Design-Patterns im Überblick

© Michael Inden, entnommen und angepasst aus dem Buch „Der Weg zum Java-Profi“, dpunkt.verlag

Bereits erprobte und regelmäßig eingesetzte allgemeingültige Verfahren zur Lösung eines Entwurfsproblems werden als *Entwurfsmuster* oder auch *Designpattern* bezeichnet und gewinnen seit einigen Jahren immer mehr an Bedeutung.

Die von der GoF vorgenommene Gliederung in die drei Kategorien Erzeugungsmuster, Strukturmuster und Verhaltensmuster übernehme ich. Die Vorstellung eines Musters beginnt mit einer kurzen Beschreibung und Motivation für dessen Einsatz. Daran schließt sich eine Darstellung der zugrunde liegenden Struktur in Form eines UML-Diagramms sowie eine Beschreibung des jeweiligen Musters anhand von Praxisbeispielen an.

Erzeugungsmustern vereinfachen den Konstruktionsprozess von Objekte. **Strukturmuster** helfen, Funktionalitäten flexibel erweitern und anpassen zu können. Durch den Einsatz von **Verhaltensmustern** strukturiert und vereinfacht man komplexe Abläufe oder Interaktionen zwischen Objekten.

18.1.1 Erzeugungsmethode

Beschreibung und Motivation

Man kann sich die ERZEUGUNGSMETHODE etwa wie eine Bestellung in einem Restaurant vorstellen. Man sagt, welches Gericht man essen möchte, nennt jedoch nicht jede Zutat. Auch die Art der Zubereitung bleibt verborgen.

Auf ein Programm übertragen bedeutet dies Folgendes: Wenn der Konstruktionsprozess komplex oder fehleranfällig ist, weil beispielsweise viele Parameter übergeben werden müssen, so wird eine Erzeugung durch einen Konstruktoraufruf von außen vermieden. Stattdessen wird die Klasse selbst für die Objekterzeugung verantwortlich gemacht und stellt zur Objektkonstruktion eine spezielle, statische Konstruktionsmethode, die Erzeugungsmethode, bereit. Um Objekterzeugungen von außen ohne Aufruf der Erzeugungsmethode sicher zu verhindern, dürfen lediglich private Konstruktoren bereitgestellt werden.

Struktur

Betrachten wir eine vereinfachte Klasse DrawingPad, die entweder als Applet oder als Applikation gestartet werden kann. Dies wird durch den Wert des Konstruktorparameters runAsApplet entschieden. Die Signatur des Konstruktors sieht wie folgt aus:

```
DrawingPad(final String title, final boolean runAsApplet,
final JApplet applet, final ConsoleParams appParams)
```

Die beiden Parameter applet bzw. appParams sind jeweils nur in einem der beiden Modi sinnvoll zu belegen. Für Applikationen bleibt daher der applet-Parameter unbe legt (null). Für Applets wird für appParams der Wert null übergeben. Damit käme es beispielsweise zu folgenden Konstruktoraufrufen:

```
new DrawingPad("Application", false, null, consoleParams);
new DrawingPad("Applet", true, applet, null);
```

Betreiben wir eine kurze Analyse, um einige Probleme bei der Konstruktion eines DrawingPad-Objekts aufzudecken.

1. Wirklich lesbar und verständlich sind die gezeigten Konstruktoraufrufe mit null-Werten und den booleschen Literalen true und false nicht.
2. Existieren optionale Parameter, so steigt die Gefahr, an falscher Stelle null-Werte zu übergeben und dadurch unerwartete Initialisierungen vorzunehmen.
3. Es ist keine Kontrolle möglich, dass alle Werte konsistent übergeben werden.

Als Verbesserung bietet sich an, zwei Erzeugungsmethoden zu nutzen: Eine für die

Konstruktion von Applets und eine für Applikationen. Diese beiden Methoden bekommen die sprechenden Namen createAsApplet() und createAsApplication().
Sie werden wie folgt realisiert:

```
private static final boolean RUN_AS_APPLET = true;

public static DrawingPad createAsApplet(final String title, final JApplet applet)
{
   final ConsoleParams NO_APP_PARAMS = null;
   return new DrawingPad(title, RUN_AS_APPLET, applet, NO_APP_PARAMS);
}

public static DrawingPad createAsApplication(final String title, final ConsoleParams appParams)
{
   final JApplet NO_APPLET = null;
   return new DrawingPad(title, !RUN_AS_APPLET, NO_APPLET, appParams);
}
```

18.1.2 Fabrikmethode (Factory method)

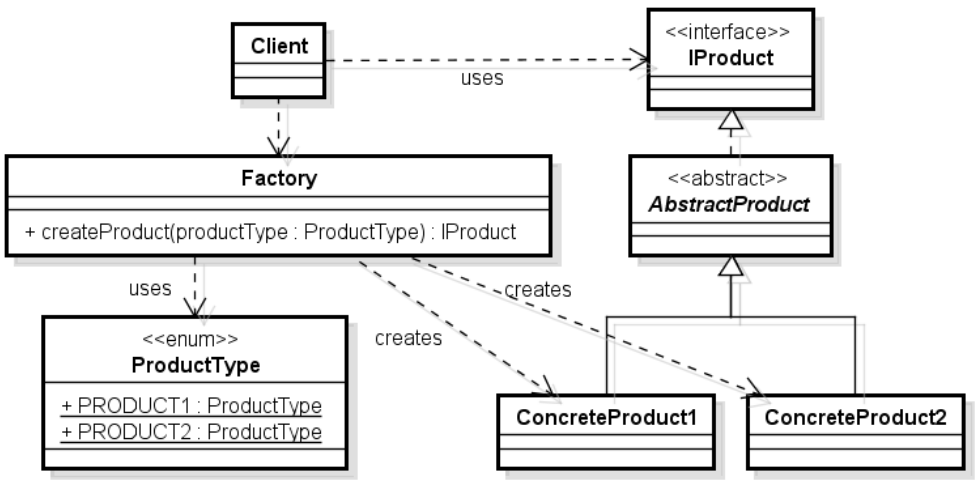
Beschreibung und Motivation

Die Idee hinter dem Muster FABRIKMETHODE ist ähnlich einer Produktion in einer realen Fabrik, die aufgrund einer Bestellung gewünschte Dinge herstellen kann. Bei einer solchen Bestellung sind nur die Artikelnummern oder Bezeichnungen der Teile, nicht aber die konkreten Ausprägungen und Realisierungen bekannt. Es findet demnach analog zum Muster ERZEUGUNGSMETHODE eine Kapselung der Objekterzeugung statt:
Objekte werden nicht direkt per Konstruktoraufruf erzeugt, sondern dieser Vorgang wird an ein spezielles Objekt, eine sogenannte Fabrik, delegiert. Die dort definierten Fabrikmethoden erzeugen Objekte verschiedenen Typs.

Der Unterschied zum Muster ERZEUGUNGSMETHODE, das den Konstruktionsprozess einer speziellen Klasse vereinfacht, liegt darin, dass beim Muster FABRIKMETHODE eine andere Klasse, die Fabrik, die Konstruktion von Objekten eines gewünschten konfigurierbaren Typs durchführt.

Struktur

Es wird eine Fabrikklasse Factory mit einer Methode createProduct(ProductType) zum Erzeugen von Objekten definiert. Dort wird anhand eines Parameters productType entschieden, welcher konkrete Typ erzeugt wird. Der gewünschte zu erzeugende Typ wird dabei entweder, wie in diesem Fall, als Parameter übergeben oder aus einer Konfiguration gelesen. Damit überhaupt verschiedene Produkte von einer Methode erzeugt werden können, müssen diese Produkte ein gemeinsames Interface IProduct, eine gemeinsame Basisklasse AbstractProduct oder beides besitzen.
Klienten arbeiten immer mit dieser Abstraktion des konkreten Produkts, wodurch es automatisch zu einer besseren Kapselung und loseren Kopplung zum verwendenden Applikationscode kommt. Abbildung 18-2 zeigt eine mögliche Realisierung als UML-Diagramm.



18.1.5 Prototyp (Prototype)

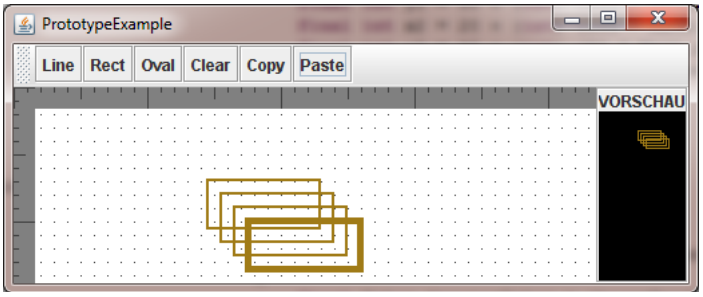
Beschreibung und Motivation

Das Vorgehen beim PROTOTYP-Muster besteht darin, basierend auf speziellen Vorlagenobjekten neue Objekte zu generieren. Man nutzt dieses Muster, wenn Instanzen einer Klasse einen großen gemeinsamen Grundstock an gleichen Werten ihrer Attribute haben und durch wenige Modifikationen gebrauchsfertig gemacht werden können. Außerdem kann der Einsatz zur Performance-Steigerung und zur ressourcenschonenden Objekterzeugung verwendet werden, wenn diese per Konstruktor zeitaufwendig ist (z. B. durch erneute Datenbankzugriffe zum Ermitteln der Werte der Attribute) und somit ein Kopieren der Ergebnisse und anschließendes Parametrieren günstiger ist.

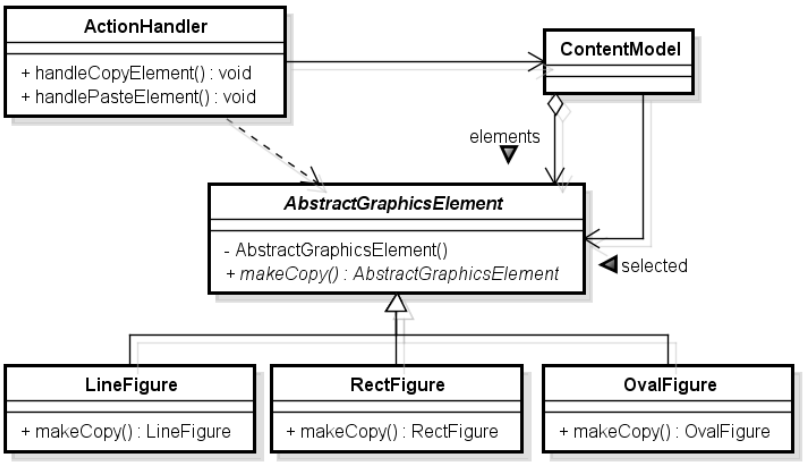
Dieses Muster besitzt zwei Varianten: Bei der *statischen* Variante dienen eine oder mehrere Kopiervorlagen (Prototypen) als Basis für neue Objekte. Bei der *dynamischen* Variante kann sogar der Typ der Kopiervorlage zur Laufzeit unterschiedlich sein.

Beispiel

Die dynamische Variante stelle ich im Folgenden am vereinfachten Beispiel eines grafischen Editors vor, der über Buttons verschiedene Aktionen, wie Figuren zeichnen, kopieren und einfügen, erlaubt. Mehrfache Kopien sind in der folgenden Abbildung für einige Rechtecke gezeigt, wobei die aktuell selektierte Figur fett dargestellt ist.



Wird der Copy- bzw. Paste-Button gewählt, so werden spezielle Methoden aufgerufen und das momentan auf der Zeichenfläche selektierte Element kopiert bzw. eingefügt. Die beteiligten Klassen und deren Zusammenwirken zeigt das Klassendiagramm.



Zur Realisierung der Copy- bzw. Paste-Aktionen werden die Methoden handleCopy-Element() und handlePasteElement() aufgerufen, um das momentan auf der Zeichenfläche selektierte Element zu kopieren. Dieses kann je nach Auswahl von unterschiedlichen Laufzeittypen sein. Damit eine einheitliche Verarbeitung möglich ist, müssen alle zu verarbeitenden Objekte einen gemeinsamen Typ besitzen. In diesem Beispiel ist dies die abstrakte Klasse AbstractGraphicsElement mit den konkreten Subklassen LineFigure, RectFigure und OvalFigure. Eine Copy- bzw. Paste-Aktion nutzt lediglich die abstrakte Basisklasse und muss dadurch die Spezialisierungen nicht kennen. Auf diese Weise können sogar beliebige, abgeleitete Elemente kopiert werden, die zum Kompilierzeitpunkt noch unbekannt sind.

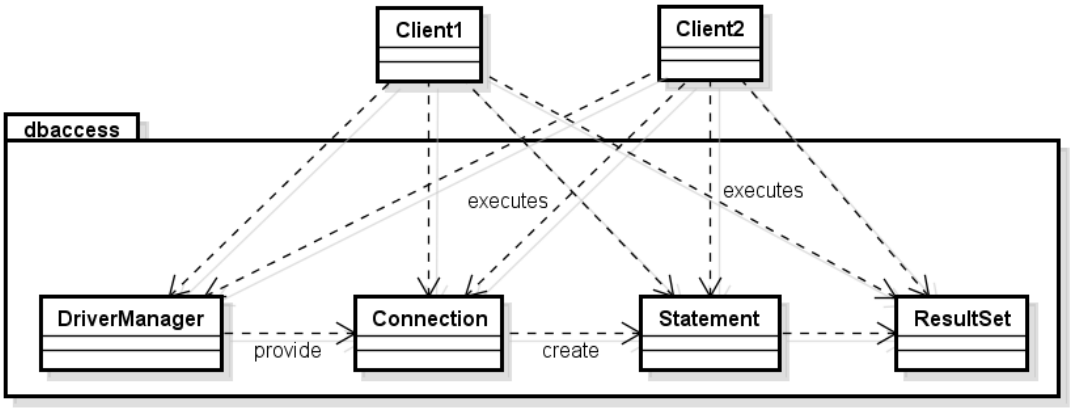
18.2.1 Fassade (Façade)

Beschreibung und Motivation

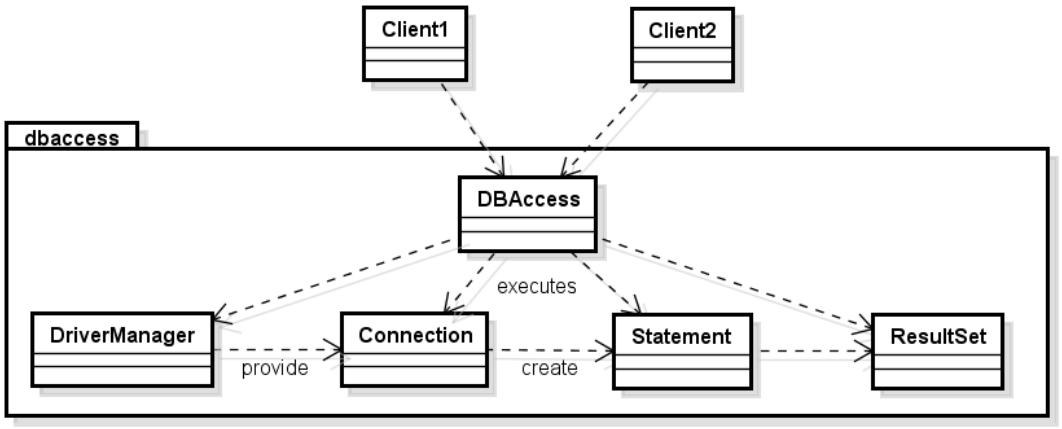
Das Entwurfsmuster FASSADE kann dabei helfen, die Komplexität eines Subsystems zu verbergen und den Zugriff darauf für externe Klassen zu vereinfachen bzw. zu steuern. Eine Fassadenklasse definiert dazu ein »High-Level«-Interface, um komplizierte, sehr feingranulare Interaktionen und Beziehungen zwischen Package-internen und -externen Klassen zu vermeiden. Bei Aufrufen durch Klienten delegiert ein Fassadenobjekt dazu Aufrufe entsprechend der Zuständigkeit an spezielle Klassen des Subsystems. Es herrscht eine gerichtete Abhängigkeit: Die Fassadenklasse kennt die Klassen des Subsystems, aber nicht umgekehrt, d. h., keine Klasse des Subsystems kennt die Fassadenklasse.

Struktur

Betrachten wir zunächst ein System mit zwei Klienten Client1 und Client2, die Datenbankzugriffe durchführen wollen. In Abbildung 18-9 erkennt man viele Abhängigkeiten von den eingesetzten Klassen. Dies liegt daran, dass die Logik und Steuerung jeweils in den Klienten erfolgt.



Mit der Anzahl der beteiligten Klienten steigen diese Abhängigkeiten immer weiter. Dies ist der Punkt, an dem man über den Einsatz des FASSADE-Musters nachdenken sollte. Führt man im Beispiel eine Fassadenklasse DBAccess ein, so lassen sich dadurch die Zugriffe auf die Datenbankzugriffsklassen strukturieren. Abbildung 18-10 zeigt dies eindrucksvoll.



Die Fassadenklasse verbirgt den Verbindungsaufbau und die Komplexität der Datenbankzugriffe vor aufrufenden Klienten. Diese nutzen nur noch eine Verwaltungsklasse und damit eine einzige Schnittstelle, die es ihnen ermöglicht, ihre Aufgaben auszuführen. Sämtliche Details der Infrastruktur sind für Aufrufer nicht mehr von Interesse. Die meisten Klassen des Subsystems können und sollten folglich Package-private definiert werden, um Implementierungsdetails nach außen auch tatsächlich zu verbergen.

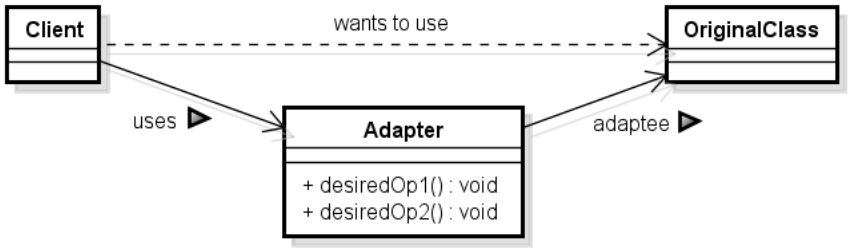
18.2.2 Adapter

Beschreibung und Motivation

Dieses Muster lässt sich sehr anschaulich am Beispiel von zwei Steckern oder Puzzleteilen motivieren, die nicht ineinander passen. Man nimmt ein Adapterstück und nutzt dieses zur Verbindung, um die nicht passenden Teile zu verbinden. Bei Software handelt man gleichermaßen: Man verwendet ein Softwarestück, das die Schnittstellen ansonsten inkompatibler Software aufeinander abbildet, um diese miteinander zu verbinden.
Der entscheidende Vorteil gegenüber einem direkten Ansprechen einer anderen Klasse ist, dass durch den Einsatz eines Adapters keine Änderungen an den Schnittstellen der bereits vorhandenen Implementierungen nötig sind. Dafür muss allerdings ein neues Softwarestück, der Adapter, entwickelt werden.

Struktur

Ein Klient Client nutzt ein Objekt vom Typ Adapter, um die Funktionalität einer vorhandenen Klasse OriginalClass mit einer für den Klienten passenden Schnittstelle verwenden zu können. Diese wird durch den Adapter zur Verfügung gestellt (vgl. Abbildung 18-11). Die Referenz auf die Originalklasse wird häufig adaptee genannt.



Beispiel

In diesem Beispiel ist eine Implementierung eines Adapters gezeigt, die eine beliebige Realisierung des Interface List<E> (etwa ArrayList<E>, LinkedList<E> etc.) mit einem Listenmodell für Swing-Komponenten kompatibel macht.

```
public static class ListToListModelAdapter extends AbstractListModel<Person>
{
   private final List<Person> personsAdaptee = new ArrayList<>();

   ListToListModelAdapter(final List<Person> persons)
   {
      this.personsAdaptee.addAll(persons);
   }

   public int getSize()
   {
      return personsAdaptee.size();
   }

   public Person getElementAt(final int i)
   {
      return personsAdaptee.get(i);
   }
}
```

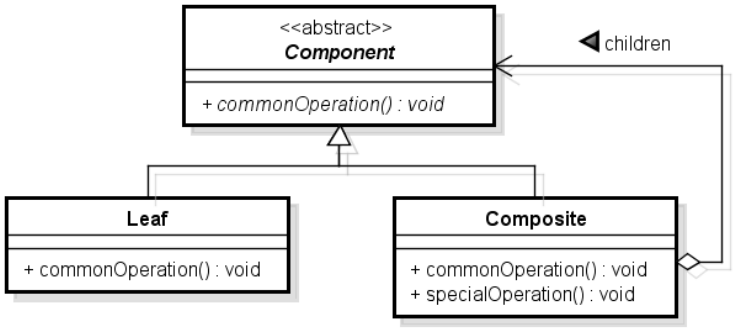
18.2.4 Kompositum (Composite)

Beschreibung und Motivation

Das KOMPOSITUM-Muster ermöglicht es, in hierarchischen Datenstrukturen sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln. Damit kann man in den meisten Fällen den Unterschied zwischen Einzelobjekten und Kompositionen außer Acht lassen.

Struktur

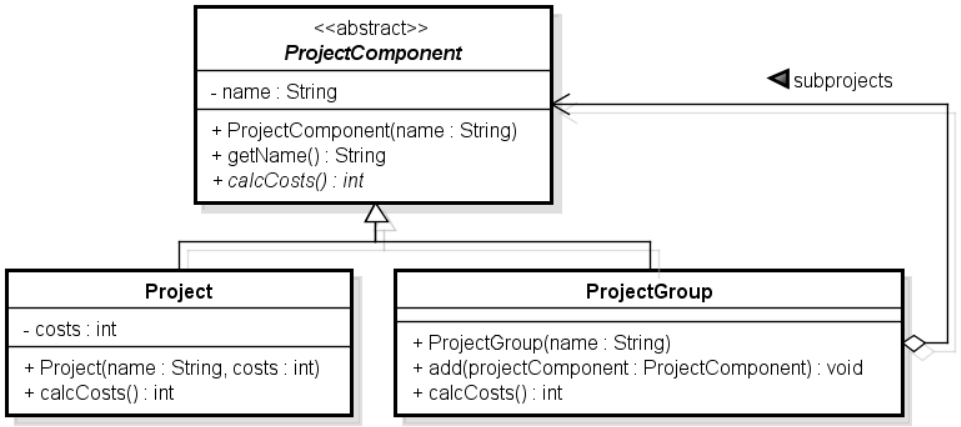
Um sowohl Einzelobjekte als auch Kompositionen einheitlich behandeln zu können, definiert eine abstrakte Basisklasse (oder alternativ auch ein Interface) Component die gemeinsame Schnittstelle für alle Akteure. Diese Schnittstelle wird jeweils sowohl von Einzelobjekten Leaf als auch vom Kompositum Composite implementiert. Das Kompositum wird in der Regel noch weitere Methoden besitzen, um beispielsweise die hierarchische Struktur aufzubauen. Abbildung 18-14 stellt das Klassendiagramm für das KOMPOSITUM-Muster dar.



Beispiel

Im Folgenden stelle ich ein vereinfachtes Beispiel aus der Praxis vor, das mithilfe des KOMPOSITUM-Musters die Projektkosten hierarchisch organisierter Projekte berechnen soll. Ziel ist es, die Kosten unabhängig von der Hierarchieebene mit dem gleichen Aufruf ermitteln zu können. Eine Projekthierarchie besteht aus Einzelprojekten (Klasse Project) oder aus Projektgruppen (Klasse ProjectGroup), die mehrere Einzelprojekte oder untergeordnete Projektgruppen verwalten können. Diese Struktur wird mithilfe des KOMPOSITUM-Musters wie folgt modelliert (Abbildung 18-15):

- Ein Einzelprojekt entspricht einem Blatt (Project).
- Eine Projektgruppe entspricht dem Kompositum (ProjectGroup). Unterprojekte werden mit der Methode add(ProjectComponent) hinzugefügt.
- Die abstrakte Basisklasse ProjectComponent definiert die gemeinsame Schnittstelle. Dazu zählt insbesondere die abstrakte Methode calcCosts().



Einzelprojekten sind bestimmte Kosten zugeordnet (als Vereinfachung im Beispiel per Konstruktorparameter übergeben). Projektgruppen ermitteln ihre Kosten durch Addition

der Kosten ihrer Unterprojekte (Einzelprojekte oder Unterprojektgruppen), die sie transparent über die Schnittstelle der Basisklasse ProjectComponent verwalten. Mit dieser Modellierung können die Kosten auf jeder Hierarchieebene berechnet werden. Die Ermittlung dieser Kosten lässt sich somit einfach rekursiv wie folgt formulieren:

```
@Override
public int calcCosts()
{
   int costs = 0;

   for (final ProjectComponent current : subprojects)
   {
      costs += current.calcCosts();
   }
   return costs;
}
```

18.3.3 Schablonenmethode (Template method)

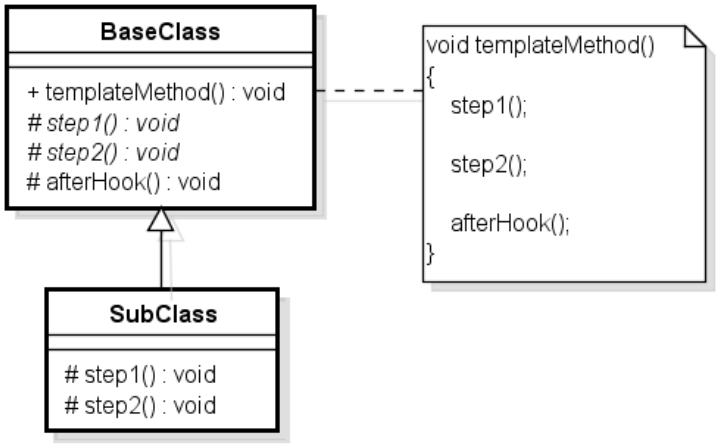
Beschreibung und Motivation

Das Muster SCHABLONENMETHODE definiert den grundsätzlichen Ablauf eines Algorithmus und erlaubt es, an einigen Stellen durch Subklassen spezielle Funktionalität einzubringen. Dazu wird ein Algorithmus zunächst in eine festgelegte Abfolge verschiedener Schritte aufgeteilt. Die Abfolge der Schritte wird in einer speziellen Methode der Basisklasse realisiert, die *Schablonenmethode* genannt wird. Sie ist in der Regel final definiert, um die grundsätzliche Abfolge vor Veränderungen zu schützen. Einige der Berechnungsschritte des Algorithmus sind in der Basisklasse noch undefiniert und werden durch abstrakte Methoden modelliert, die von Subklassen implementiert werden müssen. Möchte man Subklassen dagegen *optional* eine Veränderungsmöglichkeit anbieten, so kann man mit einer Leerimplementierung einer Methode in der Basisklasse arbeiten, die als *Hook* bezeichnet wird. Jede Subklasse kann durch Überschreiben dieser Methode bei Bedarf eigene Funktionalität ausführen.

Das beschriebene Vorgehen stellt sicher, dass die Struktur eines Algorithmus unverändert bleibt, Subklassen jedoch Möglichkeiten der Einflussnahme gegeben wird.

Struktur

Der grundsätzliche Ablauf und Algorithmus ist in der Basisklasse BaseClass in der Methode templateMethod() realisiert und umfasst die Teilschritte 1 bis 2 und einen abschließenden Hook. Diese Funktionalität wird durch die korrespondierenden Methoden step1(), step2() sowie afterHook() realisiert. Die Methode templateMethod() ist final und die Methoden step1() und step2() sind abstrakt. Die Methode afterHook() ist in der Basisklasse leer implementiert, kann aber in Subklassen mit Funktionalität gefüllt werden. Die Subklasse SubClass implementiert den variablen Teil des Algorithmus durch die Methoden step1() sowie step2() und kann so beliebige Funktionalität im Ablauf des Algorithmus beisteuern. In Abbildung 18-19 ist das zugehörige Klassendiagramm gezeigt.



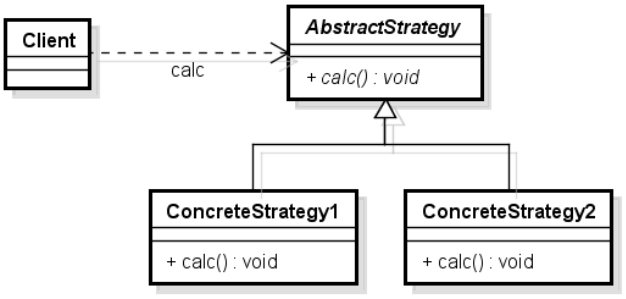
18.3.4 Strategie (Strategy)

Beschreibung und Motivation

Das STRATEGIE-Muster ermöglicht es, das Verhalten eines Algorithmus an ausgesuchten Stellen anzupassen. Im Unterschied zum Muster SCHABLONENMETHODE werden die variablen Bestandteile eines Algorithmus durch eigene Klassen statt durch überschriebene Methoden realisiert. Eine erste Implementierung unterschiedlicher Strategien könnte die Wahl von Alternativen über if-Anweisungen steuern. Ein solches Vorgehen ist allerdings schlecht erweiterbar und wird schnell unübersichtlich. Beim STRATEGIE-Muster werden daher die variablen Teile eines Algorithmus jeweils in eigenen Klassen (mit gemeinsamer Basis) gekapselt und sind dadurch austauschbar. Das konkrete Verhalten kann bei Bedarf sogar erst zur Laufzeit durch Wahl einer beliebigen vorhandenen Realisierung festgelegt werden.

Struktur

Die abstrakte Klasse AbstractStrategy definiert eine Schnittstelle mit benötigten und zu variierenden Methoden. Statt einer abstrakten Klasse kann dies alternativ durch ein Interface erfolgen. Spezielle Funktionalität wird in den Subklassen ConcreteStrategy1 und ConcreteStrategy2 implementiert.



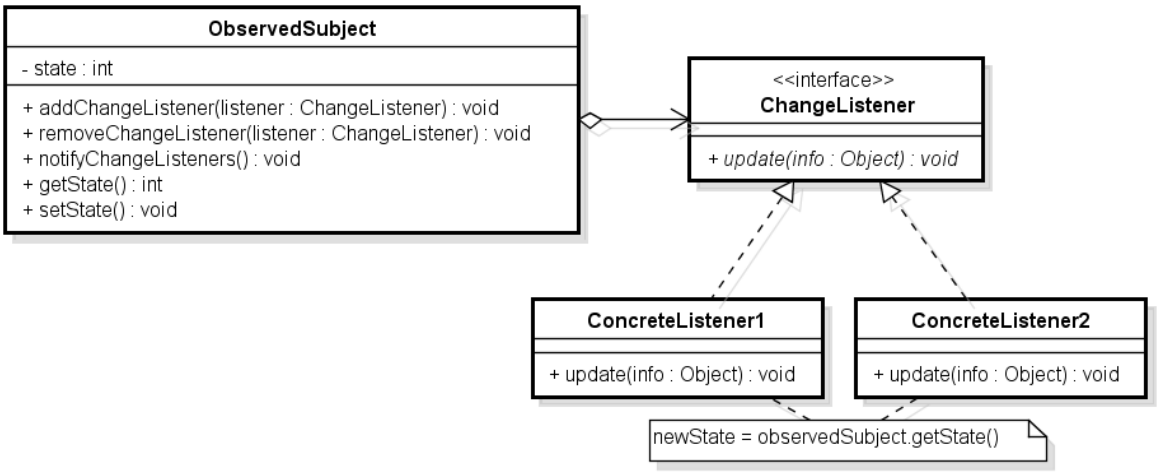
18.3.7 Beobachter (Observer)

Beschreibung und Motivation

Der Einsatz des BEOBACHTER-Musters ermöglicht es, Änderungen am Zustand eines speziellen Objekts durch andere interessierte Objekte beobachten zu können, ohne dass sich ein Objekt und seine Beobachter direkt kennen müssen. Beobachter melden sich dazu als Interessenten bei einem Objekt an und »abonnieren« damit Informationen über Veränderungen, die ihnen das Objekt in der Folge mitteilt. Sind Interessenten nicht mehr an Änderungen interessiert, so können sie sich abmelden. Man kann die diesem Muster zugrunde liegende Idee mit dem Abonnement einer
Zeitschrift vergleichen. Hat man sich als Abonnent registriert, so erhält man regelmäßig die neuesten Ausgaben, ohne ständig am Kiosk nachschauen zu müssen. Dieses Muster ist auch unter dem Namen *Publisher/Subscriber* oder *Listener* bekannt.

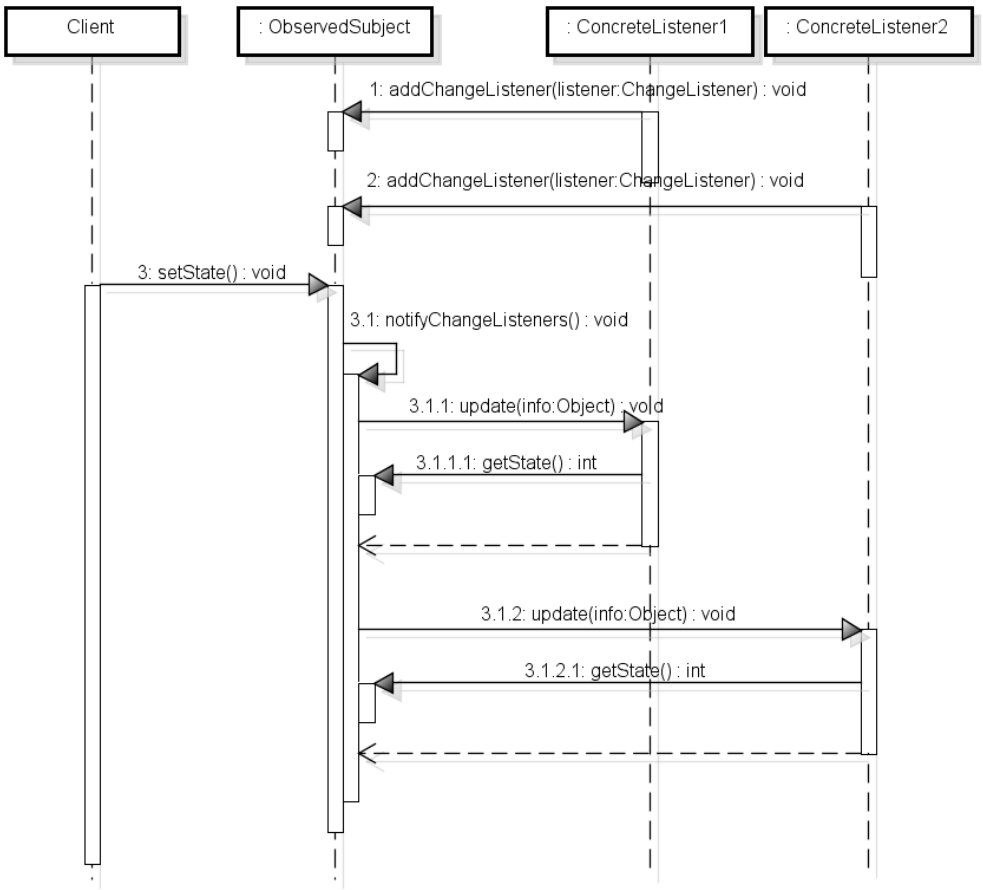
Struktur

Um Änderungen an einem beobachteten Subjekt ObservedSubject mitgeteilt zu bekommen,
können sich andere, das Interface ChangeListener erfüllende Objekte als Änderungsinteressenten bei dem ObservedSubject anmelden.



Bei einer Änderungsnachricht variiert deren Inhalt in der Art und Anzahl der übertragenen Daten. Man unterscheidet zwischen den Varianten *Pull* und *Push*. Nachdem ein Beobachter über eine Änderung informiert wurde, kann dieser weitere Informationen von dem ObservedSubject holen. Dazu existieren in der Regel diverse get()-Methoden – hier exemplarisch die Methode getState().

Ein möglicher Ablauf ist in Abbildung 18-30 in Form eines Sequenzdiagramms für das ObservedSubject und zwei registrierte Listener ConcreteListener1 bzw. ConcreteListener2 visualisiert.


Client
: ObservedSubject
: ConcreteListener1
: ConcreteListener2
1: addChangeListener(listener:ChangeListener) : void
2: addChangeListener(listener:ChangeListener) : void
3: setState() : void
3.1: notifyChangeListeners() : void
3.1.1: update(info:Object) : void
3.1.1.1: getState() : int
3.1.2: update(info:Object) : void
3.1.2.1: getState() : int