HANSER

Leseprobe

zu

„Design Patterns mit Java“

von Florian Siebler

ISBN (Buch): 978-3-446-43616-9
ISBN (E-Book): 978-3-446-44111-8

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-43616-9
sowie im Buchhandel

Inhalt

Mae govannen![1]

Design Patterns sind ein spannendes Thema!

Und Sie werden Spaß haben, wenn Sie dieses Buch durcharbeiten.

Wenn von Design Patterns die Rede ist, denken Sie sicher zuerst an das Buch von Erich Gamma und seinen drei Kollegen, die zusammen „Viererbande“ („Gang of Four“, „GoF“) genannt werden. Sie beschreiben 23 Design Patterns so grundlegend, dass ihr Buch seit vielen Jahren unverändert gedruckt wird.

Mein Buch hat die Aufgabe, Ihnen die GoF-Muster vorzustellen und Ihnen einen leichten Zugang zu ihnen zu ermöglichen. Dabei setze ich Java als Sprache für die Beispielcodes ein. Mein Ziel ist es jedoch, Ihnen nicht nur die Patterns zu zeigen; ich beabsichtige, Ihnen die dahinterstehenden Strukturen und die Wege zur Lösung zu zeigen. Ich gehe davon aus, dass Sie mit dem „Warum“ mehr anfangen können als mit dem „Wie“.

Mein Buch hat noch eine zweite Aufgabe – ich möchte ein paar Aspekte der Java-Programmierung beleuchten, die in vielen Java-Einführungen zu kurz kommen. Sie werden zum Beispiel einen eigenen LayoutManager schreiben und lernen, mit TableModel und TreeModel umzugehen. Ich zeige Ihnen ein paar Tricks aus meiner eigenen Java-Praxis; und schließlich werden Sie an vielen Stellen sehen, wie die Klassen der Klassenbibliothek arbeiten. Ich setze voraus, dass Sie Java bereits kennen und anwenden können. Sie müssen kein Java-Profi sein, um von diesem Buch zu profitieren. Sie sollten aber stabile Grundlagenkenntnisse haben und die vier Säulen der Objektorientierung (Abstraktion, Kapselung, Vererbung und Polymorphie) kennen. Außerdem sollten Sie mit den Sprachgrundlagen vertraut sein und zum Beispiel wissen, was eine innere Klasse ist. Wenn Sie die Grundzüge der GUI-Programmierung mit Swing kennen, ist das ganz optimal.

Ich stelle Ihnen in jedem Kapitel ein Muster der GoF vor. Da die Kapitel in sich abgeschlossen sind, können Sie gezielt das Kapitel lesen, das Sie am meisten interessiert. Dennoch greife ich in einigen Kapiteln auf Themen zurück, die weiter vorne besprochen wurden. Wenn Sie also mit Design Patterns bisher nichts oder nur wenig zu tun hatten, fangen Sie am besten vorne an und arbeiten sich nach hinten durch. Die Reihenfolge, in der ich Ihnen die Muster vorstelle, entspricht nicht der von Gamma gewählten Reihenfolge.

Um Muster aus der abstrakten Welt der Informatik herauszulösen, biete ich Ihnen als Einleitung an vielen Stellen Beispiele aus der realen Welt an, in denen ein Muster – ein Vorge-

[1] Frei übersetzt: „Herzlich willkommen!“ (Gruß der Elben)

hen – eingesetzt wird. Erst dann wird das Muster aus Sicht der IT besprochen. Ich möchte Ihnen damit die Sicherheit geben, dass Design Patterns kein Spezialwissen von irgendwelchen Gurus sind, sondern bewährte alltägliche Praxis beschreiben. Anschließend stelle ich Ihnen ein Problem aus der IT vor und biete dazu eine Lösung an. Diese Lösung wird funktionieren, aber sie ist bei weitem noch nicht perfekt. Also werde ich eine zweite Lösung entwickeln, die die erste verbessert. Sie werden also in fast jedem Kapitel mehr als ein Beispielprojekt finden. Wenn ein Pattern anhand zweier unterschiedlicher Projekte gezeigt wird, ist das erste Projekt recht einfach; es hat den Sinn, das Pattern zu beschreiben. Das zweite Projekt ist dann oft etwas aufwendiger und komplizierter; es zeigt, wie ein Pattern in einem größeren Kontext angewandt werden kann. Die meisten Quelltexte sind im Buch nur gekürzt abgedruckt. Sie finden die vollständigen Projekte unter

www.patternsBuch.de.

Die Codes habe ich mit NetBeans in der Version 8 auf Java 7 erstellt; Sie können die Quelltexte aber problemlos mit einer IDE Ihrer Wahl bearbeiten. Bitte öffnen Sie jedes Projekt und analysieren Sie es. Es ist sehr wichtig, dass Sie sich die Zeit nehmen, jeden einzelnen Schritt und jedes Projekt nachzuvollziehen. Das Buch ist nicht dafür gedacht, „konsumiert" zu werden; es soll Sie vielmehr zu eigener Recherche motivieren.

Die GoF setzt in ihrem Buch die Zweckbeschreibung eines Patterns an den Anfang eines Kapitels. Da die Zweckbeschreibungen die zugehörigen Patterns auf einem sehr hohen abstrakten Niveau beschreiben, sind sie wenig hilfreich, wenn man das erste Mal mit einem Pattern zu tun hat. Daher gehe ich den umgekehrten Weg und erkläre erst die Grundzüge des Patterns und zitiere danach seine Zweckbeschreibung. Ich halte diese Reihenfolge auch deswegen für didaktisch sinnvoll, weil Sie anhand der Zweckbeschreibung kontrollieren können, ob Sie die Zielrichtung des gerade besprochenen Patterns verstanden haben. Sie sollten sich auch deshalb unbedingt mit den Zweckbeschreibungen beschäftigen, weil sie so etwas wie „offizielle" Definitionen sind; Sie müssen sie einfach kennen.

Ich beteilige mich an dem Softwareprojekt „patternCoder". Sie finden dieses Projekt auf der Homepage *www.patternCoder.org*. Auf *www.patternCoder.de* arbeite ich an der deutschen Übersetzung. Sie finden dort weiterführende Links zum Thema Design Patterns und deutschsprachige Musterbeschreibungen.

Wenn Sie ein Kapitel durchgearbeitet haben, sollten Sie das Muster unbedingt mit dem patternCoder umsetzen. Das Beispiel im patternCoder wird keine neuen Erkenntnisse liefern, sondern das besprochene Pattern wiederholen. Sie können damit das Gelernte aus einem anderen Blickwinkel und in einem anderen Kontext noch einmal betrachten. Der patternCoder kann Ihnen auch in der Praxis immer wieder als „Spickzettel" dienen.

In allen Patterns-Büchern finden Sie eine Vielzahl von Klassendiagrammen. Ich habe mich bewusst entschieden, sehr sparsam mit Diagrammen umzugehen. Die Diagramme vieler Patterns gleichen sich zu sehr, um einen Erkenntnisgewinn daraus zu ziehen. Sie profitieren sehr viel mehr, wenn Sie die Beispiele im patternCoder nachvollziehen.

Die Klassendiagramme, die Sie im Buch finden, habe ich mit *www.yUML.me* erstellt. Auf die Angabe von Kardinalitäten habe ich verzichtet.

Ich möchte Sie im Buch direkt ansprechen. Dabei verwende ich ausschließlich die Anrede, die als „männliche“ Anrede gilt. Ich möchte den Text nicht unnötig aufblähen, indem ich permanent schreibe „Sie als Programmierer oder Programmiererin“. Ich halte es für unangemessen und unseriös, eine Klasse mit `class StudentIn` zu bezeichnen. Da die Bezeichner von Klassen nach der Konvention Nomen im Singular sein müssen, kann ich auch nicht in irgendwelche Behelfskonstrukte flüchten. Irgendwann muss ich `class Studierender` oder `class Studierende` coden. Noch unpassender finde ich es, aus Gründen der „Geschlechtergerechtigkeit“ grundsätzlich die weibliche Form zu verwenden, was konsequenterweise zu `class Klientin` und `class Serverin` führen würde. Daher meine Bitte: Sofern Sie eine Leserin sind, fühlen Sie sich nicht übergangen oder gering geschätzt.

Ich bedanke mich

- bei meiner Freundin **Martina Guth**. Ich habe zwei Jahre an diesem Buch gearbeitet und dabei sehr viel Energie und noch mehr Freizeit investiert. Martina hat mir in dieser Zeit sehr viel Kraft gegeben. Ich halte ihr Verständnis, ihre Zuverlässigkeit und ihre Unterstützung nicht für eine Selbstverständlichkeit – danke!
- bei **Benjamin Sigg**, der dieses Buch als Fachgutachter begleitet hat. Ich habe die Zusammenarbeit als ausgesprochen produktiv und motivierend erlebt. Benjamin hat mir immer sehr zeitnah ein umfassendes Feedback zu den Kapiteln gegeben. Darüber hinaus hat er mich mit sehr viel Programmierarbeit unterstützt; der patternCoder hätte ohne seine Unterstützung in dieser Form niemals zu einem Abschluss gebracht werden können. Vielen Dank!
- Bu toil leam taing a thoirt do **Sheumas Paterson** (Ollamh aig Caledonian University Ghlaschu) airson an taic a fhuair mi bhuaithe. ’S iomadh latha a bha sinn ag obair air pattern code còmhla agus nach mi a dh’ionnsaich tòrr bho Sheumas chòir. Tha mi uamhasach fhèin taingeil airson a chomhairle is am fiosrachadh prìseil a bha na chuideachadh mòr dhomh agus a bha air leth cudthromach agus luachmhor airson an leabhair agam. Mìle Taing agus Herzlichen Dank bhuam![2]

Erwähnen möchte ich schließlich auch meine Arbeitskollegen aus dem Referat Software-Entwicklung des Bundesamtes für Justiz. Wir haben ein ausgezeichnetes Arbeitsklima und das patternsBuch war beim Feierabendbier oft ein Thema. Auch von ihnen habe ich viel konstruktives Feedback und Input bekommen.

[2] Das ist Schottisch-Gälisch – ich bedanke mich hier bei James Paterson, Professor an der Glasgow Caledonian University.

2 Template Method Pattern

In diesem Kapitel werden Sie das Template Method Pattern kennenlernen. Es ist ein Verhaltensmuster und sehr leicht zu verstehen. Das Pattern hilft Ihnen, wenn Sie einen Algorithmus haben, der einerseits nicht geändert werden darf, andererseits aber in Unterklassen teilweise unterschiedlich implementiert werden muss.

2.1 Die Arbeitsweise von Template Method

Nehmen Sie als Beispiel die klassische Reihenfolge von Eingabe, Verarbeitung und Ausgabe. Sie möchten einen String eingeben, diesen wahlweise in Groß- oder Kleinschreibung konvertieren und anschließend auf der Konsole ausgeben.

2.1.1 Der erste Ansatz

Auf den ersten Blick scheint die Aufgabe schnell gelöst zu sein, wenn Sie zwei Klassen definieren: `Uppercase` und `Lowercase`. Den Quelltext von `Uppercase` drucke ich folgend ab. Die Methode `run()` definiert den Algorithmus: eingeben, konvertieren, ausgeben. Der Quellcode der Klasse `Lowercase` ist identisch. Lediglich der fett markierte Part unterscheidet sich.

Listing 2.1 Klasse „Uppercase“ im Projekt „Template_1“

```
public class Uppercase
{
    public final void run()
    {
        final String EINGABE = textEingeben();
        final String KONVERTIERT = convert(EINGABE);
        drucke(KONVERTIERT);
    }
```

```
    private final String textEingeben()
    {
        final String MESSAGE = "Bitte geben Sie den Text ein: ";
        return JOptionPane.showInputDialog(MESSAGE);
    }

    private final String convert(String eingabe)
    {
        return eingabe.toUpperCase();
    }

    private final void drucke(String text)
    {
        System.out.println(text);
    }

    public static void main(String[] args)
    {
        new Uppercase().run();
    }
}
```

Wenn ein Projekt Quellcode hat, der per Copy and Paste übertragen wird, ist das in der Regel ein Hinweis auf ein schlechtes Design. Es bietet sich an, doppelten Code in eine gemeinsame Oberklasse auszulagern – und genau das machen Sie im folgenden Abschnitt.

2.1.2 Der zweite Ansatz

Die abstrakte Oberklasse definiert einerseits den Algorithmus, andererseits die Methoden, die in beiden Klasse identisch sind. Da die Definition des Algorithmus für alle Unterklassen verbindlich sein soll, bietet es sich an, die Methode `run()` `final` zu deklarieren.

Listing 2.2 Abstrakte Oberklasse im Projekt „Template_2“

```
public abstract class Eingabe
{
    public final void run()
    {
        String eingabe = textEingeben();
        String konvertiert = convert(eingabe);
        drucke(konvertiert);
    }

    private final String textEingeben()
    {
        final String MESSAGE = "Bitte geben Sie den Text ein: ";
        return JOptionPane.showInputDialog(MESSAGE);
    }

    protected abstract String convert(String eingabe);

    private final void drucke(String text)
```

```
    {
        System.out.println(text);
    }
}
```

Die Unterklasse `UppercaseConverter` und `LowercaseConverter` überschreiben lediglich die abstrakte Methode `convert()`. Exemplarisch wird die Klasse `LowercaseConverter` abgedruckt.

Listing 2.3 Die Subklasse „LowercaseConverter“ im Projekt „Template_2“

```
public class LowercaseConverter extends Eingabe
{
    @Override
    protected String convert(String eingabe)
    {
        return eingabe.toLowerCase();
    }
}
```

Im folgenden Absatz wird das Projekt in Betrieb genommen.

2.1.3 Das Hollywood-Prinzip

Der Client kann nun durch Unterklassenbildung sehr leicht den gewünschten Converter auswählen.

Listing 2.4 Client im Project „Template_2“

```
public class Client
{
    public static void main(String[] args)
    {
        Eingabe eingabe = new LowercaseConverter();
        eingabe.run();
        Eingabe neueEingabe = new UppercaseConverter();
        neueEingabe.run();
    }
}
```

Der Aufruf erfolgt immer von der Oberklasse aus. In der Oberklasse ist der Algorithmus definiert und die Oberklasse ruft die entsprechende Methode in der Unterklasse auf. Dieses Vorgehen nennt die GoF *Hollywood-Prinzip*: *„Rufen Sie uns nicht an – wir rufen Sie an!“*

2.1.4 Einführen von Hook-Methoden

Sie können das Projekt mit *Hook-Methoden* erweitern. Eine Hook-Methode ist eine Methode, die optional überschrieben werden kann. In der Oberklasse wird ein Standardverhalten – oder gar kein Verhalten – definiert. Die Unterklassen können – müssen aber nicht – dieses

Verhalten übernehmen. Im Projekt „Template_3“ gibt es eine Hook-Methode. Die Methode `speichern()` gibt zurück, ob der eingegebene Text auf der Festplatte gespeichert werden soll. In diesem Fall wird die Methode `saveToDisk()` aufgerufen. Standardmäßig wird `false` zurückgegeben.

Listing 2.5 Optional die Eingabe speichern – Projekt „Template_3“

```
public abstract class Eingabe
{
    public final void run()
    {
        String eingabe = textEingeben();
        String konvertiert = convert(eingabe);
        drucke(konvertiert);
        if ( speichern() )
            saveToDisk();
    }

    // ... gekürzt ...

    protected boolean speichern()
    {
        return false;
    }

    private void saveToDisk()
    {
        System.out.println("Eingabe wurde gespeichert.");
    } }
```

Wenn die Unterklassen der Meinung sind, dass das Standardverhalten so nicht passt, steht es ihnen frei, dieses zu überschreiben. Der `UppercaseConverter` möchte beispielsweise, dass der Text immer gespeichert wird – er gibt also `true` zurück.

Listing 2.6 Klasse „UppercaseConverter“ im Projekt „Template_3“

```
public class UppercaseConverter extends Eingabe
{
    // ... gekürzt ...

    @Override
    protected boolean speichern()
    {
        return true;
    }
}
```

Der `LowercaseConverter` möchte den Anwender entscheiden lassen, ob der Text gespeichert wird.

Listing 2.7 Klasse „LowercaseConverter“ im Projekt „Template_3“

```
public class LowercaseConverter extends Eingabe
{
    // ... gekürzt ...
```

```
    @Override
    protected boolean speichern()
    {
        String frage = "Soll der Text gespeichert werden?";
        int antwort = JOptionPane.showConfirmDialog(null, frage);
        return antwort == JOptionPane.YES_OPTION;
    }
}
```

Hook-Methoden werden auch *Einschub-Methoden* genannt. Sie geben die Möglichkeit, auf den Ablauf des Algorithmus Einfluss zu nehmen.

Das war’s dann auch schon – jetzt kennen Sie das Template Method Pattern. Trotz – oder gerade wegen – seiner trivial erscheinenden Formulierung ist das Template Method Pattern ausgesprochen wichtig für die Entwicklung von Frameworks wie der Java-Klassenbibliothek. Sie entwickeln ein Grundgerüst, den Algorithmus, und darauf bauen Sie Ihre Erweiterungen auf. Auf *www.Sourcemaking.com* habe ich Bild 2.1 gefunden, das, wie ich finde, das Prinzip sehr anschaulich zeigt.

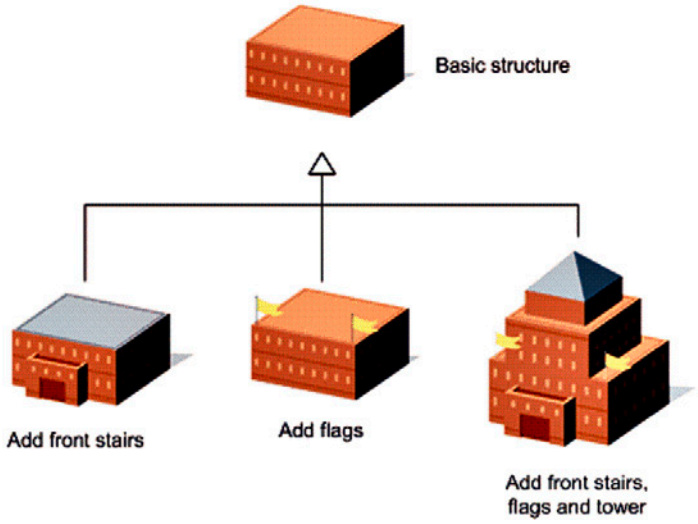

Bild 2.1 Prinzip des Template Method Patterns

2.2 Das Interface „ListModel“

Das Template Method Pattern werden Sie überall dort identifizieren können, wo Sie auf abstrakte Klassen treffen. Ich möchte an dieser Stelle kurz auf den Algorithmus eingehen, den das Interface `ListModel` vorgibt; er beschreibt, was passieren muss und welche Daten vorliegen müssen, damit eine JList Daten anzeigen kann. Eine `JList` akzeptiert jedes Objekt als Datenmodel, das vom Typ `ListModel` ist. Das Interface `ListModel` schreibt vier Methoden vor, die für die Anzeige der Daten erforderlich sind. Eine `JList` muss sich als Beobach-

ter, als ListDataListener, beim Model registrieren und de-registrieren können; dafür müssen die Methoden `addListDataListener()` und `removeListDataListener()` implementiert sein. Die JList muss die Datenbasis außerdem fragen können, wie viele Elemente angezeigt werden sollen; das wird mit der Methode `getSize()` beantwortet. Und schließlich muss die JList das Element an einer bestimmten Stelle abfragen können; hierfür gibt es die Methode `getElementAt()`, der Sie den Index des gesuchten Elements übergeben. Unter *www.patternsBuch.de* finden Sie das Projekt „ListModel_Beispiel“. Dieses Projekt zeigt die Vorgehensweise mit einem sehr simplen Beispiel.

Wenn Sie seinen Quelltext analysieren, werden Sie sehen, dass die Methoden zum Registrieren und De-Registrieren von Listenern so allgemein sind, dass sie vermutlich in allen Situationen verwendet werden können. In der Klassenbibliothek gibt es die Klasse `AbstractListModel`, in der diese beiden Methoden implementiert sind. Sofern Sie eine andere als die Standardimplementierung benötigen, sind Sie frei, diese Methoden zu überschreiben. Methoden, die die Datenbasis beschreiben, also `getSize()` und `getElementAt()`, entziehen sich einer Standardimplementierung. Sie müssen sie selbst definieren.

2.3 Der patternCoder

In diesem Abschnitt möchte ich Ihnen den patternCoder vorstellen. Der patternCoder war früher eine Erweiterung für BlueJ. Wir haben im Laufe der Zeit verschiedene Versionen erstellt, so dass der patternCoder mit gleichem Funktionsumfang als Stand-alone-Version, als Erweiterung für BlueJ und als Plug-in für Eclipse genutzt werden konnte. Im Hinblick auf dieses Buch haben wir 2013 angefangen, den patternCoder neu zu designen. Die Stand-alone-Version ist fertig, Sie finden sie unter *www.patternsBuch.de*. Die BlueJ-Version war bei Drucklegung noch nicht fertig erstellt, ist aber in Arbeit. Natürlich wird auch der Funktionsumfang des patternCoder weiterentwickelt und ausgebaut. Die neue Version des patternCoder ist unter Java 7 und höher lauffähig. Bitte schauen Sie ab und zu auf *www.patternCoder.de* vorbei; dort hinterlegen wir die jeweils neuen Versionsstände.

Den patternCoder installieren

Unter *www.patternsBuch.de* finden Sie eine zip-Datei, die das ausführbare jar-File und den Ordner *patternFiles* enthält. Entpacken Sie die zip-Datei, so dass das jar-File und der Ordner im gleichen Verzeichnis liegen. Wenn Sie den patternCoder starten, sucht er zunächst im gleichen Verzeichnis nach dem Ordner *patternFiles*. Wenn Sie diesen Ordner umbenennen oder an einen anderen Ort verschieben, prüft patternCoder, ob in der Registry ein anderer Pfad als Projektverzeichnis vorliegt. Ist das nicht der Fall, sind die meisten Funktionen ausgegraut und Sie müssen zuerst das Projektverzeichnis eingeben. Klicken Sie hierzu **strg** + **alt** + **s**. Den Pfad, den Sie eingeben, hinterlegt patternCoder in der Registry und greift beim nächsten Start darauf zu. Am einfachsten dürfte es jedoch sein, wenn Sie jar-File und patternFiles-Ordner im gleichen Verzeichnis ablegen.

Der Ordner „patternFiles“

Im Ordner *patternFiles* sind die Beschreibungen der Patterns hinterlegt. Jedes Pattern wird durch ein patternFile, eine XML-Datei, beschrieben. Im Ordner *behavioural* finden Sie beispielsweise die Datei *Template.patf.xml*. Sie beschreibt das Template Method Pattern. Bitte benennen Sie die Unterordner und die Dateien **nicht** um.

2.3.1 Der patternCoder aus Anwendersicht

Dieser Abschnitt beschreibt, wie der patternCoder funktioniert und was Sie von ihm erwarten können.

Die Ansicht „Übersicht“

Wenn Sie den patternCoder starten, sehen Sie auf der linken Seite einen Baum, in dem alle Patterns gelistet werden. Auf der rechten Seite lesen Sie eine Zusammenfassung, wie der patternCoder bedient wird.

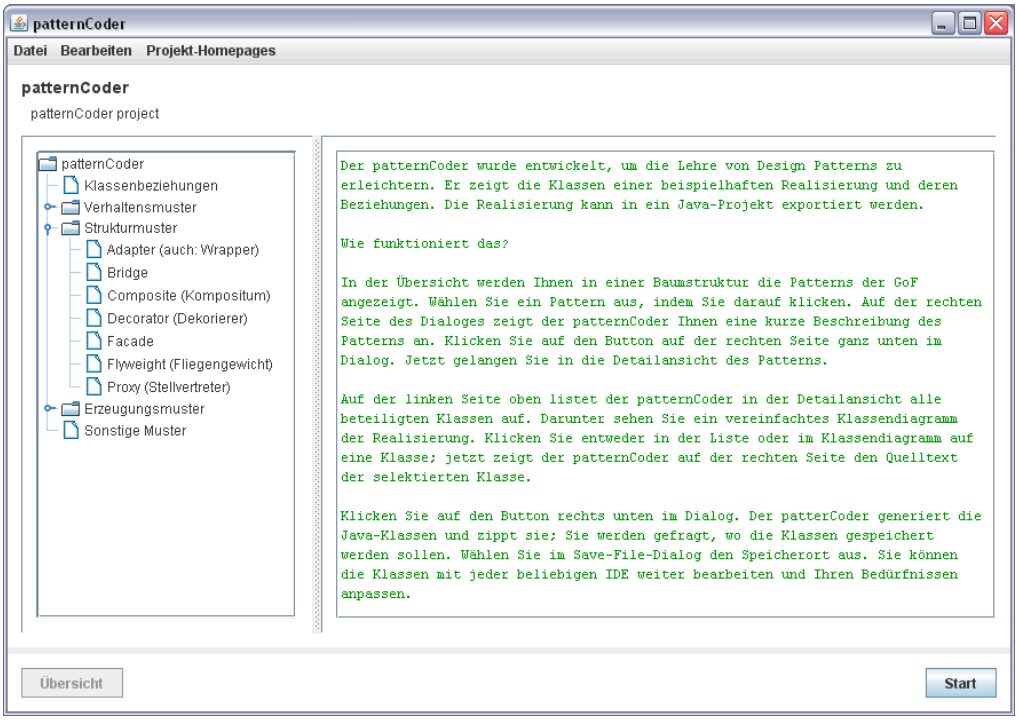

Bild 2.2 Der patternCoder nach dem Start

Wählen Sie aus der Kategorie Verhaltensmuster das Template Method Pattern. Klicken Sie einmal kurz auf den Eintrag. Jetzt wird auf der rechten Seite die Zweckbeschreibung des Patterns angezeigt. Klicken Sie nun auf den Button **Start**. Alternativ können Sie doppelt auf den Eintrag klicken, um in die Detailansicht zu wechseln.

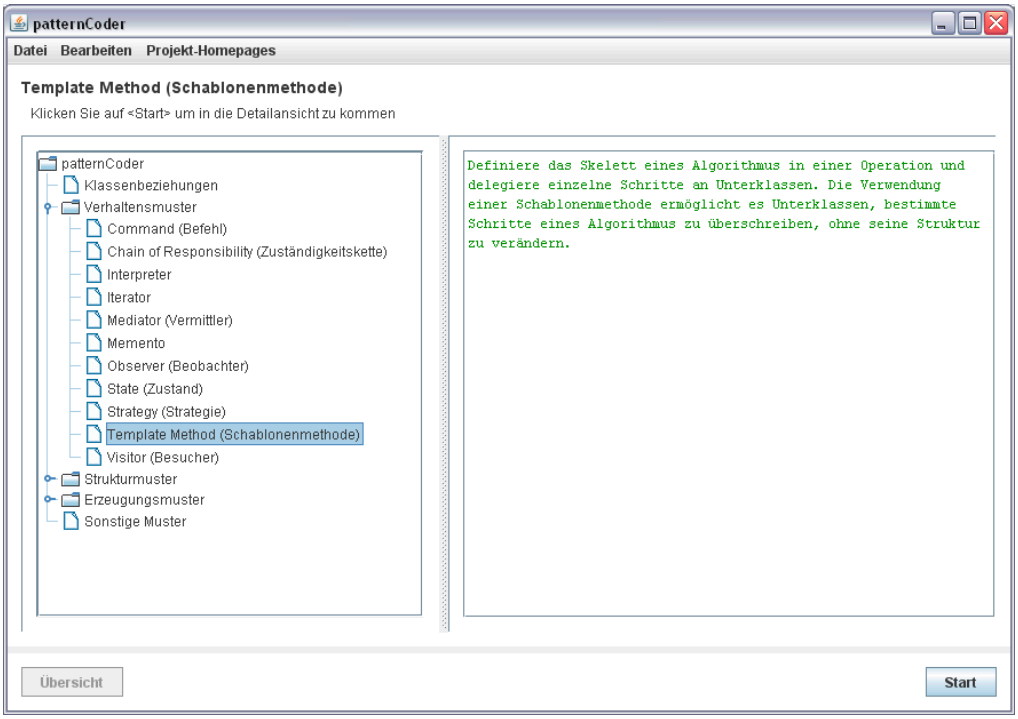

Bild 2.3 Übersichtsansicht des Template Method Patterns

Die meisten Texte sind auf Englisch hinterlegt. Wir werden sie nach und nach übersetzen. Auf *www.patternCoder.de* finden Sie die neuen patternFiles.

Die Ansicht „Detail“

Wenn Sie in die Detail-Ansicht wechseln, finden Sie im oberen linken Bereich eine Auflistung der am Pattern beteiligten Klassen. Im unteren linken Bereich sehen Sie das Klassendiagramm des hinterlegten Projekts. Auf der rechten Seite ist nach wie vor die Beschreibung des Patterns zu sehen. Wenn Sie auf eine Klasse klicken – entweder im Klassendiagramm oder in der Auflistung oben links – wird rechts der Quelltext der Klasse angezeigt. Die ausgewählte Klasse wird im Klassendiagramm grün markiert.

Oberhalb des Diagramms gibt es den Button **Klassendiagramm speichern**. Wenn Sie darauf klicken, wird das Diagramm in eine png-Datei exportiert; Sie werden von patternCoder aufgefordert, den Speicherort einzugeben. Der Button **Code generieren** ganz unten rechts exportiert die hinterlegten Java-Quelltexte. Sie werden gezippt und patternCoder lässt Sie eingeben, wo die zip-Datei gespeichert werden soll. Sofern Sie es vorziehen, dass öffnende Klammern ans Ende der vorigen Zeile geschrieben werden, entfernen Sie im Menü **Bearbeiten** das Häkchen bei **Öffnende Klammer in neuer Zeile**. Der Button **Übersicht** bringt Sie zurück zur Übersicht.

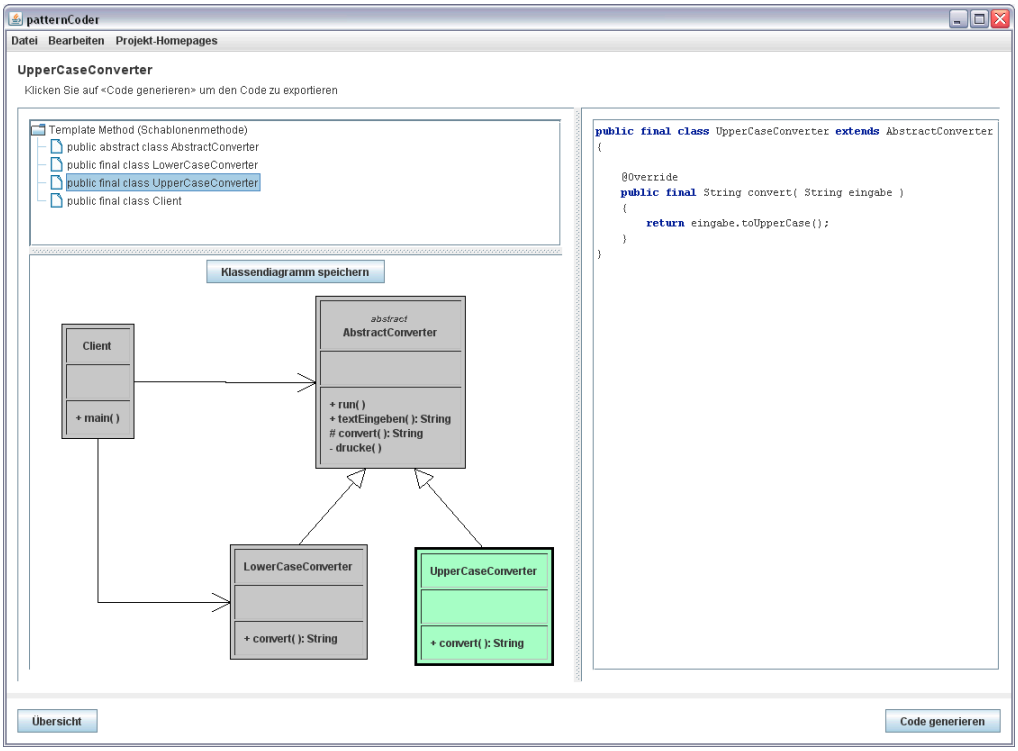

Bild 2.4 Detailansicht des patternCoder

Der patternCoder für Macher

Möchten Sie selbst patternFiles anlegen? Wechseln Sie zurück in die Übersicht. Setzen Sie im Menü **Bearbeiten** einen Haken bei **Pattern bearbeiten**.

Ein Pattern anlegen oder editieren

Wenn Sie nun einen rechten Mausklick auf den JTree setzen, erscheint ein Kontextmenü. Wenn Sie ein Pattern ausgewählt haben, können Sie es editieren oder löschen. Sie können auch ein neues Pattern anlegen. Egal, ob Sie ein bestehendes Pattern editieren oder ein neues anlegen, Sie arbeiten immer im Dialog „Ein Pattern editieren“. Wenn Sie ein Pattern anlegen, wählen Sie die Kategorie, zu der es gehört. Der Dateiname darf nur aus Buchstaben in Klein- und Großschreibung bestehen. Das Schlagwort ist der Text, der später im JTree angezeigt wird. In dem großen Feld „Beschreibung“ hinterlegen Sie die Beschreibung des Patterns. Die Kategorie und der Dateiname können später nicht mehr geändert werden.

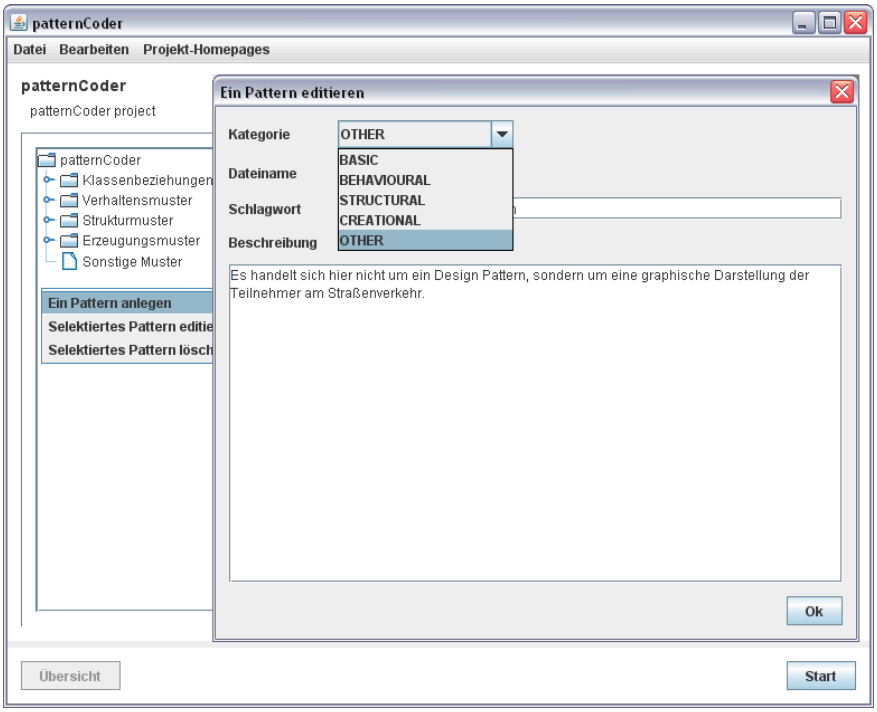

Bild 2.5 Ein Pattern anlegen oder editieren

Klicken Sie auf **Ok**, um die Änderungen zu speichern.

Eine Klasse anlegen

Markieren Sie das gerade angelegte Pattern und klicken Sie auf **Start**, um in die Detailansicht zu wechseln. Im Diagramm gibt es nun sechs ToggleButtons, die später gebraucht werden, um Beziehungen zwischen Klassen darzustellen. Klicken Sie auf die obere Hälfte der linken Seite. Das Kontextmenü bietet Ihnen an, eine Klasse zu erstellen, sie zu editieren oder sie zu löschen. Wählen Sie **Erstelle eine Klasse**, um eine neue Klasse zu erzeugen. Im Dialog „Eine Klasse editieren“, den patternCoder jetzt anzeigt, tragen Sie die Daten der Klasse ein. Sie wählen zunächst die Sichtbarkeit aus – standardmäßig ist `public` vorgesehen. Eine Klasse kann entweder `final` oder `abstract` sein – in den seltensten Fällen beides. Im patternCoder ist eine Prüfung hinterlegt, dass immer nur eine Option ausgewählt werden kann. Der Typ gibt an, ob Sie eine Klasse oder ein Interface anlegen. Geben Sie im nächsten Feld den Namen der Klasse ein. Im Eingabefeld *extends* geben Sie an, von welcher Klasse Ihre Klasse erbt – dazu später mehr. Die Combobox *implements* listet alle Interfaces auf, die Ihre Klasse implementieren soll. Einträge können Sie mit einem eigenen Dialog vornehmen, auch das wird gleich genauer beleuchtet. Die Combobox *imports* speichert alle Import-Anweisungen. Die Buttons **Editiere Attribute** und **Editiere Methoden** brauchen Sie, um Felder und Methoden anzulegen. Ein *UML comment* fügt Ihrer Klasse im Diagramm einen kurzen Kommentar hinzu. Wenn Sie die Checkbox *Erstelle Linien* selektiert lassen,

werden implements- und inherits-Beziehungen automatisch vom patternCoder in das Diagramm eingezeichnet. Diese Einstellung betrifft alle Klassen im Pattern. In das Textfeld *Documentation* tragen Sie die API-Dokumentation der Klasse ein.

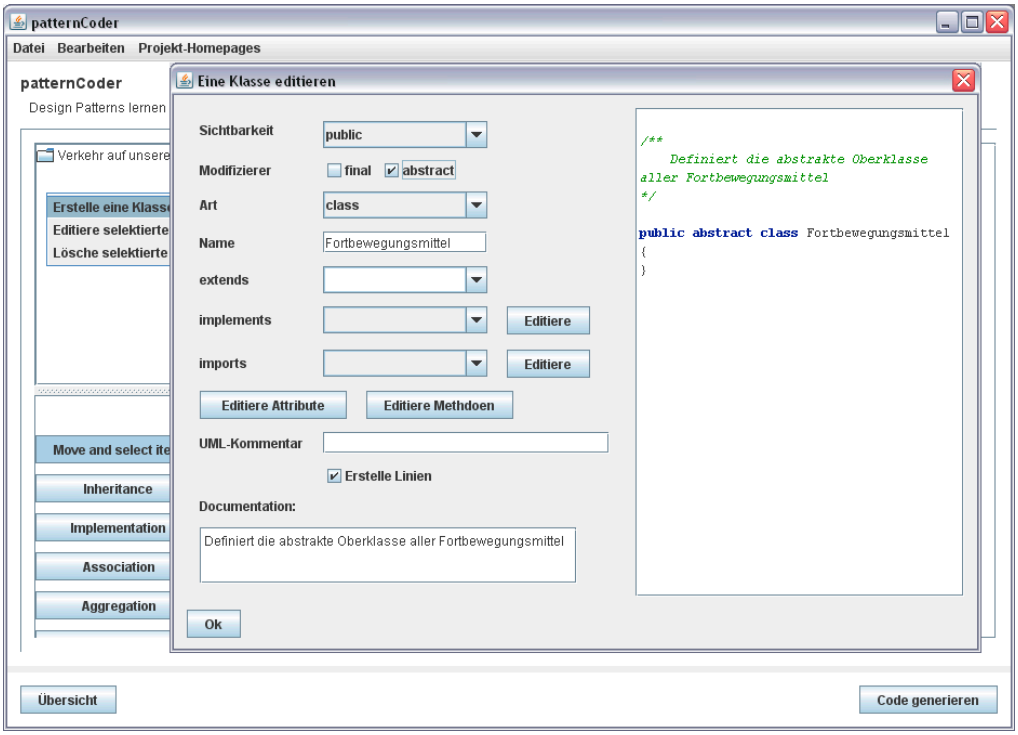

Bild 2.6 Eine Klasse anlegen oder editieren

Ein Datenfeld anlegen

Klicken Sie auf **Editiere Attribute**, um in den Dialog „Ein Attribut editieren“ zu gelangen. Klicken Sie auf **Neues Attribut**, um ein neues Feld zu erzeugen. Es wird in der Liste der Attribute auf der linken Seite des Dialogs angezeigt. Wieder geben Sie die Informationen des Felds an: die Sichtbarkeit, die Modifizierer, den Datentyp, den Bezeichner und die API-Dokumentation. Außerdem können Sie Ihr Datenfeld mit einem beliebigen Wert initialisieren.

Wenn Sie ein Attribut markieren, können Sie es mit **Auswahl löschen** wieder entfernen. Klicken Sie auf **Ok**, um zum Dialog „Eine Klasse editieren“ zurückzukehren.

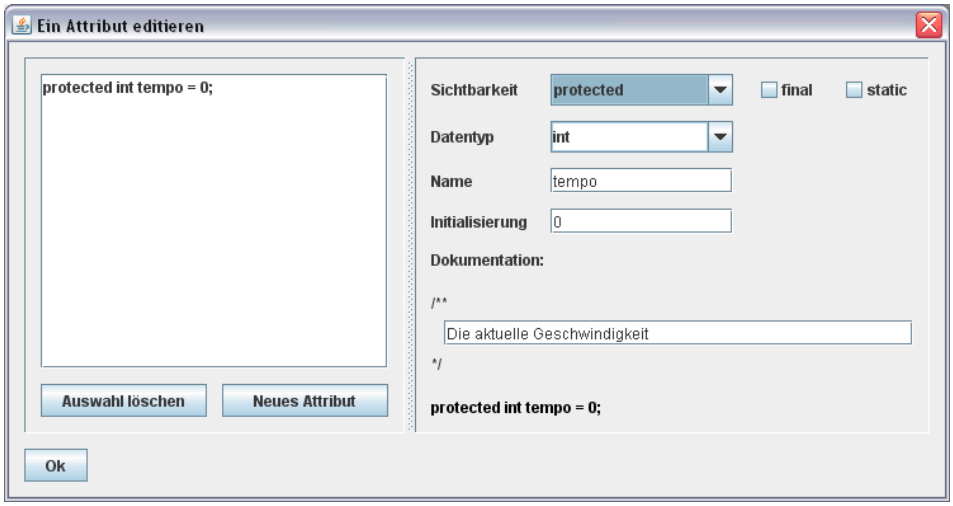

Bild 2.7 Ein Feld anlegen oder editieren

Eine Methode anlegen

Klicken Sie auf den Button **Editiere Methoden**. Der patternCoder öffnet den Dialog „Eine Methode editieren“. Auf der linken Seite werden alle deklarierten Methoden aufgelistet. Erstellen Sie mit **Neue Methode** eine neue Methode; sie wird dann links angezeigt. Sie können nun die Daten der Methode eingeben. Handelt es sich um einen Konstruktor? Überschreibt die Methode eine andere Methode? Welche Sichtbarkeit soll sie haben? Sie geben außerdem die Modifizierer an. Der Rückgabewert ist standardmäßig auf `void` gesetzt. In der Combobox sind verschiedene Vorschläge hinterlegt: `void`, die primitiven Datentypen, der Typ `String` und alle bereits angelegten Klassen im Pattern. Sie können die Einstellungen übernehmen, aber auch einen anderen Datentyp eintragen. Dokumentieren Sie den Rückgabewert und geben Sie der Methode einen Namen. Im Textfeld zwischen /** und */ tragen Sie die API-Dokumentation der Methode ein. Im unteren Textfeld hinterlegen Sie den Code der Klasse. Code können Sie nur eingeben, wenn die Methode nicht abstrakt ist und nicht in einem Interface deklariert wird.

Klicken Sie auf **Neuer Parameter**, um einen neuen Parameter anzulegen. In der Combobox wählen Sie den Datentyp. Geben Sie einen Namen für den Parameter ein und vergessen Sie die API-Doku nicht. Wenn Sie mehr als einen Parameter anlegen, können Sie die Reihenfolge per Drag and Drop ändern; markieren Sie einen Parameter und verschieben Sie ihn nach oben oder nach unten. Der Methodenkopf oberhalb des Code-Felds wird automatisch aktualisiert. Sowohl Parameter als auch Methoden können gelöscht werden. Klicken Sie dazu auf **Auswahl löschen** unterhalb der jeweiligen Liste.

In diesem Dialog sind verschiedene Abhängigkeiten hinterlegt, um Sie vor Fehleingaben zu schützen. Sie können keine zwei Parameter mit gleichem Namen anlegen. Methoden können überladen werden, müssen aber unterschiedliche Parameterlisten haben.

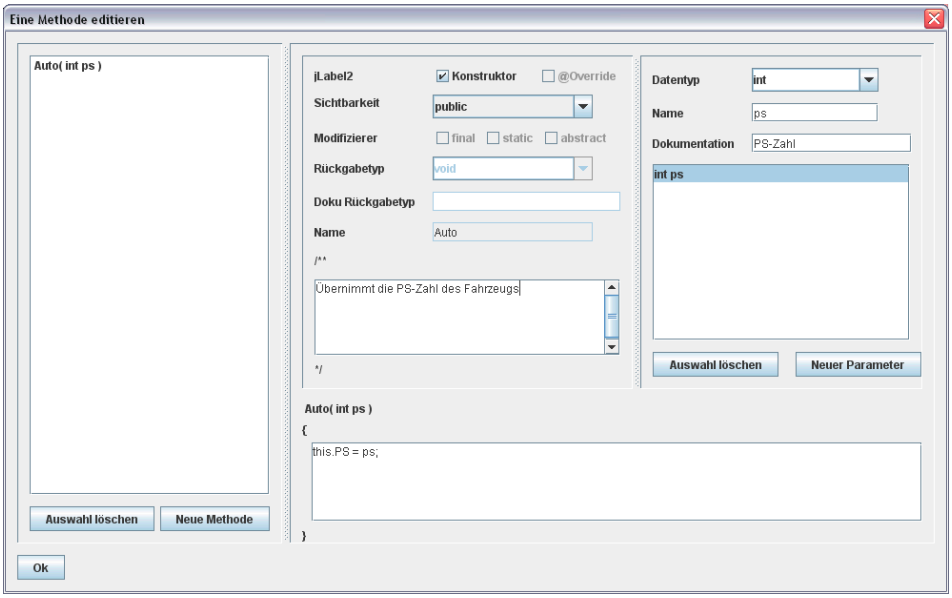

Bild 2.8 Eine Methode anlegen oder editieren

Klicken Sie auf **Ok**, um zum Klassendialog zurückzukehren. Sie werden sehen, dass patternCoder bereits den Code der Klasse erzeugt hat. Klicken Sie erneut auf **OK**, um zur Detailansicht zurückzukehren. Im Diagramm ist die Klasse bereits eingezeichnet.

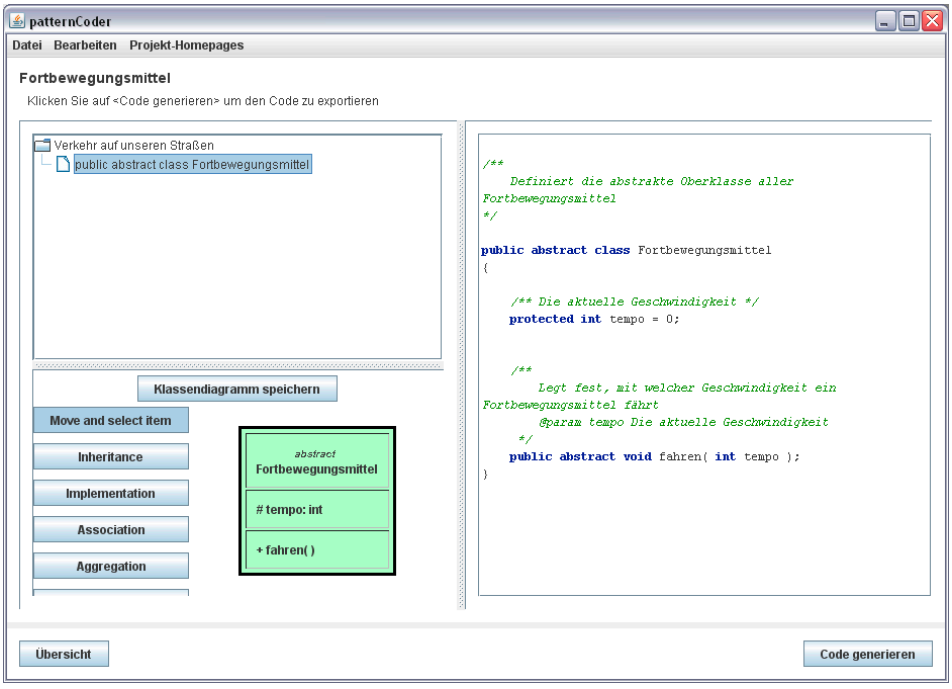

Bild 2.9 Das Pattern in der Detailansicht

Legen Sie eine neue Klasse `Auto` und das Interface `Versicherungspflichtig` an. Das Auto soll von `Fortbewegungsmittel` erben. Wenn Sie im Dialog „Eine Klasse editieren“ auf die ComboBox „extends“ klicken, sehen Sie, dass dort bereits alle bestehenden Klassen des Patterns gelistet sind. Natürlich können Sie die Einträge überschreiben. Ein Interface legen Sie an, indem Sie aus der ComboBox „Art“ den entsprechenden Eintrag wählen.

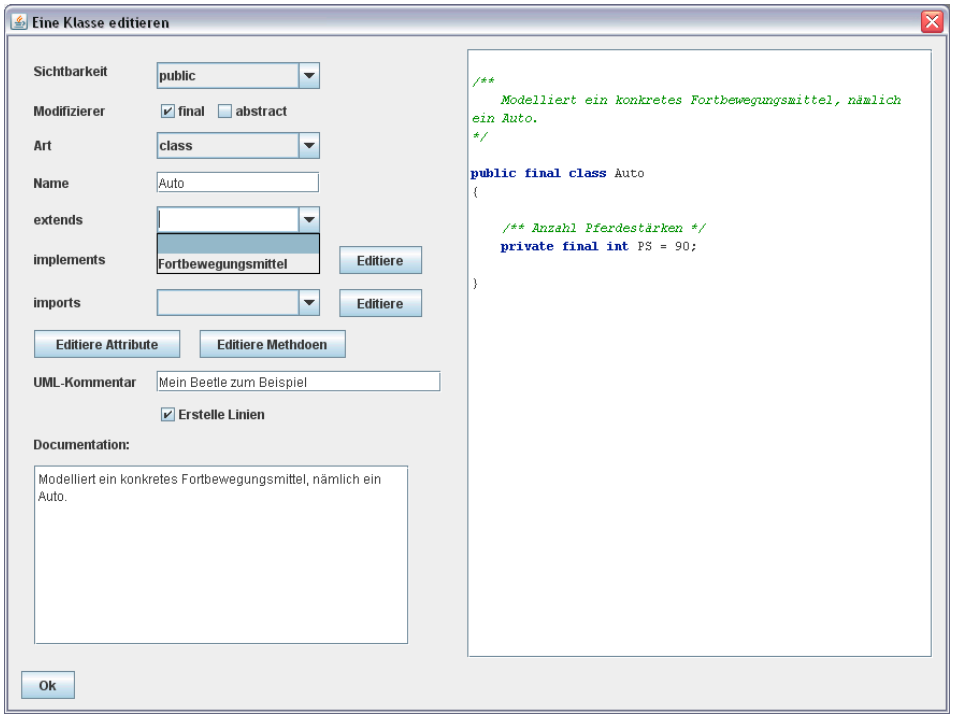

Bild 2.10 Eine andere Klasse erweitern

Ein Interface implementieren

Die Klasse `Auto` soll `Versicherungspflichtig` implementieren. Rufen Sie auf der Klasse `Auto` den Dialog „Eine Klasse editieren“ auf (Bild 2.11). Klicken Sie auf **Editiere** neben der Combobox „implements“. Der patternCoder zeigt einen neuen Dialog an, in dem Sie alle Interfaces aufgelistet finden, die von der Klasse implementiert werden; im Moment ist diese Liste leer. Klicken Sie auf das Pluszeichen, um im InputDialog ein zu implementierendes Interface einzutragen. Tragen Sie `Versicherungspflichtig` ein und schließen Sie den Dialog.

Der Dialog, um Imports einzutragen, ist identisch aufgebaut.

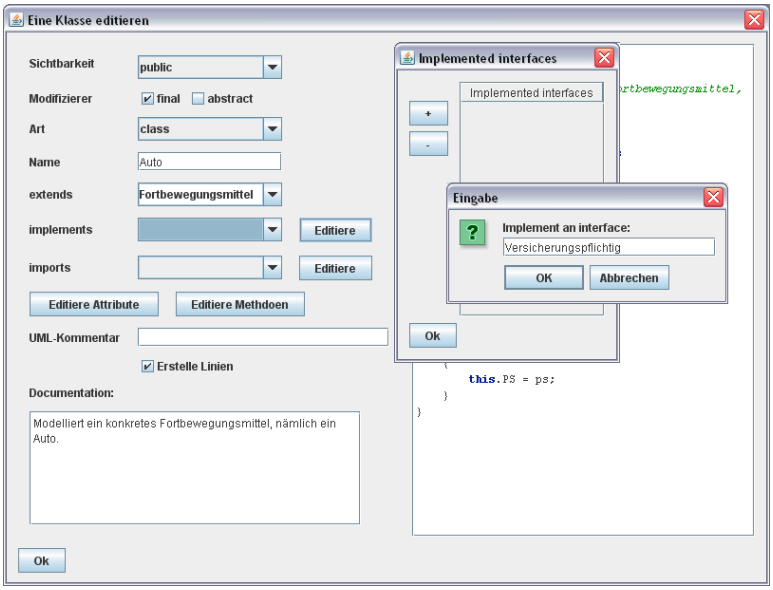

Bild 2.11 Ein zu implementierendes Interface eintragen

Das Klassendiagramm bearbeiten

Schließen Sie den Klassendialog. Der patternCoder hat die Klassen, deren Beziehungen und den UML-Kommentar in das Diagramm eingezeichnet. Vererbung und Realisierung werden zuverlässig erkannt. Mit den ToggleButtons links können Sie selbst Beziehungen zwischen Klassen einzeichnen (Bild 2.12). UML-Kommentare, Verbindungen und Klassen werden Items genannt. Das Kontextmenü ermöglicht es Ihnen, ein selektiertes Item zu löschen.

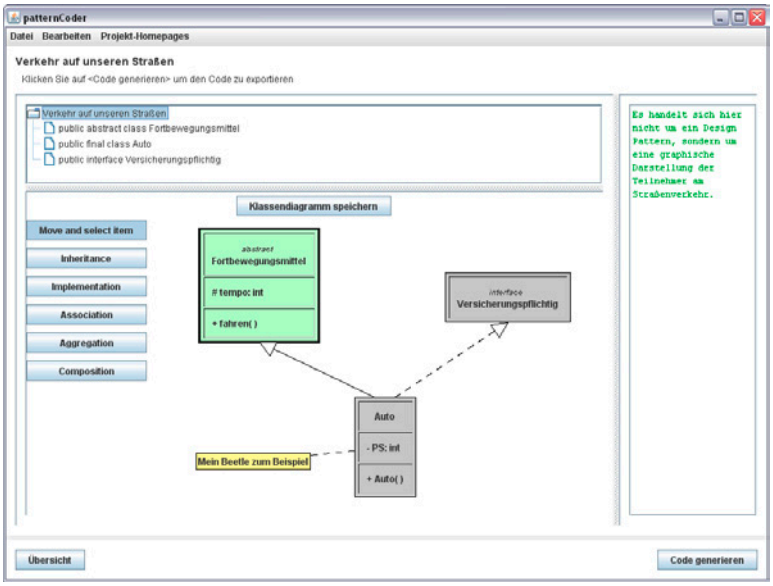

Bild 2.12 Das fertig angelegte Projekt

Um die Klassendiagramme generieren zu können, hat Benjamin Sigg die Bibliothek „GraphPainting“ entwickelt. Den Quelltext finden Sie auf *https://github.com/Benoker*.

2.4 Zusammenfassung

Gehen Sie das Kapitel nochmal stichwortartig durch!

- Sie definieren einen bestimmten Algorithmus.
- Teile des Algorithmus können von der Klasse selbst ausgeführt werden, andere Teile werden den Subklassen vorgeschrieben, die die nicht abstrakten Teile implementieren.
- Den Algorithmus beschreiben Sie in einer finalen Methode der abstrakten Oberklasse.
- Die Unterklasse bzw. die Unterklassen überschreiben die abstrakten Methoden der Oberklasse.
- Optional können Hook-Methoden – Einschubmethoden – überschrieben werden.
- Hook-Methoden erlauben es Ihnen, den Algorithmus teilweise zu variieren.

Zweckbeschreibung:

Die Gang of Four beschreibt den Zweck des Patterns wie folgt:

„Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.“

Index

G

I

J

K