung weiß man, dass Datenbankzugriffe bei solchen Systemen überwiegend nur *lesend* geschehen. Das

[7] Das Facadepattern wird im Abschnitt 2.8 besprochen.

Anzeigen von Kunden- oder Produktdaten ändert an den Daten an sich letztendlich nichts. Nur im seltensten Fall wird tatsächlich ein Datensatz geändert. Benötigt ein Anwender einen bestimmten Datensatz, muß er ihn sich somit zunächst einmal von der Datenbank besorgen, was einen Datenbankzugriff zu Folge hat. Dies ist im Prinzip nicht weiter schlimm. Tritt jedoch der Fall auf, dass mehrere Benutzer gleichzeitig Datensätze abrufen wollen, wirkt sich dies sehr schnell negativ auf die Netz-Perfomance aus. Hinzu kommt, dass viele Zugriffe ähnlich oder sogar gleich sein können. Diese Situation soll dahingehend optimiert werden, dass die Zugriffszeiten zu den Datensätzen verringert werden. Die Optimierung soll jedoch nicht auf der Seite des Anwenders vollzogen werden, da diese sonst zu aufwendig werden würde. Die Optimierung soll für die Anwendung selbst weitgehend transparent bleiben.

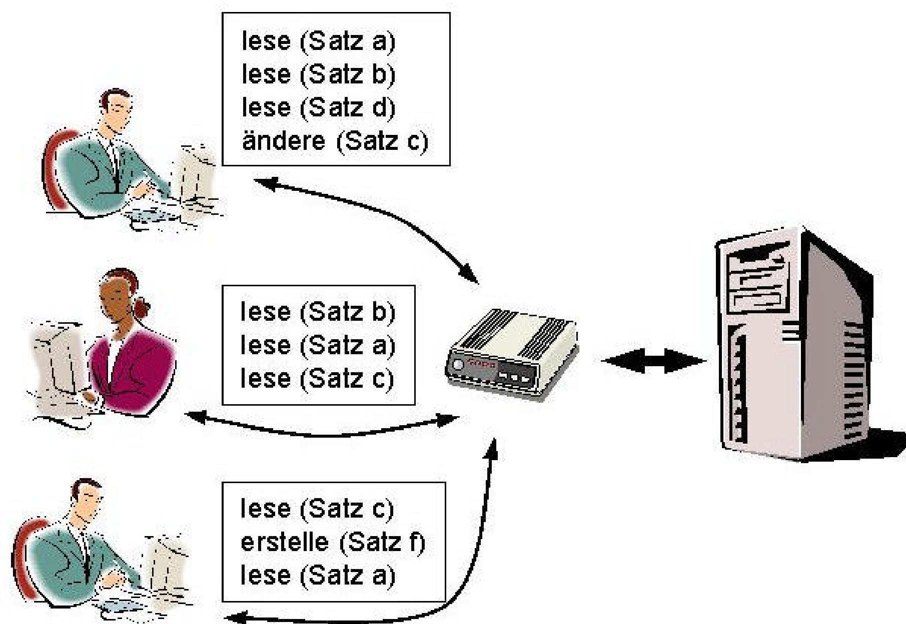


Abbildung 10 – Beispiel aus der Praxis

An dieser Stelle empfiehlt es sich, das Proxy-Pattern anzuwenden. Im Folgenden wird das Pattern anhand der vier Elemente Kontext, Problem, Lösung und Konsequenzen erläutert.

Kontext

Eine Anwendung benötigt Zugriff auf eine bestimmte Komponente. Der Zugriff auf diese Komponente könnte unter Umständen auch direkt erfolgen, ist aber vielleicht nicht der beste Ansatz.

Problem

© Goll

Ein direkter Zugriff auf eine Komponente durch eine Anwendung soll nicht möglich sein. Hierfür kann es verschiedene Gründe geben:

- Die Komponente selbst kann sich auf einem entfernten Rechner befinden. Die Anwendung soll davon jedoch nichts mitbekommen.
- Der Zugriff auf die Komponente muß aus Sicherheitsgründen zunächst autorisiert werden.
- Unnötige Zugriffe auf die Komponente sollen vermieden werden, um Ressourcen zu sparen.

Lösung

Die Anwendung kommuniziert nicht direkt mit der Komponente, sondern über einen Stellvertreter. In diesem steckt dann die ganze Logik, die benötigt wird, um:

- ein entferntes Objekt zu lokalisieren und mit ihm zu kommunizieren,
- eine Authentifikationsprüfung durchzuführen,
- unnötige Zugriffe auf die Komponente zu verhindern und der Anwendung trotzdem ein zufriedenstellendes Ergebnis zu präsentieren.

Konsequenzen

Der Proxy kann auf jedes der drei Probleme speziell angepaßt werden. Er stellt eine zusätzliche Ebene der Indirektion dar.

- Der **Remote-Proxy** verbirgt die Tatsache, dass sich eine Komponente auf einem entfernten Rechner befinden kann.
- Der **Schutz-Proxy** übernimmt zusätzliche Sicherheitsfunktionen, bevor auf die Komponente zugegriffen wird und weist den Aufrufer gegebenenfalls zurück.
- Der **Cache-Proxy** speichert unveränderte Daten einer Komponente zwischen und präsentiert diese den Anwendern.

Unter Umständen kann das Einführen einer zusätzlichen Indirektion auch wieder zu einem Verlust an Perfomance führen. Dies ist in der Regel der Fall, wenn überwiegend schreibende Zugriffe erfolgen oder wenn lediglich ein Anwender die Komponente benötigt.

© Goll

2.9.2.2 Technische Lösung

Die im vorigen Abschnitt besprochene allgemeine Lösung für das Problem, einen direkten Zugriff zu umgehen, wird nun unter einem technischen Aspekt näher beschrieben. Dies beinhaltet eine Darstellung der statischen, sowie der dynamischen Struktur der einzelnen Elemente. Dies wird wieder anhand des Beispiels der Betriebsverwaltung verdeutlicht.

Eine Problemstellung dabei könnte folgendermaßen aufgebaut sein:

Es soll ein zentraler Punkt existieren, an dem die Daten von allen Kunden verwaltet werden. Diese Daten stehen üblicherweise in einer Datenbank, welche bereits vorhanden sein soll. Das Datenbank-Management-System (**DBMS**) stellt somit kein elementares Problem dar. In einer solchen Betriebsverwaltung arbeiten jedoch häufig mehrere Endbenutzer (**Clients**) an identischen Tabellen oder sogar Datensätzen. Wie schon angedeutet, wäre technisch gesehen ein direkter Zugriff der Endbenutzer auf die Kundendaten der Datenbank realisierbar. Es stellt sich jedoch die Frage, ob diese Vorgehensweise effizient und vor allen Dingen auch sicher ist. Tritt beispielsweise der Fall auf, dass zwei oder mehrere Clients gleichzeitig denselben Datensatz direkt bearbeiten wollen, so würden lediglich die zuletzt durchgeführten Änderungen übernommen werden, beziehungsweise die zuerst durchgeführten Änderungen von dem zweiten Client überschrieben werden. Ein Direktzugriff auf die Datenbank wäre auch nicht sehr effizient, wenn man bedenkt, dass bei jedem Lesevorgang eine Verbindung zur Datenbank hergestellt werden muß und Daten aus der Datenbank gelesen werden müssen, die unter Umständen schon mehrere Male von anderen Clients benötigt wurden und sich unterdessen auch nicht geändert haben. Auf den ersten Blick gesehen mag der Aspekt der Effizienz gerne vernachlässigt werden, wenn man sich ein System vor Augen hält, das lediglich aus zwei oder drei Clients besteht, wie es in Abbildung 10 dargestellt wird. In einem realen System kommen jedoch teilweise mehrere hundert Clients gleichzeitig zum Einsatz und dann gilt es, die Zahl der Datenbankzugriffe möglichst klein zu halten, da die Datenbank den Flaschenhals in dem System darstellt.

Um das Beispiel zu vereinfachen, wird an Stelle der gesamten Datenbank lediglich ein Objekt betrachtet, das einen Teil der Datenbankfunktionen zur Verfügung stellt. Dieses Objekt stellt zunächst einmal zwei elementare Funktionen zur Verfügung:

- Einen Datensatz mit Kundendaten in die Datenbank zu schreiben
- Einen bestimmten Kundendatensatz aus der Datenbank zu lesen

Diese beiden Funktionen veranlassen einen direkten Zugriff zur Datenbank. Hierfür wird jetzt ein Proxy-Objekt erstellt, welches nach Außen hin dieselben Funktionen zur Verfügung stellt.

Im Falle eines lesenden Zugriffes prüft der Proxy jedoch erst, ob der Kunde bereits ausgelesen wurde. An diesem Punkt angelangt, gibt es zwei unterschiedliche weitere Abläufe:

A) Der Kunde wurde noch nicht aus der Datenbank gelesen:
Der Proxy leitet die Anfrage an die Datenbank weiter, speichert das Resultat dieser Anfrage zwischen und gibt es ebenfalls an den Client weiter.

B) Der Kunde wurde schon einmal aus der Datenbank gelesen:
Das Proxy-Objekt leitet die zwischengespeicherten Daten an den Client direkt weiter und verzichtet auf einen Zugriff auf die Datenbank.

Im Falle eines schreibenden Zugriffes prüft der Proxy zunächst, ob die Daten augenblicklich von einem zweiten Client verändert werden. Sollte dies der Fall sein, muß einer der beiden Schreibvorgänge verweigert werden.

Dieses Beispiel wurde sehr stark vereinfacht, um es noch anschaulich zu halten. In reellen System steckt hinter diesem einfachen Prinzip jedoch wesentlich mehr als an dieser Stelle beschrieben wird. Abbildung 11 veranschaulicht dieses Prinzip.

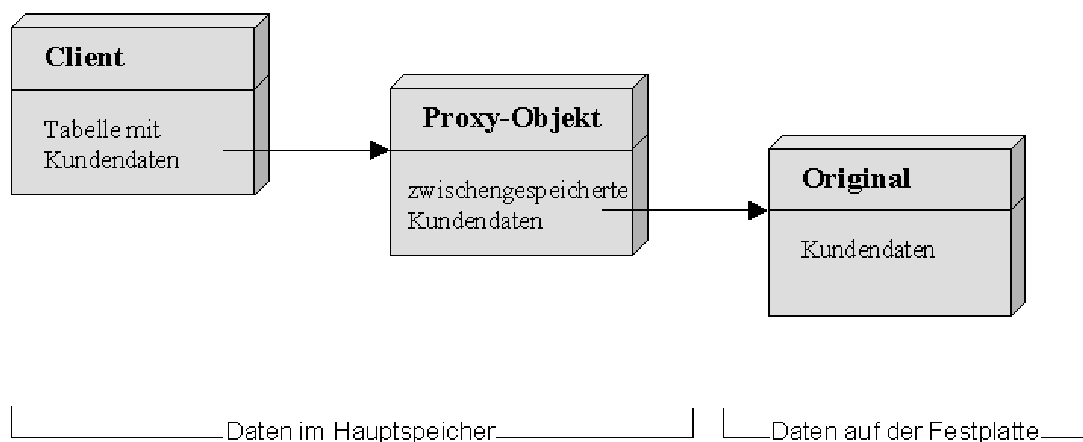


Abbildung 11 – Client und Proxy im gleichen Hauptspeicher

© Goll

Der Stellvertreter besitzt dabei dieselben Schnittstellen wie das Orginal und stellt einen korrekten Zugriff auf das Orginal sicher. Es wird somit notwendig, dass der Stellvertreter eine Referenz auf das Orginal besitzt. Die Beziehung zwischen Orginal und Proxy-Objekt ist meist eine Eins-zu-Eins-Beziehung. Dies bedeutet, dass ein Stellvertreterobjekt nur die Referenz eines bestimmten Orginales gespeichert hat. Ebenso wird ein Orginal-Objekt auch nur von einem einzigen Proxy-Objekt benutzt. Jedoch keine Regel ohne Ausnahmen. Das Proxy-Muster bietet eine Vielzahl von Varianten, wie zum Beispiel den Firewall-Proxy oder den Remote-Proxy. Bei diesen Varianten reicht eine Eins-zu-Eins-Beziehung nicht aus. Diese und auch noch andere Varianten werden im späteren Verlauf erläutert.

In der objektorientierten Modellierung werden Zusammenhänge zwischen Klassen oft mit Hilfe von CRC-Karten dargestellt. Diese Karten veranschaulichen sowohl die eigenen Verantwortlichkeiten der Klassen, sowie die Kooperation mit den anderen Klassen. Abbildung 12 zeigt die CRC-Karten für das Proxy-Muster.

Class *Client*	**Collaborators**
Responsibilities • Benutzt das vom Proxy bereitgestellte Interface für bestimmte Service • Führt zusätzlich seine eigenen Aufgaben aus	• Proxy

Class *AbstraktesOrginal*	**Collaborators**
Responsibilities • Abstrakte Basisklasse für Proxy und Orginal	-

Class *Proxy*	**Collaborators**
Responsibilities • Vertritt das Interface des Orginals für alle Clients • Realisiert einen schnellen und sicheren Zugriff auf das Orginal	-

Class *Orginal*	**Collaborators**
Responsibilities • Implementiert einen bestimmten Service	-

Abbildung 12 – CRC-Karten für einen Proxy

++++
UNDEFINED: Fehler in Bild: Service statt Services
++++

Das Klassendiagramm in Abbildung 13 zeigt noch einmal die Zusammenhänge zwischen den Klassen. Anhand des Klassendiagramms läßt sich erkennen, dass

© Goll

Proxy sowie ***Original*** von der Klasse ***AbstraktesOriginal*** abgeleitet werden. Beide Klassen erben die Schnittstellen des abstrakten Originals, in diesem Fall *funktion_1* und *funktion_2*. Die Methoden sind im abstrakten Original noch nicht implementiert worden. Eine Implementierung erfolgt erst in den abgeleiteten Klassen. Dadurch läßt sich eine unterschiedliche Funktionalität zwischen Proxy und Original realisieren, während die Schnittstellen nach Außen hin identisch sind.

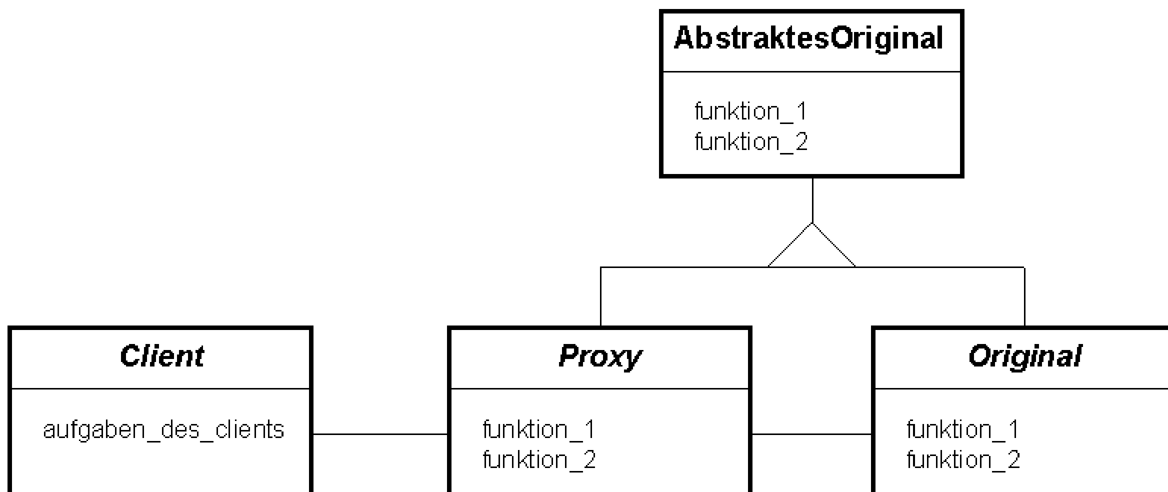


Abbildung 13 – Klassendiagramm eines Proxies

© Goll

Dynamische Sicht

Abbildung 14 zeigt ein dynamisches Szenario. Dieser Ablauf ist charakteristisch für eine Proxy-Struktur.

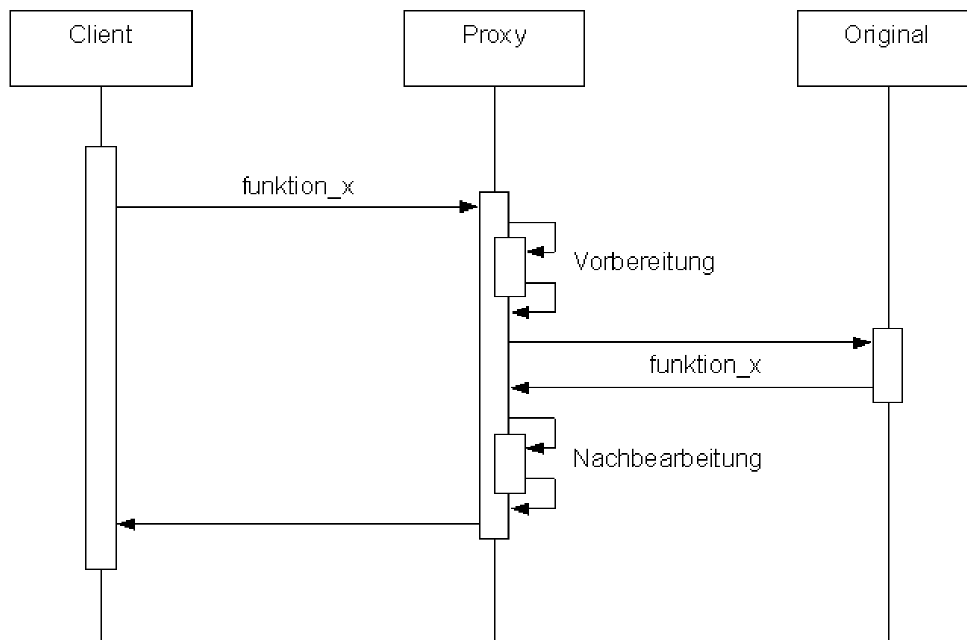


Abbildung 14 – Sequenzdiagramm eines Proxies

Vorbereitung und Nachbereitung können je nach Proxy-Variante unterschiedlich ausfallen. Der Schutzproxy zum Beispiel wird in der Vorbereitungsphase prüfen, ob der Client die nötige Berechtigung hat, um Methoden des Originals auszuführen. Der Cacheproxy hingegen hat während der Vorbereitung eine völlig unterschiedliche Aufgabe. Er würde prüfen, ob es überhaupt notwendig ist, die Methode des Originals aufzurufen oder ob er die vom Client angeforderten Daten aus seinem eigenem Speicher entnehmen kann. In diesem Fall würde auch der Aufruf beim Original entfallen.

Diese einfache Grundvariante des Proxies läßt sich anhand von fünf nacheinander ablaufenden Schritten erklären:

- Zu einem beliebigen Zeitpunkt wird der Proxy vom Client beauftragt, die Funktion *funktion_x* auszuführen. Der Client wartet auf eine Antwort vom Proxy. Vor- und Nachbearbeitung, sowie der eigentliche Aufruf des Originals bleiben dem Client jedoch verborgen.

 © Goll

- Der Proxy empfängt die Dienstanforderung und führt eine Vorbereitung durch. Dies kann – wie schon angesprochen – von Variante zu Variante unterschiedlich ausfallen. Eine weitere Möglichkeit für die Vorbereitung, welche nicht von der verwendeten Variante des Proxies abhängt, ist, ist, dass eine Prüfung geschieht, ob die Referenz auf das Original noch Gültigkeit besitzt. Das Original könnte in der Zwischenzeit beendet und wieder neu gestartet worden sein oder es befindet sich an einer anderen Stelle. In diesem Fall müßte sich der Proxy eine neue Referenz auf das Original besorgen.

- Im dritten Schritt nimmt der Proxy den Dienst des Originals in Anspruch. Dies geschieht jedoch nur, falls das Resultat der Vorbereitung dies zuläßt. Sollte die Vorbereitung ergeben haben, dass ein Aufruf nicht nötig ist oder aus Sicherheitsgründen nicht durchgeführt werden darf, entfällt dieser.

- Das Original empfängt die Anforderung und führt den Dienst aus. Das Original selbst führt somit keine Sicherheitsprüfung mehr durch. Das Ergebnis des Aufrufes wird direkt an das Stellvertreterobjekt zurückgegeben.

- Der Proxy empfängt die Antwort und führt anschließend eine Nachbearbeitung durch. Im Falle des Cache-Proxies fällt die Nachbearbeitung insofern aus, dass die neu empfangenen Daten in den internen Speicher des Stellvertreters mit aufgenommen werden, so dass bei einem erneuten Aufruf die Daten direkt aus dem Speicher genommen werden können und auf einen Funktionsaufruf beim Original verzichtet werden kann. Eine Aufgabe für die Nachbearbeitung des Schutzproxies hingegen könnte das Protokollieren des Zugriffes sein. Auf diese Weise könnte der Vorgang zu einem späteren Zeitpunkt nachvollzogen werden.

2.9.2.3 Implementierung

Die Implementierung des Proxy-Musters kann entlang der folgenden 6 Schritte durchgeführt werden:

- ***Verantwortungsbereich identifizieren***: Dies bedeutet, dass sämtliche Zusammenhänge, welche die Zugriffskontrolle (wie zum Beispiel Performancesteigerung und Sicherheitsabfragen) des Originals betreffen, extrahiert und dem Stellvertreter zugewiesen werden.

- ***Abstrakte Basisklasse einführen***: Soweit gemeinsame Anteile existieren, was die Schnittstelle des Proxies und des Originals betrifft, werden diese in einer abstrakten Basisklasse bereits definiert. Proxy und Original erben von dieser Klasse. Durch die Bereitstellung identischer Schnittstellen wirkt der Stellvertreter nach außen hin wie das Original selbst und verbirgt seine zusätzliche Funktionalität.

- ***Funktionen des Stellvertreters implementieren***: Diese Funktionen sind im Wesentlichen dieselben, welche in Schritt eins gefunden wurden.

- ***Original und Klienten von Verantwortlichkeiten befreien***: Die Funktionen, die in Schritt drei dem Stellvertreter übertragen wurden, können jetzt beim Original oder beim Client entfernt werden. Funktionen der Zugriffskontrolle werden meist beim Original zu finden sein, während Cache-Funktionen, sollten sie soweit schon implementiert gewesen sein, eher beim Client anzutreffen sind.

- ***Original und Proxy miteinander in Verbindung bringen***: Im Wesentlichen geht es bei diesem Schritt nur darum, dass der Proxy eine Referenz auf das Original zugewiesen bekommt. Bei verteilten Systemen kann dies zum Beispiel mit Hilfe von RMI oder CORBA geschehen. Wichtig dabei ist nur, dass der Stellvertreter eine Möglichkeit kennt, den Service, den das Original anbietet, auch in Anspruch zu nehmen.

- ***Klienten vom Original lösen***: Im letzten Schritt wird lediglich sicher gestellt, dass zukünftig keine direkte Verbindung zwischen Client und Original mehr besteht. Sämtliche direkte Verbindungen werden durch Client-Proxy-Verbindungen ersetzt.

© Goll

2.9.3 Ein erstes Codebeispiel

In diesem Kapitel wird eine einfache Anwendung gezeigt, welche das Proxy-Pattern implementiert. Die Anwendung wird in Java implementiert. Das erste Ziel dieses Prototypen ist es, den Aufruf des Originals vorzubereiten, und den vom Original zurückgelieferten Wert durch den Stellvertreter nachzubearbeiten.

Zunächst wird eine abstrakte Basisklasse definiert, welche die gemeinsame Schnittstelle des Originals und des Stellvertreters enthält. Die Objekte des Prototypen sollen auf mehrere virtuelle Maschinen verteilt werden können. Den benötigten Mechanismus stellt das **Java Development Kit** (JDK) in Form von **Remote Method Invocation** (RMI) zur Verfügung. RMI wurde so entworfen, dass Methodenaufrufe völlig transparent über Netzwerkgrenzen – und somit über Rechner- bzw. Virtual-Machine-Grenzen – hinweg zwischen Java-Objekten realisierbar sind. Transparent bedeutet hierbei, dass sich der Entwickler nicht mit den Details des Verbindungsaufbaus zwischen Rechnern oder der Konvertierung der zu übertragenen Daten in ein für den Transport über ein Netz geeignetes Format auseinander setzen muß, wie es zum Beispiel bei Socket-Programmierung nötig ist. Auf die Einzelheiten der RMI-Schnittstelle wird hier jedoch nicht weiter eingegangen werden. Relevante Codeteile, was das Proxy-Pattern betrifft, sind in den folgenden Beispielen jeweils fett unterlegt.

```
/*
 * AbstractOriginal.java
*/

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

abstract class AbstractOriginal
extends UnicastRemoteObject
{

  /** Konstruktor AbstractOriginal */
  public AbstractOriginal()
     throws RemoteException
  {
  }

  /** Gemeinsame Schnittstelle **/
  public abstract String service()
     throws RemoteException;
}
```

Die gemeinsame Schnittstelle `service()` ist somit für Original und Stellvertreter definiert. Sie verlangt keine Übergabeparameter und liefert einen Wert vom Java-Datentyp `String` zurück. Als nächstes folgt die Implementierung des Stellvertreters.

© Goll

```
/*
 * ServerProxy.java
*/
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;

public class ServerProxy extends AbstractOriginal {

    RemoteInterface original;

     /** Konstruktor ServerProxy */
     public ServerProxy()
     throws RemoteException {
     super();
     try
          {
          // Referenz auf Original-Objekt
          // besorgen (RMI)
          original =
          (RemoteInterface)Naming.lookup("rmi:///Original");
          }
          catch (Exception e) {
               System.out.println("Fehler !");
         }
    }

     public String service()
     throws RemoteException {

     String wert;
     try
          {
          // Vorbereitung
          wert = new String("Vorbereitung + ");
          // ...
          // Eigentlicher Funktionsaufruf beim Original
          wert = wert.concat(original.service());
          // Nachbearbeitung
          wert = wert.concat(" + Nachbearbeitung");
          // ...
          }
          catch (Exception e) {
               System.out.println("Fehler !");
          }
     return wert;
    }
}
```

Vorbereitung und Nachbearbeitung können von Variante zu Variante unterschiedlich ausfallen.

Das Original wird somit von sämtlichen Verantwortlichkeiten der Vor- und Nachbearbeitung befreit und liefert lediglich die von ihm gewünschte Antwort zurück.

```
/*
 * Original.java
*/

import java.rmi.*;
import java.io.*;

class Original extends AbstractOriginal {

     /** Konstruktor Original */
     public Original()
     throws RemoteException {
     }

     public String service()
     throws RemoteException {

          return new String("Antwort vom Original");
     }

     public static void main (String args[]) {
     try {
          // Ein neues Original-Objekt erzeugen
          Original original = new Original();

          // Original-Objekt über RMI
          // zugänglich machen
          Naming.rebind("rmi:///Original",original);

          while (System.in.read()!='s');
          System.exit(0);
          }
          catch (Exception e) {
               System.out.println("Fehler !");
          }
     }
}
```

Alles, was jetzt noch fehlt, ist die eigentliche Anwendung selbst.

```
/*
 * ClientAppl.java
*/

import java.rmi.*;
import java.util.*;

public class ClientAppl {

	// Referenz auf das Proxy-Objekt
	private ServerProxy server;

	/** Konstruktor ClientAppl */
	public ClientAppl() {
	}


	public static void main (String args[]) {

		// Neuen Proxy erzeugen
		server = new ServerProxy();
		// Proxy-Service abrufen
		String wert = new String(server.service());
		System.out.println(wert);

		}
}
```

Der Vollständigkeit halber sollte an dieser Stelle noch erwähnt werden, dass die abstrakte Basisklasse noch mit dem Kommando `rmic -v1.2 AbstractOriginal` übersetzt werden muß, damit ein Server-Stub generiert wird. Dies sind jedoch Einzelheiten, die lediglich durch die Verwendung von RMI erforderlich werden. Hier soll nur das Prinzip eines Proxy-Patterns verdeutlicht werden.

Damit ist eine erste Anwendung erstellt, welche das Proxy-Pattern implementiert. Ziel dieser Anwendung ist es, die Möglichkeiten aufzuzeigen, welche der Stellvertreter hat, einen Funktionsaufruf beim Original vorzubereiten, und das Ergebnis anschließend nachzubearbeiten. An dieser Stelle kann jede denkbare Aufgabe realisiert werden, welche eine Anwendung erfordert.

Der Aufruf beim Client würde im obigen Beispiel folgenden Rückgabewert geliefert bekommen:

Vorbereitung + Antwort vom Original + Nachbearbeitung

Dem Klienten bleibt die zusätzliche Aufbereitung des Ergebnisses durch den Proxy jedoch verborgen.

2.9.4 Varianten

Im Folgenden werden sieben Varianten des Proxy-Musters beschrieben. Die letzte bescBhriebene Variante, der Cache-Proxy, wird anschließend wieder anhand eines Codebeispiels erläutert.

2.9.4.1 Virtueller Proxy

Der virtuelle Proxy dient dazu, das Erzeugen ***teurer*** Objekte zu kontrollieren. Dies sind Objekte, die sehr viel Ressourcen benötigen, um erstellt oder geladen zu werden. Dies kann der Fall sein, wenn das zu referenzierende Objekt von der Festplatte oder über ein Netzwerk geladen werden muß. Der Stellvertreter regelt den Ladevorgang des Objektes. Dabei entscheidet er selbst, ob er das Objekt vollständig, nur teilweise oder überhaupt nicht lädt. Diese Anwendung macht vor allem bei größeren Dokumenten oder Internet-Seiten einen Sinn.

Betrachtet man einmal das Beispiel einer HTML-Seite im Internet. Wenn auf dieser Seite mehrere Bilder zu finden sind, macht es keinen Sinn, diese Bilder alle auf einmal zu laden. Es reicht aus, wenn die Bilder erst geladen werden, wenn sie sich in dem momentan angezeigten Bereich befinden. Internet-Seiten erstrecken sich oft über mehrere Bildschirme. Den Betrachter interessiert aber unter Umständen nur der Bereich, den er im Moment angezeigt bekommt. In diesem befindet sich beispielsweise der Link, der ihn zu einer anderen Seite bringen soll. Der virtuelle Proxy würde somit teure Objekte, welches Bilder in den meisten Fällen sind, erst dann laden, wenn der Betrachter seinen Anzeigebereich dementsprechend verändert.

Diese so gewonnene Performancesteigerung würde jedoch einige Nachteile mit sich bringen, wenn der Proxy lediglich entscheiden könnte, ob er Objekte nur ganz oder gar nicht laden kann. Um bei dem Beispiel der Internet-Seite zu bleiben: Würde das Objekt überhaupt nicht geladen werden, könnte es zu Fehlern in der Anzeige kommen, da die Anwendung – in diesem Fall der Internet-Browser – zumindest Informationen über Abmessungen des Bildes benötigt. Bekommt der Browser diese Informationen nicht geliefert, würde die Seite an manchen Stellen eventuell etwas verschoben wirken.

Dieses Verfahren kann generell bei jeglicher Form von Betrachter-Applikationen angewandt werden. Jedesmal, wenn es dem Betrachter ausreicht, zunächst nur einen Teil der Informationen zur Verfügung gestellt zu bekommen und er die gesamten Informationen eines Objektes erst zu einem späteren Zeitpunkt benötigt, kann der virtuelle Proxy angewandt werden.

© Goll

2.9.4.2 Schutz-Proxy

Der Schutzproxy kontrolliert den Zugriff auf das Originalobjekt. Er wird angewandt, wenn Objekte über unterschiedliche Zugriffsrechte verfügen sollen. Der Schutzproxy überprüft, ob der Aufrufer die zum Ausführen der vom Original angebotenen Services notwendigen Zugriffsrechte besitzt.

Bei der Implementierung des Schutzproxies ist darauf zu achten, dass der Stellvertreter auch eine Referenz auf den Klienten besitzt. Bei dieser Variante wird eine Instanz des Proxies meist von mehreren Anwendungen benutzt. Es entsteht somit eine sogenannte eins-zu-n-Beziehung. Die Referenz auf den Klienten muß systemweit eindeutig sein. Jedem Klienten wird dabei eine eigene Identifizierungsnummer (ID) zugeordnet. Um die Sicherheit zu erhöhen, empfiehlt es sich hierbei, die ID mit einem Paßwort in Verbindung zu bringen. Eine Möglichkeit der Realisierung beinhaltet, dass die Zugriffskontrollmechanismen, welche bereits das Betriebssystem liefert, genutzt werden. Manchmal jedoch bieten diese nicht die gewünschte Sicherheit oder Funktionalität. In diesem Fall muß die Zugriffskontrolle teilweise oder sogar vollständig neu implementiert werden. Der Entwickler muß dabei selbst abschätzen können, ob er eine Neuimplementierung der Kontrollmechanismen für nötig hält.

2.9.4.3 Synchronisierungs-Proxy

Der Synchronisierungs-Proxy wird angewandt, wenn mehrere gleichzeitige Zugriffe auf eine Komponente gegenseitig synchronisiert werden müssen. Es kann in bestimmten Fällen notwendig sein, dass ein bestimmtes Objekt von nie mehr als einem Klienten gleichzeitig bearbeitet werden darf. Beispiel hierfür sind Datensätze in einer Datenbank. Wenn mehr als ein Klient gleichzeitig versucht, einen bestimmten Datensatz zu ändern, werden die Daten früher oder später in einen inkonsistenten Zustand verfallen. Es muß somit ein Mechanismus vorhanden sein, der ein gleichzeitiges Bearbeiten gleicher Datensätze konsequent untersagt. Realisiert wird der Proxy im Allgemeinen durch die Verwendung von Semaphoren.

Es kann auch zwischen schreibenden und einem lesenden Zugriff unterschieden werden. Wenn – um bei dem Beispiel der Datenbank zu bleiben – ein Datensatz momentan verändert wird, ist es in der Tat ungünstig, wenn im selben Augenblick noch ein zweiter Klient versucht, diesen Datensatz zu ändern. Es spricht jedoch nichts dagegen, dass er den Datensatz lediglich mit lesendem Zugriff für sich in Anspruch nimmt. Es sollte jedoch darauf geachtet werden, dass sich die ihm gebotene Anzeige auch tatsächlich aktualisiert, wenn der Datensatz letztendlich geändert wird.

2.9.4.4 Remote-Proxy

Der Remote-Proxy stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adreßraum dar. Er kapselt die Informationen darüber, wo sein Original-Objekt konkret liegt. Das vorige Codebeispiel implementiert einen solchen Remote-Proxy. Das Beispiel wurde für ein verteiltes System entwickelt. Dies bedeutet, dass sich das Original-Objekt innerhalb einer anderen virtuellen Maschine befindet, welche sich nicht einmal auf demselben Rechner befinden muß wie die Client-Applikation. Die Kommunikation mit dem Original-Objekt regelt das Stellvertreter-Objekt, was wiederum für die Anwendung transparent geschieht. Der Proxy regelt die Kommunikation über RMI und baut selbständig die Verbindung zum Original auf.

Intelligente Systeme können hierbei zwischen drei Fällen unterscheiden:

- *Anwendung und Original befinden sich in demselben Prozeß*: In diesem Fall kann auf den Remote-Proxy verzichtet werden, da Anwendung und Original direkt miteinander kommunizieren können.

- *Anwendung und Original befinden sich in unterschiedlichen Prozessen aber auf demselben Rechner*: Der Proxy benötigt fortan eine Referenz auf das Original-Objekt in dem anderen Adreßraum.

- *Anwendung und Original befinden sich in unterschiedlichen Prozessen und auf unterschiedlichen Rechnern*: Der Proxy benötigt fortan eine Referenz auf das Original-Objekt in dem anderen Adreßraum sowie eine Referenz auf den anderen Rechner.

Werden moderne Systeme zur Kommunikation eingesetzt, wie zum Beispiel RMI oder CORBA, unterscheiden sich die beiden letzten Fälle nicht mehr voneinander. Referenzen, die von CORBA übergeben werden, beziehen sich automatisch immer auf ein Objekt in einem festgelegten Adreßraum und auf einem festgelegten Rechner. Es gibt jedoch auch IPC-Mechanismen, bei denen es einen Unterschied macht, ob sich das zu referenzierende Objekt auf dem lokalen oder einem entfernten Rechner befindet.

2.9.4.5 Firewall-Proxy

Der Firewall-Proxy wird eingesetzt, wenn lokale Klienten vor dem Zugriff von außen geschützt werden sollen. Er wird meist in Verbindung mit Internet-Anwendungen eingesetzt und läuft in einem Hintergrundprozeß (engl. Deamon) ab. Der Proxy prüft ein- und ausgehende Pakete auf eine Einhaltung der netzinternen Sicherheitsvorschriften. In einigen lokalen Netzen ist es beispielsweise wünschenswert, dass keine Telnet-Verbindungen zwischen dem internen Netz und dem externen Netz, d.h. dem Internet, möglich sind. Der Proxy verbietet dann jeglichen Telnet-Datenverkehr, während andere Verbindungen, wie zum Beispiel HTTP oder FTP weiterhin möglich sind. Werden die Bestimmungen nicht eingehalten oder sind die Ressourcen erschöpft, verweigert der Proxy den Zugriff. Server im Internet haben den Eindruck, als würden sie nur mit dem Proxy kommunizieren. Die interne Netzstruktur bleibt somit vor der Außenwelt gekapselt. Klienten im lokalen Netz müssen sich in aller Regel auch nicht erst beim Proxy anmelden, da der gesamte Datenverkehr mit der Außenwelt automatisch über den Proxy geleitet wird. Die Klienten merken somit nicht, dass sie über einen Firewall-Proxy geleitet werden. Um zu verhindern, dass die Klienten trotz des Firewall-Proxies eine direkte Verbindung in des externe Netz aufbauen, wird oft Packet filtering eingesetzt. Weitere Informationen über Firewalls in [KeEs99].

2.9.4.6 Counting-Proxy

Der Counting-Proxy wird eingesetzt, wenn Komponenten nicht zufällig gelöscht werden sollen. Er zählt beispielsweise Referenzen auf gemeinsam genutzte Objekte und kann diese löschen, sobald sie von keiner Applikation mehr genutzt werden. Eine weitere Anwendung des Counting-Proxy ist das Mitführen von Statistiken über die Benutzung bestimmter Komponenten. Dies ist im Wesentlichen auch nichts anderes als das Zählen von Referenzen auf ein Objekt. Jedoch kann in manchen Fällen von Interesse sein, wie oft ein Objekt von einem bestimmten Klienten benötigt wird. Der Counting-Proxy kann besonders zu Analysezwecken eingesetzt werden, wenn es darum geht, Komponenten, auf die besonders oft zugegriffen wird, eventuell auf einen eigenen Rechner auszulagern.

Generell gilt für den Counting-Proxy, dass jede originale Komponente nur von einem Proxy-Objekt referenziert werden sollte. Der Zugriff auf die Komponente darf dann auch nicht mehr direkt, also am Proxy vorbei, geschehen, da sonst Statistiken oder Zähler falsche Werte erhalten würden. Die Schnittstelle des Proxies sollte der des Originals sehr ähnlich sein, so dass ein möglichst transparenter Zugriff auf die Komponente erfolgen kann. In den meisten Fällen ist die Schnittstelle des Proxies auch identisch mit der des Originals.

2.9.4.7 Cache-Proxy

Die Variante des Cache-Proxies findet hier besondere Beachtung, da zu ihr anschließend auch ein Prototyp erstellt werden wird. Wie der Name schon sagt, dient der Cache-Proxy dem Zwischenspeichern von Daten wie es ein Cache in aller Regel macht.

Um den Cache-Proxy zu implementieren, wird der Stellvertreter um einen Zwischenspeicher für zeitweilig zu speichernde Daten erweitert. Dieser Zwischenspeicher stellt letztendlich nichts anderes dar als die Kopien von konkreten Originalen. Nun ist jedoch immer Vorsicht geboten, wenn mit Kopien gearbeitet wird, da sich der Klient somit nie sicher sein kann, ob die Daten, die er vom Proxy geliefert bekommt, tatsächlich aktuell und somit gleich dem Original sind. Der Proxy muß folglich ein Konzept beinhalten, seinen eigenen Zwischenspeicher zu warten und – falls es erforderlich – wird zu aktualisieren oder gar neu zu beschreiben. Speichereinträge im Proxy können auch mit einem Verfallsdatum versehen werden, nach dessen Ablauf die Daten automatisch für ungültig erklärt und gelöscht werden. Die meisten Internet-Browser gehen nach diesem Prinzip vor. Der Cache-Proxy macht folglich nur einen Sinn, wenn er mit Objekten kommuniziert, die häufig von Klienten benötigt werden, jedoch eher seltener verändert werden. Dies ist zum Beispiel auch wieder in einer Datenbank der Fall. Datensätze werden sehr häufig von Klienten angefordert. Das Verändern von Daten kommt in der Praxis jedoch seltener vor.

Da der Proxy jedoch auch nur über einen bestimmten Speicherplatz verfügt, können auch in ihm nicht alle anfallenden Daten zwischengespeichert werden. Um das Verhalten des Proxies bei der Speicherplatzoptimierung zu regeln, gibt es mehrere Vorgehensweisen:

- Der Proxy leert seinen gesamten Speicher und fängt wieder von vorne an, ihn zu füllen. Dies ist die am einfachsten zu realisierende Möglichkeit. Einige Internet-Browser wie zum Beispiel *Netscape* verfolgen dieses Prinzip der Zwischenspeicherung.

- Die ältesten Einträge im Zwischenspeicher werden zuerst gelöscht und durch neue Daten ersetzt. Diese Variante ist im Vergleich zur ersten schon effektiver, da nicht alle Daten auf einmal verloren gehen. Dies bedeutet jedoch einen Mehraufwand bei der Implementierung.

- Weniger häufig benötigte Speichereinträge werden zuerst gelöscht. Häufig benötigte Speichereinträge werden zuletzt gelöscht. Diese Vorgehensweise erzielt nach einiger Zeit eine optimale Belegung des Speicherplatzes, da sehr häufig benötigte Objekte auf diese Weise so gut wie nicht mehr gelöscht werden können.

Die verwendete Variante der Speicherplatzoptimierung bleibt dem Entwickler selbst überlassen.

2.9.5 Codebeispiel für einen Cache-Proxy

Die Architektur des Cache-Proxies ist dieselbe, wie bereits oben beschrieben wurde. Die Client-Applikation erzeugt ein neues Proxy-Objekt und ruft den vom Proxy angeboten Service auf.

Im ersten Beispiel konnte die Client-Anwendung jedoch nur eine einzige Funktion des Stellvertreters aufrufen. Die Antwort, die der Client bekommt, wäre bei jedem Aufruf dieselbe gewesen. Er wird deshalb ein wenig verändert, so dass die Möglichkeit besteht, unterschiedliche Datensätze vom Proxy zu empfangen. Die Funktion `service()` wird deshalb im Folgenden ein wenig abgeändert. Sie bekommt in Zukunft einen Parameter mit übergeben, der Integer-Werte von 1 bis 12 annehmen kann. Der Parameter wird folglich auch an das Original mit übergeben. Der Sinn liegt ganz einfach darin, dass das Original in Abhängigkeit vom übergebenen Parameter einen unterschiedlichen Wert zurückliefern kann. In diesem Fall wird je nach übergebenem Wert der Name des dazugehörigen Monats zurückgeliefert. Dies ist vergleichbar mit einer Anfrage an eine Datenbank. So könnte zum Beispiel der übergebene Wert eine bestimmte Kundennummer sein und die zurückgelieferte Antwort der Datenbank der zu der Kundennummer passende Kundenname.

Als erstes wird die abstrakte Basisklasse auf den neu zu übergebenden Wert angepaßt:

```
/*
 * AbstractOriginal.java
*/

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

abstract class AbstractOriginal
extends UnicastRemoteObject
{

  /** Konstruktor AbstractOriginal */
  public AbstractOriginal()
     throws RemoteException
  {
  }

  /** Gemeinsame Schnittstelle **/
  public abstract String service(int wert)
     throws RemoteException;
}
```

Der Client wird gleich dahingehend erweitert, dass er permanent den Service des Proxy anfordert. Dies wird durch die zwei fortlaufenden Schleifen realisiert. Die innere der beiden Schleifen sorgt dafür, dass nacheinander alle Nummern von 1 bis 12 mit übergeben werden. Die äußere Schleife sorgt lediglich dafür, dass die gesamte Anfrage nicht nur ein einziges Mal durchlaufen wird, sondern insgesamt 100 mal. Auf diese Weise kann eine erhöhte Last simuliert werden. Diese Last wird am Ende ausgewertet werden. Es werden Messungen vorgenommen werden, was die durchschnittliche Antwortzeit des Proxies angeht. Dies Messungen beziehen sich jedoch nur auf das hier genannte Beispiel und können keineswegs verallgemeinert werden.

```
/*
 * ClientAppl.java
*/

import java.rmi.*;
import java.util.*;

public class ClientAppl {

    /** Konstruktor ClientAppl */
    public ClientAppl() {
    }

    public static void main(String args[]) {
     try {

          String monat = null;
          // Neuen Proxy erzeugen
          ServerProxy server = new ServerProxy();

          // Startzeit festhalten
          long starttime = System.currentTimeMillis();


          // Proxy-Service veriiert abrufen
          // und Auslastung simulieren
          for (int runden=0;runden<100;runden++)
           for (int wert=1;wert<=12;wert++)
                  monat =  server.service(wert);

          // Endzeit bestimmen
          long stoptime = System.currentTimeMillis();
          System.out.println("Benötigte Zeit:"+
                         (stoptime-starttime));

     } catch (Exception e) {};
  }
```

```
}
```

Das Original muß diese Änderung dementsprechend auch erfahren und eine jeweils passende Antwort zurück liefern. Die zurückgelieferten Daten werden einem Array von Strings entnommen.

```
/*
 * Original.java
*/

import java.rmi.*;
import java.io.*;

class Original extends AbstractOriginal {

     String [] daten = { "Januar","Februar","März",
                         "April","Mai","Juni","Juli",
                         "August","September","Oktober",
                         "November","Dezember" };

    /** Konstruktor Original */
    public Original() throws RemoteException {}

    public String service(int wert)
     throws RemoteException {

     // Namen des Monats zurückliefern
     return daten[(wert-1)];
    }

    public static void main (String args[]) {
     try {
         // Ein neues Original-Objekt erzeugen
         Original original = new Original();

         // Original-Objekt über RMI
         // zugänglich machen
         Naming.rebind("rmi:///Original",original);

         while (System.in.read()!='s');
         System.exit(0);
     }
     catch (Exception e) {
         System.out.println("Fehler !");
     }
  }
}
```

Der Stellvertreter selbst muß jedoch um seine neue Funktionalität erweitert werden. Es wird ein Zwischenspeicher angelegt, der die vom Original empfangenen Daten speichert und sie bei Bedarf an die Anwendung weitergibt, sofern er sie bereits im BSpeicher hat. Vor- und Nachbearbeitungen entfallen in diesem neuen Beispiel. Einzelne Teile des Proxy, welche sich nicht von dem ersten Beispiel unterscheiden wurden mit " . . . " gekennzeichnet.

```
/*
 * ServerProxy.java
*/

// import ...

public class ServerProxy extends AbstractOriginal {

    RemoteInterface original;
    String [] cache = new String [12];
    String monat = null;

    /** Konstruktor ServerProxy */
    public ServerProxy()
     throws RemoteException {

     super();
     try
         {
          // Cache initialisieren
          for (int x=0;x<12;x++)
              cache[x]=null;

          // Referenz auf Original-Objekt
          // ...
    } catch (Exception e) {
         System.out.println("Fehler !"); }

    public String service(int wert)
     throws RemoteException {
     try
         {
          // Zuerst prüfen, ob der Wert eventuell
          // schon im Cache vorhanden ist
          if (cache[(wert-1)]!=null)
              return cache[(wert-1)];

          // falls nicht, Funktionsaufruf beim Original
          // und neu erhaltenen Wert speichern
          monat= original.service(wert);
          cache[(wert-1)]=new String(monat);

         }
```

```
      catch (Exception e) {
          System.out.println("Fehler !");
      }
      return monat;
     }
}
```

Auffallend an diesem Beispiel wird eventuell sein, dass der übergebene Wert immer Bum eins reduziert wird, sobald er in Verbindung mit den Variablen `cache` des Server-Proxies beziehungsweise `daten` des Originalobjektes in Aktion tritt. Dies entsteht lediglich dadurch, dass Monate im Allgemeinen mit Ziffern von 1 bis 12 numeriert werden, wohingegen Arrays in Java grundsätzlich bei 0 beginnen. Der Monat Februar, welches der zweite Monat im Jahr ist, bekommt somit in der internen Datenstruktur von Java-Arrays nur den Wert 1 zugewiesen. Die Numerierung der Monate in einem Array erfolgt somit von 0 bis 11. Dies soll jedoch nicht weiter stören.

Der Proxy prüft auf diese Weise vor jeder Anfrage an das Original, ob der gewünschte Datensatz nicht bereits in den internen Speicher kopiert wurde. Man beachte, dass in diesem Beispiel noch nicht die Möglichkeit einer Aktualisierung der Daten vorgesehen wurde. Sollten die Monate im Original plötzlich nicht mehr Januar, Februar, etc. lauten, sonder beispielsweise durch englische Namen ersetzt worden sein, würde der Proxy von der Änderung nichts erfahren und weiterhin die unaktualisierten Daten an den Client zurückgeben. Für das Problem der Aktualisierung wird der Cache-Proxy häufig mit dem Beobachterpattern (siehe erstes Kapitel) kombiniert. Design Patterns lassen sich in den meisten Fällen problemlos miteinander kombinieren.

Das Problem der Speicherplatzoptimierung ist in diesem Fall auch nicht gegeben, da der Proxy in jedem Fall weiß, dass er nie mehr als 12 Felder für seinen Speicher reservieren muß. Letztendlich gibt es eben nicht mehr als 12 Monate in einem Jahr. Es ist jedoch nur in den wenigsten Fällen gegeben, dass von einer festen Anzahl von möglichen Elementen ausgegangen werden kann. Oft gibt es keine definierte Obergrenze für die Anzahl möglicher und unterschiedlicher Elemente, die vom Original zurückgegeben werden können.

2.10 Observer

Das Observerpattern beschreibt, wie eine veränderliche Gruppe von Objekten über ein bestimmtes Ereignis informiert werden kann. Andere Namen sind: **Beobachter-Muster** oder **Publisher-Subscriber-Muster**.

2.10.1 Ein Einführungsbeispiel

Wenn ein Kind des Kindergartens Läuse hat, so werden die Eltern aller Kinder darüber informiert, dass der Kindergarten in den nächsten Tagen geschlossen bleibt. Im Beispiel informiert das Objekt `Kindergarten` die Elternobjekte `Müller`, `Fabian`, `Frech` und `Stanullo`. Dazu ist eine Assoziation mit den Elternobjekten nötig.

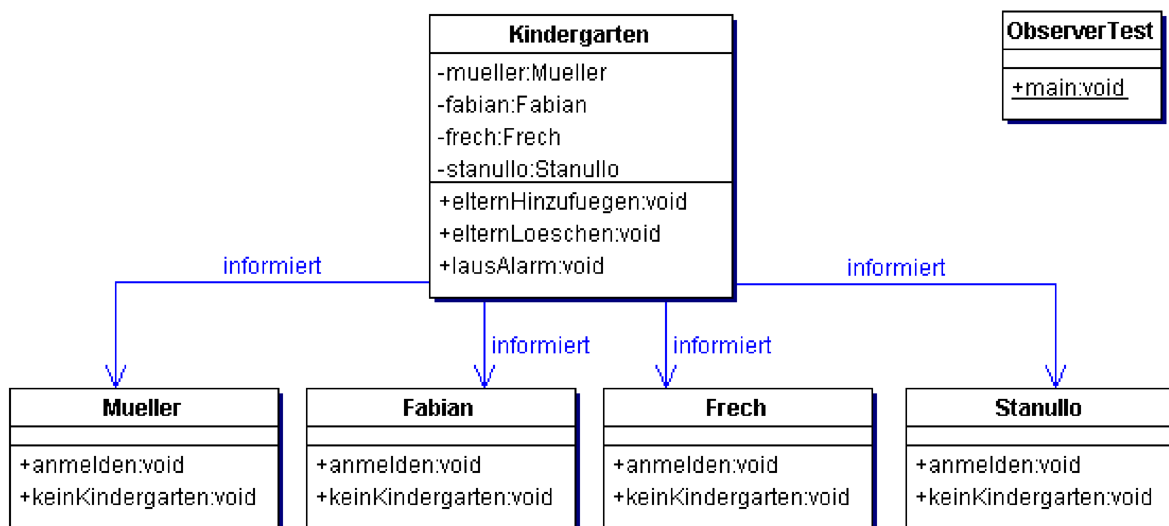


Abbildung 15: Negativbeispiel: Das Objekt Kindergarten aggregiert vier Elternobjekte

In der Methode `lausAlarm ( )` des Kindergarten-Objekts werden die Methoden `keinKindergarten ( )` der vier Elternobjekte aufgerufen.

Diese Struktur ist für den Augenblick funktionstüchtig, sie ist aber sehr unflexibel. Wenn das Kind einer anderen Familie in den Kindergarten kommt oder wenn das Kind einer Familie den Kindergarten verlässt, muss das Kindergartenobjekt neu erzeugt werden, um neue Informationen sinnvoll zu adressieren.

Das Observer-Pattern sieht einen anderen Weg der Adressierung vor. Hier können sich die `Eltern`-Objekte selbst bei dem `Kindergarten`-Objekt für interessante Ereignisse registrieren lassen und wieder abmelden. Das `Kindergarten`-Objekt speichert die Referenzen auf alle interessierten `Eltern`-Objekte in einem Vektor ab. In der Methode `lausAlarm ( )` wird nun die Methode `keinKindergarten ( )` von allen `Eltern`-Objekten aufgerufen, die in dem Vektor abgelegt wurden. Damit wird sogar eine dynamische Änderung des Verteilers zur Laufzeit möglich.

© Goll

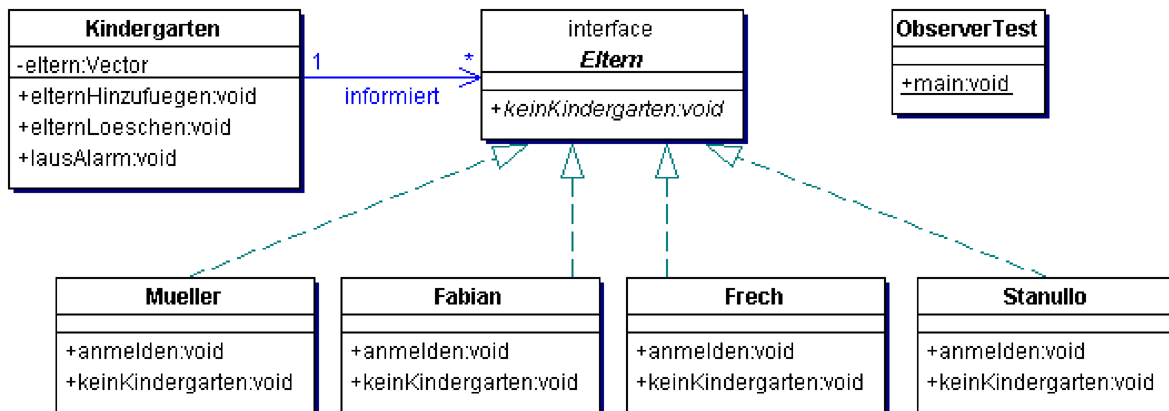


Abbildung 16: Das Kindergartenobjekt hält die Elternobjekte in einem Vektor

Die Objekte, die das `Kindergarten`-Objekt speichert, müssen von dem Typ `Eltern` sein, denn andere Objekte interessieren sich nicht für die Belange das Kindergartens. Um dies zu erreichen genügt es, wenn die `Eltern`-Objekte das Interface `Eltern` implementieren. Auf diese Weise kann sich das `Kindergarten`-Objekt sicher sein, dass alle interessierten Objekte auch die Methode `keinKindergarten ( )` implementiert haben. Ein `Eltern`-Objekt kann nun die Methode `elternHinzufuegen (this)`[8] des `Kindergarten`-Objekts aufrufen, um sich in den Verteiler aufnehmen zu lassen.

Quellcode

```
import java.util.*;

public class Kindergarten {

   private Vector eltern = new Vector ();

   public void elternHinzufuegen (Eltern e) {
     eltern.add (e);
   }

   public void elternLoeschen (Eltern e) {
     eltern.remove (e);
   }

   public void lausAlarm () {
     System.out.println ("Kindergarten: Ein Kind hat Laeuse!");
     for (int i=0; i<eltern.size(); i++) {   // alle Eltern-Objekte
       Eltern e = (Eltern) eltern.get(i);    // des Vektors werden
       e.keinKindergarten();                 // benachrichtigt
     }
```

[8] Mit „this“ übergibt das Eltern-Objekt eine Referenz auf sich selbst

```
    }
}
```

```
public interface Eltern {
  void keinKindergarten ();
}
```

```
public class Mueller implements Eltern {

  public void anmelden (Kindergarten kiga) {
    kiga.elternHinzufuegen (this);
  }

  public void keinKindergarten () {
    System.out.println ("Heute bleibt der kleine Mueller zu Hause!");
  }
}
```

```
public class Fabian implements Eltern {

  public void anmelden (Kindergarten kiga) {
    kiga.elternHinzufuegen (this);
  }

  public void keinKindergarten () {
    System.out.println ("Heute bleibt der kleine Fabian zu Hause!");
  }
}
```

```
public class Frech implements Eltern {

  public void anmelden (Kindergarten kiga) {
    kiga.elternHinzufuegen (this);
  }

  public void keinKindergarten () {
    System.out.println ("Heute bleibt der kleine Frech zu Hause!");
  }
}
```

```
public class Stanullo implements Eltern {

  public void anmelden (Kindergarten kiga) {
    kiga.elternHinzufuegen (this);
  }

  public void keinKindergarten () {
    System.out.println ("Heute bleibt der kleine Stanullo zu Hause!");
```

```
  }
}
```

```
class ObserverTest {

  public static void main (String [] args) {

    Kindergarten kiga = new Kindergarten ();

    Mueller mueller = new Mueller ();
    Fabian fabian = new Fabian ();
    Frech frech = new Frech ();
    Stanullo stanullo = new Stanullo ();

    mueller.anmelden (kiga);
    fabian.anmelden (kiga);
    frech.anmelden (kiga);
    stanullo.anmelden (kiga);

    kiga.lausAlarm ();
  }
}
```

Ausgabe des Programms:

```
Kindergarten: Ein Kind hat Laeuse!
Heute bleibt der kleine Mueller zu Hause!
Heute bleibt der kleine Fabian zu Hause!
Heute bleibt der kleine Frech zu Hause!
Heute bleibt der kleine Stanullo zu Hause!
```

Bemerkungen

Das Observerpattern kann immer dann eingesetzt werden, wenn ein Objekt Ereignisse an eine veränderliche Gruppe von Objekten weitergeben soll. Dies ist häufig bei Steuerelementen der GUI nötig. Wenn z.B. verschiedene Objekte auf einen Knopfdruck reagieren sollen, so lassen sie sich bei dem entsprechenden Schaltflächenobjekt registrieren und werden dann informiert, wenn das Ereignis „Knopfdruck" auftritt. Die gesamte Ereignisverarbeitung von AWT[9] und Swing beruht auf diesem Pattern.

Der Methode, die bei den interessierten Objekten aufgerufen wird, wird in der Regel ein `Event`-Objekt als Parameter mitgegeben, das den Hintergrund des Ereignisses genauer beschreibt.

[9] Das Abstract Window Toolkit (AWT) stellt die erste Klassenbibliothek dar, die eine Oberflächenprogrammierung unter Java ermöglicht. Die Klassenbibliothek Swing wird mit dem JDK seit der Version 1.2 ausgeliefert. Swing ist umfangreicher und leistungsfähiger als AWT [GOL1999].

© Goll

2.10.2 Kontext

Es existiert eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objektes dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

2.10.3 Problem

Eine enge Verknüpfung unter den einzelnen Objekten ist zwar technisch gesehen machbar, würde jedoch die Wiederverwendbarkeit und die Flexibilität der daraus resultierenden Anwendung stark beeinträchtigen.

2.10.4 Lösung

Das Beobachtermuster (engl. Observer) wird immer dann eingesetzt, wenn es darum geht, dass bei Änderungen des Zustands eines Objekts die davon abhängigen Objekte bezüglich dessen Zustandsänderungen[10] synchronisiert werden. Eine Klasse nimmt dabei die Rolle des Subjektes ein. Das Subjekt stellt das **beobachtbare Objekt** dar und stellt ein Interface zum An- und Abmelden von **Beobachtern**, den Observern, zur Verfügung. Die Observer, die am Zustand des Subjekts interessiert sind, melden sich am Subjekt mit der Methode `attach(Observer)` an. Das Subjekt kennt somit alle seine Observer und kann diese bei einer Zustandsänderung benachrichtigen. Die Observer stellen dabei ein Interface zur Verfügung, über welches das Subjekt die Observer über Änderungen benachrichtigen kann. Das Subjekt hat zusätzlich ein Interface zum Zugriff auf dessen internen Zustand, das meist durch die beiden Methoden `getState()` und `setState()` repräsentiert wird.

Dies mag im ersten Moment sicher etwas kompliziert klingen. An einem Beispiel wird es jedoch sehr schnell deutlich.

Das konkrete Subjekt, von dem oben noch die Rede war, könnte beispielsweise der Datenbestand in einer Tabellenkalkulation sein. Diese bietet in aller Regel eine Möglichkeit, die Daten auf unterschiedliche Art und Weise zu betrachten. In dem einen Fall möchte man gerne die Daten in Form einer Tabelle betrachten und ändern können. Eine andere Sichtweise der Daten könnte in Form eines Balkendiagramms oder einer sogenannten Tortengraphik sein. Egal jedoch welche – und vor allen Dingen wie viele – der drei Formen der Darstellung (engl. View) der Benutzer gewählt hat, sollten sie sich alle bei einer Änderung des Datenbestandes ebenso umgehend aktualisieren. Abbildung 17 veranschaulicht dieses Prinzip.

Das Problem der Aktualisierung wird dadurch gelöst, dass sich die Views über die Methode `attach(Observer)` bei dem Subjekt als Beobachter anmelden. Wird der von einer View angezeigte Datenbestand geändert, so ruft der Observable (das Subjekt) den Observer, d.h. die View, auf. Das Subjekt muß jetzt alle an ihm angemeldeten Beobachter über eine Änderung des Datenbestandes

[10] Zustandsänderungen eines Objektes sind Werte-Änderungen gewisser Datenfelder des Objektes

© Goll

benachrichtigen. Dies geschieht dadurch, dass das Subjekt bei jedem der Beobachter die Methode `update()` aufruft, die wiederum sich den neuen Datenbestand über eine Methode `getState()` des Subjektes besorgen und ihre Darstellung neu zeichnen.

In der Regel werden der Methode `update()` noch zusätzliche Parameter übergeben, damit die angemeldeten Beobachter entscheiden können, ob die durchgeführte Änderung für sie relevant ist. Auf diese Art und Weise erfährt jeder der am Subjekt angemeldeten Beobachter die Änderung und kann selbst entscheiden, ob er die Änderung über den Aufruf von `getState()` übernimmt oder nicht.

Eine zweite Möglichkeit ist, dass der geänderte Wert direkt beim Aufruf von `update()` als Parameter mit übergeben wird. Dann entfällt der zusätzliche Schritt, dass Beobachter den neuen Wert erst abfragen müssen, bevor sie ihre Darstellung aktualisieren können. Dies ist jedoch nicht immer wünschenswert. Sobald es sich um größere Objekte handelt, die aktualisiert werden müssen, verzichtet man darauf, das geänderte Objekt gleich in `update()` mit zu übergeben und läßt den Beobachter selbst entscheiden, ob er eine Aktualisierung benötigt.

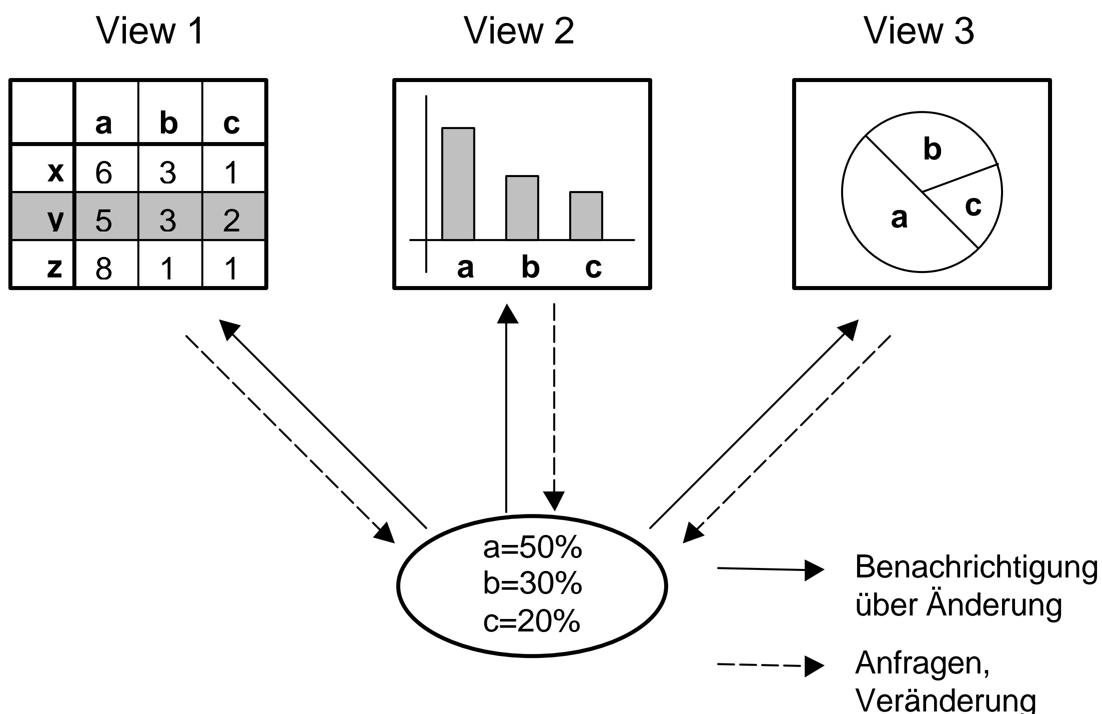


Abbildung 17 – Prinzip des Beobachter-Patterns

Abbildung 18 zeigt das Klassendiagramm für das Observer-Pattern.

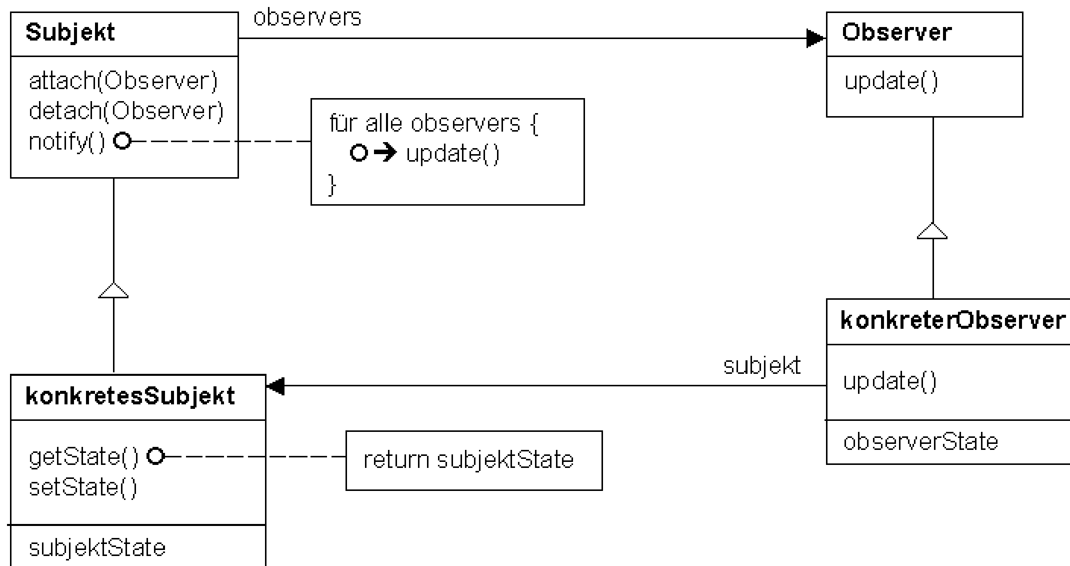


Abbildung 18 – Klassendiagramm des Beobachter-Patterns

Ein konkretes Subjekt implementiert das Interface `Subject`. Der konkrete Beobachter implementiert seinerseits das Interface `Observer`. Auf diese Weise können Subjekt und Beobachter miteinander kommunizieren, da sie gegenseitige Kenntnis darüber besitzen, welche Schnittstelle der Partner implementiert. Die konkreten Objekte können durchaus weitere Funktionen implementieren, über die der Partner keine Kenntnis besitzt.

Wird der Zustand des Subjektes von außen durch `setState()` geändert, so schickt das Subjekt an sich selbst die Nachricht `notify()`. Daraufhin werden alle registrierten Observer mit `update()` über diese Änderung benachrichtigt. Die benachrichtigten Oberserver können nun entscheiden, ob sie die Änderung abholen wollen oder nicht. Ist ein Observer an der Änderung interessiert, so holt dieser sich den neuen Zustand des Subjektes mit `getState()` ab. Dieser Vorgang wird an einem Interaktionsdiagramm in Abbildung 19 dargestellt.

© Goll

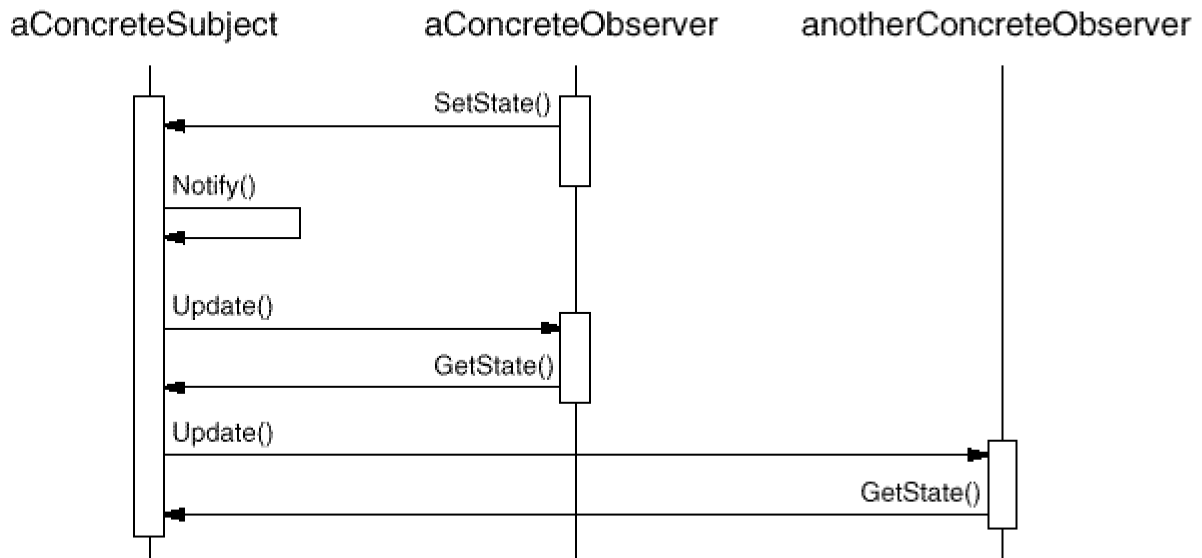


Abbildung 19 – Sequenzdiagramm für ein Beobachter-Pattern

2.10.5 Konsequenzen

Das Observer Pattern ermöglicht es, einem Objekt Mitteilungen an andere Objekte zu versenden, ohne dass das sendende beziehungsweise das empfangende Objekt genaue Kenntnis über die Klasse des Gegenüber besitzt. Es gibt jedoch einige Situationen, in denen das Beobachtermuster unvorhergesehene und ungewollte Nebeneffekte mit sich bringt:

Das Verschicken der Mitteilungen kann je nach Anzahl der Beobachter-Objekte eine längere Zeit in Anspruch nehmen. Dieser Fall kann auftreten, wenn ein beobachtetes Objekt sowohl viele direkte Beobachter als auch viele indirekte Beobachter registriert hat, die hierarchisch über die direkten Beobachter informiert werden.

Ein viel ernster zu nehmendes Problem ergibt sich, wenn die Beobachter zyklisch untereinander verknüpft sind. Die Objekte würden sich gegenseitig mit Update aktualisieren, bis der Stack-Speicher des Rechners aufgebraucht ist. Dieses Problem kann jedoch gelöst werden, indem den Beobachtern ein internes Flag hinzugefügt wird, wodurch eine gegenseitige rekursive Aktualisierung erkannt und verhindert wird.

+++++++++++++++++++
Noch zu integrieren:

Das Entwurfsmuster Observer (Beobachter) ist ein Verhaltensmuster, das gewährleistet, dass bei Änderungen des Zustands eines Objekts die davon abhängigen Objekte bezüglich dessen Zustandsänderungen synchronisiert werden. Eine Klasse nimmt dabei die Rolle des Subject ein. Das Subject stellt das beobachtbare Objekt dar und stellt ein Interface zum An- und Abmelden von Beobachtern, den Observern, zur Verfügung. Die Observer, die am Zustand des Subjects interessiert sind, melden sich am Subject mit der Methode `Attach(Observer)` an. Das Subject kennt somit alle seine Observer und kann diese bei einer Zustandsänderung benachrichtigen. Die Observer stellen dabei ein Interface zur Verfügung, über welches das Subject die Observer über Änderungen benachrichtigen kann. Das Subject hat zusätzlich ein Interface zum Zugriff auf dessen internen Zustand, das meist durch die beiden Methoden `GetState()` und `SetState()` repräsentiert wird. **Fehler! Verweisquelle konnte nicht gefunden werden.** zeigt das Klassendiagramm des Entwurfsmusters Observer.

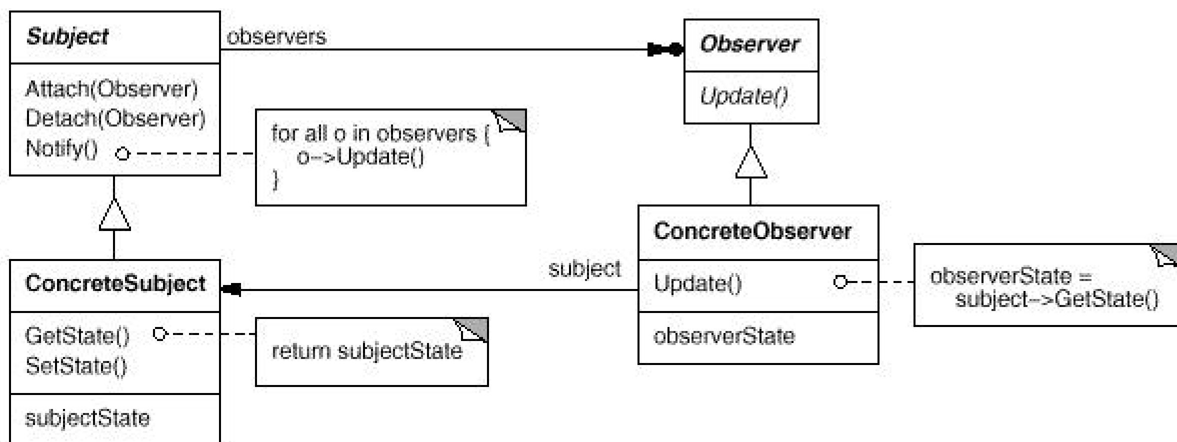


Abbildung 20: Entwurfsmuster Observer

Wird der Zustand des Subjects von außen durch `SetState()` geändert, so schickt das Subject an sich selbst die Nachricht `Notify()`. Daraufhin werden alle registrierten Observer mit `Update()` über diese Änderung benachrichtigt. Die benachrichtigten Oberserver können nun entscheiden, ob sie die Änderung abholen wollen oder nicht. Ist ein Observer an der Änderung interessiert, so holt dieser sich den neuen Zustand des Subjects mit `GetState()` ab. Dieser Vorgang ist in **Fehler! Verweisquelle konnte nicht gefunden werden.** dargestellt.

 © Goll

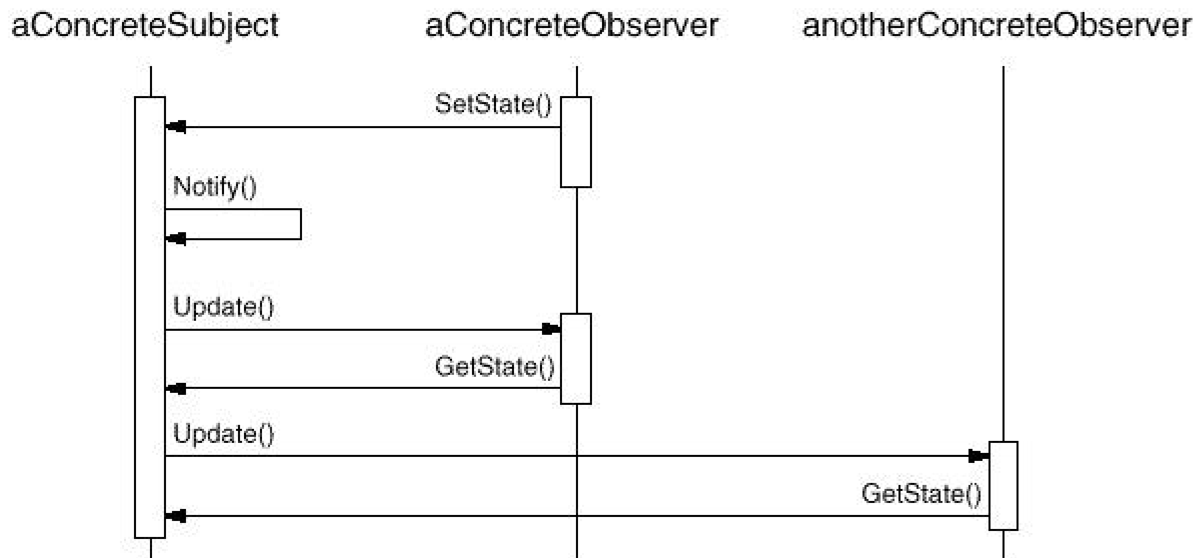


Abbildung 21: Benachrichtigungsvorgang

Das Entwurfsmuster Observer entkoppelt die beteiligten Objekte weitgehend voneinander. Das Subject kennt seine Observer nur über das Interface Observer, dass nur die Methode `Update()` enthält. Die Observer kennen zwar alle das Subject, diese Beziehung ist jedoch verständlicherweise nicht zu vermeiden. Die Observer untereinander kennen sich überhaupt nicht, was im Allgemeinen ein Vorteil ist. Beobachten jedoch viele Observer ein Subject, so kann eine Zustandsänderung des Subjects eine `Update()`-Kaskade hervorrufen, die schnell unüberschaubar wird.

Das Observer-Muster hat zahlreiche Varianten. Hauptsächlich geht es dabei um den Benachrichtigungs-Mechanismus und die An- und Abmeldung von Observern. Einige dieser Varianten sollen abschließend betrachtet werden.

Fehler! Verweisquelle konnte nicht gefunden werden. zeigt das sogenannte Pull-Model. Die Observer werden über `Update()` benachrichtigt und müssen daraufhin den Zustand des Subject selbst erfragen (Pull), indem sie die Methode `GetState()` des Subjects aufrufen. Eine andere Variante (Push-Model) ist es, den neuen Zustand des Subjects beim Aufruf von `Update()` zu übergeben (Push). Ein Aufruf von `GetState()` seitens der Observer ist dadurch nicht mehr nötig und entfällt somit. Der Vorteil ist die reduzierte Anzahl von Methodenaufrufen zwischen dem Subject und seinen Observern. Der Nachteil ist, dass jeder Observer den neuen Zustand des Subjects mitgeteilt bekommt, egal ob er im Moment daran interessiert ist, oder nicht. Ein weiterer, viel wichtigerer Nachteil ist die stärkere Kopplung zwischen Subject und Observer, da der Zustand des Subject mit in die Signatur der Methode `Update()` aufgenommen werden muss.

Ist der Zustand des Subjects sehr umfangreich und gibt es Observer, die nur an einem kleinen Teil des Zustands interessiert sind, so kann es vorteilhaft sein, das Protokoll zwischen Subject und Observer zu erweitern. Über einen zusätzlichen

© Goll

Parameter der Methode `Attach()` kann ein Observer dem Subject mitteilen, an welchem Teil des Zustands dieser interessiert ist. Der zusätzliche Parameter kann auch in die Signatur der Methode Update() aufgenommen werden, falls der Observer an mehreren Teilen des Zustands interessiert ist. Auch diese Variante reduziert im Allgemeinen die Anzahl der Methodenaufrufe zwischen Subject und Observer bei gleichzeitig stärkerer Kopplung der beteiligten Klassen.

Beobachtet ein Observer mehrere Subjects, so muss eine Referenz auf das Subject in das Protokoll der Methode `Update()` aufgenommen werden, so dass die Observer unterscheiden können, welches Subject sich geändert hat. Auch hier wird eine stärkere Kopplung zwischen Subject und Observer erzeugt.

Wird der Zustand des Subject über `SetState()` geändert, so ruft das Subject die Methode `Notify()` bei sich selbst auf. Dadurch ist sichergestellt, das jede Änderung des Zustands an alle Observer weitergereicht wird. Nimmt ein Observer jedoch mehrere Änderungen hintereinander vor, so kann es zu überflüssigen `Update()`-Kaskaden kommen. Um dies zu verhindern kann es sinnvoll sein, die Verantwortlichkeit zum Aufruf der Methode `Notify()` vom Subject in die Observer zu verlagern. Dadurch kann ein Observer mehrere Änderungen am Subject vornehmen und erst danach eine Benachrichtigung aller Observer über einen Aufruf von `Notify()` beim Subject veranlassen. Der Nachteil ist, dass ein Observer diesen Schritt vergessen kann und die Änderung dann nicht sofort an die Observer weitergegeben wird.

Ist die Aktualisierungssemantik zu komplex und wird eine sehr geringe Kopplung zwischen mehreren Subjects und Observern gewünscht, so kann eine Vermittler-Klasse eingeführt werden. Alle Observer melden sich nicht mehr direkt am Subject an, sondern an der Vermittler-Klasse. Alle Subjects melden eine Zustandsänderung nicht den Observern direkt, sondern der Vermittler-Klasse. Die Vermittler-Klasse kümmert sich um die Benachrichtigung aller interessierten Observer. Aber auch diese Methode hat weitere Vor- und Nachteile auf die an dieser Stelle nicht genauer eingegangen werden soll.

Eine ausführliche Betrachtung dieser und vieler weiterer Varianten des Entwurfsmusters Observer ist in [GoF95] und [Gra98] zu finden.

+++++++++++++++++++

2.11 Producer/Consumer

Abschließend soll noch ein sehr nützliches Entwurfsmuster auf dem Bereich der nebenläufigen Programmierung vorgestellt werden: Producer/Consumer (Erzeuger/Verbraucher). Mit Hilfe dieses Entwurfsmusters kann ein asynchrones Erzeugen und Verbrauchen von Objekten koordiniert werden. Erinnern wir uns an das Beispiel aus Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**, **Fehler! Verweisquelle konnte nicht gefunden werden.**. Es ging dabei um eine Warteschlange für Objekte der Klasse `Message`. Die dort vorgestellte Lösung hat den Nachteil, dass eine Exception geworfen wird, wenn die Warteschlange leer ist und versucht wird ein `Message`-Objekt zu entnehmen. Stellt man sich die Warteschlange im Kontext einer Anwendung vor, so zeigt sich schnell, dass diese Vorgehensweise eher ungeeignet ist. Stellen wir uns eine Komponente der Anwendung vor, die zu beliebigen Zeitpunkten `Message`-Objekte erzeugt. Diese `Message`-Objekte sollen an den Empfänger ausgeliefert werden. Um die Asynchronität zu gewährleisten, werden die `Message`-Objekte nach der Erzeugung in einer zentralen Warteschlange ablegt und ein eigenständiger Thread kümmert sich um deren Auslieferung. Wäre die Warteschlange wie in Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** implementiert, so müsste der Thread in einer Schleife versuchen ein `Message`-Objekt auszulesen und darauf vorbereitet sein eine Exception zu fangen. Abgesehen von dem erhöhten Aufwand die möglicherweise auftretende Exception fangen zu müssen, wird auch unnötig viel Rechenzeit verbraucht. Folgender Programmcode zeigt einen solchen Thread, implementiert durch die Klasse `PostOffice`:

© Goll

```
import java.util.NoSuchElementException;

public class PostOffice extends Thread {

  MessageQueue mq;

  public PostOffice(MessageQueue mq) {
    this.mq = mq;
    this.start();
  }

  public void run() {

    Message m;

    while(true) { // Polling ...
      try {
        m = mq.dequeue();
        deliver(m);
      }
      catch(NoSuchElementException e) {
        // Ignore ...
      }
    }

  }

  private void deliver(Message m) {
    // ...
  }

}
```

Wie gesagt verursacht das Polling der Warteschlange einen hohen Verbrauch an Rechenzeit. Eine elegantere Lösung stellt das Producer/Consumer Muster dar. Folgender Progammcode zeigt die angepasste Klasse `Synchronized-ObjectQueue`:

```
import java.util.LinkedList;

public class SynchronizedObjectQueue {

  private LinkedList queue = new LinkedList();

  public synchronized void enqueueObject(Object o) {
    this.queue.addLast(o);
    this.notifyAll();
  }

  public synchronized Object dequeueObject() {
    while (queue.isEmpty()) {
      try {
        this.wait(); // Thread anhalten
      }
      catch (InterruptedException e) {
        // ...
      }
    }
    return this.queue.removeFirst();
  }

}
```

Im Programmcode ist zu sehen, wie in der Methode `dequeueObject()` der aufrufende Thread angehalten wird, wenn die Warteschlange leer ist. Somit wird keine Rechenzeit verbraucht. Die Methode kehrt erst zum Aufrufer zurück, wenn ein Objekt aus der Warteschlange entnommen werden konnte. Dazu reaktiviert die Methode `enqueueObject()` alle wartenden Threads, wenn ein Objekt hinzugefügt wird. Dadurch vereinfacht sich die `run()` Methode der Klasse `PostOffice`:

```
import java.util.NoSuchElementException;

public class PostOffice extends Thread {

  // ...

  public void run() {

    Message m;

    while(true) {
      m = mq.dequeue();
      deliver(m);
    }

  }

  // ...
}
```

2.12 Facade

Über das Facadepattern kann der Zugriff auf viele Objekte einer komplexen Objektstruktur durch den Zugriff auf ein einziges Facade-Objekt ersetzt werden.

2.12.1 Ein Einführungsbeispiel

[GRA1998] Ein e-Mail Client greift auf eine komplexe Struktur von Objekten zu. Um eine neue Nachricht zu verfassen, erzeugt er verschiedene Objekte, die wiederum miteinander in Beziehung stehen.

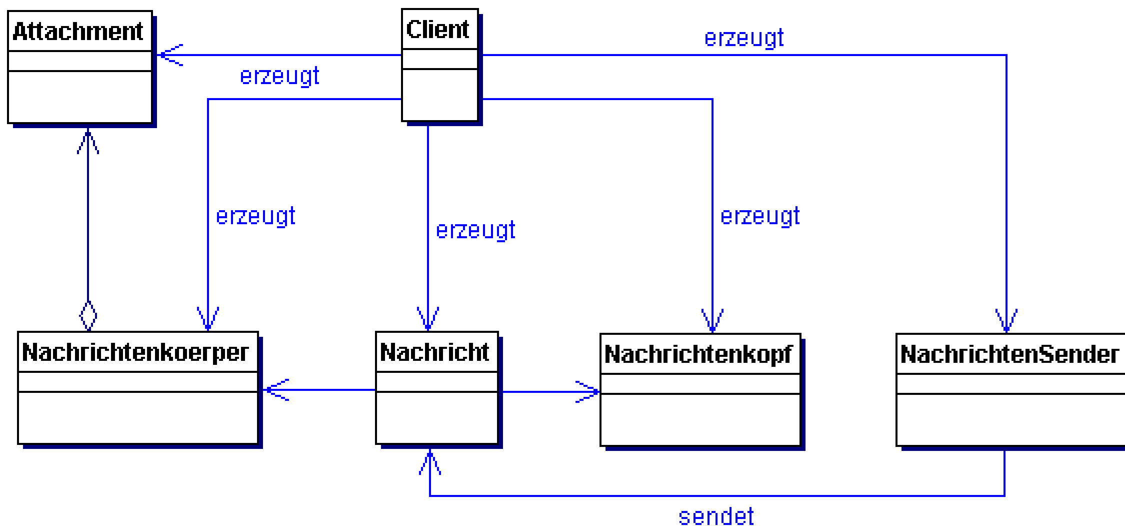


Abbildung 22: Beziehungen eines e-Mail Clients

Die Beziehungen der Objekte im Klassendiagramm können nicht vereinfacht werden. Es ist jedoch möglich, nachträgliche Änderungen zu erleichtern denn die vorliegende Architektur ist sehr unflexibel. Jedes neue Client-Objekt muss dieselben Beziehungen pflegen wie der ursprüngliche Client. Damit wird die Konzeption eines neuen Clients stark verkompliziert. Wenn der Ablauf der Nachrichtenerzeugung durch die Ersetzung von Dienstobjekten verbessert werden soll, können bisherige Client-Objekte nicht mehr verwendet werden. Sie müssen angepasst und neu erzeugt werden.

Das Facade-Pattern sieht für das e-Mail System eine einzige Schnittstellenklasse `NachrichtenErzeugung` vor, die den e-Mail Client mit der Nachrichtenerstellung verbindet.

© Goll

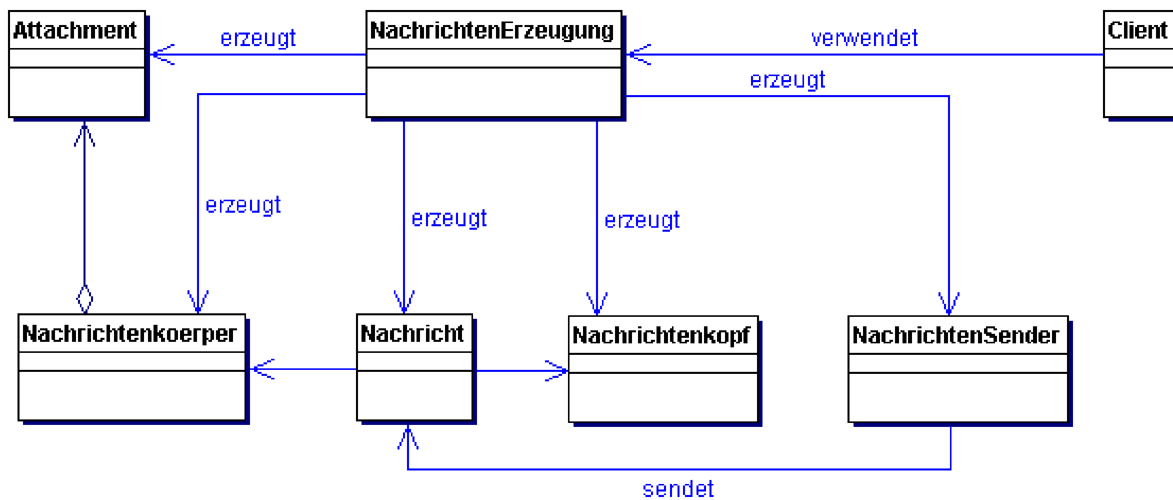


Abbildung 23: E-mail Erzeugung mit Schnittstellenklasse

Bei nachträglichen Änderungen der Objektstruktur bleibt die Schnittstellenklasse NachrichtenErzeugung erhalten. Das bisherige Client-Objekt kann also weiter verwendet werden. Ein neues Client-Objekt kann sich ebenfalls auf das Objekt NachrichtenErzeugung stützen und muss die interne Objektstruktur, die der Nachrichtenerzeugung zugrunde liegt, nicht kennen.

Quellcode

Bei diesem Design-Pattern wird auf den Quelltext eines Beispielszenarios verzichtet, da er die Funktionsweise des Patterns nicht zusätzlich verdeutlichen kann.

Bemerkungen

Das Facade-Pattern wird gerne verwendet, da es einfach zu implementieren ist. Es stellt ein typisches Pattern dar, das die Abhängigkeiten einer Architektur reduziert, damit die resultierende Software leichter zu pflegen und zu erweitern wird.

++++++
Noch zu integrieren:

Das Entwurfsmuster Facade ist ein Strukturmuster, das eine Schnittstelle für eine Menge zusammenarbeitender Klassen darstellt. Die Facade vereinfacht durch Abstraktion die Verwendung der Funktionalität eines solchen Klassenverbundes. Die Objekte (Client-Objekte), die die Funktionalität des Klassenverbundes verwenden möchten, müssen nicht von den einzelnen Klassen und deren Abhängigkeiten wissen. Sie verwenden nur die Facade. Durch die Verwendung des Entwurfsmusters Facade werden die Client-Objekte von den Details der Implementierung des Klassenverbundes entkoppelt. **Fehler! Verweisquelle konnte nicht gefunden werden.** zeigt diesen Zusammenhang.

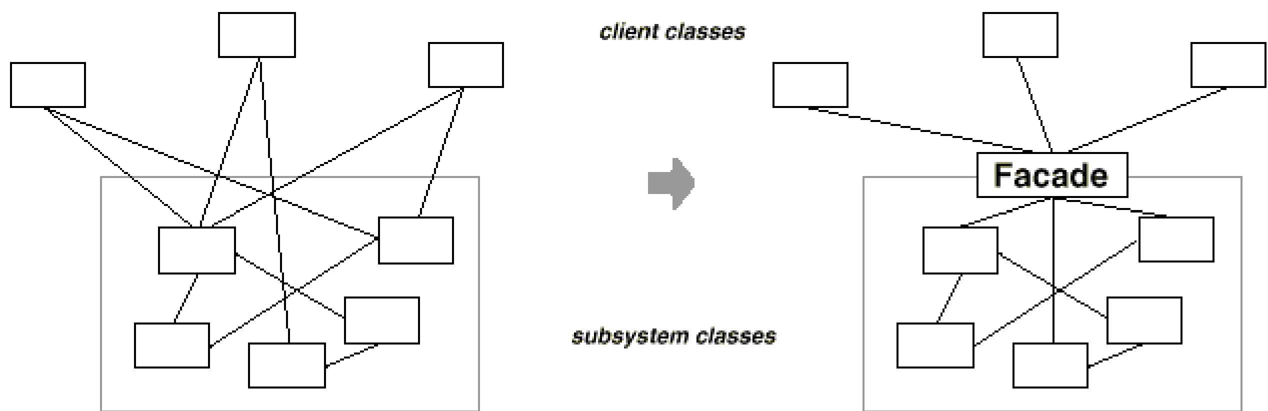


Abbildung 24: Verwendung der Funktionalität eines Klassenverbunds ohne und mit Hilfe des Entwurfsmusters Facade

Die Idee hinter diesem Entwurfsmuster ist eng mit dem Prinzip des "Information Hiding" verwandt. Beim "Information Hiding" wird das Innere einer Klasse vor der Anwendung verborgen, so dass das Innere der Klasse variiert werden kann, ohne deren Schnittstelle nach außen zu verändern. Der Klassenverbund beim Entwurfsmuster Facade entspricht dabei der Klasse beim "Information Hiding". Die Klasse Facade entspricht als Schnittstelle des Klassenverbunds der Schnittstelle der Klasse beim "Information Hiding".

Mit dem Entwurfsmuster Facade lassen sich Softwaresysteme sehr einfach in Komponenten zerlegen. Die Komponenten stellen dabei einen Klassenverbund mit hoher Kohärenz dar, d.h. die Klassen stehen funktional in einer sehr engen Beziehung zueinander und erbringen im Verbund die Leistung der Komponente. Gleichzeitig ist eine lose Kopplung zwischen den Komponenten gewährleistet.

+++++

2.13 Singleton

Das Entwurfsmuster Singleton ist ein Erzeugungsmuster mit dem sichergestellt wird, dass in einem System nur ein einziges Exemplar einer Klasse erzeugt werden kann. Alle Objekte, die dieses Singleton-Objekt benötigen (Client-Objekte), verwenden ein und die selbe Referenz auf das Singleton-Objekt. Erreicht wird dies, indem andere Objekte keine Möglichkeit haben, ein Singleton-Objekt zu erzeugen. Dazu werden alle Konstruktoren einer Singleton-Klasse mit dem Zugriffsmodifizierer `private` gekennzeichnet, so dass andere Klassen keinen Zugriff auf diese Konstruktoren haben. Objekte, die das Singleton verwenden möchten, wenden sich nun direkt an eine bestimmte Klassenmethode der Singleton-Klasse, die Methode `getInstance()`. Diese Methode liefert eine Referenz auf das einzige existierende Singleton-Objekt zurück. Diese Referenz sollte von den Client-Objekten niemals zwischengespeichert, sondern bei jedem Zugriff erneut durch einen Aufruf der Methode `getInstance()` ermittelt werden. Wird diese Referenz zwischengespeichert, so kann das zu Problemen nach der Zerstörung eines Singleton-Objekts führen. Wird das Singleton-Objekt aus dem Speicher entfernt, bleiben alle Client-Objekte, die diese Referenz zwischengespeichert haben, mit einer ungültigen Referenz zurück. In Programmiersprachen wie Java, die eine eigene Speicherverwaltung haben, ist dies zwar kein Problem. Es hat sich jedoch als guter Stil herausgestellt eine Referenz auf ein Singleton-Objekt nie zwischenzuspeichern.

Die Erzeugung des einzigen Exemplars der Singleton-Klasse übernimmt das Singleton selbst. Hierfür gibt es zwei unterschiedliche Vorgehensweisen. Bei der ersten Variante, wie sie in [GoF95] zu finden ist, wird das Singleton-Objekt erst erzeugt, wenn ein Client-Objekt zu erstenmal die Methode `getInstance()` aufruft. Bei der zweiten Methode, u.A. in [Gra98] zu finden, wird das Singleton-Objekt beim Laden der Klasse erzeugt, unabhängig davon ob das Singleton jemals gebraucht wird, oder nicht. Um die Vor- und Nachteile beider Methoden besser verstehen zu können, soll nun zuerst der Quellcode einer typischen Singleton-Klasse vorgestellt werden.

```
class ToolTipManager {

  private static ToolTipManager instance;

  private ToolTipManager () {
  }

  public static ToolTipManager getInstance() {
    if (instance == null) { instance = new ToolTipManager (); }
    return instance;
  }

  public void operation() {
    // eigentliche Funktionalität des Singleton
  }

}
```


© Goll

Das Beispiel zeigt, dass die Singleton-Klasse zur Verwirklichung der Singleton-Funktionalität eine Referenz auf das Singleton-Objekt beinhalten muss (`instance`). Diese Referenz muss eine Klassenvariable sein (`static`), da auf diese in der Klassenmethode `getInstance()` zugegriffen wird. Die Methode `getInstance()` muss eine Klassenmethode sein, da Client-Objekte auf diese Methode zugreifen, bevor überhaupt ein Singleton-Objekt existiert. Ein weiterer Grund dafür ist, dass die Referenz auf ein Singleton-Objekt im Allgemeinen nicht von Client-Objekten zwischengespeichert wird, sondern jedesmal erneut durch die Methode `getInstance()` erfragt wird. Ein Aufruf der Methode `operation()` sieht damit folgendermaßen aus:

```
ToolTipManager.getInstance().operation();
```

Hierbei wird über den Klassennamen `ToolTipManager` auf die Klassenmethode `getInstance()` zugegriffen, um eine Referenz auf das Singleton-Objekt zu erhalten. Über diese Referenz wird dann die Instanzmethode `operation()` aufgerufen, die die eigentliche Funktionalität des Singleton bereitstellt.

Der interessanteste Teil einer Singleton-Implementierung ist jedoch der Rumpf der Methode `getInstance()` selbst, sowie die Deklaration der Referenz `instance`. An diesen beiden Stellen unterscheiden sich die oben genannten beiden Varianten zur Erzeugung des Singleton-Objekts. Die hier vorgestellten Variante erzeugt, wie in [GoF95], das Singleton-Objekt erst dann, wenn die Methode `getInstance()` aufgerufen wird. Der Vorteil bei dieser Variante liegt darin, dass die Objekterzeugung wirklich nur dann erfolgt, wenn tatsächlich ein Client-Objekt das Singleton-Objekt benötigt. Der Nachteil ist eine leichte Verschlechterung der Performance bei weiteren Zugriffen auf die Methode `getInstance()`, da jedesmal der Vergleich `(instance == null)` ausgeführt werden muss, der nach dem ersten Aufruf sowieso immer negativ ausfällt.

Ist der (fast immer überflüssige) Vergleich und die damit einhergehende Verschlechterung der Performance unerwünscht, so kann auf die zweite Variante des Singleton Musters [Gra98] zurückgegriffen werden. Hier erfolgt die Erzeugung des Singleton-Objekts "statisch", d.h. sofort bei Programmstart oder im Fall von Java beim Laden der Klasse durch den Klassenlader. Der Vorteil bei dieser Variante ist, dass der o.g. Vergleich in der Methode `getInstance()` gänzlich entfällt. Folgendes Beispiel zeigt diese Variante:

```
class ToolTipManager {

  private static ToolTipManager instance = new ToolTipManager ();

  private ToolTipManager () {
  }

  public static ToolTipManager getInstance() {
    return instance;
  }
```

```
  public void operation() {
    // eigentliche Funktionalität des Singleton
  }

}
```

Es ist zu sehen, dass sich die Deklaration der Klassenvariablen `instance` von der ersten Variante unterscheidet. Es wird sofort ein Exemplar der Klasse `ToolTipManager` erzeugt und nicht erst beim Aufruf von `getInstance()`. Somit vereinfacht sich auch der Rumpf der Methode `getInstance()`. Der Vergleich entfällt und die Referenz auf das Singleton-Objekt kann sofort zurückgegeben werden. Die Performance leidet so nicht, allerdings wird unnötig Speicher verbraucht für den Fall, dass das Singleton-Objekt nie gebraucht wird.

Es ist wie so oft zwischen Performance und Speicherbedarf abzuwägen und die richtige Variante für die aktuelle Situation auszuwählen. Ist nicht sicher, ob das Singleton-Objekt überhaupt jemals benötigt wird oder ist geringer Speicherverbrauch wichtiger als eine höhere Performance, so ist sicherlich die Variante aus [GoF95] geeigneter. Wird das Singleton-Objekt in jedem Fall gebraucht oder ist eine höhere Performance wichtiger als geringer Speicherbedarf, so ist die Variante aus [Gra98] die bessere.

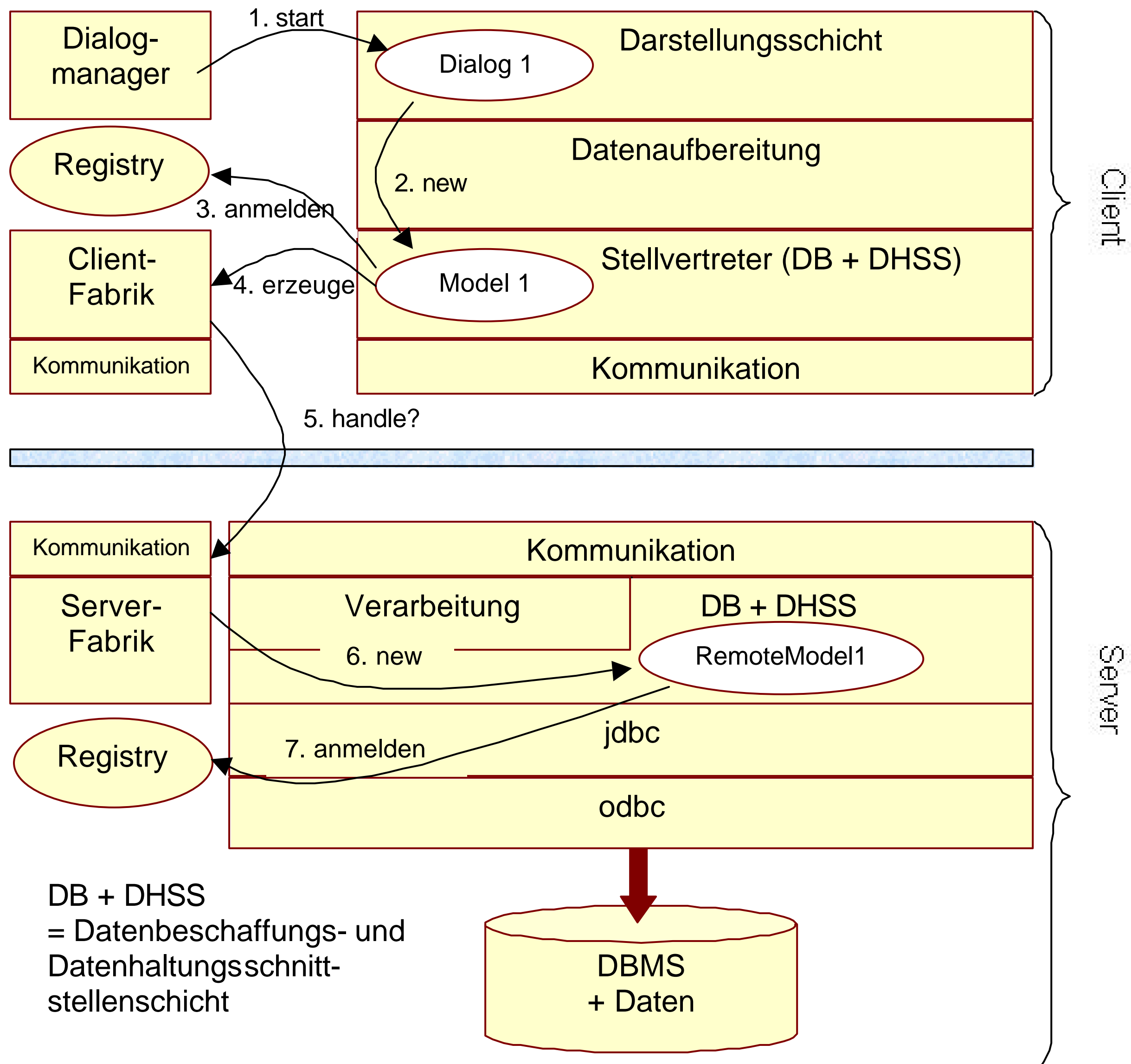
Bestehen innerhalb eines Systems jedoch Abhängigkeiten zwischen mehreren Singleton-Objekten in der Form, dass die Reihenfolge der Objekterzeugung relevant ist, ist die zweite Variante [Gra98] des Entwurfsmusters ungeeignet. Es kann keine Aussage darüber gemacht werden, welche Singleton-Objekte wann und vor allem in welcher Reihenfolge vom Laufzeitsystem geladen und damit erzeugt werden. Soll die Reihenfolge der Erzeugung also gesteuert werden, so muss die Variante aus [GoF95] gewählt werden.

Es gibt noch weitere Varianten des Singleton-Musters. Zum Beispiel wäre es denkbar, die Anzahl der Singleton-Objekte nicht auf eins zu begrenzen, sondern auf eine beliebige andere Anzahl. Diese Idee wird auch im Entwurfsmuster Object Pool [Gra98] eingesetzt, um eine bestimmte Anzahl von Objekten zu verwalten und wiederzuverwenden.

© Goll

2.14 Abstrakte Fabrik

Beginnen wir mit einem Schichtenmodell. In diesem Modell sollen immer nur die Objekte in den Schichten aktiv sein, die unbedingt zur Erfüllung der Aufgabe benötigt werden. Zur Schaffung von Objekten dienen hierabstrakte Fabriken.



 © Goll

Beispiel für eine abstrakte Fabrik in Java

Die Abstrakte Fabrik soll gleich anhand eines konkreten Beispiels betrachtet werden. Dabei wird auf die zu Anfang völlig unverständlichen und formalen Umschreibungen, wie sie in Büchern über Entwurfsmuster zu finden sind, verzichtet.

Es soll ein Informationssystem entwickelt werden, das Daten in einer Datenbank verändern, einfügen und löschen kann. Das Informationssystem soll sowohl als Einzelplatzsystem, als auch als Client-Server-System zur Verfügung gestellt werden. Wenn das Informationssystem in der Client-Server-Konfiguration läuft, sollen von allen Clients die gemeinsamen Daten auf dem Server benutzt werden. Denken wir zuerst einfacher halber an eine Adressverwaltung. Die Adressverwaltung kann privat oder geschäftlich benutzt werden. Wird sie privat genutzt werden, so wird auf den lokalen Daten auf dem Rechner gearbeitet. Wird sie geschäftlich genutzt, so benutzt sie die Daten des Servers. Es soll möglich sein, bei laufender Anwendung zwischen der privaten und geschäftlichen Nutzung umzuschalten. Es soll weiterhin möglich sein, das System nur als Einzelplatzsystem oder als Client-Server-System zu benutzen. Eine weitere günstigere Variante soll auch noch in Betracht gezogen werden, bei dieser soll keine Datenbank, sondern Files zur Speicherung der Daten verwendet werden.

Wenn geschickt programmiert wird, müssen lediglich die Klassen, die auf die unterschiedlichen Speichermedien (lokale Datenbank, entfernte Datenbank oder lokales Filesystem) zugreifen, in den unterschiedlichen Varianten programmiert werden. Der Rest des Systems ist für alle Varianten gleich. Wie das gemacht wird, soll das nachfolgende Diagramm veranschaulichen:

© Goll

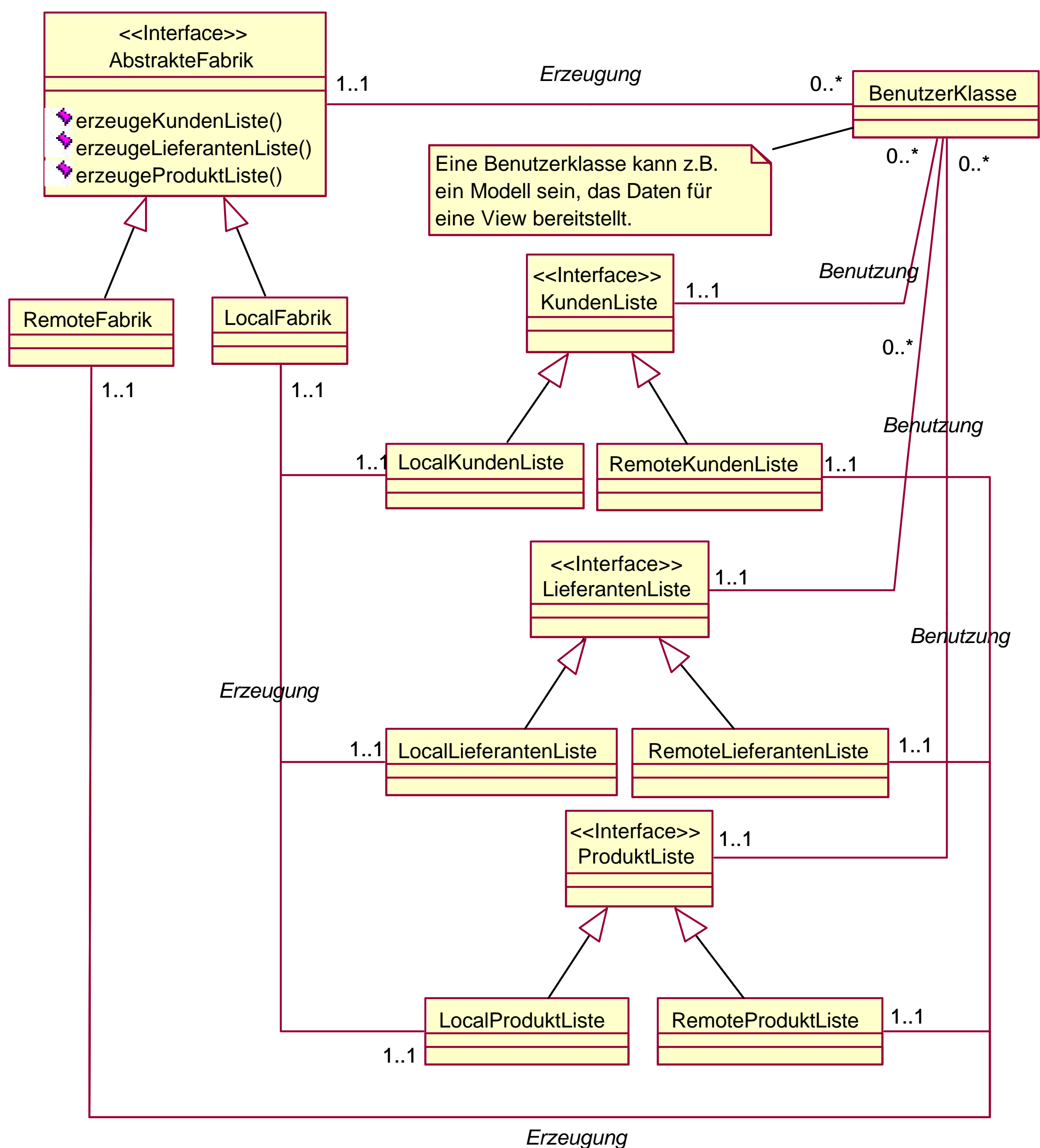


Bild 2-1 Klassendiagramm für ein Erzeugungsmuster Abstrakte Fabrik

Das Interface AbstrakteFabrik

In diesem Interface wird für jedes unterschiedliche Objekt, das von der Fabrik erzeugt werden soll eine Methode zur Verfügung gestellt. Das Interface muss erweitert werden, wenn die abstrakte Fabrik zusätzliche Objekte erzeugen soll. Damit müssen auch alle Klassen, die dieses Interface implementieren, erweitert werden. Wie dieser Nachteil umgangen werden kann, wird in einer weiteren Variante diskutiert.

© Goll

Die RemoteFabrik

Die `RemoteFabrik` implementiert die Schnittstelle `AbstrakteFabrik`. Sie sorgt für die Erzeugung der Remote-Objekte (`RemoteKundenListe`, `RemoteLieferantenListe`, `RemoteProduktListe`).

Die LocalFabrik

Die `LocalFabrik` implementiert ebenso die Schnittstelle `AbstrakteFabrik`. Sie ist für die Erzeugung der lokalen Objekte für die Einzelplatzvariante zuständig. Die `LocalFabrik` implementiert also die Erzeugung von Objekten der Klassen `LocalKundenListe`, `LocalLieferantenListe`, `LocalProduktListe`.

Die Interfaces KundenListe, LieferantenListe und ProduktListe

Diese Interfaces legen die Methoden fest, die eine Implementierung von Listen aus client-seitiger Sicht auf jeden Fall haben muss. Es werden die Methoden festgelegt, die alle verschiedenen Implementierungen einer Datenliste, ob Remote oder Lokal auf jeden Fall haben müssen.

Die „Lokalen Klassen“

Die „Lokalen Klassen“ sind diejenigen Klassen, die die Daten für das Einzelplatzsystem zur Verfügung stellen. In dem obigen Diagramm sind dies die `LocalKundenListe`, die `LocalLieferantenListe` und die `LocalProduktListe`. Diese Klassen implementieren das entsprechende Interface in der Ausprägung für die Einzelplatzvariante.

Die „Remote Klassen“

Die „Remote Klassen“ sind diejenigen Klassen, die die Daten für das Client-Server-System zur Verfügung stellen. Im obigen Diagramm sind dies die `RemoteKundenListe`, die `RemoteLieferantenListe` und die `RemoteProduktListe`. Diese Klassen implementieren das entsprechende Interface in der Ausprägung der Client-Server-Variante.

Je nach dem, in welchem Modus das Informationssystem läuft, wird entweder die `RemoteFabrik` oder die `LocalFabrik` instantiiert. Angesprochen werden beide über die gemeinsame Schnittstelle.

```
AbstrakteFabrik refFabrik;

if(modus.equals("Einzel"))
   refFabrik = new LocalFabrik();
else
   refFabrik = new RemoteFabrik();
```

Über die Referenz `refFabrik` können dann die einzelnen konkreten Listen erzeugt werden. Das folgende Codestück demonstriert die Erzeugung von den drei unterschiedlichen Listen:

© Goll

```
KundenListe refKunden = refFabrik.erzeugeKundenListe();
LieferantenListe refLieferant = refFabrik.erzeugeLieferantenListe();
ProduktListe refProdukt = refFabrik.erzeugeProduktListe();
```

Die Methode `erzeugeKundenListe()` gibt dabei als Rückgabewert den Schnittstellentyp `KundenListe` zürück. Die anderen beiden Methoden entsprechend `LieferantenListe` und `ProduktListe`. Welche konkreten Listen nun erzeugt wurden, hängt davon ab, auf welche konkrete Fabrik die Referenz `refFabrik` zeigt. Die BenutzerKlassen, die nun die verschiedenen Daten benötigen, können nun über die Referenzen `refKunden`, `refLieferant` bzw. `refProdukt` mit den Daten arbeiten, ohne zu wissen, ob sie mit Objekten der Einzelplatzvariante, oder mit Objekten der Client-Server-Variante reden.

Nachteilig an der obigen Variante ist, dass wenn das System erweitert werden soll, das Interface `AbstrakteFabrik` erweitert werden muss. Eine stabilere Variante würde verlangen, dass nicht für jede neue Liste eine neue Methode dem Interface hinzugefügt werden muss, sondern dass die komplette abstrakte Fabrik unverändert bleiben kann. Dies kann durch folgende Abwandlung erreicht werden, und ist vor allem in Java sehr einfach und sicher zu implementieren:

Die Abänderung ist wie in **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen ist sehr gering. Es hat sich lediglich das Interface der Abstrakten Fabrik geändert. Das Interface stellt nur noch eine Methode mit folgender Schnittstelle zur Verfügung:

```
interface AbstrakteFabrik
{
   Object erzeuge(String className) throws IllegalAccessException,
                                           ClassNotFoundException,
                                           InstantiationException;
}
```

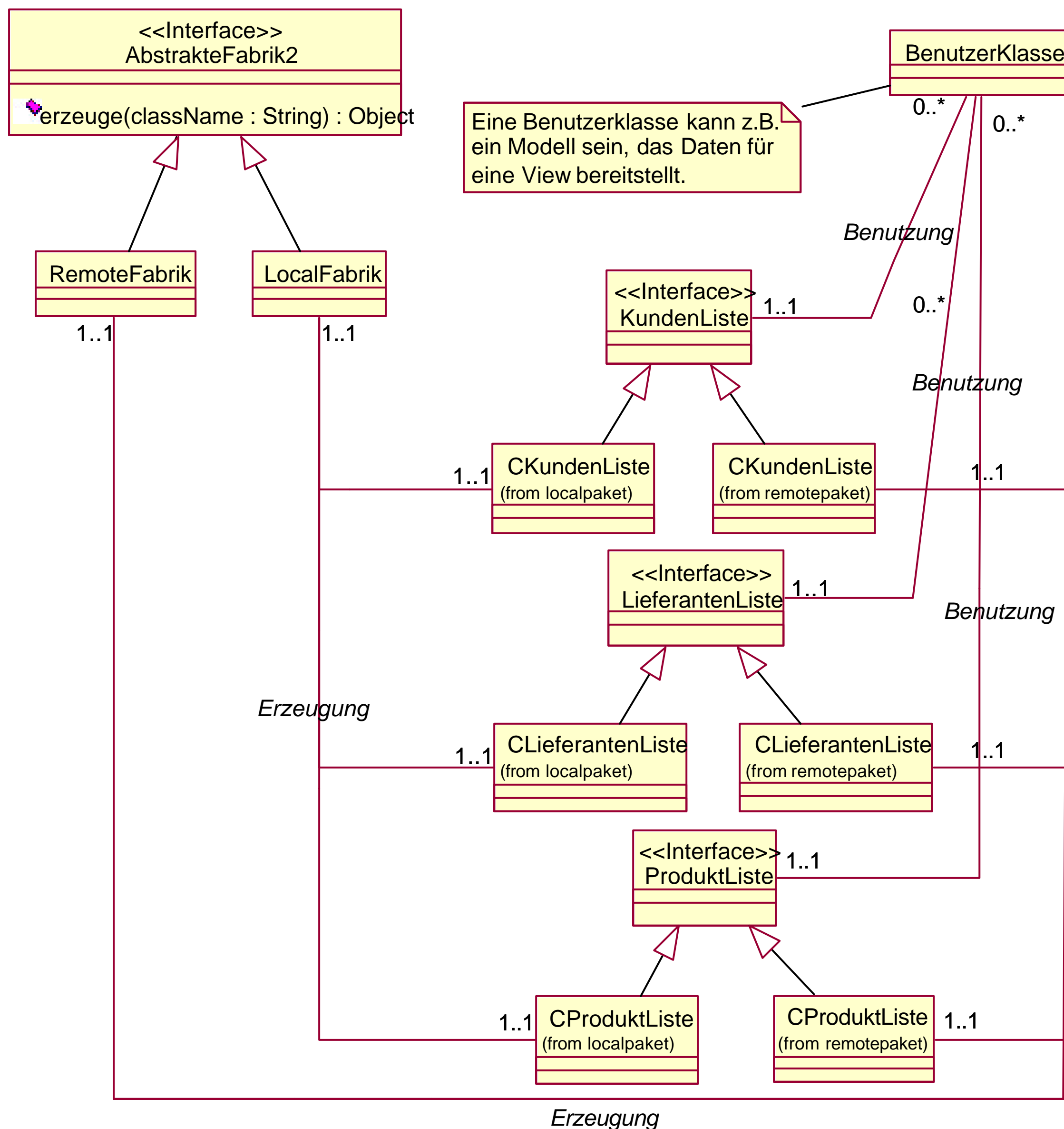


Bild 2-2 Abgewandeltes Klassendiagramm des Erzeugungsmusters Abstrakte Fabrik

Die konkreten Fabriken, die dieses Interface implementieren, haben die Aufgabe, aus dem übergebenen Klassennamen entweder eine Instanz für den lokalen Zugriff oder eine Instanz für den Remote-Zugriff zu erzeugen. Eine Implementierung einer konkreten Fabrik könnte wie folgt aussehen:

```
class LocalFabrik implements AbstrakteFabrik
{
   public Object erzeuge(String className)
                        throws IllegalAccessException,
                               ClassNotFoundException,
                               InstantiationException
   {
      String completeName = "localpaket."+className;
```


© Goll

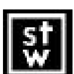
```
        Class instance = Class.forName(completeName);
        // Neues Objekt einer Listenklasse erzeugen
        Object obj = instance.newInstance();
        return obj;
    }
}
```

Das Erzeugen einer Datenliste aus der Benutzerklasse heraus könnte folgendermaßen aussehen:

```
KundenListe (KundenListe) refKunden =
                             refFabrik.erzeuge("CKundenListe");
LieferantenListe (LieferantenListe) refLieferant =
                             refFabrik.erzeuge("CLieferantenListe");
ProduktListe (ProduktListe) refProdukt =
                             refFabrik.erzeuge("CProduktListe");
```

Zu beachten ist in **Fehler! Verweisquelle konnte nicht gefunden werden.**, dass die Klassen, die die verschiedenen Listen implementieren, zwar gleich heißen, aber in unterschiedlichen Paketen liegen. Wenn das System nun um eine Auftragsliste erweitert werden soll, muss lediglich ein Interface für die neue Auftragsliste vereinbart werden und dieses Interface in der Einzelplatz- und in der Client-Server-Variante implementiert werden. An den abstrakten Fabriken ändert sich nichts mehr, die Schnittstellen bleiben stabil!

© Goll



© Goll

3 Architectural and design patterns

In literature you can not find a clear distinction between architectural and design patterns. In general it can be said that architectural patterns describe an overall system structure and often use several design patterns to accomplish their task. The following definitions differentiate architectural patterns from design patterns:

Architectural patterns *are suitable for the* ***coarse grained*** *structuring of software systems. They divide up a system into sub systems (components), specify the responsibilities of the sub systems found* and *define the rules for the organisation of the relations between the sub systems.*

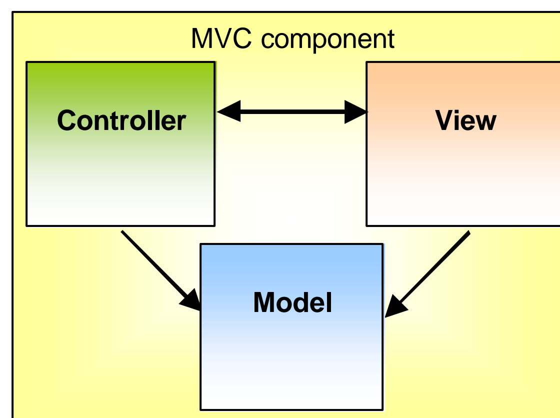
Design patterns *are suitable for the* ***fine grained*** *structuring of special system parts. They either refine sub systems or components or describe the organisation of the relations between sub systems or components, which must obey the rules of the underlying architectural pattern.*

As an example of an architectural pattern in the next chapter the Model-View-Controller (MVC) pattern is introduced. The **MVC pattern specifies the structure of an interactive software system** and has often been used for the architecture of graphical user interfaces.

3.1 Architectural pattern MVC

The core of interactive software systems is based on functional requirements on the system. However the **user interfaces** are often **subject to change**. The look and feel of an application often has to be adopted to the individual requirements of a customer. Therefore an architecture is required which allows changes at the user interfaces without a great impact on the application specific functionality. One architecture which meets this requirement is the Model View Controller (MVC) pattern[11].

The MVC pattern divides an application into 3 parts as shown in the next figure:



[11] The MVC pattern has been developed from Trygre Reenskang. See also Reenskang (1996).

© Goll

FIGURE 3-1 MVC component with the parts Model, View and Controller

An application consists normally of many MVC-components[12]. View and controller of a MVC triangle form together the parts representing the user interface. The arrows in FIGURE 3-1 describe the “uses” relationship and the 3 parts (Model, View, Controller) are classes in the object-oriented paradigm. The different classes of a MVC component have the following tasks to fulfil:

- The **model** realises the **core functionality** of the application and informs dependent classes when the data has changed.
- The **view** is responsible for the **data representation** to the user. It generates and initialises the corresponding controller and observes the model.
- The **controller** interprets the **user input** and translates these events into method calls of the model and representation requests to the view.

3.2 Design patterns inherent in the MVC pattern

Design patterns are used for the fine grained structuring of system parts. Often a lot of **design patterns work together to build up a framework** for the architecture of a system. The design patterns **Observer**, **Factory Method, Template Method**, **Strategy** and **Composite** are used in the MVC architectural pattern. These patterns will be described on the next pages.

[12] A MVC component is also called MVC triangle.

Design pattern Observer

The **Observer pattern** is used for the **update change mechanism between model and view**. The data of one model can be represented in more than one view. Therefore when the data in the model changes, all views have to be updated. Because the views are subject to change whereas the model remain stable in its functionality it is meaningful that the model does not know the kind and amount of views interested in representing its data. FIGURE 3-2 shows the two abstract classes `Observer`[13] and `Observable`. The class `Model` inherits from the abstract class `Observable` and the two classes `View1` and `View2` inherit from the class `Observer`.

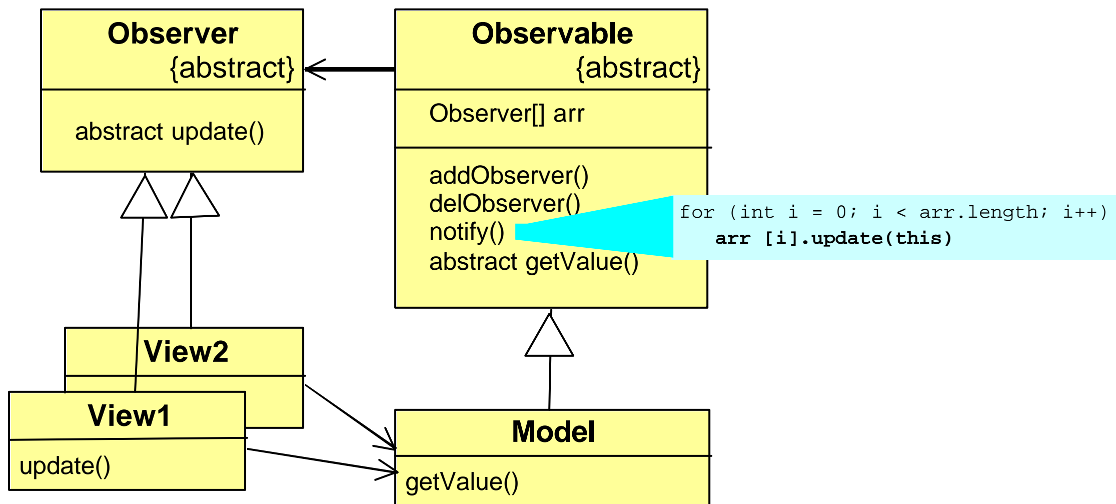


FIGURE 3-2 The Observer pattern is used for the update change mechanism of Model and View

When the views are generated they register themselves as observers of the model by calling the model's method `addObserver()`. Whenever data changes in the model, the model calls its method `notify()`. The method `notify()` calls for each observer in the array `arr` the method `update()`. Now the individual views are informed that data changed in the model and they can decide if it is necessary to get the new data by calling the method `getValue()` of the model.

Design pattern Factory Method (virtual constructor)

The Factory method is a generation pattern. In FIGURE 3-3 the abstract classes `View` and `Controller` can be seen. A view generally generates its corresponding controller. But the abstract class `View` cannot know which specific controller a sub-class wants to generate. This problem is solved by the factory method.

[13] Because there is only one abstract method in the class `Observer` it can also be implemented in a simple interface `Observer`.

© Goll

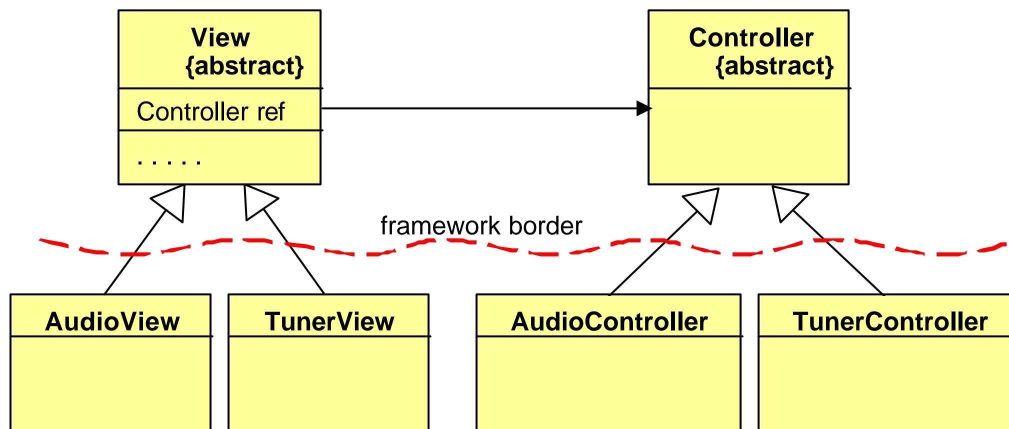


FIGURE 3-3 View and controller relationship in the Factory Method

The generation of the specific controller is not done in the abstract class `View`, but in the deferred classes `AudioView` and `TunerView`. Therefore `AudioView` writes in its constructor `ref = new AudioController()` and `TunerView` writes in its constructor `ref = new TunerController()`.

The factory method is often used in frameworks, because the existing abstract classes in the framework cannot know which specific sub-classes have to be generated in an individual case. The curved line in FIGURE 3-3 indicates the framework border which separates the framework classes from the user defined application specific classes.

Design pattern Template Method

In the pattern **Template Method** the **skeleton of an algorithm is implemented in an abstract super class** and **single steps are delegated to** the more specific **sub classes**. The Template Method is nearly always inherent in frameworks, because it is the mechanism to implement as much as possible in the framework classes and delegate specific things to the application specific sub classes. A view for example - independent if it is a button view or a text field view - always needs the methods `increase()` and `decrease()`. The algorithm is always the same. But the contents of the view area of a button is different from those of a text field. FIGURE 3-4 shows the usage of the Template Method. The abstract class `View` implements the methods `increase()` and `decrease()`. Both methods use the method `repaint()` to fill up the new area. But the method `repaint()` is only implemented in the sub classes.

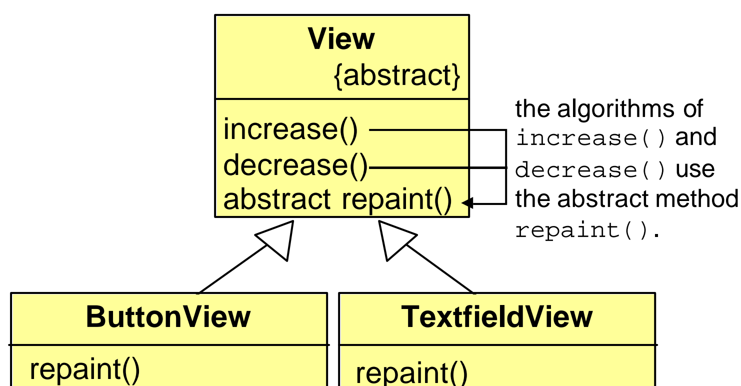


FIGURE 3-4 The abstract class `View` uses the Template Method

Design pattern Strategy

The Template Method uses inheritance to vary parts of an algorithm whereas the **Strategy pattern uses delegation to vary a complete algorithm**. FIGURE 3-5 shows the class `TextView` which uses a `TextController` for the interpretation of the user input.

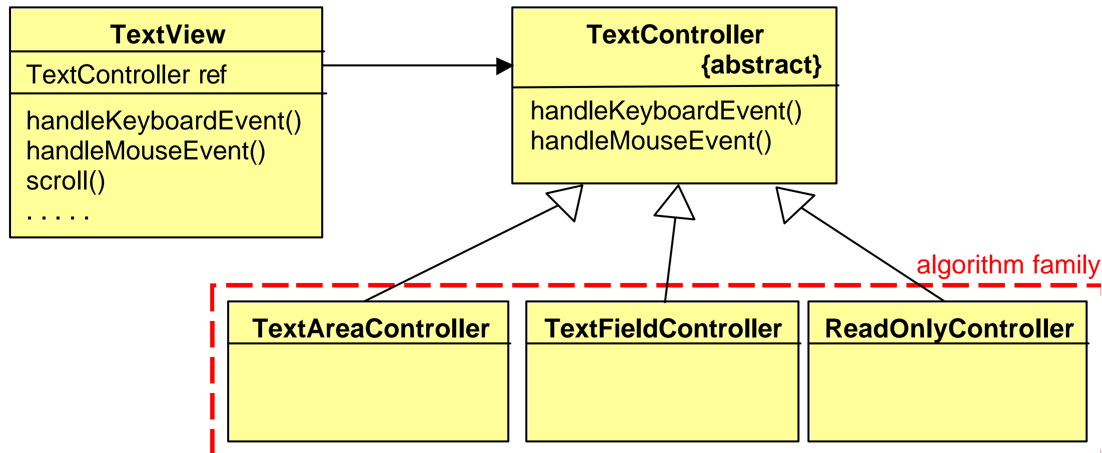


FIGURE 3-5 One text view can use different algorithms for interpretation of user input

But the algorithm of the interpretation varies dependent on the usage of the text view: A text view with an editable area of one input line uses an object of class `TextFieldController`, a text view with an editable area of various input lines uses an object of the class `TextAreaController` and a text view with a read only area uses an object of the class `ReadOnlyController`. The assignment of the controller to the view can also change dependent on the state at runtime. The **strategy pattern extracts an algorithm into sub classes** whereas the **interface of the algorithm is defined in an abstract super class**. An object which wants to use an algorithm of the algorithm family can easily choose and exchange the algorithm when necessary.

Design pattern Composite

The **pattern Composite comes into play when** a **composite object is used in the same manner as the parts the composite object is build up**. This means that for example a window which contains buttons, text areas, labels, etc. has the same methods and the same behaviour as the individual elements. In this case the classes can be arranged like in FIGURE 3-6.

© Goll

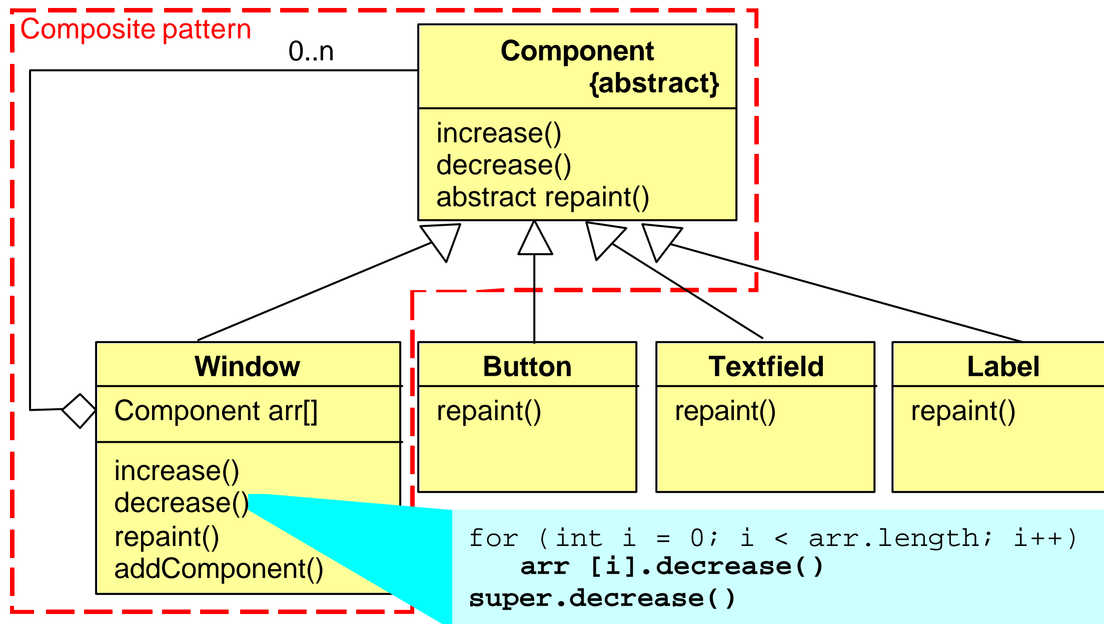


FIGURE 3-6 The Composite pattern for the arrangement of views

The patterns covered in this chapter were not described in details. More information on design patterns can be found in the books of Gamma (1995) and Buschmann (1998), which are indeed highly recommendable.

© Goll

4 Ein zusammenhängendes Anwendungsbeispiel

Folgendes Beispiel zeigt die Verwendung der drei Entwurfsmuster:

- Schnittstellen und Varianten
- Delegation
- Singleton
-

4.1 Anforderungen

Es soll ein **Versandsystem** entworfen werden. Es gibt **Briefe** und **Pakete**. Beide sollen über einen zentralen **Briefkasten** versendbar sein.

4.2 Analyse

Entfällt :-)

4.3 Entwurf

Es wird eine Klasse `Versendbar` entworfen, die Methoden zum Ermitteln des Absenders und des Empfängers hat. Davon abgeleitet gibt es eine Klasse `Brief` und eine Klasse `Paket`, die die Schnittstelle von `Versendbar` jeweils um eine klassenspezifische Schnittstelle erweitern. Der zentrale Briefkasten wird als Singleton in der Klasse `Briefkasten` implementiert.

4.4 Implementierung

Im Folgenden wird die Implementierung des Versandsystems gezeigt.

© Goll

Klasse `Versendbar`:

```
public abstract class Versendbar {

  private String empfaenger = "";
  private String absender = "";

  public void adressieren(String empfaenger, String absender) {
    this.empfaenger = empfaenger;
    this.absender = absender;
  }

  public String empfaenger() {
    return this.empfaenger;
  }

  public String absender() {
    return this.absender;
  }

  public void versenden() {
    Briefkasten.getInstance().versende(this);
  }

}
```

Klasse `Brief`:

```
public class Brief extends Versendbar {

  private String text = "";

  public void beschriften(String text) {
    this.text = text;
  }

  public String text() {
    return this.text;
  }

}
```

Klasse `Paket`:

```
public class Paket extends Versendbar {

  private String inhalt = "";

  public void befuellen(String inhalt) {
    this.inhalt = inhalt;
  }

  public String inhalt() {
    return this.inhalt;
  }

  }
```


© Goll

Klasse `Briefkasten`:

```
public class Briefkasten {

  private static Briefkasten instance = new Briefkasten();

  private Briefkasten() {
  }

  public static Briefkasten getInstance() {
    return instance;
  }

  public void versende(Versendbar v) {
    System.out.println("Versende von '" + v.absender() +
                       "' an '" + v.empfaenger() + "' ...");
  }

}
```

Klasse `Versandsystem`:

```
public class Versandsystem {

  public static void main(String[] argv) {
    new Versandsystem();
  }

  public Versandsystem() {

    Versendbar v1 = new Brief();
    Versendbar v2 = new Paket();

    v1.adressieren("John Doe", "Jane Doe");
    v2.adressieren("GI Joe", "GI Jane");

    v1.versenden();
    v2.versenden();

  }

}
```