Entwicklung eines Oberflächendesigners für aspektorientierte Anwendungsentwicklung im Model-View-Controller Architekturstil

Diplomarbeit von
Karsten Meier
(Matr. Nr: 150 964)

Technische Universität Berlin
Fakultät IV; Institut für Softwaretechnik
Lehrstuhl: Prof. Dr. Stefan Jähnichen

Betreuer: Stephan Herrmann

27. November 2006

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ziele

ObjectTeams/Java (OT/J) [OTJ06a] ist eine aspektorientierte Erweiterung von Java. Für OT/J existiert bereits eine Sammlung von Tools in Form von Erweiterungen (Plugins) der ***eclipse*** Entwicklungsumgebung [ECL06a]. Diese Tool-Sammlung soll nun um einen Formeditor erweitert werden, der es erlaubt, Dialoge für diese Sprache generieren zu lassen, die, in eine MVC-Architektur integriert, die besonderen Eigenschaften dieser Sprache ausnutzen. Um dieses Ziel zu erreichen, werden die folgenden Schritte nötig sein:

- Untersuchung, wie MVC mit OT/J realisiert werden kann. Hierzu wird zunächst ein Konzept erstellt und dieses anhand eines kleinen Beispiels untersucht.
- Entwicklung einer Toolunterstützung, so dass in ***eclipse*** Dialoge für OT/J erstellt werden können, die für die Verwendung mit MVC vorbereitet sind. Sofern es möglich ist, wird diese Unterstützung bereits ein komplettes Framework für den entsprechenden Dialog erzeugen.

In den folgenden Abschnitten wird nun zuerst ein Blick auf die Grundlagen geworfen:

1.2 Aufgaben eines Formdesigners

Dialoge sind die Benutzerschnittstelle eines Programms. Sie haben die Aufgabe, den Anwender möglichst intuitiv durch die einzelnen Arbeitsschritte

des Programms zu führen: Sie fragen Daten ab, präsentieren Ergebnisse und helfen dem Anwender, allgemein gesprochen, zu dem gewünschten Ergebnis zu gelangen.

Dies erreichen sie, indem sie dem Anwender eine Reihe von Widgets (Controls), wie Eingabefelder, Knöpfe, Auswahlflächen, Dropdownlisten, Tabellen, uvm., präsentieren, um Informationen abzufragen oder darzustellen.

Damit dies gut gelingt, gibt es viele Anforderungen an das Design und das Verhalten der Dialoge: Sie sollten sich an das Design der Plattform anpassen, Daten in einer intuitiven Reihenfolge abfragen, sofort auf Fehler hinweisen und evtl. sogar Vorschläge zur Korrektur geben, etc. Selbst verhältnismäßig kleine Dialoge haben damit bereits eine Menge Aufgaben zu erfüllen, die man dem Dialog nicht unbedingt ansieht.

Diese Anforderungen an Dialoge haben natürlich Folgen in der Implementierung: Der Programmcode wird schnell umfangreich und kompliziert. Die Programmierung eines Dialogs umfasst eine ganze Reihe von verschiedenen Aufgaben, die mehr oder weniger umfangreich sind. Einige dieser Aufgaben, wie z.B. die Widgetanordnung (oder: Design, Layout) können durch entsprechende Werkzeuge deutlich vereinfacht werden. Ein weit verbreitetes Werkzeug für Dialoge ist ein Formdesigner: Der Programmierer (oder auch ein Designer) malt einen Dialog, indem er Widgets graphisch anordnet und Eigenschaften der Widgets definiert. Das Ergebnis wird ihm sofort präsentiert und es wird Programmcode zur Erstellung eines solchen Dialogs generiert. Diesen per Hand zu programmieren wäre ein langwieriger Prozess, bis der Dialog das gewünschte Aussehen hat.

1.3 Zielsprache: ObjectTeams/Java (OT/J)

In dieser Diplomarbeit soll ein Formdesigner für OT/J realisiert werden, der die Vorteile und Stärken dieser Sprache nutzt. Außerdem soll gleichzeitig der Versuch unternommen werden, aus den zur Verfügung stehenden Daten ein Framework zu generieren, in dem der Entwickler nur noch verhältnismäßig wenig selbst programmieren muss, nämlich die Anbindung an das Datenmodell sowie die Fensterlogik. Da hierbei insbesondere die Eigenschaften der Sprache OT/J zum Tragen kommen, folgt nun eine kurze Einführung in die aspektorientierte Programmierung und in die Sprache OT/J selbst.

1.3.1 Aspektorientierte Programmierung (AOP)

Aspektorientierte Programmierung (AOP) ist ein relativ neues Konzept, das die objektorientierte Programmierung (OOP) durch das Konzept von Aspekten erweitert. Der Ansatz der aspektorientierten Programmierung begegnet dem Problem des sogenannten **crosscutting Code**: Einige Anforderungen an ein Programm verhindern eine saubere Trennung in einzelne Module. Solche Anforderungen können nicht in einem eigenen Modul zusammengefasst werden, sondern werden über das gesamte System verstreut, wodurch die Implementierung der Businesslogik durch diese „verunreinigt“ wird. Beispiele für solche Anforderungen sind Rechteprüfung, Logging oder Tracing: Schnittstellenmethoden müssen vor der eigentlichen Ausführung zunächst eine Rechteprüfung durchführen. Wird ein logging oder tracing benötigt, ist dies auch noch vorher bzw. nachher zu tun. Es entsteht in der eigentlichen Methode eine Menge Code, der mit der eigentlichen Aufgabe der Methode nichts zu tun hat. Das hat Folgen:

- Die eigentliche Businesslogik „verschwindet“ unter diesem crosscutting Code - die Wartung der Businesslogik wird deutlich erschwert.
- Die Wartung der verteilten Implementierungen wird durch die Verteilung selbst sehr teuer - im schlimmsten Fall müssen sämtliche Module modifiziert werden.
- Viel Code wird durch die Verteilung mehrfach geschrieben.

Der aspektorientierte Ansatz bietet nun ein Konzept, um auch solche „crosscutting“ Anforderungen sauber von der Businesslogik zu trennen. Diese Trennung hat zur Folge, dass viele Konzepte (Muster, Pattern), die in der objektorientierten Welt wohlbekannt sind, für aspektorientierte Programme von Grund auf neu überdacht werden müssen.

1.3.2 ObjectTeams/Java (OT/J)

ObjectTeams/Java (OT/J) ist eine aspektorientierte Spracherweiterung von Java. OT/J erweitert Java um die Konzepte für **Teams** und **Rollen**, wobei die Rollen die eigentlichen Aspekte im Sinne von AOP sind.

Ein Team ist im Wesentlichen eine Art Kombination eines Packages mit einer Klasse: Wie bei einem Package ist die Hauptaufgabe eines Teams das

Zusammenfassen von Klassen (hier: Rollen). Teams können jedoch, wie Klassen, Methoden und Attribute besitzen und auch vererbt werden.

Rollen hingegen sind Klassen, die jedoch an andere Klassen gebunden werden können und bilden somit die Aspekte.

Eines der klassischen Beispiele für AOP ist das Tracing: Es soll jeweils zu Beginn und am Ende einer Methode eine Ausgabe erfolgen. Dieses Problem ist mit OT/J recht einfach zu lösen. Angenommen, wir haben eine Klasse A mit den Methoden doA() und doB(). Diese sollen entsprechende Ausgaben erzeugen. Das Tracing ist sogenannter „crosscutting"-Code, der mittels eines Aspekts (ab nun: Rolle) gelöst werden soll. Der Code sähe in OT/J etwa folgendermaßen aus:

```
1  public team class TraceTeam {
2      public class ATracer playedBy A {
3          public void beginTrace() {
4              ...
5          }
6          public void afterTrace() {
7              ...
8          }

10         beginTrace <- before doA;
11         beginTrace <- before doB;
12         endTrace <- after doA;
13         endTrace <- after doB;
14     }

16     public TraceTeam() {
17         activate(ALL_THREADS);
18     }
19 }
```

Das ist ein sehr einfaches Tracing, z.B. wissen die Trace Methoden noch gar nicht, welche Methode eigentlich gerade getraced wird, aber es zeigt schon die wesentlichen Elemente.

Die **TraceTeam** Klasse ist durch das Schlüsselwort *team* als Team gekennzeichnet. Es hat alle Eigenschaften einer Klasse und kann somit von anderen Klassen und Interfaces erben sowie Attribute, Methoden und Unterklassen enthalten. In diesem Beispiel wird nur ein Konstruktor definiert.

Im Unterschied zu normalen Klassen, die implizit immer von **Object** erben, erben Teams immer implizit von **Team**. Eine der wichtigsten Methoden

von Team ist die Methode **activate()**: Teams in OT/J können jederzeit aktiviert bzw. deaktiviert werden. Was das genau bedeutet, wird bei der Rolle gleich verständlich. Festzuhalten ist hier nur, dass das Team in seinem Konstruktor sich gleich aktiviert.

Die Klasse **ATracer** ist als Teil eines Teams eine Rolle. Die Rolle wird zunächst ebenfalls wie eine normale Klasse deklariert, jedoch wird sie über ***playedBy A*** an die Klasse A gebunden - dieses ist nun die ***Basisklasse***. Durch diese Verbindung kann die Rolle nun an bestimmte Elemente gebunden werden. In diesem Fall definiert sie zwei (*callin*) Methoden, die jeweils an entsprechende Methoden der Basisklasse gebunden werden. Die Bindungen (Zeilen 10-13) beschreiben, wie diese Methoden an Methoden der Basisklasse gebunden werden. Diese *callin-Bindungen* bedeuten, dass die beginTrace() Methode aufgerufen wird *bevor* eine der Methoden doA() oder doB() der Basisklasse aufgerufen wird. afterTrace() hingegen wird jeweils *nach* dem Aufruf der Methoden doA() oder doB() aufgerufen.

Wird nun eine Instanz des umschließenden Teams erzeugt und aktiviert, dann wird für jede Instanz von A eine Rolle ATracer erzeugt[1] und in den beschriebenen Fällen die entsprechenden Trace Methoden aufgerufen. Ist das Team nicht aktiviert, so werden entsprechende callins auch nicht ausgeführt. In dem Beispiel hieße das, dass z.B. über einen Knopf in der Oberfläche das Tracing ein- und ausgeschaltet werden kann.

Neben den beschriebenen callin-Typen (*after*, *before*) existiert noch eine dritte Form: *replace*. Ein replace-Callin ersetzt die gebundene Methode komplett und wird statt der Basismethode aufgerufen. Das oben beschriebene Tracing könnte auch mittels eines replace-Callins erreicht werden. Der Code der Rolle sähe dann etwa folgendermaßen aus:

```
1  public class ATracer playedBy A {
2      public void trace() {
3          ... // do begin trace logging
4          try {
5              base.trace();
6          }
7          finally {
8              ... // do end trace logging
9          }
10     }
```

[1] Genaugenommen werden Rollen erst erzeugt, wenn das Basisobject das erste Mal *geliftet* wird. Wann dies genau passiert, ist in der Sprachdefinition [OTJ06b] nachzulesen.

```
12      trace <- replace doA;
13      trace <- replace doB;
14  }
```

Die trace() Methode führt nun das Tracing aus und kümmert sich um den Aufruf der ursprünglichen Methode durch das Kommando *base.trace()*. OT/J wird hier nun, je nach Ursprung des Aufrufs, *doA()* oder *doB()* aufrufen. Das Beispiel hat aber einen Haken: Wenn die Basismethoden nun Parameter hätten, würde dieser Weg so nicht funktionieren.

Eine Rolle kann auf alle Elemente ihrer Basisklasse zugreifen, wenn diese explizit in der Rolle definiert werden. Rollen sind hiermit insbesondere in der Lage, auch auf geschützte Elemente der Basisklasse zuzugreifen. Diese Definitionen erfolgen in Form von **callouts**. Die Syntax für einen callout ist dem eines callin ähnlich:

```
1 doX -> doSomething
2 setY -> set y;
3 getY -> get y;
```

Die erste Bindung heißt, dass bei Aufruf der Rollenmethode stattdessen die Methode *doSomething()* in der Basisklasse aufgerufen wird. Die beiden folgenden Bindungen sind jeweils Zugriffe auf das Attribut y der Basisklasse. Das bedeutet, dass bei einem Aufruf von *getY()* das Attribut y der Basisklasse ausgelesen wird.

Bei den einzelnen Bindungen können statt der Methodennamen auch komplette Signaturen angegeben werden. In Fällen, in denen die Signatur nicht eins zu eins abgebildet werden kann, besteht die Möglichkeit, ein Parametermapping zu definieren.

Obwohl innerhalb der Rolle das Basisobjekt über das Schlüsselwort *base* zur Verfügung steht, verhindert OT/J es, über dieses Schlüsselwort Methoden des Basisobjekts aufzurufen. Dies kann man durch callout-Bindungen umgehen, aber auch durch manuelles **lowering**. Die Begriffe **lifting** und **lowering** bezeichnen die Prozesse, wie die Rolle zu einem Basisobjekt (*lifting*) bzw. wie das Basisobjekt zu einer Rolle gefunden wird (*lowering*). Beides wird in der Regel automatisch durchgeführt, kann aber auch manuell erfolgen. Will man in einer Rolle auf das Basisobjekt zugreifen, ohne callout-Bindungen zu verwenden, muss man die Rolle demnach durch ein manuelles *lowering* in das Basisobjekt umwandeln. Im Falle des ATracers von oben sähe dies etwa so aus:

```
1    A myBase = this;
2    ... // use myBase
```

Es gibt noch mehr Möglichkeiten, bei denen ein *lifting* oder *lowering* durchgeführt wird. Auch gibt es einige Spracheigenschaften mehr, die hier noch nicht genannt wurden, wie z.B. **Guards**. Die bisherigen Ausführungen sollten zunächst einen groben Überblick über die Möglichkeiten und die grundlegendsten Aspekte von OT/J geben. Details können in der Sprachbeschreibung von OT/J [OTJ06b] nachgelesen werden.

1.3.3 Vergleich: C#, Java, OT/J

Die Vorteile des aspektorientierten Programmierens sind anhand des Pattern *Observer* (vgl. [GAM95], Abbildung 1.1) am einfachsten zu sehen. Dieses Pattern ist zum einen eines der wichtigsten in OOP und zum anderen ein zentrales Muster bei der Implementierung einer auf dem *Model-View-Controller* Stil basierenden Architektur (vgl. Kapitel 2).

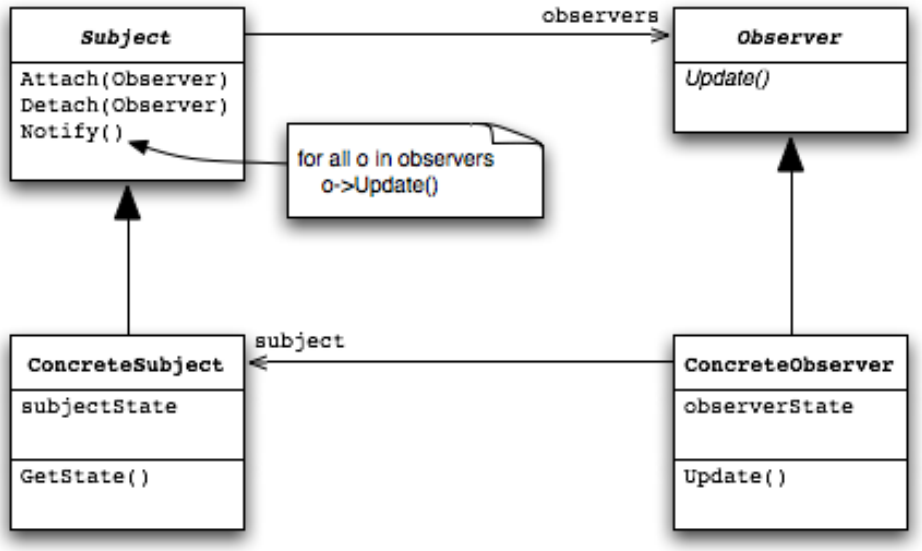

Abbildung 1.1: Subject Observer Pattern

Java

Java bietet keinerlei Konzepte, die bei der Implementierung des Observer-Pattern im Besonderen helfen. Daher muss in Java jedes Detail implementiert werden. Das Subject wird in einfachster Form folgendermaßen implementiert:

```
1  public class DataObject {
2      // Änderungen an diesem Attribut sollen gemeldet werden:
3      private int data;
4
5      public DataObject(int data) {
6          this.data = data;
7      }
8
9      public int getData() {
10         return data;
11     }
12
13     public void setData(int data) {
14         this.data = data;
15         // benachrichtige Beobachter
16         notifyDataChanged();
17     }
18
19     public void calcData() {
20         ... // führe Berechnung durch
21
22         // benachrichtige Beobachter
23         notifyDataChanged();
24     }
25
26     // Subject Implementierung
27     ArrayList<IObserver> observers;
28
29     protected void notifyDataChanged() {
30         if(observers != null)
31             // iteriere über Observer und melde die Veränderung:
32             for(IObserver observer : observers)
33                 observer.update(this);
34     }
35
36     public void attach(IObserver observer) {
37         if(observers == null)
38             observers = new ArrayList<IObserver>();
```

```
39          observers.add(observer);
40      }

42      public void detach(IObserver observer) {
43          if(observers == null)
44              throw new InvalidMethodException("detach needs
                    attach before!");
45          observers.remove(observer);
46      }
47  }
```

Diese einfache Implementierung eines kleinen Subjects zeigt recht deutlich die Nachteile. Sämtliche Methoden, die Änderungen am Attribut *data* vornehmen, müssen sich nach der Änderung um die Benachrichtigung der Observer kümmern: das ist der sogenannte *crosscutting*-Code.

Ein weiteres Problem ist, dass derjenige, der dieses Subject implementiert, nicht unbedingt weiß, an welchen Dingen ein Beobachter interessiert sein wird. Ein zentraler Aspekt des Subject-Observer-Pattern ist ja eben die Abkopplung der Beobachter. In der Folge wird er sich also an allen Stellen um eine Benachrichtigung kümmern, um möglichst jedes Beobachter-Szenario abzudecken. Es wird aber nicht unbedingt von allen Gebrauch machen, mit der Folge, dass hier eine Menge Code produziert wird, der potentiell nicht mal benötigt wird. Es entstehen Performance- und Wartungsprobleme, die im Grunde überflüssig sind.

Der Beobachter ist hier weniger von Bedeutung, da dessen Aufgaben immer gleich sind: Um bei Änderungen von Daten Aktionen ausführen zu können, muss er sich bei dem entsprechenden Subject anmelden, die gemeldeten Änderungen entsprechend bearbeiten und sich wieder beim Subject abmelden. Hier ist keinerlei „crosscutting“-Code vorhanden, daher sieht die Implementierung des Observers, bis auf formelle Details, immer gleich aus. Der Vollständigkeit halber folgt hier ein kleiner Beispiel-Observer für das oben beschriebene Subject:

```
1  public interface IObserver {
2      void update(DataObject data);
3  }

5  public class ConcreteObserver implements IObserver {

7      public void startObserve(DataObject data) {
8          data.attach(this);
```

```
9     }

11    public void stopObserve(DataObject data) {
12        data.detach(this);
13    }

15    public void update(DataObject data) {
16        System.out.println("got update! Data: " + data.
              getData());
17    }

19 }
```

C#

C# bietet ein Sprachelement, das bei der Implementierung des Observer Pattern hilft: das *Delegate*.

Ein Delegate ist zunächst ein besonderer Typ: Er beschreibt sich durch eine komplette Signatur. Wenn man so will, ist es ein Signatur-Typ. Eine Instanz dieses Typs kann man nun ähnlich wie Funktionspointer in C/C++ (vgl. §7.7 in [STR97]) verwenden: Man weist ihr eine *Funktion*[2] zu und kann diese Delegate-Instanz dann verwenden, als wäre sie selbst eine *Funktion*.

Tatsächlich kann ein Delegate aber mehr als ein Funktionspointer. Denn zum einen kann einem Delegate jede Methode zugeordnet werden, die der Signatur des Delegate entspricht, und zum anderen kann ein Delegate auch als Container für eine ganze Menge von Methoden gleicher Signatur herhalten. Die Verwendung als Container hat nur eine Einschränkung: Wird das Delegate „aufgerufen“, so ist der return-Wert per Definition nicht vorhersehbar[3].

Die Verwendung eines Delegates als Container für Methoden ist für die Subject-Observer Implementierung sehr hilfreich: Das Subject stellt nur eine

[2] Der Begriff *Funktion* ist hier nur ein begrifflicher Kompromiss. In Java und C# gibt es diesen eigentlich gar nicht. In C/C++ wird hier jedoch zwischen Methode („Funktion einer Klasse“) und Funktion unterschieden. Im Zusammenhang mit Funktionspointern ist diese aber auch in C/C++ nicht sehr eindeutig. Hier können nämlich einem Funktionspointer im Grunde keine Methoden zugeordnet werden, es gibt aber die Ausnahme von statischen Methoden, bei denen es dann doch geht.

[3] Obwohl man durch Tests durchaus eine Vorhersehbarkeit feststellen kann, die zumindest für die getestete .NET-Runtime gilt. Solche Dinge gilt es natürlich zu vermeiden!

Delegate-Instanz zur Verfügung, auf der sich jeder beliebige Observer anmelden kann. Die Benachrichtigung selbst reduziert sich auf den Aufruf des Delegate - eine eigene Observer-Listenverwaltung entfällt ganz.

Ein weiterer Vorteil entsteht durch die spezielle Anonymisierung der Observer-Methode. Da das Delegate ausschließlich eine Signatur vorgibt, ist kein Interface für den Observer notwendig. Außerdem kann der Observer seine Methoden für die Benachrichtigung benennen wie er will - solange die Signatur stimmt. Das wiederum hat zur Folge: Keine Interface-Inflation, keine *name-clashes* bei der Beobachtung von verschiedenen Subjects und keine unnötigen Implementierungen von nicht benötigten Observer-Interface Methoden.

Das Ganze hat nur einen einzigen Haken: Die An- und Abmeldung (besonders die Abmeldung) sieht ein wenig gewöhnungsbedürftig aus (siehe Beispiel).

Der allgemein anerkannte Java-Codingstyle gibt vor, dass Attribute, entsprechend dem OO-Paradigma, gekapselt (privat) sind und nach außen über getter- und setter-Methoden verfügbar gemacht werden. Damit wird eine Lücke im Sprachumfang von Java geschlossen, die es in C# nicht gibt: Properties (Eigenschaften). Properties sind spezielle Methoden, die aus der Sicht des Benutzers aussehen wie Attribute. Die meisten Properties beziehen sich auch auf ein Attribut und sind daher den getter- und setter-Methoden in Java ähnlich. Daher sind diese Properties in C# der ideale Punkt für das Melden von Änderungen.

Die Implementierung eines Subjects mit denselben Aufgaben, wie das Java-Beispiel von oben, sieht nun wie folgt aus:

```
1  public class DataObject {
2      private int data;
3
4      public Data(int data) {
5          this.data = data;
6      }
7
8      public int Data {
9          get {
10             return data;
11         }
12         set {
13             data = value;
14             // benachrichtige observer:
15             NotifyDataChanged();
```

```
16        }
17    }


20    public void CalcData() {
21        ... // führe Berechnungen durch

23        // benachrichtige observer:
24        // Hier gibt es zwei Varianten:
25        // Variante 1: Verwendung des Property
26        Data = newCalculatedData;
27        // Variante 2; Direkte Attributänderung und manuelle Notifizierung
28        data = ...;
29        NotifyDataChanged();
30    }

32    // Nun die Implementierung der Subject Methoden:
33    // Einführung des delegate
34    public delegate int NotifyDelegateType(Data data);
35    public NotifyDelegateType NotifyDelegate;

37    protected void NotifyDataChanged() {
38        if(NotifyDelegate != null)
39            NotifyDelegate(this);
40    }
41 }
```

Wie man sieht, hilft das Delegate ein wenig, die Masse an Code zu reduzieren. Die einzelnen Benachrichtigungen an Stellen, an denen das Attribut verändert wird, sind weiterhin nötig, können aber durch konsequente Verwendung des Property für *data* auch noch reduziert werden.

Der Observer wäre in diesem Fall im Wesentlichen identisch mit der Java-Variante. Die An- und Abmeldung an das Subject erfolgt hier direkt über das Delegate und könnte wie folgt aussehen - Hier ist auch die oben erwähnte gewöhnungsbedürftige Anmeldung an Delegates zu sehen:

```
1 public class ConcreteObserver {

3    public void startObserve(DataObject data) {
4        data.NotifyDelegate += new CalcData.
             NotifyDelegateType(update);
5    }
```

```
7     public void stopObserve(DataObject data) {
8        data.NotifyDelegate -= new CalcData.
             NotifyDelegateType(update);
9     }

11    public void update(DataObject data) {
12       System.out.println("got update! Data: " + data.
             getData());
13    }

15 }
```

ObjectTeams/Java (OT/J)

In OT/J hingegen ist in der Datenklasse keinerlei zusätzlicher Code vonnöten. Der „crosscutting“-Code entfällt vollkommen. Entsprechend sähe hier das „Subject“ nur noch folgendermaßen aus:

```
1  public class DataObject {
2     // Änderungen an diesem Attribut sollen gemeldet werden:
3     private int data;

5     public DataObject(int data) {
6        this.data = data;
7     }

9     public int getData() {
10       return data;
11    }

13    public void setData(int data) {
14       this.data = data;
15    }

17    public void calcData() {
18       ... // führe Berechnung durch
19    }
20 }
```

Im Gegensatz zu den Beispielen von Java und C# ist hier jedoch der Observer von besonderem Interesse, da die Anmeldung nun nicht mehr über Methoden oder Delegates in der Subject-Klasse erfolgt, sondern er allein für

alles verantwortlich ist. In OT/J ist der Observer eine Rolle und besteht daher aus einer Methode, die bei Änderungen aufgerufen werden soll, und Bindungen. Diese Bindungen ersetzen quasi die Anmeldung, die bei den anderen Sprachen notwendig ist.

Der Observer würde nun folgendermaßen implementiert werden:

```
1  public team class ObserverTeam {

3      private class Observer playedBy Data {
4          // Die callback Methode, die bei Änderungen an Data aufgerufen werden
5          // soll:
6        public void update() {
7              // bearbeite Notifizierung
8          }

10         // Variante 1: update wird immer gerufen, nachdem das Attribut data
11         // gesetzt wird
12         // (ACHTUNG: In der aktuellen OT/J Version noch nicht möglich)
13         // Beachte: Diese Form bricht die Kapselung von Data
14         update <- after set data;

16         // Variante 2: update wird nach jeder Methode aufgerufen, die Änderungen
17         // an Data durchführt:
18         update <- after setData;
19         update <- after calcData;
20     }

22     public ObserverTeam() {
23         // aktiviere das team, um ab jetzt callins gemeldet zu bekommen
24         activate();
25     }
26 }
```

Es erfolgt in dem Sinne keine Anmeldung mehr, sondern der Observer beschreibt nur noch, wann er aufgerufen werden möchte. Die OT/J Runtime hat nun die Aufgabe, die entsprechenden Aufrufe zu realisieren. Das Subject ist davon überhaupt nicht betroffen - es existiert kein „crosscutting“ Code mehr.

Allerdings muss man bei diesem Beispiel auch erwähnen, dass dieser Observer sich durchaus von den anderen Beispielen unterscheidet, was z.T. auch dessen Verhalten angeht:

1. Der Observer existiert nur innerhalb eines Teams. Die Benachrichtigungen erfolgen nur solange wie das Team auch aktiv ist.

2. Der Observer kann ausschließlich Objekte der Basisklasse beobachten. Die Observer der anderen Beispiele waren hier nicht derart eingeschränkt.

3. Dieses konkrete Beispiel hat zur Folge, dass *alle* Instanzen der Klasse *Data* beobachtet werden. Das Team stellt sicher, dass für jede Instanz von *Data* auch eine Instanz von Observer erzeugt wird. Eine selektive Anmeldung, wie das bei den anderen Beispielen möglich ist, funktioniert hier nicht.

4. Ein solcher Observer kann nicht mehrere Objekte gleichzeitig beobachten.

Die Einschränkungen sind in dem Konzept von Teams und Rollen begründet. Die drei letzten Einschränkungen sind jedoch mit OT/J Mitteln durchaus lösbar. Über *guards* (vgl. OT/J Sprachbeschreibung [OTJ06b]) lassen sich Data-Instanzen filtern, für die Rollen erzeugt werden (Punkt 3). Durch eine andere Interpretation des Teams und des Observers lässt sich auch eine Lösung für die Einschränkungen 2 und 4 finden: Wenn man annimmt, dass das Team der eigentliche Observer ist, kann über *guard predicates* und verschiedene Rollen in der Summe das gleiche Ergebnis erzeugt werden wie bei den Beispielen oben.

Eine Implementierung könnte dann so aussehen:

```
1  public team class Observer {

3      private class DataObserver playedBy Data
4          base when (hasRole(base))
5      {
6          ... // Implementierung wie oben
7      }

9      public void startObserve(Data as DataObserver data) {
10         // declared lifting: Nun ist data geliftet und hasRole() liefert true: callins sind
               aktiv
11     }

13     public void stopObserve(Data data) {
14         unregisterRole(data);
15         // hasRole() liefert nun false: keine callins mehr für data
```

```
16      }

18      public ObserverTeam() {
19        activate();
20      }
21  }
```

Ich möchte hier nicht weiter auf dieses Thema eingehen. Problemlösungen in OT/J sehen nicht selten deutlich anders aus als in OO-Sprachen. Mit diesem kleinen Ausflug wollte ich jedoch ein Gefühl für die andere Herangehensweise an Probleme in OT/J vermitteln. Mehr Informationen und mehr Diskussionen über dieses Thema sind über die Homepage von ObjectTeams [OTJ06a] verfügbar.

1.4 Oberflächendesigner

Bei der Frage nach der Realisierung von Oberflächen-Designern ist ein Blick auf andere aktuelle Konzepte sinnvoll. Neben klassischen Ansätzen, in denen Oberflächen in der Zielsprache auch kodiert werden, wird ein kurzer Blick auf einen vermutlich zukunftsträchtigen Ansatz mit XML als Beschreibungssprache geworfen. Als Beispiel für die Java-Welt stelle ich kurz vor, wie NetBeans [NB06a] Oberflächen behandelt. Daraufhin wird beschrieben, wie Oberflächen mit Microsoft Visual Studio (.NET) erzeugt werden.

1.4.1 Allgemeines

Klassen, die Fenster oder Dialoge beschreiben, unterscheiden sich i.d.R. von herkömmlichen Klassen insbesondere durch schlecht lesbaren Code. Programmierrichtlinien, die Code lesbarer und damit auch gut zu warten machen sollen, scheitern bei solchen Klassen. Gründe dafür sind recht offensichtlich: Dialoge bestehen aus diversen Widgets, die jeweils eine ganze Reihe von Properties besitzen und zudem auch noch diverse Events feuern können. Die Dialogklasse (meist ist es auch nur eine) muss diese Widgets erzeugen, konfigurieren und steuern. Dadurch entsteht eine Klasse mit sehr langen Methoden und sehr vielen Attributen. Folge: Der Code ist schlecht lesbar. Ein Programmierer, der einen solchen Dialog per Hand schreiben muss, wird eine ganze Weile benötigen, diesen zu erstellen.

Sinn und Zweck eines Oberflächen-Designers ist es nun, dem Entwickler die Möglichkeit zu geben, Dialoge und Fenster bequem in einem graphischen Editor zu erstellen. Der Oberflächendesigner generiert aus diesen Informationen dann den entsprechenden Code, der für die Erstellung des Fensters zur Laufzeit erforderlich ist. Mehr noch: Der Designer erlaubt auch das Erstellen von Eventhandlern und z.T. auch das Binden an Daten.

Oberflächen-Designer gibt es reichlich. Daher stelle ich zunächst eine kleine Auswahl vor:

1.4.2 eclipse - Visual Editor

Der Visual Editor ist ein Teilprojekt von eclipse und (derzeit) ein GUI Designer für Java-Oberflächen. Er ist weitestgehend unabhängig vom GUI-Framework, das eingesetzt werden soll. Er unterstützt von Hause aus SWT, AWT und Swing[4].

Wie auch aktuelle GUI-Designer, serialisiert er all seine Informationen in Form von (Java-) Code. Es wird hierzu mindestens eine Methode generiert, die *createSShell()* heißt. Umfasst der Dialog auch Container-Widgets, wie z.B. eine Groupbox, wird deren Initialisierung in eine eigene Methode ausgelagert. Sämtliche Widgets werden zudem in Form von Attributen generiert.

Events werden in Form von anonymen Klassen gefangen. Die eigentliche Event-Methode ist dann vom Entwickler auszufüllen. Eigene Event-Methoden, wie sie häufig in anderen GUI-Designern generiert werden, gibt es hier nicht.

Der VE unterstützt Code-Roundtrip. Das heißt: Änderungen am generierten Code werden vom VE interpretiert und in seine Design-Präsentation übernommen. Fehlerhafter Code oder Code, mit dem der VE nichts anfangen kann, führt dann aber dazu, dass er keine Präsentation mehr erstellen kann.

1.4.3 NetBeans

NetBeans ist eine Entwicklungsumgebung, die SUN unter [NB06b] zum freien download zur Verfügung stellt. Der GUI Designer von NetBeans sieht auf den ersten Blick dem VE sehr ähnlich: eine Design- und Code-Ansicht, eine

[4]Da ich mich in dieser Arbeit auf das SWT-Framework beschränke, gelten die Aussagen über den VE für die SWT-Implementierung. Sie stimmen dennoch weitestgehend mit den anderen Frameworks überein, müssen aber nicht.

Palette, ein Property-Fenster und eine Baum-Darstellung des Dialogs. Die Palette ist hier jedoch auf AWT und Swing beschränkt. Das Zeichnen eines Dialogs läuft hingegen deutlich flüssiger. Zum einen sind kaum bis gar keine Verzögerungen zu bemerken, zum anderen ist die Präsentation korrekt: Der VE hat zum Teil Probleme, Fenster korrekt darzustellen. Unter Linux beispielsweise, ist die Präsentation und das tatsächliche Ergebnis nicht ganz dasselbe. Es scheint hier gewisse Eigenschaften des WindowManagers nicht darstellen zu können.

NetBeans unterstützt jedoch keinen Code-Roundtrip. Der generierte Code wird durch NetBeans geschützt und ein Kommentartext weist explizit darauf hin, dass manuelle Änderungen am Code vom Designer wieder überschrieben werden. Der Schutz besteht hierbei darin, dass der Code zunächst nicht sichtbar ist (zugeklappt) und außerdem in diesem Bereich keine Änderungen vorgenommen werden können.

Ansonsten ähnelt der Code dem des VE: Es wird eine zentrale Methode generiert, die den gesamten Code für das Layout enthält. Im Gegensatz zum VE werden aber auch Container-Widgets hierhin generiert. Der VE generiert für Container-Widgets hingegen eigene Methoden, die den Sub-Container erzeugen. Events werden ein klein wenig anders generiert: Die Klasse, die das Event empfängt, ist auch hier anonym, die Empfängermethode delegiert das Event jedoch an eine Methode der Dialogklasse. Der VE hingegen generiert lediglich die anonyme Event-Klasse. Da bei NetBeans der Initialisierungs-Code schreibgeschützt ist, ist die VE-Variante hier natürlich gar nicht erst möglich.

1.4.4 MS Visual Studio .NET

VS.NET enthält einen sehr mächtigen und für den Benutzer sehr komfortablen Dialogeditor. Wie der VE und auch NetBeans serialisiert er seine gesamten Informationen als reinen Code. Zu diesem Zweck legt er in der betreffenden Klasse für jedes Dialogelement ein Attribut an und implementiert die gesamte Initialisierung nach den Vorgaben des Benutzers in einer Methode namens „InitializeComponents“. Wie auch in NetBeans wird hierhin der gesamte InitialisierungsCode generiert, also auch der für Container-Widgets.

Diese Methode wird im Normalfall vor dem Entwickler verborgen, ist jedoch auch ohne weiteres einseh- und editierbar - wobei durch Kommentare der Betrachter unmissverständlich auf die Gefahr hingewiesen wird, dass manuelle Änderungen potentiell zu Problemen mit dem Editor führen können. Ein Code-Roundtrip, wie im VE, ist hier jedoch möglich.

Events werden über Delegates realisiert (vgl. 1.3.3) und entsprechend anders registriert. Für jedes zu empfangene Event wird eine eigene Methode angelegt und diese bei dem entsprechenden Delegate in der Initialisierungs-Methode registriert.

Im aktuellen Visual Studio (2005 für das Framework 2.0) nutzt der Editor als zusätzliche „Sicherheit“ das neu eingeführte Konzept von „Partial Classes“: Klassendefinitionen können damit über mehrere Dateien verteilt werden. Der Dialogeditor generiert all den Code, den ein Entwickler besser nicht anfassen soll, in eine eigene Datei und versteckt sie normalerweise vor dem Entwickler. Dennoch ist es dem Entwickler weiterhin möglich, diesen Code per Hand zu verändern.

1.4.5 Zusammenfassung

Die aktuellen GUI-Designer gehen im Wesentlichen den gleichen Weg: Sämtliche Informationen für das Dialog-Layout werden ausschließlich in Form von Code vorgehalten. Frühere Ansätze gingen hier zum Teil andere Wege: Auf der Win32-Plattform gab es für Dialog-Layouts sogenannte Ressourcen-Dateien, die diese Informationen enthielten. Dialog-Klassen referenzierten die Widgets hier über IDs. Durch diese Trennung war es, zumindest theoretisch, möglich, die Aufgaben bei der Entwicklung von Dialogen besser zu trennen als heute: Ein Designer konnte sich dem reinen Design widmen und unabhängig vom Programmierer arbeiten, und der Programmierer brauchte sich nicht um das Design kümmern und konnte sich auf die Implementierung konzentrieren. Auch wurde der Code nicht durch etliche Zeilen für das Layout unnötig lang.

Um dem Problem der langen Klassen zu begegnen, gehen einige Ansätze heute wieder einen Schritt zurück und suchen Wege, wie das Layout vom Code getrennt werden kann. Die Einführung von *Partial Classes* war ein erster Schritt, aber aktuelle Entwicklungen gehen nun soweit, das Layout komplett in XML auszulagern. Hier sei nur kurz auf **XUL** [XUL06] und die **Windows Presentation Foundation** [MS06]verwiesen, die diesen Weg beschreiten.

Der VE wird in Kapitel 3 noch genauer untersucht, da er die Grundlage dieser Arbeit ist. Im folgenden Kapitel wird jedoch zunächst der Model-View-Controller Architekturstil, in Hinblick auf dessen Realisierung und die dabei auftretenden Probleme, beschrieben.

Kapitel 2

Einführung: Model-View-Controller Stil

2.1 Vorwort

Der Begriff *Stil* in diesem Zusammenhang ist etwas ungewöhnlich und bedarf der Erklärung: In der Literatur findet man in der Regel eher die Bezeichnung *Architektur* oder auch *Pattern*, seltener auch *Paradigma*. Grund für dieses Namenswirrwarr ist das Problem, dieses *Ding* richtig einzuordnen. Für ein *Pattern* ist das, was mit MVC beschrieben wird, eigentlich zu grob: Patterns definieren sehr genau, wie Klassen und Objekte miteinander in Beziehung stehen, welche Rollen sie haben und wie sie zusammenspielen. Das jedoch macht MVC nicht, hier geht es eher um das konzeptionelle Zusammenspiel.

Der Begriff Architektur trifft die Sache schon besser, ist aber im Grunde dasselbe auf etwas höherer Abstraktionsebene. MVC ist jedoch nicht eine feste Vorgabe, sondern eher eine Beschreibung von Komponenten, die bei einer GUI Anwendung zusammenspielen.

Das Wort *Paradigma* scheint hingegen schon zu allgemein zu sein. MVC ist schließlich keine grundlegende Denk- sondern eher eine Betrachtungsweise.

Aufgrund dieser begrifflichen Ungenauigkeiten habe ich mich daher für das Wort *Stil* entschieden. Dieser Begriff scheint mir, ohne hiermit eine neue Diskussion entfachen zu wollen, am passendsten und ist hinreichend neutral.

2.2 MVC Allgemein

Der MVC-Architekturstil ist sehr allgemein gehalten und in der Abbildung 2.1 dargestellt. Das wesentliche Merkmal dieses Modells ist die Dreiteilung des Systems in Model, View und Controller. Das Modell beschreibt die darzustellenden Daten, die Oberfläche und den Controller.

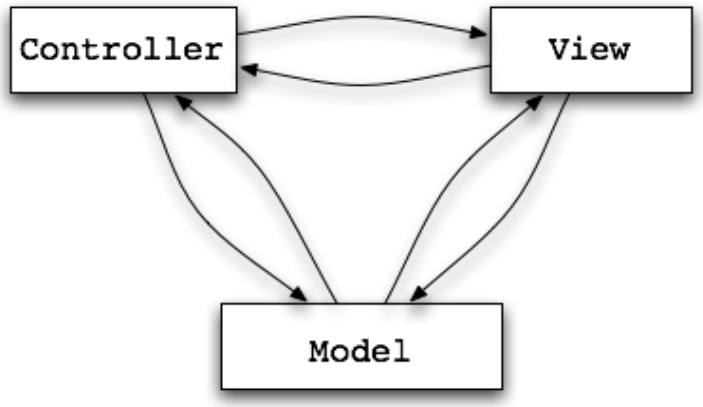

Abbildung 2.1: Model View Controller

Insbesondere in der objektorientierten Programmierung macht dieses Modell einige Probleme und steht teils sogar in Konflikt mit anderen übergeordneten Programmiermodellen. Daher wird zunächst dieser Bereich betrachtet.

2.3 MVC in OOP

Es gibt eine ganze Reihe Probleme im Zusammenhang mit Oberflächen, speziell im Zusammenhang mit MVC, und OOP bzw. deren Programmierstilen:

2.3.1 Kapselung

Ein entscheidender Faktor bei der Implementierung von Klassen in einer objektorientierten Sprache ist die Kapselung. Eine Klasse soll ihre Daten kapseln und anderen nur unbedingt notwendige Aktionen zur Verfügung stellen. Hierbei steht im Allgemeinen die Businesslogik im Vordergrund. Oberflächen haben aber in der Regel andere Anforderungen an eine solche Klasse als die

Businesslogik -z.B. soll ein Monitor durchaus den inneren Zustand eines Objektes visualisieren können, in der Businesslogik soll dieser jedoch verborgen (oder gekapselt) werden.

2.3.2 n-Tier Architekturen

n-Tier Architekturen (Schichtenarchitekturen) geben vor, dass eine Applikation aus mehreren Schichten besteht. Jede agiert ausschließlich mit ihren Nachbarn, indem sie die Schnittstelle der tieferen Schicht verwendet und eine Schnittstelle für höhere Schichten anbietet. Eine gängige Schichtenarchitektur ist die 3-Schichtenarchitektur (Abbildung 2.1), die zwischen Daten, Businesslogik und Präsentation unterscheidet.

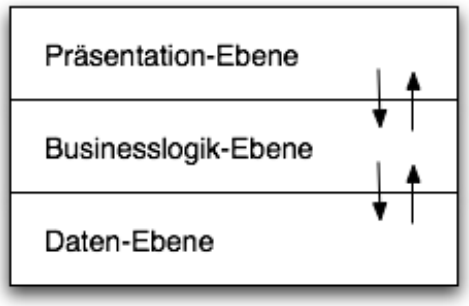

Abbildung 2.2: 3-Schichten-Architektur

So gut das Modell auch aussieht: Da die Oberfläche häufig auch Informationen aus der Datenschicht darstellen muss, die von der Businesslogik nicht angeboten werden, folgt, dass die Businesslogik-Schicht Daten aus der Datenschicht durchreichen muss. Es wird also allein zum Zweck der Präsentation unnötiger Code benötigt. Eine Alternative zum Durchreichen der Informationen der Datenschicht an die Präsentations-Schicht ist die Variante, der Präsentations-Schicht direkten Zugriff auf die Daten zu ermöglichen. Das hat jedoch zur Folge, dass das reine Schichtenmodell an dieser Stelle „aufgeweicht“ werden muss (Abbildung 2.3).

2.3.3 Benachrichtigungen

MVC beschreibt neben der Trennung der drei Komponenten auch deren Beziehungen zueinander: Sie sind jeweils bidirektional. Das heißt eben auch,

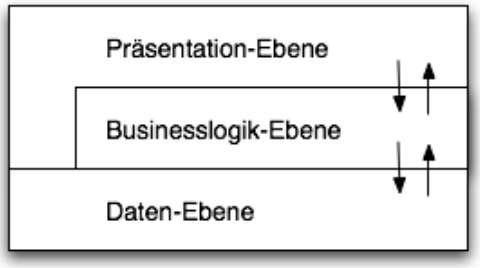

Abbildung 2.3: Aufweichen der 3-Schichten-Architektur

dass das Modell etwas an die View und/oder an den Controller schicken können muss. Entsprechend der Probleme der Kapselung (vgl. 2.3.1) haben wir hier das Problem, dass die Klassen des Modells nicht nur Daten nach außen geben, sondern zudem auch noch Code für die Benachrichtigung an die View und/oder an den Controller enthalten müssen.

Ein weiteres Problem in diesem Bereich ist auch die bidirektionale Beziehung zwischen View und Controller. In herkömmlichen Systemen enthalten GUI-Klassen sämtliche Widgets. Wird nun ein Controller extern realisiert, so wird er zwangsläufig auf die meisten Elemente der Oberfläche zugreifen oder zumindest über alle möglichen Aktionen dort informiert werden müssen. Ein nicht unerheblicher Aufwand.

2.3.4 Folgerung

In OOP ist es kaum möglich, wirklich sauber eine Oberfläche zu realisieren. Interfaces werden erweitert, Daten verfügbar gemacht und nicht selten wird logikfremder Code in Businessobjekte integriert. Um wenigstens das letzte der genannten Probleme zu umgehen, wird daher in den meisten Fällen ein Spezialfall des MVC implementiert: das Document-View-Model. Hier verschmelzen Controller und View zu einer Klasse. In GUI-Frameworks trifft man bei komplexeren Widgets (optionale) Controller an, die über fest definierte Interfaces die Kommunikation zwischen Widget und den Daten organisieren. Das entlastet zwar den Anwender, aber es muss weiterhin logikfremder Code implementiert werden.

2.4 MVC in AOP

Die Vorteile von AOP gegenüber OOP sind genau die, die benötigt werden, um die Probleme in OOP zu lösen. Die Kapselung von Daten, die stört, wenn GUI-relevante Daten versteckt werden, kann man mit AOP umgehen. Crosscutting-Code aufgrund der Implementierung des Observer-Pattern fällt gänzlich weg. Wie MVC in AOP, speziell mit OT/J, realisiert werden kann, ist Teil dieser Arbeit und wird später noch genauer untersucht (vgl. Kapitel 4).

2.5 Zusammenfassung

Das wesentliche Problem bei der Realisierung von MVC mit OOP liegt in der Kapselung der Daten und dem Problem, dass das Modell Änderungen an den Controller und die View propagieren muss und somit Aufgaben hat, die nicht in die Domäne des Datenmodells gehören. Neben den Erläuterungen hier, wurde dieses Thema schon in [AOSD03] behandelt und die Probleme von OOP aufgezeigt (Abschnitt 2.2: „Weak points of MVC based design“).

Kapitel 3

Visual Editor

In diesem Kapitel wird der VisualEditor (VE) vorgestellt. Zunächst wird gezeigt, wie der VE Dialoge aus Anwendersicht erzeugt und wie der generierte Code strukturiert ist. Im Anschluss wird das VE Projekt in groben Zügen beschrieben.

3.1 Die Anwendersicht

Der Visual Editor ist heute in der Lage, Dialoge zu zeichnen und als Code zu generieren. Er nutzt außer der Java-Datei keine weiteren, um die Dialoginformationen zu speichern. Damit er aus der Java-Datei den Dialog wieder darstellen kann, verwendet er verschiedene Pattern, um den passenden Code zu finden. Für die eigentliche Darstellung des Dialogs startet der VE eine weitere Virtual Machine, in der ein Dialog entsprechend der Vorlage erzeugt wird. Dieser wird dort ausgeführt und ein Bild davon erzeugt. Dieses Bild wird im Editor von eclipse wieder dargestellt. Jede Änderung am Dialog wird so zu der separaten VM geschickt, dort wieder in ein Bild umgewandelt und zurückgeschickt.

Hier ist übrigens der entscheidende Unterschied zwischen .NET und Java in der GUI Entwicklung zu sehen. In .NET kennt jedes Widget zwei Modi: Einen Laufzeitmodus und einen Designmodus. Im Designmodus kann sich das Widget anders darstellen und auch Eigenschaften anbieten, die zur Laufzeit nicht vorhanden sind. Ebenfalls sind hier spezialisierte Konfigurationseditoren möglich, die insbesondere bei komplexen Widgets viele Vorteile haben.

Zurück zum Visual Editor: Zunächst werfe ich einen Blick auf den vom Visual Editor generierten Code.

Nach dem Anlegen einer Visual Class sieht der Code folgendermaßen aus (eine SWT-Shell):

```
1  public class MyDialog {
2
3      private Shell sShell = null;
4
5      /**
6       * This method initializes sShell
7       */
8      private void createSShell() {
9          sShell = new Shell();
10         sShell.setText("Shell");
11         sShell.setSize(new Point(300, 200));
12         sShell.setLayout(new GridLayout());
13     }
14
15 }
```

Als erstes sieht man, dass ein Attribut für die Shell generiert wird. Diese kann von der Anwendung später verwendet werden, um den Dialog auch darzustellen. Des Weiteren wird eine Methode erzeugt, die für die Erzeugung und Konfiguration der Shell zuständig ist. Sämtliche Änderungen an den Eigenschaften der Shell im VE werden hierhin generiert bzw. von hier gelesen.

Nun kommt ein Button hinzu:

```
1  public class MyDialog {
2
3      private Shell sShell = null;
4      private Button button = null;
5
6      /**
7       * This method initializes sShell
8       */
9      private void createSShell() {
10         GridLayout gridLayout = new GridLayout();
11         gridLayout.numColumns = 1;
12         sShell = new Shell();
13         sShell.setText("Shell");
14         sShell.setLayout(gridLayout);
15         sShell.setSize(new Point(300, 200));
16         button = new Button(sShell, SWT.NONE);
17     }
```

```
19 }
```

Was ist passiert? Es kam ein Attribut für den Button hinzu. Der VE erzeugt demnach für jedes Widget ein Attribut. Außerdem wird der Button in createSShell, wie auch die Shell, erzeugt und konfiguriert.

Nun noch ein Container-Widget mit einem Button darin:

```
1  public class MyDialog {

3      private Shell sShell = null;
4      private Button button = null;
5      private Group group = null;
6      private Button button1 = null;

8      /**
9       * This method initializes sShell
10      */
11     private void createSShell() {
12         /* Code, wie oben */
13         createGroup();
14     }

16     /**
17      * This method initializes group
18      *
19      */
20     private void createGroup() {
21         group = new Group(sShell, SWT.NONE);
22         group.setLayout(new GridLayout());
23         button1 = new Button(group, SWT.NONE);
24     }

26 }
```

Hier ist nun einiges passiert: Wieder wird für jedes Widget ein Attribut angelegt. Da nun ein zweiter Button hinzugekommen ist, wird dem Namen einfach ein Zähler angehängt. Für die Groupbox wird jetzt jedoch nicht der Code zum Erzeugen und Konfigurieren in createSShell generiert, sondern ausgelagert in eine eigene Methode namens createGroup. Die Namen der Attribute können natürlich über die Eigenschaften editiert werden. Nennt man die Gruppe z.B. myGroup, so wird das Attribut entsprechend benannt, und die Methode wird zu *createMyGroup()*.

Sämtliche Widgets werden derart generiert: Entweder als Container mit eigener *createXYZ()*-Methode oder als einfaches Widgets innerhalb der *create()*-Methode des entsprechenden Containers.

Nun nochmal zurück zum einfachen Button. Eine Sache muss noch betrachtet werden: Events. Also bekommt der Button aus dem zweiten Beispiel noch ein Event:

```
1  public class MyDialog {
2
3      private Shell sShell = null;
4      private Button button = null;
5      /**
6       * This method initializes sShell
7       */
8      private void createSShell() {
9          GridLayout gridLayout = new GridLayout();
10         gridLayout.numColumns = 1;
11         sShell = new Shell();
12         sShell.setText("Shell");
13         sShell.setLayout(gridLayout);
14         sShell.setSize(new Point(300, 200));
15         button = new Button(sShell, SWT.NONE);
16         button.addSelectionListener(new org.eclipse.swt.
               events.SelectionAdapter() {
17             public void widgetSelected(org.eclipse.swt.
                   events.SelectionEvent e) {
18                 // TODO Auto-generated Event stub widgetSelected()
19                 System.out.println("widgetSelected()");
20             }
21         });
22     }
23
24 }
```

Events werden, wie weiter oben schon erwähnt, als anonyme Klassen generiert. Die Event-Methode enthält dann einen Kommentar und eine Konsolenausgabe: Der Entwickler muss hier später seine Implementierung vornehmen.

3.2 Das VE Projekt

Der Visual Editor ist als **eclipse**-Plugin realisiert und besteht selbst aus mehreren Plugins. Das Kern-Plugin heißt **org.eclipse.ve.java.core** und ist für diese Arbeit das Plugin, das die größte Beachtung findet. Hier wird das interne Modell definiert und der eigentliche Editor beschrieben. Es gibt neben diesem Plugin noch weitere für jedes unterstützte GUI-Framework. Hierbei ist für diese Arbeit nur das SWT Plugin relevant.

3.2.1 Basis Plugins

Das Core-Plugin baut auf verschiedenen anderen Plugins auf, die zunächst in aller Kürze beschrieben werden.

Java Development Tools (JDT)

Das **Java Development Tools** Plugin (JDT) ist das Kern-Plugin für den Bereich der Java Entwicklung in eclipse. Für den VE speziell relevant ist die Eigenschaft, Java-Code als abstraktes Modell (AST - Abstract Syntax Tree) zu organisieren. Das JDT umfasst ansonsten alles, was zur Entwicklung von Java mit eclipse notwendig ist: ein Editor, Projektverwaltung, Parser, Refactorings, usw.

Für diese Diplomarbeit spielt das JDT noch eine ganz spezielle Rolle: Für die Object Teams Integration in eclipse wurde eine eigene Variante des JDT entwickelt: Das OTDT (Object Teams Development Tools). Das OTDT umfasst alle Eigenschaften des JDT zuzüglich der Unterstützung für Object Teams. Da das OTDT das JDT ersetzt, ist der VE damit *im Prinzip* bereits OT-fähig.

Obwohl in der Dokumentation des VE davon gesprochen wird, dass der VE sprachunabhängig ist, wird sich nachher zeigen, dass große Teile des VE-Kerns eben gerade auf dem JDT Plugin basieren - mit all den Vor- aber auch Nachteilen. Ein Vorteil ist z.B., dass durch diese Abhängigkeit die ganze Bandbreite an Tools für die Verwendung von OT gleich mit bekannt ist. Nachteil ist jedoch, dass der VE eben gerade mit den zusätzlichen Eigenschaften des OTDT nur bedingt umgehen kann. Da dies einer der wesentlichen Punkte dieser Diplomarbeit ist, folgt später dazu noch mehr.

Eclipse Modeling Framework (EMF)

Das **Eclipse Modeling Framework** (EMF) ist ein Plugin, das sich im Wesentlichen um Code Generierung kümmert. Es gehört der eher seltenen Gruppe von Plugins an, die ausnehmend gut und umfangreich dokumentiert sind. Neben einigen online-Ressourcen gibt es sogar ein Buch, auf das ich hier explizit verweisen möchte: *eclipse Modeling Framework* [EMF06]

Aus einem vorgegebenen Modell generiert EMF zunächst einmal Java-Code für dieses Modell. Dieser Code umfasst eine Reihe von Interfaces mit entsprechenden Zugriffsmethoden auf die modellierten Eigenschaften sowie Implementierungsklassen. Aber EMF bietet noch mehr: Die generierten Klassen unterstützen automatisch Serialisierung. Objekte können so ohne zusätzlichen Aufwand gespeichert werden. Des Weiteren definiert EMF ein Adapter-Konzept. Adapter haben u.a. die Möglichkeit, Einfluss auf die adaptierten Klassen zu nehmen, dienen gleichzeitig als *Observer* und werden als solcher, bei Änderungen am adaptierten Objekt, benachrichtigt.

Java EMF Model (JEM)

Das **Java EMF Modell** (JEM) ist ein EMF Modell für die Sprache Java. Es entspricht in weiten Teilen dem AST (Abstract Syntax Tree) des JDT. Als EMF Modell umfasst es alle zusätzlichen Vorteile des EMF.

Graphical Editing Framework (GEF)

Das Graphical Editing Framework (GEF) stellt, wie der Name schon sagt, die Umgebung für graphische Editoren bereit. Es bietet viele allgemein gültige Operationen bereits an, auch eine MVC-Architektur. GEF implementiert bereits einen graphischen Editor samt Palette. Der Entwickler braucht sich daher nur noch um seine Modell-Elemente und deren Präsentation kümmern.

3.2.2 Das VE Modell

Der VE verwendet für seine Arbeit nicht nur ein, sondern gleich mehrere Modelle. Das VE-Modell ist ein erweitertes JEM Modell und bildet die zentrale Basis für den VE. Sämtliche Komponenten, wie der Editor und die Steuerung für die externe Virtual Machine, bauen auf diesem Modell auf. Die Abbildung 3.1 verdeutlicht dies - es ist ein Ausschnitt aus einem Diagramm aus [ULC05].

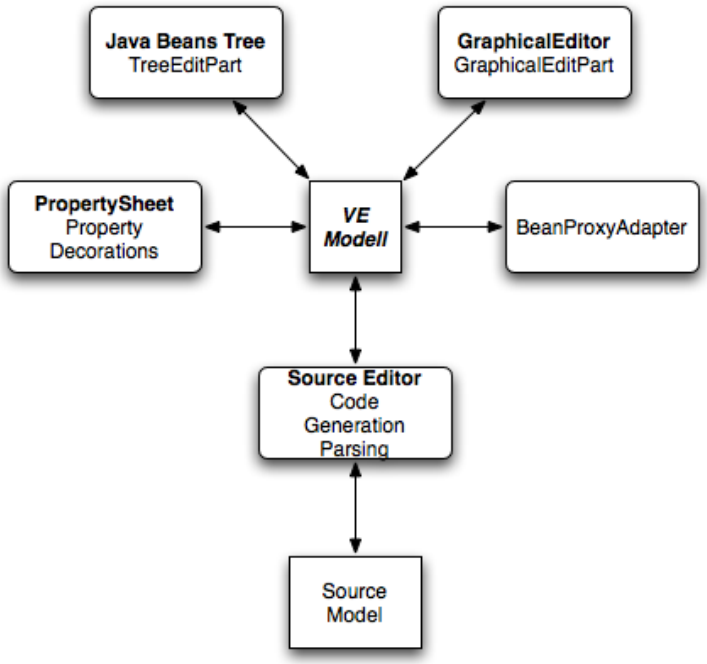

Abbildung 3.1: VE Modell: Basis für VE Komponenten (aus [ULC05])

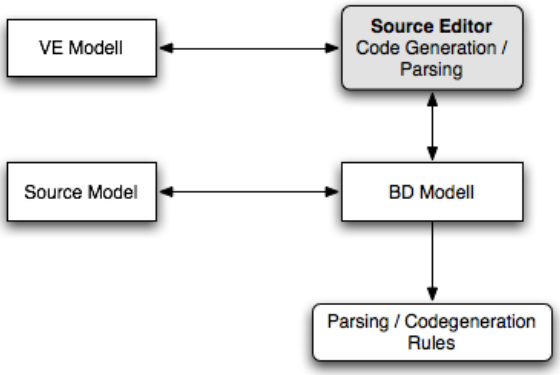

Abbildung 3.2: VE: Die Modelle (aus [ULC05])

Das VE-Modell ist unabhängig vom Code. Für die Abbildung auf den Code kennt der VE dazu noch das Bean-Declaration-Modell, das quasi als Brücke zwischen dem Code und dem VE-Modell dient. Dieses Modell ist jedoch kein EMF-Modell. Die Zusammenhänge werden in Abbildung 3.2 dargestellt.

Obwohl der VE im Prinzip durchaus ein offenes Modell hat, so sind diese Modelle, auch das Java-abhängige, in einem Plugin zusammengefasst. Das Plugin bietet auch keine Möglichkeit, ein eigenes Modell (einfach) zu integrieren. Die bestehenden Plugin-Extensionpoints dienen lediglich der Erweiterung der Widgetpaletten und der Integration anderer Frameworks. Auf diese Problematik wird im Kapitel 5 noch genauer eingegangen.

3.3 Zusammenfassung

Der VE generiert ein Minimum an Code für die Darstellung von Dialogen und bietet einen guten Ansatz, ein MVC Modell darauf aufbauend zu generieren. Die interne Struktur ermöglicht es theoretisch, auch andere Sprachen zu generieren und die Generierung zu beeinflussen. Da die angedachten Erweiterungen des VE sich ausschließlich auf das Anbinden anderer Frameworks und Widgets konzentrieren, wurde die Unabhängigkeit jedoch nicht so realisiert, wie die Modelle es darstellen. Selbst kleine Änderungen in der Art der Generierung, wie z.B. die View-Klasse als eine nested-class zu generieren, scheitert an der inkonsequenten Trennung der Modell-Komponenten. Es wird also bei der Realisierung ein besonderes Augenmerk auf diese Problematik zu legen sein.

Kapitel 4

Model-View-Controller für ObjectTeams

Nachdem die Grundlagen klar sind, wird es nun Zeit, ein Modell (Ziel-Modell) zu entwickeln, das schließlich vom VE für OT/J generiert werden soll. Dies erfolgt in zwei Schritten: Zunächst wird das Modell unter Nicht-Beachtung des VE erstellt und in einem zweiten Schritt unter Berücksichtigung der Rahmenbedingungen komplettiert.

Für die Betrachtung ist es hilfreich, ein paar Kategorien von möglichen Dialogen zu definieren. Dialoge lassen sich in vielerlei Hinsicht kategorisieren. Hier wird der Schwerpunkt auf die Komplexität des Verhaltens gelegt und somit im Wesentlichen auf die Komplexität des Controllers.

1. Die Information: Ein Dialog, der unabhängig von Daten existiert und keine nennenswerten Benutzerinteraktionen erwartet, z.B. eine Art Message Box, ein Informationsfenster mit statischem Text.

2. Der Monitor: Ein Dialog, der ausschließlich Daten visualisiert. Evtl. visualisiert er auch Änderungen in den Daten. Eine Benutzerinteraktion ist nicht vorgesehen.

3. Der Eingabedialog: Der Dialog besteht weitestgehend aus Eingabefeldern. Es gibt zwischen den Feldern keine Abhängigkeiten.

4. Der komplexe Dialog: Der Dialog enthält verschiedene Eingabefelder, die zum Teil voneinander abhängig sind, z.B. eine Checkbox, die gewisse Dialogteile sperrt oder freigibt.

Die erste Kategorie ist eine, die hier keine weitere Rolle spielt: Der Aufwand eines MVC-Modells für diese Kategorie würde keinerlei Nutzen bringen, da hier weder ein Data-Modell vonnöten ist noch ein Controller, der die View steuern soll. Die anderen sind insbesondere aufgrund des Umfangs des Controllers in Hinsicht auf ein allgemeines MVC-Modell interessant und sollten bei der Erörterung des Modells die Grundlage bilden.

4.1 Das VE-unabhängige Modell

Das allgemeine MVC Modell, bestehend aus den Komponenten Model, View und Controller, beschreibt, dass die Komponenten miteinander jeweils paarweise in einer bidirektionalen Beziehung stehen können. Es stellt sich für das OT/J -MVC-Modell nun die Frage, ob alle diese Beziehungen benötigt werden und wie die benötigten Beziehungen aussehen. OT/J bietet durch die *playedBy*-Beziehung eine neue und wichtige Beziehung, die jedoch in einer Hinsicht beschränkt ist: eine Rolle kann nur eine solche Beziehung haben, allerdings können viele Rollen eine Basis adaptieren. Daher ist es nötig, die einzelnen Komponenten und deren Beziehungen, die das allgemeine MVC-Modell vorschlägt, genauer zu untersuchen.

Zunächst aber wird ein Katalog von Fragen aufgestellt, die für die Realisierung wichtig sind. Diese werden danach Grundlage für die Diskussion der Komponenten und deren Beziehungen sein:

1) Wie erfolgt der Transfer der Daten vom Modell zur View?

2) Wer bereitet die Daten zur Präsentation vor?

3) Wie erfolgt der Transfer der Daten von der View in das Modell?

4) Wie werden Eingaben auf Korrektheit überprüft?

5) Entstehen Datenfluss-Zyklen? Wie werden diese verhindert?

4.1.1 Die Modell-Komponente

Das Grundziel von aspektorientierten Sprachen ist die Vermeidung von *crosscutting* Code (vgl. Abschnitt 1.3.1). Im Sinne vom MVC heißt dies, dass das ursprüngliche Modell, dass die Daten für den Dialog liefert (in der Folge Core-Modell genannt) von eben diesem *crosscutting* frei gehalten werden soll

- in diesem Fall eben Code für die Meldung von Änderungen im Core-Modell. Daraus folgt zunächst, dass das Core-Modell als Basis für OT/J -Rollen dienen wird. Nun entsteht aber eine neue Frage:

6) Welche Komponente bzw. Komponenten haben Rollen für das Core-Modell?

Aus dem allgemeinen MVC-Modell würde sich ergeben, dass sowohl der Controller als auch die View derartige Rollen enthalten. Geht man von einem komplexen Core-Modell aus, das insbesondere die Daten, die die View benötigt, nicht direkt liefert, so müssen die Daten über ein Rollen-Modell aus dem Core-Modell zunächst aufbereitet werden. Geht man nun davon aus, dass die Daten sowohl vom Controller als auch von der View beobachtet werden, würde dies bedeuten, dass beide Komponenten ein Rollen-Modell implementieren müssten. Diese Modelle wären in vielen Dingen ähnlich und in der Folge gäbe es Redundanzen. Das erscheint wenig sinnvoll.

Ein besserer Ansatz ist, eine Komponente zu bauen, die das Core-Model derart abstrahiert, dass es den Anforderungen der GUI entspricht, also sowohl denen des Controllers als auch der View. Diese Komponente wird im Folgenden **Modell-Interface** genannt und ist die Schnittstelle des Zielmodells zum Core-Modell. Der Zugriff auf das Core-Modell erfolgt ausschließlich über diese. Die Frage 6 ist somit schon beantwortet. Aber es entsteht eine neue Frage, die später noch beantwortet werden muss:

7) Welche Komponente bzw. Komponenten hat bzw. haben Rollen für das Modell-Interface?

Die Aufgaben des Modell-Interface liegen nun in der Datenaufbereitung für die View und den Controller. Ferner bietet es Möglichkeiten, Veränderungen an den Daten beobachten zu können. Die Datenaufbereitung ist zwangsläufig abhängig vom Core-Modell und kann auch erhebliche Umstrukturierungen zur Folge haben, damit die View und der Controller mit den Daten besser umgehen können.

Da das Modell-Interface nun die Aufgabe hat, Daten aus dem Core-Modell view- und controllergerecht aufzubereiten, liegt es nahe, auch gleich die Frage 2 zu beantworten: Die Aufbereitung der Daten für die Präsentation kann nun direkt durch die Modell-Abstraktion abgedeckt werden.

Die Frage nach der Verifikation (4) ist hingegen noch nicht so eindeutig, denn hier spielt der Controller noch eine wesentliche Rolle. Die Diskussion wird also dorthin verlegt.

4.1.2 Die Model-View Beziehung(en)

Hier ist zunächst die Frage (1) zu klären: Wie kommen die Daten in die View oder präziser: Wie gelangen die Werte in die Widgets? Im allgemeinen MVC-Modell gibt es zwei Wege: Direkt aus dem Modell in die View oder indirekt über den Controller. Der letztere Weg wäre wichtig, wenn der Controller die Daten zunächst noch aufbereiten würde. Das ist jedoch nicht (mehr) nötig, da dies durch die Daten-Abstraktion schon abgedeckt ist. Also bleibt der direkte Weg. Bleibt die Frage, ob nun die View sich die Daten holt oder das Daten-Modell diese in die View schreibt. Es gibt eine ganze Reihe von Gründen, die für die erste Variante sprechen:

- Das Daten-Modell hat die primäre Aufgabe, Daten aufzubereiten. Da es dies für den Controller und die View durchführt, liegt es nahe, diese Komponente bzgl. der View und des Controllers passiv zu halten.
- Das Daten-Modell hat mindestens für den Controller eine Schnittstelle für die Daten. Diese könnte genauso gut von der View verwendet werden.
- Die View hat die Hoheit über ihre Widgets. Das Daten-Modell sollte damit also frei von Informationen über Widgets und deren Verhalten bleiben.
- Würde das Daten-Modell aktiv die Daten in die View schreiben, müsste die View zusätzlich ein Interface bieten, das es dem Daten-Modell erlaubt, die Werte der Widgets zu setzen. Alternativ würde das Daten-Modell eine Rolle für die View haben und damit die Widget-Hoheit der View wiederum in Frage stellen.

Zusammengefasst ist festzustellen: Die View ist eine Rolle des Daten-Modells und kann über dessen Interface die benötigten Informationen abfragen.

4.1.3 Die Controller-Model Beziehung(en)

Aus Sicht eines Code-Generators, der ein Framework aus den Informationen eines Dialogs generieren soll, ist diese Beziehung eine, über die noch nichts bekannt ist. Dennoch müssen alle Möglichkeiten offen bleiben. Ein sehr wahrscheinlicher Fall ist, dass der Controller über Änderungen im Modell benachrichtigt werden will. In diesem Fall ist die Beziehung hier die gleiche wie

zwischen der View und dem Data-Modell. Ob weitere Abhängigkeiten bestehen, bleibt dann der konkreten Implementierung überlassen.

4.1.4 Die View-Controller Beziehung(en)

Der Controller hat die Aufgabe, die View zu steuern. Das macht er auf zweierlei Arten: Er reagiert zum einen auf Events, die die View von ihren Widgets erhält und zum anderen auf Änderungen im Modell. Die Reaktionen auf Events kann z.B. bedeuten, Daten zu aktualisieren, aber auch andere Widgets zu aktivieren oder zu deaktivieren. Bei Änderungen von Daten muss er ebenfalls überprüfen, ob andere Widgets davon betroffen sind - die Aktualisierung mit Daten verbundener Widgets führt bereits die View durch.

Die Änderungen von Daten in den Widgets hat unter Umständen zur Folge, dass der Controller diese auf Korrektheit hin überprüfen muss und ggf. Falscheingaben entsprechend visualisiert. Die Prüfung, ob Eingaben korrekt sind, ist eine Frage der Modell-Konsistenz und gehört somit in die Modell-Abstraktion.

Auf der anderen Seite muss die View Events an den Controller weiterleiten. Die konkrete Realisierung ist stark davon abhängig, was mit dem VE und dem SWT-Framework möglich ist. Aus Sicht des Code-Generators, der ausschließlich die Informationen des GUI-Designers hat, ist hier zunächst nur der Weg von der View zum Controller, zum Zweck der Delegation der Events, notwendig. Da die View, wie schon beschrieben, eine Rolle des Daten-Modells ist, bleiben nur zwei Varianten übrig: die Delegation über eine Instanzbeziehung oder der Controller wird eine Rolle der View.

Der zweite Ansatz ist eher der schlechtere: Da der Controller höchstwahrscheinlich eine Rolle des Daten-Modells sein wird, hätte es zur Folge, dass der Controller in mehrere Rollen aufgeteilt werden müsste. Ein zweiter Punkt ist die Art und Weise, wie SWT Events realisiert und (vorweggenommen) wie der VE heute Events generiert: Dies wird über eine anonyme Klasse realisiert (vgl. Abschnitt 3.1), was bedeutet, dass im Grunde keine Methode in der View für ein Event notwendig ist. Wäre der Controller eine Rolle der View, so müsste die View entsprechende Methoden anbieten.

Fügt man diese Argumente zusammen, so folgt, dass zwingend eine Beziehung von der View zum Controller notwendig ist, und zwar in Form einer Instanzbeziehung, und dass der Controller bei Bedarf auch die View kennen kann.

4.1.5 Zyklen

Aus den bisher besprochenen Eigenschaften des Modells entsteht das in Abbildung 4.1 dargestellte Modell. Die *playedBy*-Beziehungen sind, wenn man den Kontrollfluss betrachtet, bidirektional: über *callins* verläuft der Kontrollfluss nämlich entgegen der Beziehungsrichtung. Da das Modell alle Kategorien von Dialogen beschreibt, ist daher davon auszugehen, dass der Controller alle gebotenen Möglichkeiten nutzt. Es ist also unter anderem auch folgendes Szenario denkbar:

Im Core-Modell erfolgt eine für den Dialog relevante Änderung. Über *callins* wird der DataAdapter entsprechend aktiv. Wiederum über ein *callin* erfährt der Dialog von der Änderung und aktualisiert das passende Widget (ggf. auch mehrere). Die Änderung hat zur Folge, dass das Widget ein Event wirft, welches diese Änderung wieder meldet. Die View fängt dieses Event und delegiert die Behandlung an den Controller. Dieser wiederum wertet das Event aus und wird nun über den DataAdapter den (für ihn) neuen Wert eintragen. Der DataAdapter wird diesen Wert nun in das Core-Modell schreiben, das sich damit wieder ändert - der Prozess fängt wieder vorne an.

Es ist also davon auszugehen, dass Zyklen dieser Art höchstwahrscheinlich entstehen. Um einen solchen Zyklus zu verhindern, muss eine Komponente in der Lage sein, diesen zu bemerken und abzubrechen. Im Prinzip kann dies jede einzelne Komponente:

Die View kann bei einer Benachrichtigung und Änderung des Widgets das folgende Event ignorieren und nicht an den Controller delegieren. Da der Controller aber evtl. auf solche Änderungen Aktionen durchführen muss, müsste er selbst über ein *callin* vom DataAdapter über die Änderung informiert werden.

Der Controller könnte ebenfalls über ein *callin* über die Änderung im DataAdapter informiert werden und in der Folge das von der View gelieferte Event ignorieren.

Abhängig davon, wie die *callins* für die Datenänderung beschaffen sind, könnte der DataAdapter beim Setzen desselben, schon bekannten, Wertes die *callin* -gebundene Methode nicht aufrufen.

Unter Umständen ist ein Mechanismus, ähnlich dem Lösungsansatz im DataAdapter, bereits im Core-Modell implementiert, so dass der Zyklus bereits hier unterbrochen wird.

4.1.6 Zusammenfassung

Das zunächst vom VE unabhängige Modell ist eine Konkretisierung des allgemeinen MVC-Modells. Dieses Modell ist in der Abbildung 4.1 dargestellt.

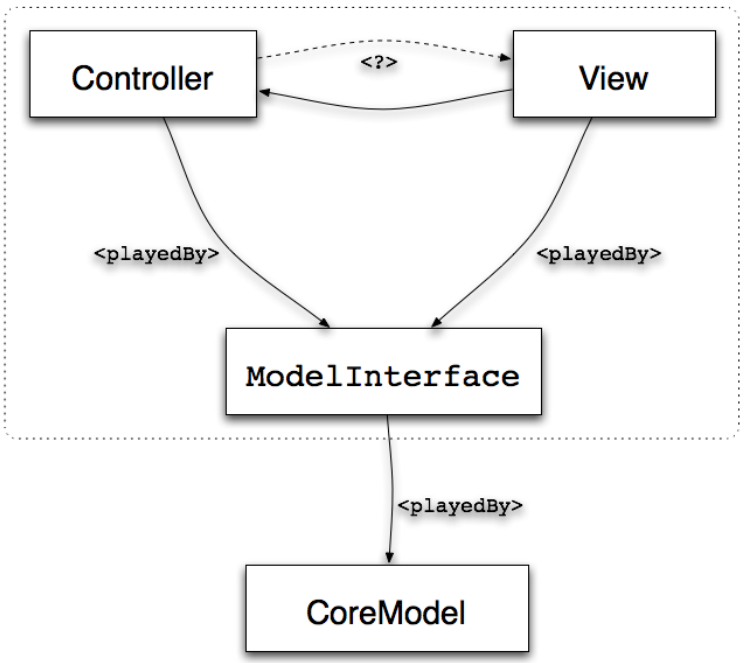

Abbildung 4.1: VE-unabhängiges Modell

Abschließend sind nun die eingangs aufgestellten Fragen zu beantworten:

1) *Wie erfolgt der Transfer der Daten vom Modell zur View?*
   Die View holt sich ihre Daten selbst.

2) *Wer bereitet die Daten zur Präsentation vor?*
   Im Zuge der Aufbereitung des Core-Modell durch das Modell-Interface werden hier auch die Daten für die Präsentation vorbereitet.

3) *Wie erfolgt der Transfer der Daten von der View in das Modell?*
   Die Änderungen von Daten in der View werden von SWT in Form von Events gemeldet. Da der Controller für die Bearbeitung der Events zuständig ist, fällt der Transfer der Änderungen in das Modell in den Aufgabenbereich des Controllers.

4) *Wie werden Eingaben auf Korrektheit überprüft?*
Auch die Kontrolle der Korrektheit von Daten erfolgt durch die Bearbeitung der Events und läuft daher über den Controller. Da allerdings das Modell-Interface schon für die Datenaufbereitung zuständig ist, ist dies auch der Ort, der die Korrektheit von Daten überprüfen kann. Infolgedessen wird die eigentliche Prüfung in der Modell-Abstraktion durchgeführt. Der Controller ist für die Delegation der Prüfung an das Modell sowie für eine eventuelle Visualisierung des Ergebnisses zuständig.

5) *Entstehen Datenfluss-Zyklen? Wie werden diese verhindert?*
Datenfluss-Zyklen werden höchstwahrscheinlich auftreten. Das Verhindern dieser Zyklen ist jedoch stark von der konkreten Realisierung des Controllers und des Data-Model abhängig und muss entsprechend bei der Realisierung geklärt werden.

6) *Welche Komponente hat Rollen für das Core-Modell?*
Der Zugriff auf das Core-Modell erfolgt ausschließlich über das Modell-Interface.

7) *Welche Komponente bzw. Komponenten hat bzw. haben Rollen für das Modell-Interface?* Sowohl der Controller als auch die View werden Rollen vom Modell-Interface.

4.2 Das Ziel-Modell

Das konkrete Modell für die Verwendung mit dem VE unterliegt einigen Anforderungen, die sich aus der Verwendung des VE im Zusammenhang mit OT/J ergeben. Es gibt zunächst drei zentrale Vorgaben:

1. Die Code-Generierung des VE ist nur mit relativ aufwändigen Mitteln anzupassen. Es ist daher von Vorteil, die Generierung soweit wie möglich von der vom VE generierten Klasse zu trennen.

2. Ein prinzipielles Problem mit generiertem Code ist, dass er mit Code interagieren muss, der vom Anwender geschrieben wird. Hierbei muss geklärt sein, inwiefern der Anwender generierten Code ändern darf bzw. wo codiert werden muss.

3. Neben dem Anwender, der den VE verwendet und die Oberfläche entwickelt, sollte man den Anwender, der den Dialog im Programm starten will (oder muss), nicht außer acht lassen. Es sollte ihm so einfach wie möglich gemacht werden. Idealerweise in etwa so:
   `new MyDialog(rootData).show()`

4.2.1 Teams

Ein erster Schritt, das Modell aus Abbildung 4.1 zu konkretisieren, ist, Teams zu identifizieren. Da das MVC Modell aus diversen Klassen besteht, ist es, aufgrund der Anforderung 3, sinnvoll, das komplette Ziel-Modell in ein Team zu kapseln - dadurch wird das komplexe Modell gekapselt und der Anwender des Dialogs kann es verwenden, als würde es sich um eine einzelne Klasse handeln.

Die einzelnen Komponenten des MVC-Modell eignen sich ebenfalls als Teams: Die Modell-Komponente wird eine Abstraktion des Core-Modells beinhalten. Abhängig vom Core-Modell kann es nötig sein, das Modell-Interface durch eine ganze Reihe von Rollen zu realisieren. Für die Verwendung durch die View und den Controller wird zudem ein (zentrales) Interface benötigt, das einen zentralen Zugriff auf die Daten erlaubt. Das Modell-Interface wird also unter Umständen in mehrere Teile aufgeteilt: Ein Interface für die View und den Controller, sowie ein Rollen-Modell um auf das Core-Modell zuzugreifen und dieses ggf. auch umzustrukturieren. Wie bereits im VE-unabhängigen Modell beschrieben, verwenden die View und der Controller Rollen, um auf die Daten zuzugreifen. Infolgedessen muss die Modell-Komponente in einem eigenen Team liegen, da OT/J verlangt, dass die Basis einer Rolle nicht im gleichen Team sein darf wie die Rolle.

Der Umfang des Controllers ist von vornherein nicht einzuschätzen. In der einfachsten Form muss er sich lediglich um das Starten und das Beenden der View kümmern. In der anderen Richtung kann er jedoch beliebig komplex werden und auch über ein eigenes Controller-Modell verfügen. Die Entscheidung, ob für den Controller ein eigenes Team vonnöten ist, kann also erst in der konkreten Situation entschieden werden. Aus der Sicht des VE - also was generiert werden kann - ist der Controller ebenfalls weitestgehend unbekannt. Der VE kann aus den Informationen, die die konkrete View hergibt, ausschließlich die Events aufzählen, die der Controller behandeln muss. Folglich kann aus dem GUI-Designer lediglich ein Interface mit Methoden für das Empfangen von Events erzeugt werden. Es muss lediglich sichergestellt sein, dass der Entwickler daraus später einen vollwertigen Controller

implementieren kann, der sowohl Zugriff auf die View als auch auf die Daten hat.

Die View selbst besteht im Grunde aus einer einzigen Klasse. Diese Klasse enthält im ursprünglichen VE lediglich Attribute für alle Widgets sowie eine oder mehrere Methoden, die die View erstellt - also Widgets erzeugt und anordnet sowie Event-Handler (derzeit noch anonym) anmeldet. Die View dennoch in ein eigenes Team zu kapseln ist sinnvoll, um es der Implementierung des Controllers zu ermöglichen, die View als Basis zu nutzen - wie schon erwähnt, erlaubt OT/J es nicht, dass eine Rolle (hier: Controller) und die Basis (hier: View) in dem gleichen Team liegen. Für die Delegation der Events an den Controller selbst ist keine *playedBy*-Beziehung nötig: Das Team kennt den Controller, so dass die View diesen verwenden kann, um die Events über anonyme Event-Handler dorthin weiterzuleiten.

Aus diesen ersten Gedanken entsteht nun das in Abbildung 4.2 dargestellte Modell.

4.2.2 VisualEditor und Code-Trennung

Dieses Modell muss nun noch an den VE angepasst werden. Aus den Informationen des VE kann ein Großteil des Modells generiert werden. Eines der Hauptprobleme mit generiertem Code, der durch manuell geschriebenen Code erweitert werden muss, ist, dass diese Mischung es dem Generator unter Umständen unmöglich macht, noch zu erkennen, welcher Code nun generiert wurde und evtl. ersetzt werden kann und welcher benutzerspezifische Logik enthält und eben nicht ersetzt werden soll. Es gibt ein paar Lösungen dazu, z.B. wird in NetBeans der generierte Code derart markiert, dass der Editor eine Veränderung durch den Programmierer verhindert. Andere generieren zusätzliche Kommentare, die darauf hinweisen, dass der Code ggf. durch den Generator wieder überschrieben wird.

Ich werde versuchen, dieses Problem durch Trennung in verschiedene Klassen zu lösen: Generierte Klassen sollen durch den Programmierer nicht angefasst, sondern durch Vererbung mit der individuellen Logik erweitert werden. Dadurch wird das Modell zwar zunächst noch umfangreicher, aber die Punkte, die durch den Entwickler verändert werden sollen, sind nun klar abgegrenzt vom generierten Code.

Eine weitere Trennung erfolgt durch die Ausnutzung von Role-Files. OT/J erlaubt, dass einzelne Rollen in eine eigene Datei ausgelagert werden können. Das ist hier sehr hilfreich: alle Klassen, die der Entwickler später ändern soll

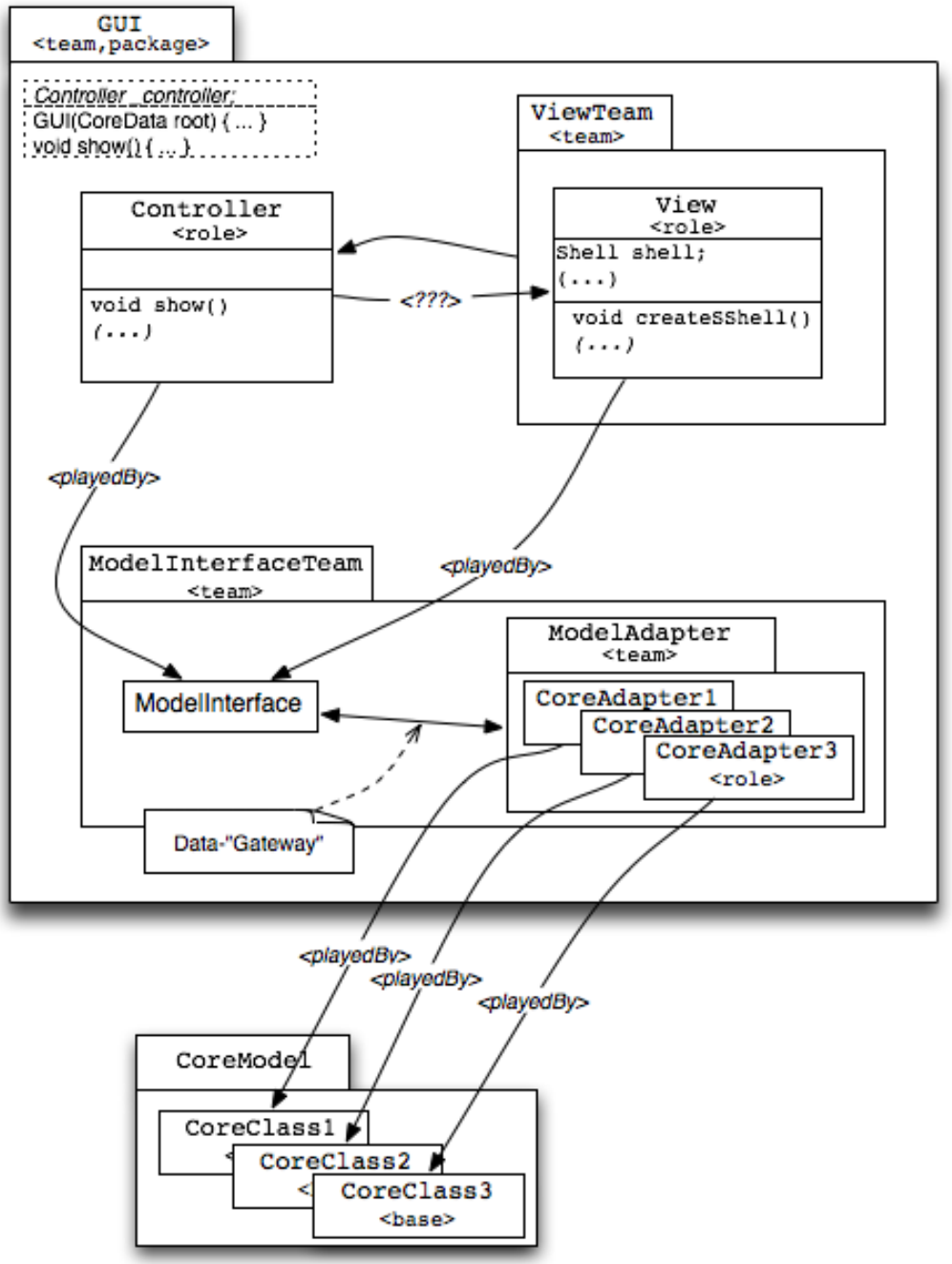

Abbildung 4.2: MVC für OT/J - Teams

und kann, werden so in eigene Dateien ausgelagert. Der generierte Code ist hiermit sauber von zu änderndem Code getrennt. In Abbildung 4.3 sind die Klassen, die separat generiert werden, grau hinterlegt (Klassen/Teams, die optional vom Entwickler implementiert werden können und nicht automatisch angepasst werden, wurden grau dargestellt).

4.3 Datenfluss

Nun ist noch zu klären, wie der Datenfluss vom Modell-Interface in die View erfolgt. Zunächst werden einfache Widgets betrachtet, die einen einzelnen Wert darstellen - also Checkboxen, Radiobuttons, Textfelder usw.

4.3.1 Einfache Daten

Für diese Felder werden zwei Dinge im Modell-Interface benötigt: Eine Methode zum Auslesen des aktuellen Wertes und eine, die eine Änderung des Wertes signalisiert. Die Methode zum Auslesen des Wertes ist dafür zuständig, den Wert des Feldes zu ermitteln und diesen zurückzuliefern. Wo dieser Wert herkommt und wie er sich zusammensetzt, muss der GUI-Entwickler implementieren und wird daher im abstrakten Modell-Interface entsprechend als abstrakte Methode generiert. Die Meldung einer Änderung wird hingegen als leere Methode generiert, die von der GUI-Logik im Falle einer Änderung des Werts aufgerufen wird. Die View wird sich in Form eines *callin* an diese Methode binden und das entsprechende Widget aktualisieren. Nur der Vollständigkeit halber: Werden Daten in der View verändert, so ist der Controller über seine Eventmethoden dafür verantwortlich, dass diese Daten in das Modell geschrieben werden. Für einfache Daten ist somit der Datenfluss komplett geklärt.

Komplizierter wird der Datenfluss für Widgets, die Mengen von Daten, evtl. sogar strukturiert, darstellen. Hierzu zählen Listboxen, Comboboxen, Tabellen und Bäume.

4.3.2 Einfache Listen

Listboxen und Comboboxen verwalten im Kern eine Liste von Strings. Die Widgets können, wie eine gewöhnliche Liste auch, einzelne Elemente hinzufügen und entfernen. Entsprechend werden Methoden benötigt, die Änderungen an den Daten melden. Für das initiale Füllen der Box wird zudem

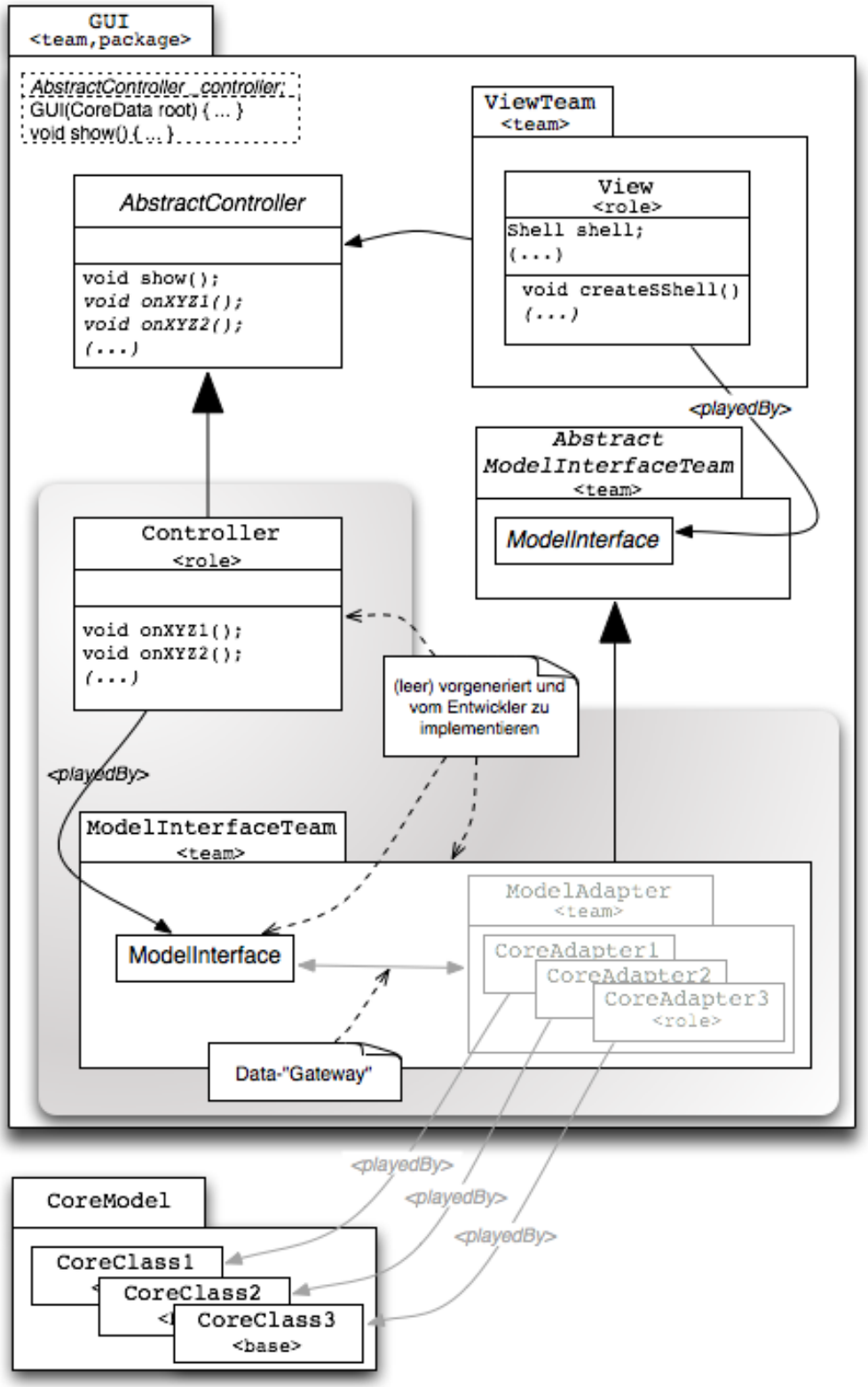

Abbildung 4.3: MVC für OT/J und VE

noch eine Methode benötigt, die eine komplette Liste der Strings liefert. Wie auch bei den einfachen Widgets werden für die Meldung von Änderungen leere Methoden in das abstrakte Modell-Interface generiert. Für eine Liste namens *myList* werden folgende Methoden generiert:

```
1 abstract String[] getMyListData();
2 void notifyMyListDataAdded(String value, int index);
3 void notifyMyListDataRemoved(int index);
4 void notifyMyListDataRenamed(String oldValue, String
      newValue, int index);
```

4.3.3 Strukturierte Daten

Tabellen und Bäume fallen unter die Kategorie *strukturierte* Daten, da diese Widgets für ihre einzelnen Elemente eine eigene Klasse einführen. Im Falle einer Tabelle ist es das *TableItem* und für Bäume das *TreeItem*. Für beide Widgets gibt es in der JFace-Erweiterung entsprechende Viewer-Klassen, die die Handhabung der Widgets mehr im Sinne von MVC handhaben. Da der VE und auch der hier zu entwickelnde GUI-Designer nur auf SWT basieren, kommen diese Ansätze hier nicht in Frage.

Die zentrale Frage ist: Wer erzeugt diese Item-Objekte? Durch die Datennähe dieser Objekte läge es nahe, diese auch im Modell-Interface zu erzeugen. Andererseits sind diese Objekte stark Widget bezogen und gehören demnach eher zur View. Events, die die Widgets melden, liefern ebenfalls das betreffende Item-Objekt mit und müssen entsprechend im Controller verarbeitet werden. Da der Controller, je nach Anwendung, mehr oder weniger direkt mit den Widgets arbeiten muss, kann dieser auch ohne weitere Bedenken mit diesen Item-Objekten umgehen. Daraus folgt im Grunde schon, dass die Erzeugung der Items in der Verantwortlichkeit der View liegt.

Die Datenbeschaffung und die Meldung über Änderungen könnten im Grunde ähnlich denen der einfacheren Widgets aussehen. Aber es gibt noch ein weiteres Problem: Wie werden die Daten bei Aktionen auf der Oberfläche identifiziert? Woher weiß das System, welche Daten dazugehören, wenn ein Item selektiert wurde? In einer Tabelle kann es z.B. vorkommen, dass mehrere Einträge exakt gleich aussehen. Wenn man nun bedenkt, dass die einzelnen Einträge über eine Liste von Objekten im Datenmodell erstellt und zudem in der Oberfläche sowieso für jeden Eintrag eigene Objekte erstellt werden müssen, kann man also diese beiden Dinge einfach verknüpfen. SWT

bietet hier die Möglichkeit, Widgets und auch Items mit zusätzlichen (anonymen) Daten zu versehen: entweder über *set/getDataObject()* für ein einzelnes Objekt oder in Form von Key/Value Paaren. Dieser Mechanismus ist zwangsläufig anonym durch die Verwendung von *Object*. Könnte man ein Table- oder TreeItem spezialisieren, wäre dies eine gute Alternative. Leider ist dies in SWT nicht erlaubt.

Um das Problem der Zuordnung zu lösen, wird im abstrakten Modell-Interface eine Klasse eingeführt, die die Daten für ein Item und Notifizierungs-Methoden für die Meldung von Änderungen bereitstellt. Diese Klasse muss dann später im Modell-Interface für die konkreten Einträge implementiert werden und wird im Item als Data-Object unter einem festgelegten Key registriert.

Tabelle

Die Realisierung wird nun am Beispiel einer Tabelle dargestellt. Die beschriebene abstrakte Daten-Klasse sieht wie folgt aus (AbstractTableDataEntry):

```
1  public abstract team class AbstractModelInterfaceTeam {
2      public abstract class AbstractTableDataEntry {
3
4          public abstract String[] getColumns();
5
6          public boolean isGrayed() {
7              return false;
8          }
9          public boolean isChecked() {
10             return false;
11         }
12
13         protected void notifyDataChanged() {}
14         protected void notifyRemoved() {}
15     }
16 }
```

Neben dieser Datenklasse wird noch eine Klasse benötigt, die nun die Verbindung mit dem TableItem herstellt: der TableConnector[1]. Theoretisch

[1]Die OT/J Entwickler haben einen Katalog von Design-Patterns erstellt, der unter anderem ein Pattern namens **Connector** beschreibt. Dieses Pattern ist hier jedoch nicht gemeint. Der Begriff Connector passt dennoch sehr gut zu seiner Aufgabe, so dass ich ihn bewusst so belassen habe.

gäbe es für den Connector die Möglichkeit, eine Besonderheit von OT/J auszunutzen, die eine Art Mehrfachvererbung zulässt. In diesem Fall würde der Connector von TreeItem erben und gleichzeitig eine Rolle der Datenklasse sein. Der Connector würde nun über Methodenbindungen die Verbindung von den Daten zu den entsprechenden Methoden des TableItems herstellen. Leider erlaubt SWT nicht, die Item-Klassen zu spezialisieren[2].

Also muss ein geringfügig anderer Weg gewählt werden: Der Connector wird zu einer Rolle der Datenklasse, erzeugt ein passendes TableItem und aktualisiert dieses bei Änderungen in den Daten. Damit nun nicht alle TableItems automatisch zu einer Rolle werden, sondern nur diejenigen, die zu einem Widget des Dialogs gehören, werden ein Registrierungs-Mechanismus und ein passender *guard* verwendet. Die Erzeugung des Items erfordert die passende Table als Parameter im Konstruktor. Daher erhält der Connector eine Methode *setTable(Table table)*, in der das Item erstellt und mit den Daten verknüpft wird.

Disposable Items

Um einen Eintrag aus einer Table zu löschen, reicht es aus, dem Item ein *dispose()* zu schicken. Werden Einträge im Datenmodell gelöscht, so wird dem Connector dies über die Methode *notifyRemoved()* mitgeteilt. Der Connector kann daraufhin dem entsprechenden Item *dispose()* schicken und sich selbst als Rolle abmelden. Da das *dispose()* jedoch theoretisch von außerhalb kommen kann, ist es nötig, dieses separat zu beobachten. Zu diesem Zweck wird eine weitere Rolle eingefügt, die als Basis ein *Item* hat und ein after *callin* für *dispose()* implementiert. Da das Item als Superklasse für Table- und TreeItem fungiert, ist hier also eine gemeinsame Behandlung möglich. Dieser DisposeWatcher wird dann für jeden Connector erzeugt und meldet diesem, wenn ein *dispose()* aufgerufen wurde.

Die Struktur, die sich daraus ergibt, ist in Abbildung 4.4 dargestellt. Der Code für den TableConnector sieht nun folgendermaßen aus:

```
1 public team class ViewTeam {

3     ...
4     // key für die Item-Data Verknüpfung
```

[2]In der Tat sind die Item-Klassen nicht als *final* markiert, sondern die Klassen selbst werfen im Konstruktor eine Exception, wenn die tatsächliche Klasse nicht aus dem package `org.eclipse.swt.widgets` stammt! Genaugenommen betrifft dieses Verhalten sogar *alle* Klassen, die von *Widget* erben: vgl. `Widget.checkSubclass()`.

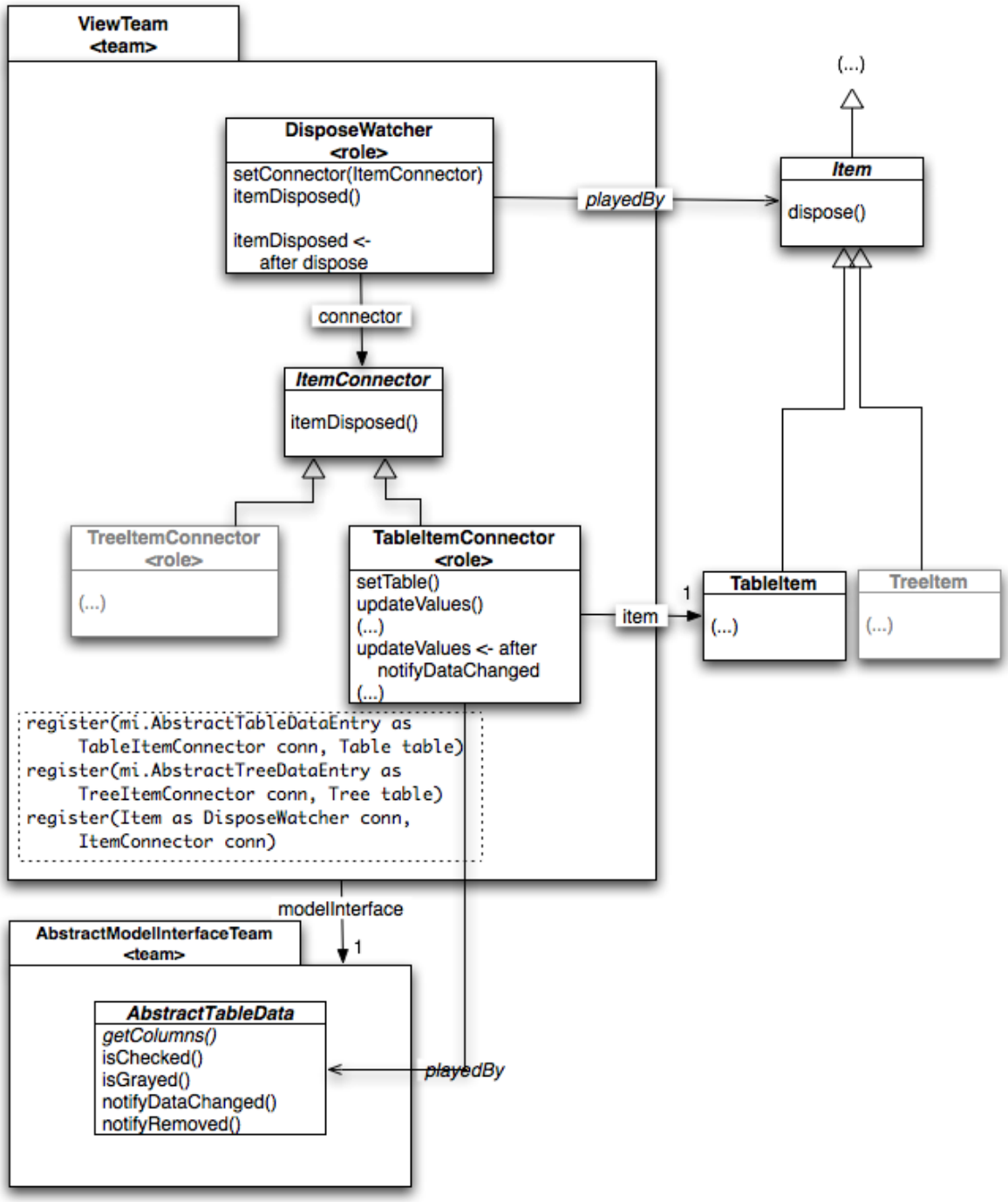

Abbildung 4.4: Strukturierte Daten

```
5       public final String DATA_KEY = "__OT/
            J_MVC_FRAMEWORK_TABLEDATAENTRY__";

7       // Diese Klasse wird von den Connectoren geerbt;
8       protected abstract class ItemConnector {
9           public void itemDisposed() {
10              ViewTeam.this.unregisterRole(this);
11          }
12      }

14      // Diese Rolle beobachtet einen dispose()-Aufruf auf Items
15      protected class DisposeWatcher
16          playedBy Item
17          base when (hasRole(base, DisposeWatcher.class)) {

19          private ItemConnector connector;

21          public void setConnector(ItemConnector connector) {
22              this.connector = connector;
23          }

25          void itemDisposed() {
26              if(connector != null)
27                  connector.itemDisposed();

29              ViewTeam.this.unregisterRole(this);
30          }

32          itemDisposed <- after dispose;
33      }

35      public void register(Item as DisposeWatcher watcher,
36                           ItemConnector connector)  {
37          watcher.setConnector(connector);
38      }


41      protected class TableItemConnector
42          extends ItemConnector
43          playedBy AbstractTableDataEntry<@mi>
44          base when (hasRole(base, TableItemConnector.class))
45       {
46          private TableItem item = null;
```

```
48          public void setTable(Table table) {
49              AbstractTableDataEntry<@mi> data = this;

51              item = new TableItem(table, SWT.LEFT);
52              item.setData(DATA_KEY, data);
53              ViewTeam.this.register(item, this);

55              updateValues();
56          }

58          void updateValues() {
59              if(item != null) {
60                  item.setText(getColumns());
61                  item.setChecked(isChecked());
62                  item.setGrayed(isGrayed());
63              }
64          }

66          void dispose() {
67              item.dispose();
68          }

70          private abstract String[] getColumns();
71          private abstract boolean isChecked();
72          private abstract boolean isGrayed();

74          getColumns -> getColumns;
75          isChecked  -> isChecked;
76          isGrayed   -> isGrayed;

78          updateValues <- after notifyDataChanged;
79          dispose <- after notifyRemoved;
80      }

82      // registriere über declared lifting (siehe Text)
83      protected void register(
84          AbstractTableDataEntry<@modelInterface>
85          as TableItemConnector connector)
86      {}

88      ...
```

```
90 }
```

Der Code bedarf einiger Erläuterungen, insbesondere, da hier bisher nicht beschriebene OT/J Eigenschaften ausgenutzt werden: Ein *base guard*, wie in Zeile 17, wird ausgewertet, wenn OT/J eine Basis automatisch liften will, und zwar unmittelbar vor der Ausführung eines *callins*. Der *guard* oben hat nun zur Folge, dass *callins* nur ausgeführt werden, wenn zu dem entsprechenden Basisobjekt schon eine Rolle existiert. Ohne diesen *guard* würde die OT/J -Runtime eine neue Rolle erzeugen. Im Prinzip ist dieser *guard* nicht unbedingt nötig. Die Beschränkung auf das verknüpfte Modell-Interface reicht eigentlich schon aus. Es ist aber denkbar, dass ein komplexer Dialog diverse Tabellen darstellt, von denen aber nur ein Teil sichtbar sind. Für die nicht sichtbaren Tabellen müssen demnach keine Items erzeugt werden. Der *guard* wird also in einigen Fällen die Performance erhöhen.

Das Team organisiert Rollen und hat Methoden, um darauf Einfluss zu nehmen: *boolean hasRole(Object aBase)* liefert *true*, wenn es zu der angegebenen Basis bereits eine Rolle gibt - die hier verwendete Variante mit einem *class* Parameter schränkt die Rollen-Suche auf die übergebene Klasse ein. *unregister(Object aBase)* entfernt eine Rolle aus der Liste der Rollen. Durch *declared lifting*, wie dies in Zeile 84 erfolgt, wird der *guard* nicht ausgewertet, sondern das Lifting erzwungen. In der Folge liefert *hasRole()* für die registrierten Basisobjekte nun *true* und die *callins* sind aktiv.

Was nun noch fehlt ist, dass die View initial und bei neuen Tabelleneinträgen die entsprechenden DataEntries im passenden Team registriert. Die Initialisierung der View wird in Abschnitt 4.4 noch genauer behandelt.

Die Erstellung des oben aufgeführten Codes ist, wie zu ersehen, mit nicht unerheblichem Aufwand verbunden. Wenn man aber bedenkt, dass dies alles generiert wird und der Entwickler sich nur noch um die Datenbeschaffung (Implementierung des AbstractTabelDataEntry) und ein paar wenige Methodenaufrufe kümmern muss, relativiert sich der Aufwand schnell. Ein kleines Beispiel dazu: Angenommen, es soll eine Liste von Personen mit zwei Spalten, „Name“ und „Vorname“, dargestellt werden. Der Code im Modell-Interface sähe nun so aus:

```
1 public team class ModelInterfaceTeam
2     extends AbstractModelInterfaceTeam
3 {
4     // Die Datenquelle im Core-Modell
5     private DataRoot data;
```

```
7       // Das Team hat genau ein ModelInterface, dass als Basis für Controller
8       // und der View dient
9       private ModelInterface mi = new ModelInterface();

11      public class ModelInterface {
12          // Diese Methode ist abstrakt im AbstractModelInterfaceTeam
13          // definiert und muss hier implementiert werden.
14          protected AbstractTableDataEntry[] getPersonsData()
15           {
16              return getPersonEntries(data.getPersons());
17          }

19          ...
20      }

22      // Diese Methode lifted ein Array von Persons
23      private AbstractTableDataEntry[] getPersonEntries(
24          Person[] as PersonDataEntry[] entries)
25       {
26          AbstractTableDataEntry[] res = entries;
27          return res;
28      }

30      protected class PersonDataEntry
31          extends AbstractTableDataEntry
32          playedBy Person
33      {
34          public String[] getColumns() {
35              return new String[]
36                  {getName(),
37                   getFirstName()};
38          }
39          abstract String getName();
40          abstract String getFirstName();

42          String getName() -> String getName();
43          String getFirstName() -> String getFirstName();

45          notifyDataChanged <- after setName;
46          notifyDataChanged <- after setForename;
47      }

49      ...
```

```
50 }
```

Im *AbstractModelInterfaceTeam* wurde eine abstrakte Rolle generiert, die für die Tabelle *persons* eine Methode enthält, um die (initialen) Einträge für die Tabelle zu holen. Diese Methode (*getPersonsData()*) muss nun vom Entwickler hier implementiert werden. Das ModelInterfaceTeam erhält bei der Erzeugung ein Objekt, das quasi als Quelle für die Core-Daten dient. Die Methode *getPersonsData()* verwendet diese Quelle, um die Liste von Personen aus dem Core-Modell zu holen und liftet diese Liste zu einer Liste von *PersonDataEntry*. Diese Klasse muss von *AbstractTableDataEntry* erben und mindestens die Methode *getColumns()* implementieren, die wiederum ein String-Array mit den Spaltendaten zurückliefert. Durch die *played-By*-Beziehung zum Core-Objekt *Person* werden die Werte der Spalten über *callout*-Bindungen bestimmt. Zuletzt wird noch die *notifyDataChanged()* an alle Methoden gebunden, die die Daten für die Spalten ändern.

Bäume

Für einen Baum sieht die Lösung im Kern genauso aus. Allerdings entsteht ein neues Problem, wenn es um das Aufklappen (Expandieren) eines Knotens geht. Nicht selten ist es sinnvoll, die Unterknoten erst bei Bedarf nachzuladen. Insofern muss man davon absehen, beim Baumaufbau bereits den gesamten Baum abzufragen, sondern jede Ebene erst dann aufzubauen, wenn diese auch dargestellt werden soll. Demzufolge liegt es nahe, die abstrakte Datenklasse für Bäume zunächst um entsprechende Methoden zu erweitern[3]:

```
1  public abstract class AbstractTreeDataEntry {
2      ...
3      public abstract boolean hasChildren();
4      public abstract AbstractTableDataEntry[] getChildren();

6      // getParent() wird null für die obersten Elemente liefern
7      public abstract AbstractTreeDataEntry getParent();

9      public void notifyChildAdded(AbstractTreeDataEntry
           child) {}
10     ...
11 }
```

[3]Ich beschränke mich bei den Baumbeispielen auf die Strukturelemente. Darzustellende Werte, Icons etc. sowie deren Notifizierungs-Methoden werden analog zur Table realisiert, aber in diesen Code-Ausschnitten ignoriert.

Der Entwickler kann bei der Implementierung selbst entscheiden, wann die Kinder eines Knoten erzeugt werden. Der Connector hat die Aufgabe, den Baumknoten entsprechend darzustellen. Ein SWT-Tree hat die Eigenschaft, Knoten nur dann aufklappbar darzustellen, wenn der Knoten auch tatsächlich mindestens einen Unterknoten hat. Da Unterknoten aber erst beim Aufklappen tatsächlich bestimmt werden sollen, muss der Connector einen Dummy-Knoten einfügen, solange er noch nicht aufgeklappt wurde[4].

```
1  protected team class ViewTeam {

3      // die zwei register-Methoden, die je nach zweitem Parameter eine
4      // Wurzel oder einen Kind-Knoten erzeugen
5      public void register(AbstractTreeDataEntry<@mi> as
6                  TreeItemConnector conn, Tree parent)
7      {
8          conn.setTree(parent);
9      }

11     public void register(AbstractTreeDataEntry<@mi> as
12                 TreeItemConnector conn, TreeItem parent)
13     {
14         conn.setParent(parent);
15     }

17     // Wird ein Knoten expandiert, ruft die View diese Methode auf:
18     public void expanded(TreeItem item) {
19         AbstractTreeDataEntry<@mi> data =
20             (AbstractTreeDataEntry<@mi>) item.getData(
                   DATA_KEY);
21         if(data != null)
22             expanded(data);
23     }

25     private void expanded(AbstractTreeDataEntry<@mi> as
26                 TreeItemConnector conn)
27     {
28         conn.expand();
29     }

31     // Der abstrakte Connector implementiert alle relevanten Methoden.
32     protected class TreeItemConnector
```

[4]Dummy-Knoten für die Anzeige, dass Unterknoten existieren, wirken etwas archaisch. Dennoch ist es leider noch immer nötig - vgl. [GUO05].

```
33          extends ItemConnector
34          playedBy AbstractTreeDataEntry<@mi>
35          base when (hasRole(base, TreeItemConnector.class))
36      {
37          // Das verknüpfte TreeItem
38          private TreeItem item = null;

40          // Zustandsinformation, ob der Knoten schon einmal expandiert wurde
41          // bzw. ob getChildren() schon einmal aufgerufen worden ist:
42          private boolean  everExpanded = false;

44          // der dummy Knoten, der dafür sorgt, dass bei vorhandenen Kindern
45          // der Knoten expandierbar ist
46          private TreeItem dummy = null;

48          public void setTree(Tree tree) {
49              AbstractTreeDataEntry <@mi> data = this;

51              item = new TreeItem(tree, SWT.NULL);
52              item.setData(DATA_KEY, data);

54              // create DisposeWatcher
55              register(item, this);

57              initTree();
58          }

60          public void setParent(TreeItem parent) {
61              AbstractTreeDataEntry <@mi> data = this;

63              item = new TreeItem(parent, SWT.NULL);
64              item.setData(DATA_KEY, data);

66              // create DisposeWatcher
67              register(item, this);

69              initTree();
70          }

72          private void initTree() {
73              if(hasChildren())
74                  dummy = new TreeItem(this, SWT.NULL);
75              updateNode();
```

```
76          }

78          protected void updateNode() {
79              setText(getNodeName());
80              if(hasChildren() == false) {
81                  removeDummy();
82                  everExpanded = false;
83              }
84          }

86          public void expand() {
87              removeDummy();

89              for(AbstractTreeDataEntry<@mi> child :
                    getChildren()) {
90                  TreeTeam.this.register(child, this);
91              }

93              everExpanded = true;
94          }

96          private void removeDummy() {
97              if(dummy != null) {
98                  dummy.dispose();
99                  dummy = null;
100             }
101         }

103         protected void nodeAdded(
104                     AbstractTreeDataEntry<@mi> child) {
105             if(everExpanded)
106                 TreeTeam.this.register(child, this);
107             else
108                 if(dummy == null)
109                     dummy = new TreeItem(this, SWT.NULL);
110         }

112         void dispose() {
113             item.dispose();
114         }

116         protected abstract String getNodeName();
117         protected abstract boolean hasChildren();
```

```
118         protected abstract AbstractTreeDataEntry<@mi>[]
                getChildren();

120         getNodeName -> getName;
121         hasChildren -> hasChildren;
122         getChildren -> getChildren;

124         // labeled callins: für evtl. benötigte precedences und zum Überschreiben
125         dataChanged:    updateNode <-
126                 after notifyDataChanged;
127         childAdded:         nodeAdded  <-
128                 after notifyChildAdded;
129         nodeRemoved: dispose           <-
130                 after nodeRemoved;
131     }

133 }
```

4.4 Initialisierung / Start

In Abschnitt 4.2 wurde gefordert, dass der Programmierer, der einen solch generierten Dialog starten will, diesen wie eine einfache Dialog-Klasse verwenden können soll. Damit ein Aufruf, wie `new MyDialog(rootData).show()` funktioniert, muss nun, als letzter Punkt, die Initialisierung des kompletten Teams geklärt werden.

Das GUI-Team organisiert im Grunde exakt jeweils eine Instanz des *ModelInterfaceTeam*, des *Controller* und des *ViewTeam*. Man kann nun ausnutzen, dass sowohl der Controller als auch die View Rollen des *ModelInterface* sind. Durch entsprechende *callins* kann darauf verzichtet werden, Instanzen dieser Klassen zu erzeugen, solange die entsprechenden Teams aktiv sind. Dazu erhält das Modell-Interface zwei leere Methoden, die quasi als Trigger für Aktionen in der *View* und dem *Controller* dienen: *init()* und *start()*. Ein Seiteneffekt ist, dass das konkrete Modell-Interface damit in der Lage ist, jeweils noch weitere Operationen auszuführen.

```
1 public team class GUITeam {

3     // Die Modell-Interface Instanz
4     private final AbstractModelInterfaceTeam mi;
```

```
6       // Die Controller-Instanz (Erklärung folgt)
7       private AbstractController controller = null;

9       public MyDialogTeam(Object rootData) {
10          // Erzeuge Modell-Interface
11          modelInterface = new ModelInterfaceTeam(rootData);

13          // Erzeuge ViewTeam
14          new ViewTeam();
15          activate();

17          modelInterface.init();
18      }

20      public void show() {
21          mi.start();
22      }

24      protected abstract team class
            AbstractModelInterfaceTeam {

26          protected ModelInterface modelInterface = null;

28          public void init() {
29              if(modelInterface != null)
30                  modelInterface.init();
31              else
32                  // Fehler! Spezialisierung muss das Attribut setzen!
33          }

35          public void start() {
36              if(modelInterface != null)
37                  modelInterface.start();
38              else
39                  // Fehler! Spezialisierung muss das Attribut setzen!
40          }

42          public abstract class ModelInterface {
43              void init() {}
44              void start() {}

46              ...
47          }
```

```
48      }

50      protected team class ModelInterfaceTeam {
51          public ModelInterfaceTeam(Object rootData) {
52              // rootData auswerten oder merken
53              modelInterface = new ModelInterface();
54              activate();
55          }
56      }
57  }
```

Die *View* ruft im Konstruktor zunächst die eigene *createSShell()*-Methode auf, die den Dialog mit seinen Widgets erstellt (aber noch nicht darstellt). Des Weiteren erhält sie eine Methode *initData()*, die als *after callin* an die *init()* Methode des Modell-Interface gebunden wird. Diese ist nun dafür verantwortlich, die Widgets mit Leben zu füllen und nutzt dazu das generierte abstrakte Modell-Interface.

Der *Controller* erhält ebenfalls eine *init()*-Methode, die an die *init()*-Methode des Modell-Interface gebunden wird. Damit die View auf diesen Controller zugreifen kann, ist es nur noch nötig, die Controller-Instanz im GUI-Team zu speichern Diese Konstruktion hat zur Folge, dass nach dem Aufruf von *init()* im Konstruktor des GUITeam alle Objekte existieren, die für das Funktionieren des Dialogs vorhanden sind.

```
1  protected team class ViewTeam {

3      public ViewTeam() {
4          activate();
5      }

7      public class View
8          playedBy ModelInterdace<@mi>
9      {
10         // custom lifting Konstruktor
11         View(ModelInterdace<@mi> b) {
12             createSShell();
13         }

15         ...

17         initData() {
18             // Fülle Widgets mit Daten
19             ...
```

```
20          }

22          void show() {
23              // starte Dialog
24              ...
25          }

27          initData <- after init;
28          show     <- after start;
29      }
30  }

32  protected class Controller
33      extends AbstractController
34      playedBy ModelInterdace<@mi>
35  {
36      // custom lifting Konstruktor
37      Controller(ModelInterdace<@mi> b) {
38          GUITeam.this.controller = this;
39      }

41      ...

43      // nur um gebunden zu werden:
44      void init() {}

46      init <- after init;
47  }
```

4.5 Zusammenfassung

Das beschriebene Modell ist umfangreich, erlaubt es aber, die Implementierung des Dialogs durch den Entwickler auf das Wesentliche einzuschränken: die Implementierung der Dialog-Logik (durch den Controller) und die Beschaffung der Daten.

Eine ähnliche Diskussion wird in [AOSD03] geführt, konzentriert sich jedoch primär auf die Modell-View-Beziehung. Modell und View sind hier voneinander gänzlich unabhängig und werden durch eine dritte Komponente miteinander verknüpft. Diese Komponente ist hier ein Connector. Im Grunde hat der Connector dieselbe Aufgabe wie das Modell-Interface. Allerdings ist

der Connector für das Füllen der Widgets verantwortlich, was hier die View selbst durchführt. In dem Papier wird aber nur ein verhältnismäßig einfacher Fall beschrieben, der gut funktioniert, wenn die Daten im (Core-) Modell weitestgehend so vorliegen, wie sie für die View benötigt werden. Wenn dem aber nicht so ist, dann muss der Connector die Aufbereitung zusätzlich übernehmen und er würde sich dem Modell-Interface in seiner Struktur annähern.

Das Papier geht auf den Controller nur beschränkt ein. Die Frage der potentiellen Redundanz des Modell-Zugriffs durch Rollen wird daher auch nicht betrachtet. Auch Fragen, wie z.B. transaktionale Dialoge bzgl. der (temporären) Datenhaltung funktionieren, bleibt ungeklärt. Daher ist der Ansatz des Modell-Interface die allgemeinere Lösung, die, durch eine Abstraktion der Core-Daten, alle Möglichkeiten offen lässt.

Ein letzter Kritikpunkt an der Lösung, wenn auch verhältnismäßig unrelevant, jedoch durchaus diskussionswürdig, ist, dass der Connector sich nicht so recht in die Dreiteilung des MVC einordnen lässt, da er weder ausschließlich zur View, zum Controller oder zum Modell gehört. Das hier vorgestellte Modell ist in diesem Sinne „sauberer“: Das Modell-Interface ist für das Modell zuständig, das ViewTeam für die View und die Controller-Klassen für den Controller.

Kapitel 5

Die Realisierung

Die Realisierung des GUI-Designers nach den beschriebenen Vorgaben besteht grob aus drei Teilen: Zunächst muss der VisualEditor mit OT/J -Teams und Rollen umgehen können, insbesondere ist er ursprünglich nicht in der Lage, mit einer Visual-Class umzugehen, die in einer anderen Klasse geschachtelt ist. Da Rollen aber zwangsläufig in einem Team geschachtelt sind, war hier Änderungsbedarf. Der zweite Schritt besteht in der Anpassung des initialen Erzeugens einer Visual Class. Hier erfolgt vom Benutzer die Auswahl, ob die Visual Class für OT/J oder klassisch für Java erzeugt werden soll. Der dritte Teil umfasst die Arbeit, das Framework bei Änderungen, die durch den VE durchgeführt werden, anzupassen, also z.B. auf das Hinzufügen oder Löschen von Widgets zu reagieren und entsprechend das Modell-Interface anzupassen.

Die Realisierung erfolgt in Form eines eclipse Plugins. Um es vorwegzunehmen: Das Plugin wird als OT/J -Plugin realisiert und verwendet für einige Aufgaben Rollen, die Klassen aus dem VE adaptieren. Die Teile 1 und 2 werden komplett über Rollen realisiert. Da diese überschaubar sind, werden sie in einem Paket zusammengefasst. Teil 3 umfasst komplexere Strukturen, die entsprechend ihren Aufgaben in verschiedene Pakete verteilt werden (Abbildung 5.1).

5.1 Teil 1: VE mit Rollen, Teams und geschachtelten Klassen

Das MVC Framework, sofern es AOP-Eigenschaften ausnutzen will, sollte die Dialogklasse in Form einer Rolle generieren, um ein sauberes Modell

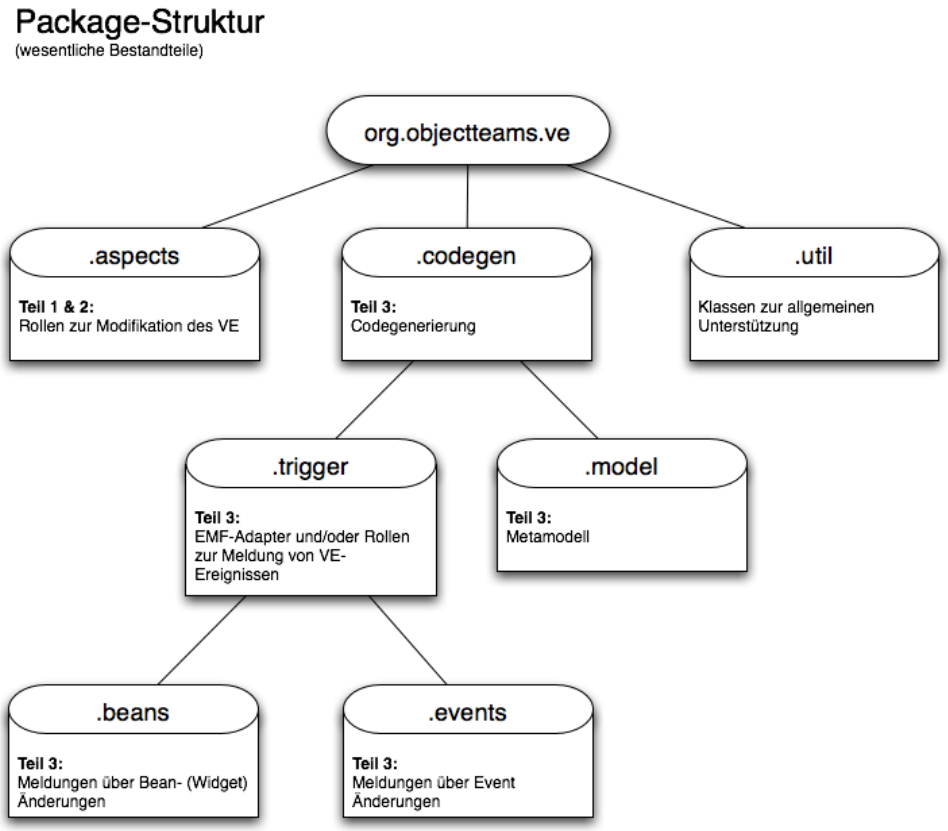

Abbildung 5.1: Paketstruktur

zu beschreiben (vgl. 4). Das bedeutet, dass der VE zum einen mit einer verschachtelten Dialog-Klasse umgehen können muss und zum anderen sich nicht durch OT/J -Schlüsselworten stören lassen darf. Wie Versuche gezeigt haben, scheint der VE mit OT/J -Schlüsselwörtern klar zu kommen, allerdings nicht mit verschachtelten Klassen.

Ein genauerer Blick auf dessen Implementierung oder auch ein Durchsuchen der VE-Newsgroup zeigt es recht schnell: Der VE ist nicht dafür vorgesehen, mit verschachtelten Klassen umzugehen. Auch andere Entwickler haben diese Eigenschaft vermisst, aber aus unerklärlichen Gründen hat das VE-Team bis heute nichts in diese Richtung unternommen.

Der VE geht prinzipiell davon aus, dass die VisualClass die oberste und „erste“ Klasse einer CompilationUnit[1] ist - sie wird im VE als **MainType** be-

[1]Eine *CompilationUnit* (Übersetzungseinheit) ist im Sinne des JDT eine Modellrepräsentation einer .java-Datei.

zeichnet. Das Finden der VisualClass erfolgt immer auf eine von zwei Arten: Über eine AST-CompilationUnit (AST-CU) oder über eine ICompilationUnit des Java-Model (JM-CU) . Beide haben, außer deren semantischer Bedeutung, nicht viel gemeinsam: Es sind zwei Repräsentanten einer .java-Datei in zwei verschiedenen abstrakten Modellen des Java-Codes - mehr zu diesen beiden Modellen folgt in Abschnitt 5.3.

Die Bestimmung des MainType erfolgt an etlichen Stellen im VE. Die VE-Entwickler haben eine Utility-Klasse implementiert (*CodeGenUtil*), die, neben vielen anderen Funktionen, die Bestimmung des MainType anbietet; aus unerklärlichen Gründen jedoch nur für die Verwendung mit einer AST-CU. Daraus folgt, dass überall, wo eine JM-CU verwendet wird, der VE den MainType über eine Code-Zeile, wie „`type = cu.types[0]`“ bestimmt. Die Implementierung in *CodeGenUtil* für eine AST-CU sieht, das sei hier noch bemerkt, im Wesentlichen genauso aus.

Beide Ansätze haben ihre Nachteile, wenn es darum geht, die Bestimmung des MainType intelligenter zu gestalten. Es gibt drei mögliche Wege:

1. Der VE-Code wird komplett kopiert und die Änderungen direkt im VE vorgenommen. Der Vorteil: Man hat vollen Zugriff auf alle nötigen Dinge. Sind benötigte Methoden oder Attribute versteckt, kann man deren Sichtbarkeit ändern. Die Nachteile kommen erst später zum Tragen. Man muss jede Aktualisierung des VE mitführen und die gemachten Änderungen dort wieder hinzufügen. Außerdem stellt sich die Frage der (konfliktfreien) Verteilung des „neuen“ VE.

2. Der VE Code wird über ein eigenes Plugin von außen über Spezialisierungen und eigenen speziellen eclipse-Actions, die eben dieses Plugin, anstatt des Originals verwenden, „korrigiert“. Aber das ist schwieriger, als es zunächst den Anschein hat: Da CodeGenUtil eine statische Klasse mit statischen Methoden ist, hilft Zentralisierung hier nicht. Diese Klasse kann nicht spezialisiert oder einfach ersetzt werden. Es müssen alle Stellen gefunden werden, an denen der direkte Zugriff auf den MainType über AST-CU oder JM-CE erfolgt, und diese ersetzt werden - das sind reichlich viele.

   Ein weiterer Punkt ist die Kapselung: Viele der Methoden, die den MainType bestimmen, sind privat, package-visible oder verwenden derartige Methoden. Um diese ersetzen zu können, müssen entweder alle Verwendungen dieser Methoden verändert oder gleich die ganze Klasse durch eine eigene ersetzt werden. Das hat nicht nur zur Folge, dass hier

viel Code aus dem Original kopiert werden muss, sondern auch, dass viele Methoden eine ganze Reihe anderer Punkte nach sich ziehen, die ebenfalls verändert werden müssen.

Der Aufwand hierfür ist sehr hoch, und man hat trotzdem das Problem der Wartung wie bei einer kompletten Kopie. Im Gegenteil: Die Wartung dürfte sogar noch schwieriger werden, da ein direkter Vergleich zwischen zwei Projekten hier gar nicht möglich ist und statt dessen jedes kopierte Code-Fragment separat untersucht werden muss.

3. Wenn man sich anschaut, was die Ursache der Probleme der zweiten Variante sind, nämlich hauptsächlich die Kapselung, und man sich überlegt, dass mit OT/J eben gerade diese aufgebrochen werden kann, so liegt der Gedanke nahe, OT/J dafür zu „missbrauchen“, die Unzulänglichkeiten des VE mittels Rollen auszugleichen. Diese Verwendung von OT/J ist im Grundsatz fragwürdig: Rollen sind eigentlich nicht dazu gedacht, um Kapselungen zu brechen, sondern primär um crosscutting-Code zu verhindern. Es wird also eine Randerscheinung von OT/J ausgenutzt. Andererseits ist das Aufbrechen der Kapselung in anderen Fällen sogar erwünscht (vgl. Abschnitt 2.4). Ein anderer Aspekt ist, dass OT/J , auf eclipse-Plugins angewendet, noch eine sehr junge Idee ist und die Gefahr besteht, dass es noch völlig unbekannte Probleme aufwerfen kann.

Die Auswahl des zu verwendenden Wegs fällt auf die OT/J Variante. Durch die Mechanismen von replace-callins müssen, wie bei der ersten Variante, ausschließlich die Methoden verändert werden, die den MainType bestimmen und nicht über CodeGenUtil abgedeckt sind. Diese Anzahl hält sich in Grenzen. Allerdings reduziert sich der Wartungsaufwand maximal auf die ersetzten Methoden. Da deren Zahl aber gering ist, dürfte die Wartung nur minimal größer sein als bei der ersten Variante (größer, weil die Änderungen hier nicht durch einen Projektvergleich in der Versionsverwaltung durchgeführt werden kann). Die Nachteile dieser Variante werden durch die verhältnismäßig schnelle Lösung in Kauf genommen.

Die entstandene Lösung ist verhältnismäßig klein: Für jede Klasse, die eine Methode besitzt, die angepasst werden muss, wurde ein Team mit einer (oder selten auch mehrerer) Rolle geschrieben, die die entsprechende Methode über ein replace-callin ersetzt. Es wurden so zwölf Teams implementiert, die zumeist nur eine Methode ersetzen. In zwei Fällen war etwas mehr zu tun, da die zu ersetzenden Methoden sehr lang sind und entsprechend viele Seiteneffekte haben, die entsprechend behandelt werden mussten.

5.2 Teil 2: Eine neue VisualClass anlegen

Um eine neue Visual-Class anzulegen, spezialisiert der VE den Wizard, der im JDT für das Anlegen von Java-Klassen zuständig ist. In diesen Dialog fügt er ein paar weitere Felder ein, wie z.B. die Auswahl des GUI-Framework. Wird der Dialog abgeschlossen, so wird entsprechend eine Visual Class angelegt. Die Erzeugung der Visual Class erfolgt in zwei Schritten: Zunächst wird eine Java-Klasse über den ursprünglichen Java-Class-Wizard angelegt und danach erhält diese Klasse (im Fall von SWT) das Shell-Attribut sowie die *createSShell()* Methode, die diese Shell anlegt.

Die Erweiterung um die Generierung eines OT/J -MVC Framework umfasst das Hinzufügen einer Auswahlbox, über die der Anwender entscheiden kann, ob das Framework mit generiert werden soll. Ist dies der Fall, wird zudem noch das initiale Framework erzeugt.

Die Einführung der Auswahlbox ist verhältnismäßig einfach: der Wizard definiert bereits verschiedene Methoden, die jeweils einen Teil des Dialogs erzeugen, insbesondere auch für die Liste der Auswahlboxen rechts unten neben der Style-Auswahl (siehe linker Teil in Abbildung 5.2). Um nun eine Auswahlbox für das MVC-Framework hinzuzufügen, reicht es aus, über eine Rolle diese Methode, mittels eines *after callin*, um das Erzeugen der Auswahlbox für das Framework zu erweitern. Diese muss als erstes die Style-Box etwas verlängern und kann dann die Auswahlbox unter den anderen erzeugen (vgl. Abbildung 5.2).

Wird nun der Dialog beendet und es soll ein MVC-Framework mit generiert werden, so übernimmt die OT/J -Erweiterung des Dialogs die Erzeugung aller Klassen und liefert dem VE-Wizard die erzeugte View-Rolle, um diese, wie oben, mit VE-Code zu füllen. Da der original Dialog hier die *createType()*-Methode des Java-Class-Wizards nutzt, wird diese durch ein *replace callin* ersetzt. Das Plugin `org.objectteams.otdt.ui`, das zur eclipse-Integration von OT/J gehört und einen eigenen Wizard implementiert, um Teams und Rollen anlegen zu können. Dieser Wizard basiert ebenfalls auf dem Java-Class-Wizard, hat aber die Erzeugung der Klassen ausgelagert. Die ersetzte *createType()*-Methode verwendet nun diesen ausgelagerten Mechanismus, um das MVC-Framework (zunächst leer) zu erzeugen. Die dabei erzeugte View-Rolle (vgl. Abbildung 4.3) wird anschließend dem VE übergeben, um ihm die Gelegenheit zu geben, seine Modifikationen vorzugeben.

Da die Konsistenz des Framework sichergestellt werden muss, solange der Dialog bearbeitet wird, und diese Konsistenz auch alle (fest) generierten

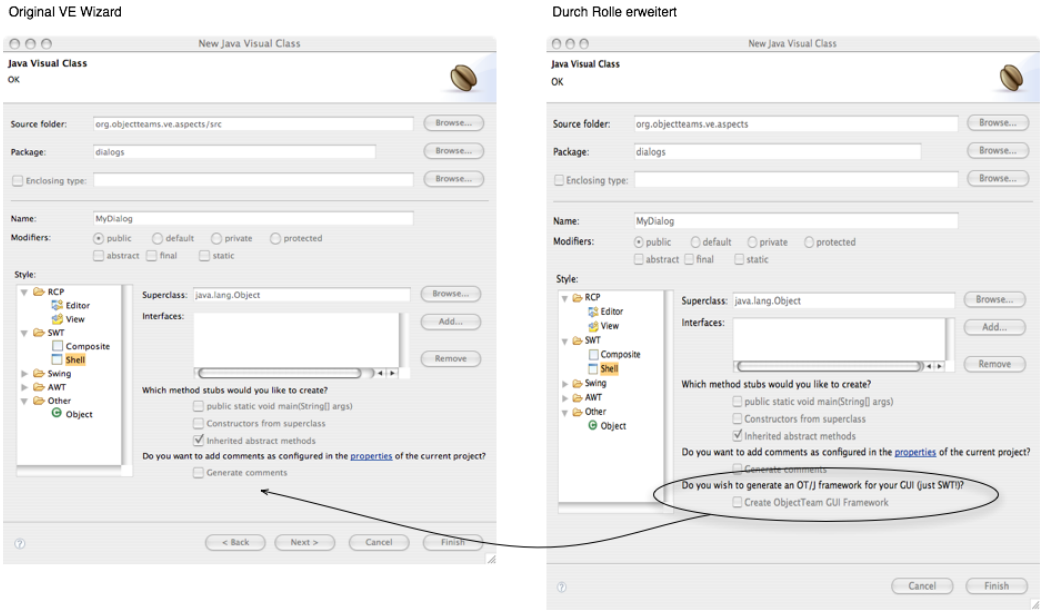

Abbildung 5.2: VisualClass Wizard

Methoden und Attribute umfasst, reicht es bis hierhin, das Framework in Form von leeren Klassen zu generieren. Wenn der Editor für den Dialog geöffnet wird, sei es durch den Wizard oder durch den Benutzer, greifen nun die Mechanismen des dritten Teils:

5.3 Teil 3: Die Framework-Konsistenz

Um während der Bearbeitung einer VisualClass durch den Benutzer das MVC-Framework konsistent und aktuell zu halten, wurde ein Meta-Modell entwickelt. Dieses Meta-Modell erstellt sich aus Informationen, die der VE bereitstellt, und aus Informationen des JDT/OTDT. Da der VE Änderungen am Dialog und an dessen Code in seinen Modellen widerspiegelt, wird das Meta-Modell an diese derart gebunden, dass es über jede Änderung an der Dialogstruktur informiert wird. Jede Meldung hat zur Folge, dass das Metamodell seine Strukturen entsprechend anpasst und ggf. Code-Änderungen im Framework durchführt. Die Meldungen über Änderungen können in verschiedener Weise realisiert werden: Das VE-Modell ist ein EMF-Modell (vgl. Abschnitt 3.2.2) und kann daher über EMF-Adapter mit Beobachtern versehen werden. Das BD-Modell hingegen kann ausschließlich über Rollen beobachtet werden. Welche dieser Methoden verwendet wird, ist abhängig

davon, welche Methode den wenigsten Aufwand erzeugt. Das ist jedoch nicht immer klar zu erkennen. Beispielsweise wird ein Event aus dem VE-Modell auch temporär entfernt. Eine Lösung über einen EMF-Adapter hätte aber zur Folge, dass das Meta-Modell entsprechend mehrfach benachrichtigt wird. Andererseits werden diverse Aktionen über komplexe anonyme Klassen realisiert, so dass die Verwendung einer Rolle hier nicht in Frage kommt, da zu viel Code kopiert werden müsste. Die Wahl der Realisierungsart ist also stark von den inneren VE-Realisierungen abhängig.

Für die Code-Manipulation, sowohl lesend als auch schreibend, gibt es prinzipiell zwei Wege: Über den AST (Abstract Syntax Tree) oder über das Java-Modell (JM). Der AST ist eine feingranulare Repräsentation des Java-Codes. Die Erzeugung eines AST ist verhältnismäßig aufwändig, die Verwendung hat aber auch diverse Vorteile. Änderungen, die am AST vorgenommen werden, können recht einfach wieder in ein Dokument geschrieben werden. So gesehen funktioniert der AST transaktional. Es ist auch möglich aus Codefragmenten einen (Teil-) AST zu erzeugen und diesen dann z.B. in einen anderen kopieren.

Das JM hingegen ist nicht so fein granular wie der AST und basiert nicht auf Dokumenten. Statt dessen ist es eher eine Art permanentes Modell des Projekts. Da nicht zu jedem Zeitpunkt alle Informationen schon vorhanden sind, werden Teile des Modelle erst bei Bedarf aufgelöst. Wenn im JM etwas erzeugt wird, so ist dieses Element auch sofort da und muss nicht zuerst noch in ein Dokument umgewandelt werden. Das JM ist damit in der Regel schneller, aber eben nicht so komfortabel.

Diese Modelle stammen aus dem JDT. Das OTDT erweitert das JDT um OT/J-Eigenschaften[2]. Zu diesen Erweiterungen gehört u.a. auch ein erweitertes JM und ein erweiterter AST. Beide Modelle enthalten entsprechend Einträge, die OT/J-spezifisch sind. Für den VE sind diese Erweiterungen transparent: er arbeitet lediglich auf den Java-Elementen. Für die Konsistenzhaltung des Frameworks werden jedoch auch die Eigenschaften der Erweiterungen verwendet.

In beiden Modellen repräsentiert eine *CompilationUnit* (CU) eine Java-Datei. Betrachtet man das MVC-Modell, so stellt man fest, dass alle Strukturen, die für den Anwender nicht zum Verändern gedacht sind, in derselben Java-Datei liegen, oder besser: in der selben CU - nämlich in derjenigen, in der auch der Code steht. Da hier der bei weitem meiste Code generiert

[2]Genaugenommen ersetzt das OTDT das JDT und ist dabei abwärtskompatibel, so dass andere Plugins, die auf dem JDT basieren, auch mit dem OTDT zusammenarbeiten können.

und verwaltet werden muss, liegt es nahe, nicht zuletzt aufgrund der transaktionalen Eigenschaften, den AST zu verwenden. Der VE verwendet zur Code-Analyse selbst einen AST. Leider kann dieser jedoch nicht mit verwendet werden ohne den VE noch weiter zu modifizieren: Den AST gibt es in zwei (weitestgehend inkompatiblen) Versionen: JLS2 und JLS3 (JLS: Java Language Specification). Der VE verwendet noch JLS2. Dieser ist jedoch für OT/J nicht komplett implementiert und kann daher nicht verwendet werden. Es ist also nötig, für das Meta-Modell einen JLS3-AST zu nutzen.

Die Code-Generierung in den anderen CUs ist eher schlicht: Es werden Methoden aus den abstrakten Klassen vorgeneriert, die i.d.R. dazu gedacht sind, vom Programmierer mit Leben gefüllt zu werden. Folglich ist es hier nur notwendig, Methoden zu generieren - es müssen keine Code-Stücke verändert werden (vorgenerierte Methoden dürfen sogar im Nachhinein nicht mehr verändert oder gelöscht werden, da potentiell der Entwickler diese bereits verändert hat und/oder selbst verwendet).

Die Code-Generierung erfolgt in zwei Varianten: Der Code ist immer gleich (z.B. für den Table/Tree-Connector - vgl. Abschnitt 4.3.3) oder muss Stück für Stück generiert werden (z.B. die Dateninitialisierung der View - vgl. 4.4). Für größere Code-Stücke wird ein Verfahren genutzt, das zusammen mit EMF angeboten wird: JJET (Java Emmitter Templates). JJET-Dateien sind JSP-ähnliche Dateien und beschreiben Java-Code-Stücke. Aus diesen wird eine Klasse erzeugt, die Code, basierend auf vorgegebenen Parametern, in Form eines String generiert.

Für die Generierung in den AST wird bei größeren oder komplexeren Code-Stücken das JJET-Verfahren genutzt. Aus dem generierten String wird ein Teil-AST generiert und dieser in den Haupt-AST eingefügt. Einfachere Code-Generierung erfolgt i.d.R. über einfachere String-Konkatenationen von Konstanten (zentral definiert) und variablen Werten.

Die Code-Generierung des VE muss nur für ein einziges Element modifiziert werden: Den Event-Handler. Event-Handler werden im Original als anonyme Klassen generiert, deren Event-Methode(n) einen Kommentar und eine Konsolen-Ausgabe enthalten. Dieser Code muss so verändert werden, dass das Event an den Controller weitergereicht wird (vgl. 4.2.1). Der VE selbst generiert diesen Code über ein JJET. Entsprechend wird die Methode zum Bestimmen dieses JJET über eine Rolle ersetzt, die bei einem MVC-Modell ein eigenes JJET liefert, welches die Delegation an den Controller definiert.

5.4 Zusammenfassung

Die erwarteten Probleme bei der Realisierung, durch die Verwendung von OT/J auf Plugins, haben sich nur bedingt bestätigt: Anfänglich gab es zwar einige Schwierigkeiten, diese konnten aber verhältnismäßig schnell gelöst werden. Probleme mit OT/J gab es vielmehr in der Entwicklung des Modells, was zu Verzögerungen in der Realisierung führte.

Die Implementierung des Meta-Modells hatte so ihre Tücken: Zum einen das Finden von sinnvollen Triggern und zum anderen dadurch, dass der VE und das Meta-Modell quasi gleichzeitig Code generieren wollen, mit der Folge, dass manchmal der VE seine Code-Zuordnungen verliert. Die Implementierung konnte zum Ende nicht mehr abgeschlossen werden. Da die Realisierung aber alle wesentlichen Bestandteile enthält, ist die Komplettierung des Codes nur eine Frage der Zeit.

Kapitel 6

Fazit

Dieser Arbeit umfasste, zurückblickend, drei wesentliche Teile (neben der Erstellung des Textes): die Herleitung des MVC-Modell für OT/J , die Modifikation des VE und die Realisierung der Generierung des MVC-Modells. Jeder dieser Teile verdient nun eine abschließende Betrachtung.

6.1 Das Modell

Das Modell entstand auf intuitive Weise und wurde sukzessiv präzisiert. Dennoch gab es nicht unerhebliche Schwierigkeiten. So wurde diese Arbeit begonnen, bevor OT/J einen Release-Stand erreichte. Entsprechend schwierig war es, das Modell in kleinen Tests auf Tauglichkeit hin zu überprüfen. Durch die speziellen Kombinationen von verschachtelten Rollen, die Aufteilung in verschiedene Role-Files, implizite Vererbung usw., entstanden viele Konstellationen, die den OT/J -Compiler überforderten. Trotz der intensiven Kommunikation mit den OT/J -Entwicklern (in Person: Stephan Herrmann), erreichte OT/J erst knapp zwei Wochen vor Beendigung dieser Arbeit einen Stand, der es erlaubte, das Modell soweit zu testen, dass sichergestellt war, dass das gedachte Modell auch funktioniert. Die in Abschnitt 5.1 erwähnte Gefahr, die es mit sich bringt, eine solch neue Technologie einzusetzen, wurde insofern bei den Tests für das Modell bestätigt.

Eine weiteres überraschendes Problem war die Einschränkung von SWT, dass Items nicht spezialisiert werden können. Die Lösung, wie sie in [AOSD03] beschrieben wird, konnte daher nicht übertragen werden. Wieso SWT diese Einschränkung hat und zudem noch auf diese Art und Weise realisiert (vgl. Abschnitt 4.3.3), bleibt eine offene Frage.

6.2 VE-Änderungen

Eigentlich war zu erwarten, dass es Probleme bereiten würde, wenn der VE OT/J- anstatt Java-Code auswerten muss. Tatsächlich kommt er jedoch mit OT/J im Prinzip zurecht. Allerdings kann der VE nicht damit umgehen, wenn die VisualClass geschachtelt ist. Die View (=VisualClass) im MVC-Modell (vgl. Kapitel 4) ist eine Rolle und damit zwangsläufig eine geschachtelte Klasse.

Die gemachten Einschränkungen im VE sind dazu noch völlig unnötig: Der VE bietet bereits Extension-Points an, um neue „Styles“ (SWT, AWT...) durch ein externes Plugin eintragen zu können. Wieso kann nicht auch dieses Plugin dem VE aus einer CompilationUnit die passende GUI-Klasse bestimmen? In der Newsgroup des VE-Projekts wurde schon öfter nach verschachtelten Klassen gefragt - ohne Ergebnis.

6.3 Framework-Generierung

Aufgrund der schon erwähnten Probleme mit OT/J ist es leider nicht gelungen, das komplette GUI-Modell aus einer VisualClass zu erzeugen. Die realisierten Teile sind jedoch in der Lage z.B. Events in der Form zu behandeln, dass ein eigener Event-Handler Code und passende Methoden in den beiden Controller-Klassen generiert werden. Damit ist das Modell im Grunde klar und muss in der Folge (nur) noch um die fehlenden Elemente erweitert werden.

Ein Problem mit dem realisierten Generator ist, dass die Erzeugung aus AST, JM und BDM u.U. sehr lange dauert - es wurden schon Zeiten von 30s für kleine Dialoge gemessen, wobei die Zeiten z.T. stark schwanken. Etwa die Hälfte der Zeit geht allein durch die Erstellung des AST verloren, der Rest durch das Aufbauen des Meta-Modells, was allerdings auch Code-Änderungen zur Folge haben kann. Für Folgearbeiten wäre es daher wünschenswert, die Performance-Probleme genauer zu analysieren und ggf. andere Wege zu finden.

Literaturverzeichnis

[AOSD03] Model-View-Controller and Object Teams: A Perfect Match of Paradigms
Stephan Herrmann, Matthias Veit
AOSD 2003 Conference
(2nd International Conference on Aspect-Oriented Software Development, March 2003, Boston)

[ARC01] Inside C Sharp
Tom Archer
Microsoft Press
ISBN: 0-7356-1288-9

[ECL06a] Eclipse Homepage: http://www.eclipse.org

[ECL06b] Contributing to eclipse - Principles, Patterns and Plug-Ins
Erich Gamma, Kent Beck
Addison Wesley Verlag
ISBN 0-321-20575-8

[EMF06] eclipse Modeling Framework
Budinsky, Steinberg, Merks, Ellersick, Grose
Addison Wesley Verlag
ISBN: 0-13-142542-0

[FLA05] Java in a Nutshell
David Flanagan
O'Reilley
ISBN: 0-59600-773-6

[GAM95] Design Patterns - Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Addison Wesley Verlag
ISBN: 0-201-63361-2

[GUO05] Professional Java Native Interfaces with SWT/JFace
Jackwind Li Guojie
Wrox (John Wiley & Sons, Ltd)
ISBN 0-470-09459-1

[MS06] Das Oberflächenkonzept von ,,Windows Vista" (ehemals ,,Longhorn") wird von Microsoft als ,,Windows Presentation Foundation" (ehemals ,,Avalon") bezeichnet. Informationen dazu sind hier zu finden:
http://msdn.microsoft.com/windowsvista/building/presentation/default.aspx

[NB06a] NetBeans Homepage
http://www.netbeans.org/index.html

[NB06b] NetBeans Download Seite
http://www.netbeans.org/downloads/index.htm

[OTJ06a] ObjectTeams/Java Homepage
http://www.objectteams.org

[OTJ06b] ObjectTeams/Java Languag Specification
http://www.objectteams.org/def/0.9/index.html

[RAM03] AspectJ in Action
Ramnivas Laddad
Manning Verlag
ISBN: 1-930110-93-1

[SEL03] Windows Forms Programmin in C Sharp
Chris Sells und Michael Weinhardt
Addison Wesley Verlag
ISBN: 0-321-11620-8

[STR97] C++ Programming Language
Bjarne Stroustrup
Addison Wesley Verlag
ISBN: 9-780201-889543

[ULC05] Implementing of ULC Visual Editor for Eclipse
http://www.eclipse.org/vep/WebContent/docs/VEpapers/ulc.pdf

[VEP06a] Visual Editor Project Homepage
http://www.eclipse.org/ve

[VEP06b] Extending The Visual Editor: Enabling support for a custom widget
http://eclipse.org/articles/Article-VE-Custom-Widget/customwidget.html

[VEP06c] Newsgroup für Fragen rund um den Visual Editor
news://news.eclipse.org/eclipse.tools.ve

[XUL06] XUL - XML User interface Language
http://www.mozilla.org/projects/xul/

Abbildungsverzeichnis