Unterstützung modellgetriebener Entwicklungsprozesse - Fujaba

Sven Kulle
242 913

Betreut von Dipl.-Inf. Ulrike Ranger

Zusammenfassung

Diese Seminararbeit beschreibt die visuelle Programmierung mit dem CASE-Werkzeug Fujaba. In Fujaba können Softwarearchitekturen und deren Implementierungen entworfen werden. Die Entwicklung findet zum Großteil auf visuelle Weise statt, wozu dem Entwickler unterschiedliche UML-Diagramme zur Verfügung stehen. In einem weiteren Schritt kann der Entwickler aus den spezifizierten Diagrammen mittels Fujaba Java Quellcode erzeugen. Es ist weiterhin möglich, Fujaba durch Plugins zu erweitern. Diese Arbeit stellt ein Komponenten-Plugin für Fujaba vor. Das Komponentendiagramm des Plugins stellt einzelne Komponenten, ihre Schnittstellen und die Abhängigkeiten zwischen den Schnittstellen dar.

Inhaltsverzeichnis

1 Motivation **8-3**

2 Komponenten **8-4**

2.1 Ziel der komponentenbasierten Softwareentwicklung 8-5

2.2 Anforderungen und Eigenschaften von Softwarekomponenten . 8-5

2.3 Komponentenmodelle 8-7

3 Beispielszenario **8-8**

4 Fujaba **8-9**

4.1 Klassendiagramme 8-11

4.2 Aktivitätsdiagramme 8-13

4.3 Beispiele 8-17

4.4 Weitere Diagramme und Funktionalitäten 8-19

5 Spezifikation von Komponenten **8-20**

5.1 Spezifikation und Verwendung von Komponenten 8-20

5.2 Fujaba und das JavaBean Modell 8-22

6 Zusammenfassung **8-23**

1 Motivation

Fujaba ist ein CASE-Werkzeug. “Computer-Aided-Software-Engineering“-Werkzeuge unterstützen den Softwareentwicklungsprozeß vor allem in der Entwicklungsphase *Entwurf*. Der Entwickler kann zur Spezifizierung der Softwarearchitektur und ggf. zur Spezifizierung der Implementierung auf visuelle Notationsmöglichkeiten des CASE-Werkzeugs zurückgreifen. Meist nutzen CASE-Werkzeuge UML-Diagramme zur Modellierung des Softwaresystems, einige nutzen aber auch SA/SD bzw. ERM/SERM. Einige CASE-Werkzeuge können nicht nur zur Spezifizierung des Softwaresystems genutzt werden. Sie bieten die Möglichkeit, aus der Spezifizierung des Softwaresystems Quellcode zu generieren. Eine Übersicht über gängige CASE-Werkzeuge und deren Notationen findet sich in [15].

Visuelle Programmierung, d.h. Quellcode aus visuell erstellten Diagrammen zu generieren, ist bei einfachen oder sehr spezifischen Anwendungen eine echte Alternative zur textuellen Programmierung. Die visuelle Programmierung bietet verschiedene Vorteile, z.B. abstrahiert sie von einer konkreten Programmiersprache wie C++ oder Java. Anhand der verschiedenen Diagramme eines CASE-Werkzeugs, kann sich der Entwickler relativ schnell und leicht einen Überblick über das Softwaresystem verschaffen. Weiterhin läßt sich beispielsweise UML als visuelle Programmiersprache leichter erlernen als textuelle Programmiersprachen. So sind auch nicht Fachleute in der Lage, mit Hilfe von ausdrucksstarken visuellen Modellierungssprachen Software bzw. Teile eines Softwaresystems zu verstehen und zu entwickeln.

Fujaba unterstützt den Entwickler sowohl bei dem Entwurf der Softwarearchitektur als auch bei dessen Implementierung. Es nutzt zur Modellierung der Struktur und des Verhaltens der Software UML-Diagramme. In einem weiteren Schritt kann Fujaba aus diesen Diagrammen Java Quellcode erzeugen. Es lässt sich insofern in das Seminarthema *Unterstützung modellgetriebener Entwicklungsprozesse* einordnen, da die Software nicht mit einer herkömmlichen Programmiersprache, wie z.B. Java sondern aus einem Modell (UML-Diagramme) heraus generiert wird. Die automatische Generierung der Software aus UML-Diagrammen hat nach [14] den Vorteil, daß bei fehlerfreier Quellcode-Generierung die Korrektheit der Implementierung in Hinsicht auf die Spezifikation garantiert werden kann. Fujaba kann durch Plugins [3] erweitert werden. Das *RealtimeStatechart* Plugin der *Fujaba Real-Time Tool Suite* [3] stellt ein *RealtimeStatechart*-Diagramm bereit. Die *Fujaba Tool Suite Reverse Engineering* [3] besitzt ein Plugin zur Spezifizierung von *design patterns*. Ein Plugin für die Darstellung von Softwarekomponenten (Komponentendiagramm) wird in dieser Arbeit vorgestellt.

In der heutigen Softwareentwicklung werden häufig Softwarekomponenten entwickelt und eingesetzt [7]. Softwarekomponenten sind wiederverwendbare Softwarebausteine bzw. Teile einer Software, die eine gewisse Funktionalität anbieten. Softwarekomponenten werden häufig durch UML-Komponenten-Diagramme visuell dargestellt. Weiterhin gibt es visuelle Entwicklungswerkzeuge, wie den *Bean Builder* [9], die die Entwicklung von Softwaresysteme durch Kombination und Verbindung von Instanzen von Softwarekomponenten ermöglichen. Die Entwicklung findet ausschließlich visuell statt. Soweit Softwarekomponenten die Angabe von Eigenschaftsparametern erlauben, kann die Funktionalität bzw. das Verhalten der Komponente verfeinert und den Wünschen des Nutzers angepasst werden. Benutzer eines Softwaresystems können durch die Eigenschaftsparameter einer Schaltflächen-Softwarekomponente beispielsweise die Farbe der Schaltfläche ihren Vorstellungen anpassen. Ein Beispiel zur Erstellung eines komplexen Softwaresystems mittels Softwarekomponenten findet sich in [1]. In diesem Beispiel wird eine Softwarekomponente entwickelt, die die Ergebnisse eines Meßinstrumentes auf dem PC visuell darstellt. Die Softwarekomponente in diesem Beispiel besteht ausschließlich aus der visuellen Repräsentation der gemessen Daten, d.h. der eigentliche Meßvorgang findet nicht in dieser Softwarekomponente statt. Softwarekomponenten abstrahieren noch einmal von Programmiersprachen im üblichen Sinn, da der Zugriff auf die gekapselten Funktionen nur über eine Schnittstelle möglich ist. Sie wirken sich vereinfachend auf den Entwicklungsprozeß aus.

Diese Seminararbeit beschreibt die Erstellung eines Softwaresystems mit dem CASE-Werkzeug Fujaba. In der Diplomarbeit [14] wurde ein Plugin zur Spezifizierung von Softwarekomponenten entwickelt, welches in 5.1 vorgestellt wird.

Die Arbeit gliedert sich wie folgt: Kapitel 2 ist eine Einführung in Softwarekomponenten. Kapitel 3 beschreibt das Beispielszenario dieser Arbeit. Es handelt sich hierbei um ein Mancala-Spiel. Kapitel 4 stellt die Entwicklung einer Softwarearchitektur und deren Implementierung mit dem CASE-Werkzeug Fujaba vor. Beispiele und Erläuterungen dieses Kapitels beziehen sich auf das Beispielszenario. Das Kapitel 5 beschäftigt sich mit der Spezifizierung von Softwarekomponenten in Fujaba. In Kapitel 6 werden die wichtigsten Aspekte der Seminararbeit zusammengefasst.

2 Komponenten

Dieses Kapitel erläutert den Begriff der Softwarekomponenten. Es werden verschiedene Definitionen für Softwarekomponenten vorgestellt, sowie als Beispiel das JavaBeans-Komponentenmodell erläutert.

2.1 Ziel der komponentenbasierten Softwareentwicklung

[1] vergleicht Softwarekomponenten mit Halbfabrikaten aus der industriellen Produktion. Dort ist es üblich, Halbfabrikate bzw. Zwischenprodukte von verschiedenen Herstellern zu beziehen und in einem eigenen Produktionsschritt zusammen zu setzen. Dies kann aus mehreren Gründen von Vorteil sein. Beispielsweise kann die eigene Entwicklung und Produktion eines Halbfabrikats extrem aufwendig und kostenintensiv sein, so daß sich der Einkauf von einem spezialisierten Hersteller lohnt.

Übertragen auf die Softwareentwicklung bedeutet dies, daß Softwaresysteme mit Hilfe von vorgefertigten Softwarebausteinen bzw. Softwarekomponenten entwickelt werden können. Die Komponenten können wie Halbfabrikate von spezialisierten Softwareunternehmen eingekauft und genutzt werden. Für die Softwareentwicklung bringt dies eine erhebliche Verbesserung in Bezug auf die *Entwicklungszeit* und *-kosten*, sowie die *Wiederverwendung* von Softwareteilen. Die Funktionalität von eingekauften oder vorhandenen Komponenten muß nicht neu entworfen und entwickelt werden. Komponenten lösen ein spezielles Problem. Existiert eine Lösung, so muß für das selbe Problem in einem anderen Softwaresystem die Lösung nicht nochmal gefunden werden. Der Entwickler kann auf die bereits vorhandene Komponente zurückgreifen. Ein weiterer Vorteil der Softwarekomponenten liegt in deren Austauschbarkeit. Nutzt ein Softwareentwickler eine bestimmte Komponente, so kann er diese durch eine Komponente eines anderen Herstellers austauschen, sofern diese die selbe Grundfunktionalität und Schnittstelle bietet. Die leichte Austauschbarkeit von Komponenten wird durch Schnittstellen erreicht. Schnittstellen abstrahieren von der Realisierung einer Komponente und definieren eindeutig den Zugriff auf die Funktionalität der Komponente.

2.2 Anforderungen und Eigenschaften von Softwarekomponenten

Obwohl das Ziel der komponentenbasierten Softwareentwicklung eindeutig ist, existiert für den Begriff der Softwarekomponenten keine eindeutige Definition. [1] versteht unter einer Komponente folgendes:

> „[...] eine Komponente (component) ist ein abgeschlossener, binärer Software-Baustein, der eine anwendungsorientierte, semantische zusammengehörende Funktionalität besitzt, die nach außen über Schnittstellen zur Verfügung gestellt wird. Beim Entwurf [...] wurde auf hohe Wiederverwendung Wert gelegt."

Im Vergleich dazu definiert [13] Softwarekomponenten folgendermaßen:

> „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“

Es lassen sich noch eine Reihe weiterer Definitionen für Softwarekomponenten finden. Es ist festzuhalten, daß die Definitionen für Softwarekomponenten eher allgemeinen Charakter haben. Der interne Aufbau von Komponenten ist nicht festgelegt. Die wesentlichen Aspekte der Definitionen können wie folgt zusammengefasst werden: Softwarekomponenten stellen kein Softwaresystem für sich dar, sie sind viel mehr als Teilsystem zu betrachten. Komponenten bilden Lösungen für spezielle Probleme und stellen wesentlich mehr Funktionalität zur Verfügung als eine Klasse aus der OO-Programmierung. In komplexen Softwaresystemen müssen häufig Probleme gelöst werden, für die bereits Lösungen in Form von unterschiedlichen Softwarekomponenten existieren.

Weiterhin ist die Kombination d.h. Komposition von Komponenten möglich. Komponenten werden i.d.R. nicht alleine genutzt, sondern in einer Komponentenumgebung betrieben. Das heißt, daß Softwaresysteme nicht nur eine einzige Softwarekomponente benutzen, sondern meist mehrere Softwarekomponenten. Softwarekomponenten können untereinander Abhängigkeiten besitzen. Beispielsweise kann zur Ausführung einer Softwarekomponente A eine Softwarekomponente B benötigt werden. Der Schnittstellenbegriff ist ein wichtiger Aspekt in Bezug auf Komponenten. Sämtliche Funktionalität einer Komponente wird über eine Exportschnittstelle gekapselt und anderen Komponenten zur Verfügung gestellt. Ein Aufruf einer Methode, die nicht in der Exportschnittstelle der Komponente aufgeführt wird, ist nicht möglich. Auch das Erweitern einer Komponente ist nicht möglich. Komponenten sind als *Blackbox* aufzufassen, deren Implementierung dem Benutzer verborgen bleibt.

Der Aspekt der Verteilung, also eine verteilte Anwendung die auf mehreren Rechnern oder Prozessoren läuft, ist auch bei dem Entwurf und der Nutzung von Komponenten wichtig. Es gibt heute viele Anwendungsteile die nicht zentral auf einem Rechner laufen, sondern z.B. über das Internet verteilt auf anderen Rechnern betrieben werden. Soll eine verteilte Anwendung bestehend aus Softwarekomponente entwickelt werden, so müssen Softwarekomponenten entsprechende Mechanismen bereitstellen, so daß sie miteinander kommunizieren können. Durch die Kapselung der Funktionalität und den Begriff der Schnittstelle werden die Komponenten zu unabhängigen Softwarebausteinen, die relativ unkompliziert als Teilsystem genutzt werden können.

2.3 Komponentenmodelle

Aus dem Aspekt der Verteilung können zwei unterschiedliche Arten von Komponentenmodellen abgeleitet werden. Zum einen gibt es Komponentenmodelle, die Konzepte wie Persistenz, Nebenläufigkeit und Verteilung umsetzen und dem Entwickler entsprechende Unterstützung bieten. Komponentenmodelle die sich für die genannten Aspekte anbieten sind:

- COM+ von Microsoft [5]
- Enterprise JavaBeans von Sun [10]
- CORBA-Modell der OMG [6]

Nutzt ein Entwickler solch ein Komponentenmodell und entwickelt eine verteilte Software, so kann er sich rein auf die Anwendungslogik konzentrieren und braucht nicht selbst die genannten Konzepte zu implementieren.

Zum anderen gibt es Komponentenmodelle, die die oben genannten Konzepte nicht umsetzen. Diese Modelle sind von ihrem Aufbau her einfacher und eignen sich für den Entwurf von Software, die nur auf einem Rechner betrieben wird. Es bieten sich folgende Komponentenmodelle an:

- JavaBeans von Sun [11]
- COM/ActiveX von Microsoft [5]

Als Beispiel wird hier das JavaBean-Modell als ein einfaches Komponentenmodell vorgestellt. Sun definiert JavaBeans wie folgt:

> „A Java Bean is a reusable software component that can be manipulated visually in a builder tool.“ [12]

JavaBeans sind wiederverwendbare Softwarebausteine, die in einem Werkzeug visuell miteinander kombiniert werden können, um Softwaresysteme zu entwerfen. Damit das JavaBean-Modell die obige Definition optimal unterstützen kann, müssen einige Regeln beachtet werden. So wird z.B. die Exportschnittstelle der JavaBeans, die über die Funktionalität und die Eigenschaften sowie Ereignisse der JavaBean Auskunft gibt, lediglich durch Namenskonventionen erstellt. Auf der Homepage [11] kann nachgelesen werden, inwiefern die Namenskonvention beachtet werden muß. Als Beispiel sei hier erwähnt, daß für private Attribute

einer Klasse eine *get*- und *set*-Methode implementiert werden muß. Durch Einhalten der Namenskonventionen können Entwicklungswerkzeuge Auskunft über die JavaBean erhalten. Dazu durchsuchen sie die Java Klassen der Softwarekomponente nach Methoden und Attributen. Öffentliche Methoden werden beispielsweise durch das Entwicklungswerkzeug erkannt und dem Benutzer der Softwarekomponente zur Verfügung gestellt. Nicht öffentliche Methoden bleiben dem Benutzer hingegen verborgen. Andere Komponentenmodelle nutzen hierfür eine Schnittstellenbeschreibungssprache (*interface definition language*) und erzeugen eine Schnittstellendatei, über die Entwicklungswerkzeuge und Entwickler an die gewünschten Informationen gelangen. Die Fähigkeit der JavaBeans Informationen nach Außen zu geben wird als *Introspektion* bezeichnet.

Neben der Introspektion bieten JavaBeans auch die Möglichkeit an, nachträgliche Anpassungen vorzunehmen. Beispielsweise können einfache Eigenschaften wie Balkenfarbe oder Beschriftung einer Schaltfläche in einem gewissen Rahmen vom Nutzer der Softwarekomponente geändert werden. Neben diesen einfachen Eigenschaften unterstützen JavaBeans noch indizierte Eigenschaften, gebundene Eigenschaften und Eigenschaften mit Nebenbedingungen. Gebundene Eigenschaften werden genutzt, um mit Hilfe des *Observer-Patterns* externen Beobachtern Änderungen von Attributwerten mitzuteilen. Am Beispiel des Meßinstrumentes aus Kapitel 1 bedeutet dies, daß die Softwarekomponente zur visuellen Repräsentation von Meßdaten sich als Beobachter (*Observer*) bei dem Meßinstrument anmelden kann. Sobald das Meßinstrument neue Messungen vornimmt kann es die Ergebnisse seinen *Observern* mitteilen. Die *Observer* werden so automatisch über Neuerungen informiert. Bei Eigenschaften mit Nebenbedingung können die *Observer* ein Veto einlegen und so die Änderung der Eigenschaft verhindern. Indizierte Eigenschaften sind zusammengesetzte Datentypen, wie z.B. *Listen* oder *Arrays*. In solchen Datenstrukturen werden mehrere Werte in einer Eigenschaft gespeichert. Der Zugriff auf einzelne Daten erfolgt über einen Index.

3 Beispielszenario

Das Beispielszenario dieser Arbeit basiert auf einem Fujaba Tutorial, das auf der Internetseite [2] zur Verfügung steht. Das Tutorial beschreibt die Entwicklung eines Mancala Spiels mit Fujaba. Bei dem Spiel handelt es sich um ein einfaches Brettspiel, das von zwei Spielern gespielt wird. Die genauen Regeln und der Aufbau des Spiels können aus dem Tutorial entnommen werden. Die Entwicklung des Spiels mit Fujaba ist auf Grund des einfachen Spielprinzips leicht.

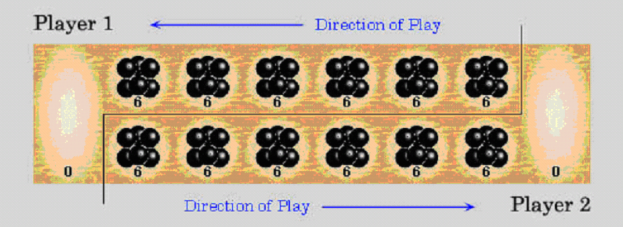

Abbildung 1: Spielaufbau

Das Spiel läßt sich wie folgt beschreiben: Jeder der beiden Spieler hat vor sich sechs normale Spielsteinbehälter stehen. Rechts von diesen sechs Behältern befindet sich der siebte Spielsteinbehälter, es ist der sog. Kalaha Behälter. In jedem der sechs normalen Behälter befinden sich zu Beginn des Spiels jeweils sechs Spielsteine. Der Spieler, der den ersten Zug macht, wählt einen beliebigen eigenen Spielsteinbehälter, außer dem Kalaha Behälter, und nimmt alle Spielsteine heraus. Danach verteilt er sie jeweils einzeln der Reihe nach rechts auf die anderen Behälter inklusive seines Kalaha Behälters, jedoch nicht in den Kalaha Behälter des Gegners. Wird der letzte Spielstein in einen seiner eigenen Behälter gelegt, so ist der Spieler nochmal an der Reihe. Ist der Behälter zusätzlich leer, so wird der Spielstein aus diesem Behälter und alle Spielsteine aus dem gegenüberliegenden Behälter des Spielgegners in den eigenen Kalaha Behälter gelegt. Das Spiel endet, wenn ein Spieler keine Spielsteine mehr in seinen Behältern, außer dem Kalaha Behälter, hat und somit keinen Spielzug machen kann. Sieger ist der Spieler, der am meisten Spielsteine in seinem Kalaha Behälter hat.

4 Fujaba

Das Akronym Fujaba steht für „**F**rom **U**ml to **J**ava **A**nd **B**ack **A**gain“. Fujaba ist ein CASE-Werkzeug das Entwickler während des Softwareentwicklungsprozesses unterstützt. Der Entwickler kann u.a. die Softwarearchitektur und deren Implementierung mit UML Klassendiagrammen und den Fujaba Aktivitätsdiagrammen spezifizieren, die eine Kombination aus UML Kollaborations- und Aktivitätsdiagrammen darstellen. Fujaba kann anhand dieser Spezifikationen Java Quellcode generieren.

Fujaba wird von der „*Fujaba Development Group*“ der Universität Paderborn in Zusammenarbeit mit anderen Universitäten entwickelt. Nach [4] wurde die ursprüngliche Implementierung 1997 von drei Studenten in einer gemeinsamen Diplomarbeit begonnen. Fujaba ist in der Programmiersprache Java geschrieben und

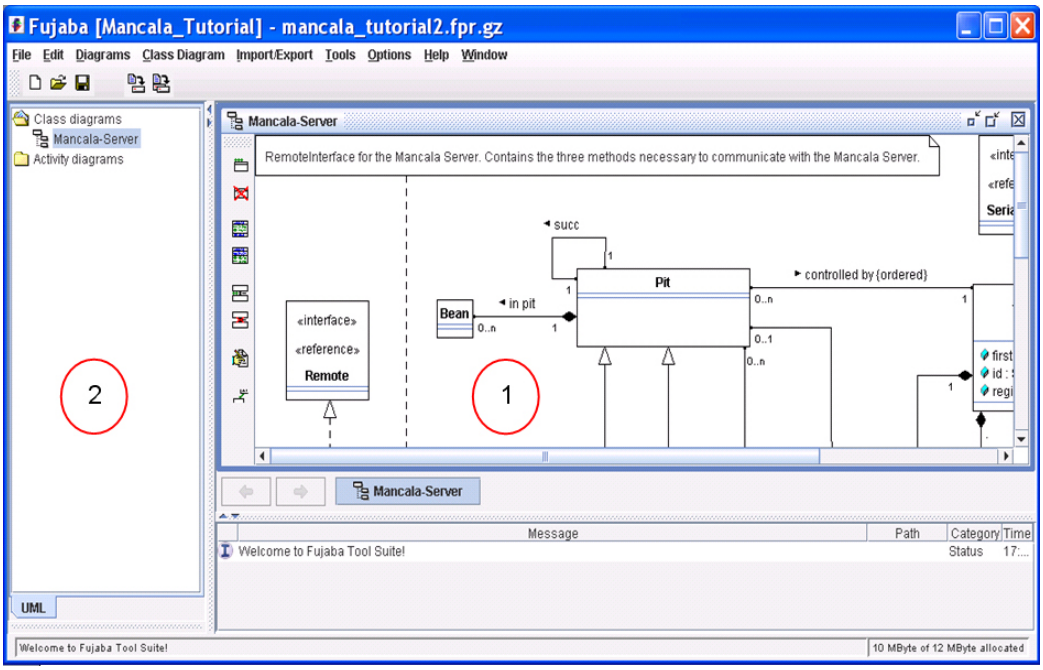

Abbildung 2: Fujaba - Oberfläche

somit plattformunabhängig. Auf [3] steht Fujaba zum Download bereit. Außerdem findet sich dort weitere Literatur zu Fujaba.

Fujaba basiert intern auf Graphtransformationssystemen. Das Datenmodell, bestehend aus Klassen und Assoziationen, stellt ein Typsystem eines Graphen dar. Fujaba kann auf diesen Graphen (Instanz des Modells) Transformationen ausführen, um Änderungen an der Objektstruktur vorzunehmen. Durch Graphtransformationen können so auch Objekte zur Laufzeit erzeugt oder gelöscht werden. Fujaba nutzt Graphen, da diese sich besonders gut zur Darstellung der Objektstruktur eigenen. Durch die eindeutige Semantik lassen sich Änderungen gut modellieren. Auch die Quellcode-Generierung wird durch die eindeutige Graphsemantik ermöglicht. Weiterhin wird daran gearbeitet, daß Fujaba UML Diagramme aus Java Quellcode generieren kann [16].

Die Oberfläche von Fujaba sieht wie in Abbildung 2 aus. Im oberen Bereich befindet sich eine Menü- und Symbolleiste, über die alle Funktionen von Fujaba erreicht werden. ① zeigt die Editorfläche, in der alle Eingaben zum Erstellen von Diagrammen vorgenommen werden. Die Bedienung von Fujaba findet zum großen Teil auf visuelle Weise statt. Auf der linken Seite der Editorfläche werden Verknüpfungen zu den meist genutzten Funktionen dargestellt. Diese sind jeweils von der gewählten Diagrammart abhängig. ② stellt die vom Entwickler erstell-

ten Diagramme in einer Übersicht dar. Der Entwickler kann sich so schnell einen Überblick über seine erstellten Diagramme verschaffen. Wählt er ein Diagramm in dieser Übersicht aus, so wird dieses auf der Editorfläche dargestellt.

In den nächsten beiden Abschnitten werden die wichtigsten Merkmale von Klassen- und Aktivitätsdiagrammen und deren Erstellung mit Fujaba erläutert.

4.1 Klassendiagramme

Mit Fujaba Klassendiagrammen, wie in Abbildung 3, wird die statische Struktur, d.h. die Architektur eines Softwaresystems dargestellt. Fujaba Klassendiagramme entsprechen den UML Klassendiagrammen. Klassendiagramme bestehen aus *Klassen* und *Assoziationen*. Klassen werden, wie in der UML üblich, durch Rechtecke dargestellt. Sie sind in drei Abschnitte unterteilt. Im oberen Abschnitt steht der Klassenname und die Stereotypen. Im mittleren Abschnitt werden die Klassenattribute festgelegt. Im unteren Abschnitt werden die Methoden der Klasse aufgelistet. Die Sichtbarkeiten der Klassenattribute und Methoden werden durch andere Symbole als in der UML üblich dargestellt.

Entwickler können zur Definition von Attributen nicht nur deren Datentyp und Sichtbarkeiten festlegen, sondern auch mit welchem Anfangswert diese initialisiert werden. Per *checkbox* kann der Entwickler auswählen, ob bei einer späteren Codegenerierung *get*- bzw. *set*-Methoden für das Attribut automatisch angelegt werden sollen. Weiterhin kann ein Attribut als *final*, *static* und *transient* markiert werden. *Final* Attribute sind Wertkonstanten und können zur Laufzeit nicht geändert werden. Attribute, die als *static* markiert sind, sind für alle Objekte dieser Klasse gleich. *Transient* bedeutet, daß diese Attribute beim serialisieren nicht beachtet werden. Sie werden also nicht persistent gespeichert.

Zur Spezifizierung der Methodensignatur kann der Entwickler neben der Sichtbarkeit und dem Rückgabetyp auch eine Liste von Parametern angeben. Methoden können ebenfalls mit einer zusätzlichen Information, wie z.B. *final*, versehen werden. Auch hier ist die Bedeutung dieser zusätzlichen Informationen wie bei den Attributen zu verstehen.

Beispielhaft ist die Klasse zur Modellierung des Spielfeldes in Abbildung 3 dargestellt. In dieser Klasse sind die beiden Attribute *beginnerNr* und *turn* angelegt. Das Attribut *beginnerNr* gibt den Spieler an, der den ersten Spielzug machen darf. *turn* zählt die gespielten Runden. Beide Attribute sind vom Typ *Integer* und besitzen die Sichtbarkeit *public*.

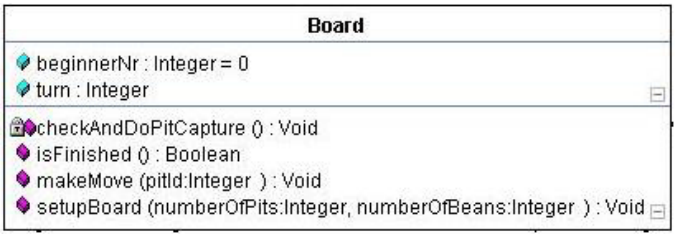

Abbildung 3: Darstellung der Klasse Board

Zur Prüfung, ob der letzte Spielstein in einen leeren Behälter gelegt wurde und somit alle Spielsteine des entsprechenden Behälters des Spielgegners eingenommen werden dürfen, dient die Methode *checkAndDoPitCapture()*. Die Methode *isFinished()* prüft, ob das Spiel beendet ist. Der Rückgabewert der Methode ist ein *Boolean* Wert. *checkAndDoPitCapture* ist eine Hilfsmethode, sie ist daher mit der Sichtbarkeit *private* versehen. *makeMove(pitId:Integer)* führt einen Spielzug aus, dazu wird ihr als Parameter der gewählte Spielsteinbehälter übergeben. *setupBoard(numberOfPits:Integer, numberOfBeans:Integer)* initialisiert das Spielfeld. Die Anzahl der Spielsteine pro Spielsteinbehälter und die Anzahl der Spielsteinbehälter werden als Parameter übergeben. Alle Methoden bis auf *checkAndDoPitCapture* besitzen die Sichtbarkeit *public*.

Neben der Generalisierung bzw. Spezialisierung einer Klasse kann zwischen zwei Klassen auch eine Assoziation bestehen. In Abbildung 4 existiert eine Generalisierung zwischen der Klasse *PlayerPit* und der Klasse*Pit*. Allgemein beschreibt eine Assoziation eine Verbindung zwischen zwei Klassen. Diese kann unterschiedliche Ausprägungen haben. Es gibt einfache (gerichtete/ungerichtete) Assoziationen sowie die aus der UML bekannte Aggregation und Komposition. Die *Aggregation* und *Komposition* drücken eine *Teile-Ganzes*-Beziehung aus. Bei der Aggregation existieren die *Teile* auch unabhängig vom *Ganzen*. Die Komposition hingegen bedeutet, das die *Teile* nur solange existieren, wie auch das *Ganze* existiert. Nach [4] setzt Fujaba allerdings Kompositionen bei der Codegenerierung als einfache Assoziation um.

Assoziationen können genauer spezifiziert werden, u.a. mit Kardinalitäten, Rollennamen und Bedingungen. Bedingungen drücken eine bestimmte Ordnung auf assoziierten Instanzen aus. Die Bedingung *ordered* drückt z.B. aus, daß es ein erstes und ein letztes Element gibt. Bezüglich der Kardinalitäten von Assoziationen sagt [4] weiter: „In Fujaba findet keine Überprüfung statt, ob die Kardinalitäten zur Laufzeit verletzt werden, das heißt mehr Instanzen assoziiert werden als in der Kardinalität angegeben.
In Abbildung 4 existiert eine Kompositionsbeziehung zwischen einem Spielstein

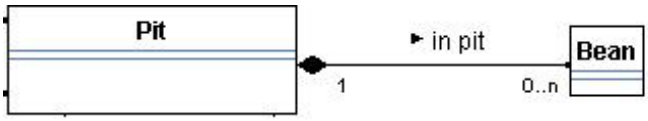

Abbildung 4: in *pit*-Kompositionsbeziehung zwischen *Pit*-Klasse und *Bean*-Klasse

(*Bean*) und einem Behälter für Spielsteine (*pit*). Die Kompositionsbeziehung ist an der ausgefüllten Raute der Verbindung zu erkennen. An den jeweiligen Enden der Assoziation sind Kardinalitäten angegeben. Die Assoziation drückt also die Beziehung zwischen Spielsteinbehältern und Spielsteinen aus. Einem Spielsteinbehälter können beliebig viele (0..n) Spielsteine zugeordnet werden. Die Komposition drückt weiterhin aus, daß durch löschen eines *Pit*-Objektes alle assoziierten *Bean*-Objekte auch gelöscht werden.

Das komplette Klassendiagramm des Beispielszenarios enthält neben einigen weiteren Kompositionen einfache Assoziationen, Implementierungen (Einbindung von *interfaces*) und die Generalisierungen. Das *interface MancalaServerRemote* besitzt die beiden Stereotypen *«interface»* und *«RemoteInterface»* sowie einen Kommentar. Stereotypen beschreiben den Verwendungszweck der Klasse. *«interface»* drückt beispielsweise aus, das es sich bei dieser Klasse um ein Interface handelt. *Static* Methoden werden in Fujaba durch einen unterstrichenen Methodennamen dargestellt.

4.2 Aktivitätsdiagramme

Klassendiagramme dienen zur Spezifizierung der statischen Struktur eines Softwaresystems. Methoden werden in diesem Diagramm nur durch ihre Signatur definiert. Eine genaue Spezifizierung findet mit den Fujaba Aktivitätsdiagrammen statt. Sie sind nach [4] eine Mischung aus UML Aktivitätsdiagrammen und UML Kollaborationsdiagrammen und beschreiben die Dynamik eines Softwaresystems.

4.2.1 Kontrollfluß von Methoden

Zur Beschreibung des Kontrollflusses einer Methode stehen dem Entwickler sog. Aktivitäten zur Verfügung, die er durch Transitionen verbindet. Aktivitäten können z.B. *story patterns*, vgl. Abschnitt 4.2.2 oder *statement* Aktivitäten sein. Während *story patterns* Transformationen auf dem Datenmodell visuell modellieren, wird in *statement* Aktivitäten expliziter Java Quellcode festgehalten. Der Kon-

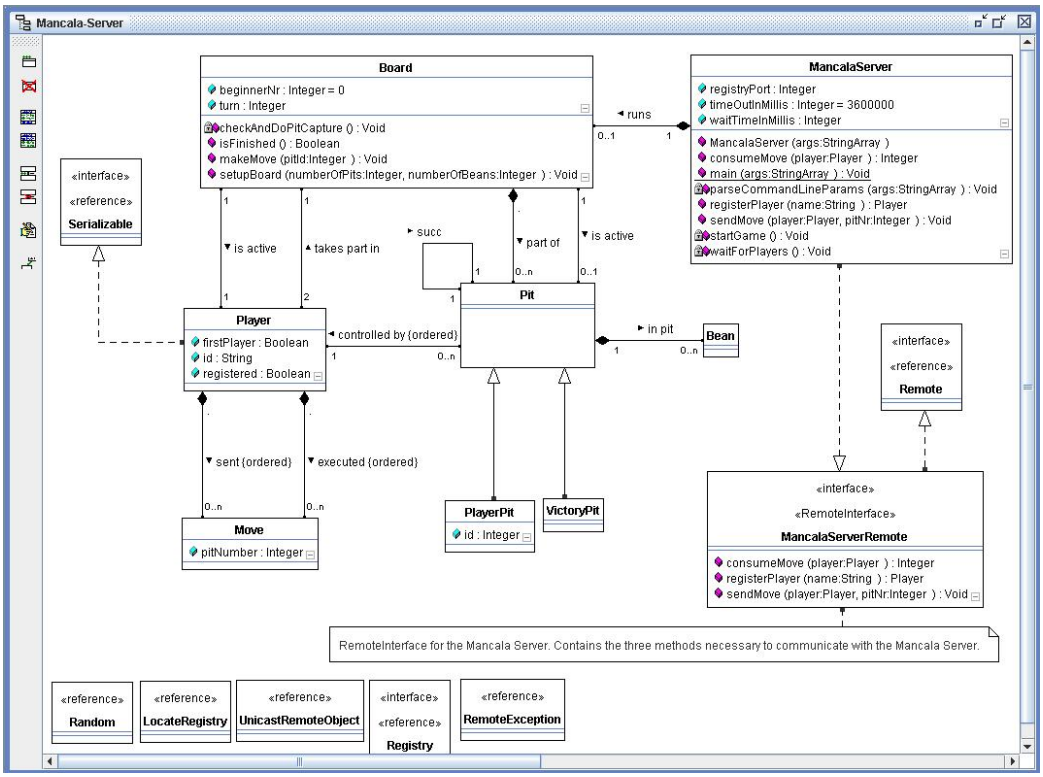

Abbildung 5: Klassendiagramm des Beispielszenarios

trollfluß beginnt in einem eindeutig festgelegten Startknoten und endet in einem Endknoten. Endknoten können beliebig oft vorkommen. In Fujaba können Transitionen mit Bedingungen versehen werden. Die folgende Liste gibt einen Überblick:

- *none*: Standardfall, eine Transition ohne Bedingung.
- *success*: Drückt aus, daß die Aktivität über diese Transition verlassen wird, wenn sie gültig ausgeführt wurde.
- *failure*: Der Kontrollfluß läuft über diese Transition, falls die Aktivität nicht gültig ausgeführt werden konnte.
- *each time*: Mit dieser Bedingung werden Schleifen modelliert. Zu jedem gefundenen Matching einer *for each* Aktivität werden bestimmte Operationen ausgeeführt, die über die ausgehende *each time* Transition spezifiziert sind. Fujaba geht dabei iterativ vor, d.h. es wird ein Matching bestimmt, und für dieses die Operationen ausgeführt. Im Anschluß daran wird das nächste

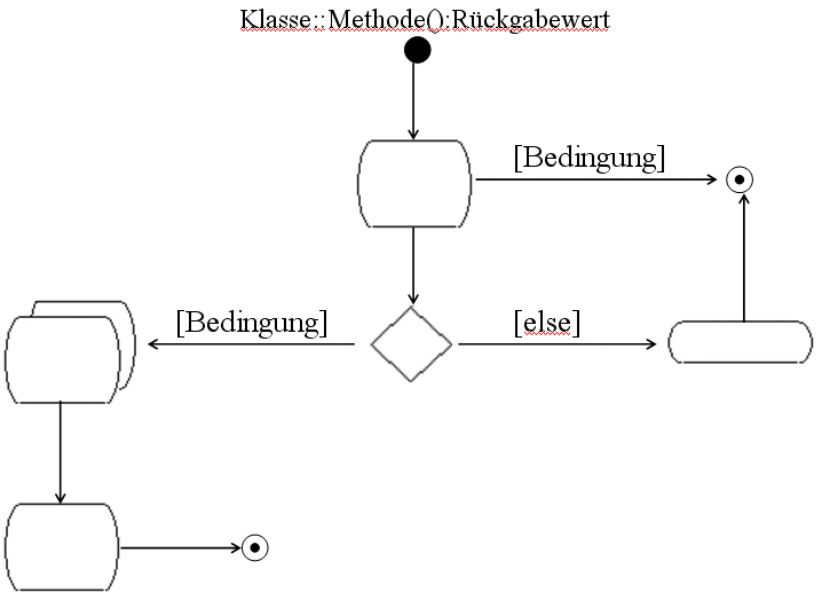

Abbildung 6: Beispiel eines Aktivitätsdiagramms

Matching gefunden und bearbeitet. Dies erfolgt solange, bis kein Matching mehr gefunden werden kann.

- *end (for all)*: Nach allen Iterationen einer *for each* Aktivität verläßt der Kontrollfluß diese über die *end (for all)* Transistion.
- *else*: Der Kontrollfluß läuft über diese Transition, falls keine von der gleichen Aktivität ausgehende Transition, die mit einer logischen Bedingung versehen ist, erfüllt werden kann.
- *boolean expression*: Der Kontrollfluß läuft nur über diese Transition, falls die Boolean Bedingung eingehalten werden kann.

Die Abbildung 6 zeigt ein mögliches Aktivitätsdiagramm. In dem Diagramm existieren neben dem Startknoten zwei Endknoten sowie die verschiedenen Aktivitäten. Die Raute kann für Verzweigungen innerhalb des Diagramms genutzt werden.

4.2.2 Story Patterns

Story patterns stellen Änderungen an der Objektstruktur dar. Damit es erfolgreich ausgeführt wird, muß zunächst eine Belegung auf der Objektstruktur ge-

sucht werden, die das *story pattern* erfüllt. Die modellierten Änderungen müssen dann auf der gefunden Objektstruktur ausgeführt werden können. Fujaba unterscheidet zwischen *gebundenen* und *ungebundenen* Objekten. Gebundene Objekte referenzieren ein eindeutiges Objekt im Laufzeitgraphen, das z.B. durch Parameterübergabe bekannt ist. Häufig wird das *this* Objekt benutzt. Es stellt die Referenz auf die aufgerufene Instanz dar und ist immer verfügbar. Ungebundene Objekte werden lediglich durch einen Namen und einen Datentyp angegeben. Sie werden gebunden, sobald eine entsprechende Objektbelegung im Laufzeitgraphen gefunden wurde.

Objekte der Aktivität werden durch Rechtecke symbolisiert, die horizontal geteilt sind. In der oberen Hälfte wird bei ungebundenen Objekten der Name und der Typ des Objektes vermerkt. Bei gebundenen Objekten wird nur der Name vermerkt. In der unteren Hälfte werden Attributänderungen bzw. -anforderungen notiert. Objekte werden durch Kanten (*links*) miteinander verbunden. *Links* symbolisieren die Bedingung, daß zwei Objekte innerhalb der Objektstruktur existieren und miteinander assoziiert sind.

Objekte der Aktivität können durch *Modifier* und *Constraints* näher beschrieben werden. *Modifier* geben an, ob ein Objekt neu erstellt bzw. ein vorhandenes Objekt gelöscht werden soll. *Constraints* sind Bedingungen an Objekte. Sie lassen sich laut Fujaba-Dokumentation [8] wie folgt beschreiben:

- *None*: Fujaba überprüft das modellierte Objekt des *story patterns* und alle Objekte, die über einen *link* mit diesem in Verbindung stehen auf Durchführbarkeit. Tritt dabei ein Fehler auf, wird das *story pattern* über eine *failure*-Transition verlassen. Tritt kein Fehler auf, so wird das *story pattern* über eine *success*-Transition verlassen.

- *Optional*: Optionale *links* bzw. optionale Knoten müssen während der Ausführung des *story patterns* nicht existieren. So kann z.B. sehr einfach modelliert werden: Falls ein *link* existiert, so soll dieser gelöscht werden.

- *Negative*: In Fujaba können Modellierungselemente auch negativ verwendet werden. Wenn ein *link* als negativ gekennzeichnet ist, überprüft Fujaba, ob zwischen den beiden angrenzenden Objekten kein *link* des entsprechenden Typs besteht. Falls der *link* doch bestehen sollte, wird das *story pattern* über die *failure* Transition verlassen. Auch Knoten können als negativ markiert werden, so daß diese bei der Ausführung des *story patterns* nicht existieren dürfen. Abbildung 7 zeigt dazu ein Ausschnitt aus einem Aktivitätsdiagramm des Beispielszenarios. Hier ist das gebundene Player Objekt *aktivePlayer* über die negative *controlled by* Kante mit dem noch ungebundenen

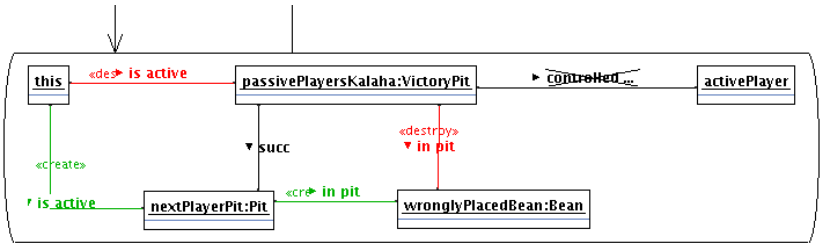

Abbildung 7: Beispiel eines negativen links

VictoryPit Objekt *passivePlayersKalaha* verbunden. Dieses Muster drückt aus, das ein Siegerbehälter gesucht wird, der nicht dem aktiven Spieler gehört. Es ist also der Siegerbehälter des Gegners gemeint. In einem Matching des Musters existiert somit keine Verbindung zwischen dem aktiven Spieler und dem gefundenen Siegerbehälter.

- *Set*: Bei dem Objekt der Aktivität handelt es sich um eine Menge von Objekten. Auf alle Objekte der Menge werden die in dem *story pattern* modellierten Änderungen ausgeführt.

4.3 Beispiele

Es folgt ein Beispiel, wie Fujaba Aktivitätsdiagrammen genutzt werden können, um auf einer Objektstruktur Objekte und deren Beziehungen zueinander zu suchen bzw. zu prüfen. Dazu wird die Methode *isFinished():Boolean* der Klasse *Board*, wie in Abbdilung 8 zu sehen implementiert. *isFinished* prüft, ob das Spiel beendet ist. Änderungen an der Objektstruktur sind dazu nicht erforderlich. Das Spiel ist beendet, falls der aktive Spieler keine Spielsteine mehr besitzt. Die Prüfung findet daher folgendermaßen statt: Der Kontrollfluß der Methode beginnt im Startknoten, der über eine Transition mit einem *story pattern* verbunden ist. Das *story pattern* sucht eine Konfiguration auf der Objektstruktur, die eindeutig belegen kann, ob das Spiel beendet ist. Der Ablauf des *story patterns* gliedert sich in drei Schritte.

Zuerst wird der aktive Spieler gesucht und gebunden. Dies geschieht durch die „*is active*“ Kante zwischen dem gebunden *this* Objekt und dem ungebunden Spieler-Objekt *activePlayer*. Da die entsprechende *is active* Assoziation im Klassendiagramm zwischen der *Board* Klasse und der *Player* Klasse eine 1:1 Beziehung ist, kann Fujaba eindeutig entscheiden, welcher der beiden Spieler der aktive Spieler ist.

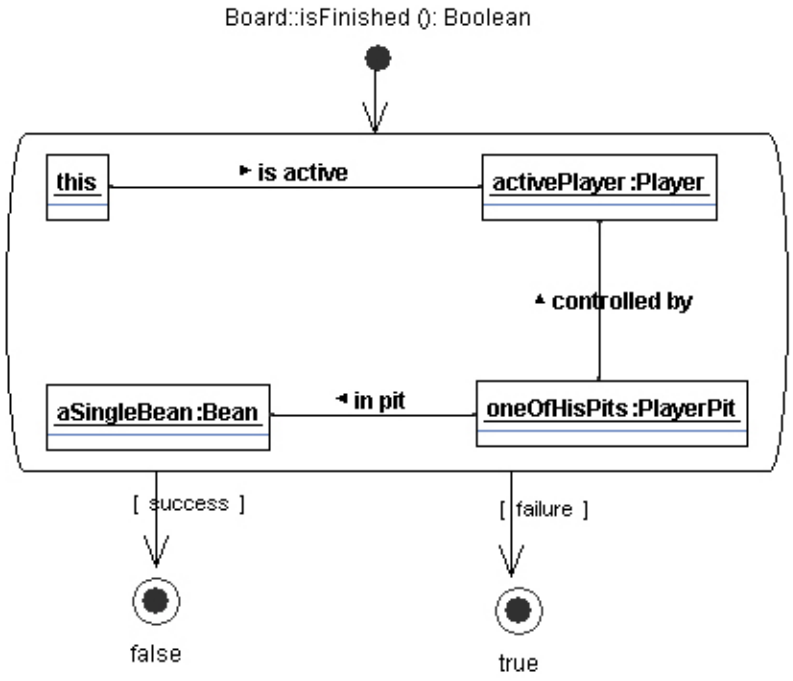

Abbildung 8: Aktivitätsdiagramm der Methode isFinished()

Im zweiten Schritt werden alle Spielsteinbehälter (*oneOfHisPits*) vom Typ *PlayerPit* des aktiven Spielers auf ihren Inhalt überprüft. Der Kalaha Behälter vom Typ *VictoryPit* ist von der Prüfung ausgeschlossen. Es werden hierfür zunächst zwei weitere Objekte gesucht: Das *oneOfHisPits* Objekt und das *aSingleBean* Objekt. Beide Objekte sind nicht gebunden. Zur Suche der *oneOfHisPits* wird die *controlled by* Kante zwischen *activePlayer* und *oneOfHisPits* genutzt (vgl. Klassendiagramm). Im ersten Schritt wurde bereits der aktive Spieler gebunden, so daß die Kante zwischen dem *activePlayer* Objekt und dem *oneOfHisPits* Objekt sich auf den aktiven Spieler bezieht. Das *oneOfHisPits* Objekt ist ungebunden, d.h. es wird ein beliebiges *OneOfHisPit* Objekte, das dem aktiven Spieler gehört, gebunden. Fujaba arbeitet bei der Suche nach Objekten nichtdeterministisch, d.h. der Entwickler kann nicht vorhersagen welches Objekt gefunden und gebunden wird. Für das nun gebundene Objekt wird durch die *in pit* Kante ein *Bean* Objekt gesucht, so daß *aSingleBean* gebunden werden kann.

Der dritte und letzte Schritt ist der wichtigste, denn er bestimmt, ob das Spiel beendet ist. Wird im zweiten Schritt zu keinem *oneOfHisPits* Objekt ein *Bean* gefunden, so kann das *story pattern* nicht erfüllt werden. Es gibt also keine Objektbelegung, die das *story pattern* erfüllt. Das bedeutet mit anderen Worten, daß der aktive Spieler keinen Spielsteinbehälter hat, in dem sich ein Spielstein befin-

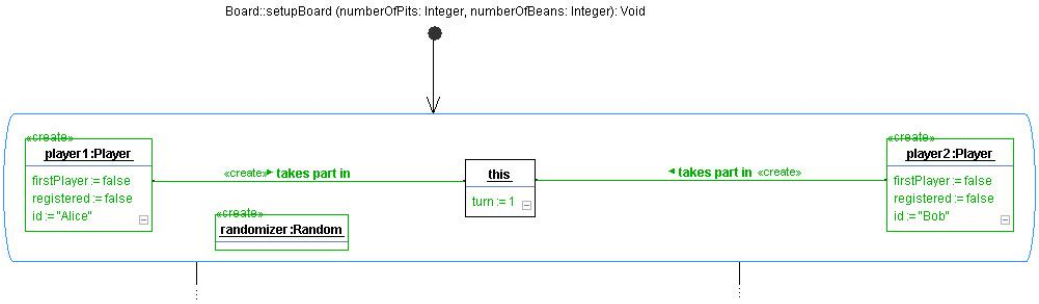

Abbildung 9: Ausschnitt aus setupBoard: Initialisierung des Spielfeldes

det. Das Spiel ist somit beendet. Der Kontrollfluß verläßt das *story pattern* über die *failure*-Transition in den *true*-Endknoten. Falls im zweiten Schritt eine gültige Objektbelegung gefunden wurde, so bedeutet dies, daß der aktive Spiele mindestens einen Spielstein in einem seiner Spielsteinbehälter (außer Kalaha) hat. Das Spiel ist noch nicht beendet. Das *story pattern* wird über die *success* Transition in den *false*-Endknoten verlassen.

Fujaba ist nicht nur in der Lage, auf Objektstrukturen zu suchen, sondern auch Objekte zu löschen bzw. zu erstellen. Zu beachten ist, daß Objekte nicht direkt gelöscht werden. In Java werden Objekte durch den *garbage collector* automatisch gelöscht. Zum Entfernen eines Objekts wird dies auf *null* gesetzt.

Die Abbildung 9 zeigt einen Ausschnitt aus der Methode *setupBoard*, die zur Initialisierung des Spielfeldes dient. Nach dem Aufruf der Methode läuft der Kontrollfluß in das dargestellte *story pattern*. In diesem *story pattern* werden die beiden Spieler Objekte *player1* und *player2* vom Typ *Player* erzeugt. Beide Objekte sind jeweils über eine *takes part in* Kante mit dem *this* Objekt assoziiert. Das *story pattern* erzeugt nicht nur die beiden Objekte, sondern auch die Assoziationen zwischen den Spieler Objekten und dem *this* Objekt. Objekte die in einem *story pattern* erzeugt werden, sind zum einen grün markiert und zum anderen mit dem Wort *create* versehen. Objekte die gelöscht werden, sind rot markiert und mit dem Wort *destroy* versehen.

4.4 Weitere Diagramme und Funktionalitäten

Neben den vorgestellten Klassen- und Aktivitätsdiagrammen kann Fujaba durch Plugins um zusätzliche Diagramme und Funktionalität erweitert werden. Plugins können über einen Fujaba Menüpunkt ausgesucht und installiert werden. Nach erfolgreicher Installation erscheinen die zusätzlichen Diagramme im Menüpunkt

Diagrams. Auf [3] gibt es Fujabaversionen mit bereits integrierten Plugins. So bietet die Real-Time Version von Fujaba laut [3] u.a. *component diagrams*, *Real-time coordination patterns* und *Real-Time Statecharts* an.

5 Spezifikation von Komponenten

Die Erstellung von Softwaresystemen mittels Fujaba wurde im vorherigen Kapitel beschrieben. Es reichen bereits Klassendiagramme und Aktivitätsdiagramme aus, um ein Softwaresystem visuell zu entwerfen und Java Quellcode zu generieren. Es bietet sich ferner an, daß Entwickler Softwaresysteme mit Fujaba entwickeln, die sich Komponententechnologie zu Nutze machen.

Komponententechnologie bedeutet zum einen, daß Entwickler in Fujaba Komponenten spezifizieren können. Ein beliebiges Komponentenmodell kann jedoch nicht genutzt werden, da Fujaba nur Java Quellcode erzeugen kann. In Frage kommen somit nur solche Komponentenmodelle, die auf der Programmiersprache Java basieren, wie das JavaBean Modell. Es ist ein einfaches und leicht umzusetzendes Komponentenmodell.

Zum anderen ist mit Komponententechnologie folgendes gemeint: Entwickler nutzen zur Entwicklung eines Softwaresystems fertige Softwarekomponenten. Das Softwaresystem besteht aus einer Komposition von ausgewählten Softwarekomponenten (Komponentenumgebung). Es ist dabei unwichtig, ob diese vorher mit Fujaba spezifiziert oder von einem Dritt-Entwickler zur Verfügung gestellt worden sind. In einem Komponentendiagramm werden die Softwarekomponenten visuell repräsentiert. Entwickler können in diesem Diagramm Softwarekomponenten anpassen und miteinander verbinden.

5.1 Spezifikation und Verwendung von Komponenten

In diesem Abschnitt wird geprüft, inwiefern Fujaba Komponententechnologie unterstützt. Die aktuelle Version von Fujaba (Version 5) unterstützt nicht die Entwicklung von Softwaresystemen mittels Softwarekomponenten. Es existiert keine Diagrammart, in der fertige Softwarekomponenten geladen und miteinander kombiniert werden können. Erweiterte Versionen von Fujaba, beispielsweise die Fujaba Real-Time Version, bieten ebenfalls nicht die Möglichkeit an, bereits entwickelte Softwarekomponenten zu nutzen.

Komponententechnologie umfaßt jedoch nicht nur die Nutzung von Softwarekomponenten zur Entwicklung von Softwaresystemen, sondern auch die Spezifizierung von Softwarekomponenten. Es stellt sich die Frage, ob Fujaba zur Spezifizierung und Entwicklung von Softwarekomponenten genutzt werden kann.

Die Diplomarbeit [14] beschäftigt sich mit der Entwicklung und Nutzung von Softwarekomponenten mit Fujaba. Erläuterungen in dieser Arbeit werden anhand eines Beispielszenarios verdeutlicht. Bei dem Beispielszenario handelt es sich um ein komponentenbasiertes Versionierungs- und Konfigurationssystem.

Im Verlauf der Diplomarbeit wird die Softwarekomponente zum *checkin* von Quellcode (Checkin-Dienst) entworfen. Die spezifizierte Komponente wird in ein Komponentendiagramm eingebettet und die Planung des Betriebs dieser Komponente wird vorgenommen. Das Komponentendiagramm stellt Softwarekomponenten und ihre Verbindung untereinander dar. *Komponenten*, *angebotene Schnittstellen*, *benutzte Schnittstellen* und *Kanten* können genutzt werden, um ein Komponentendiagramm zu entwerfen. Planung des Betriebs der Komponente bedeutet, daß das Zusammenspiel der unterschiedlichen Komponenten, aus denen das Softwaresystem besteht, betrachtet wird. Es soll möglich sein, aus diesem Diagramm zusätzlichen Quellcode zu generieren. Dieser Quellcode ist nötig, um die einzelnen Komponenten zu verbinden.

Als Komponentenmodellen standen unter anderem Web-Services, Enterprise JavaBeans und Jini zur Auswahl. Aus Gründen, wie Fehlertoleranz etc., hat sich der Verfasser für das Jini Modell entschieden. Entscheidend ist, daß bei dem späteren Entwurf der Checkin-Dienst Komponente, diese mit Fujaba spezifiziert wurde. Zur Spezifizierung der *checkin*-Komponente wurden Klassendiagramme, Aktivitätsdiagramme und Zustandsdiagramme genutzt. Zur Einbettung der Komponente in ein Komponentendiagramm und zur Planung des Betriebs der Komponente wurde ein Plugin für Fujaba entwickelt. Die Softwarekomponente kann jedoch nicht sofort genutzt werden. Hierzu sind noch einige Schritte erforderlich. Der durch Fujaba generierte Quelltext muß mit dem BuildTool-Werkzeug [14] kompiliert und in mehrere jar-Archive aufgeteilt werden. Außerdem benötigen Jini Dienste eine XML Beschreibung der Komponente. Diese kann nicht mit Fujaba erzeugt werden, sondern muß manuell erstellt werden.

Das bereits erwähnte Plugin, zur Darstellung und Beschreibung der Konfiguration von Komponenten in Fujaba, steht leider nicht zum öffentlichen Download bereit. Auf der Website [3] lassen sich bezüglich des Plugins auch keine Hinweise finden. Die vorliegende Version des Plugins befindet sich leider in einem nicht brauchbaren Zustand. Fujaba bietet zwar das Komponentendiagramm an, dieses ist aber nicht vollständig dokumentiert. So fehlt neben der Dokumentation auch Beschriftungen von Symbolen etc. Weiterhin bietet das Diagramm nicht die Fä-

higkeit, fertige Softwarekomponenten zu Nutzen und damit ein Softwaresystem zu entwickeln.

Als Fazit kann festgehalten werden, daß Fujaba bisher nicht in der Lage ist, mit fertigen Softwarekomponenten ein neues Softwaresystem zu entwickeln. Die Spezifizierung von einzelnen Softwarekomponenten beschränkt sich auf Komponentenmodelle, die auf der Programmiersprache Java basieren. Das Jini Modell, läßt sich zum größten Teil mit Fujaba umsetzen. Die Spezifizierung des Quellcodes kann mit Fujaba vorgenommen werden. Die Kompilierung der Quelltexte, das anschließende Packen und die Beschreibung der Jini Komponente wurde manuell vorgenommen. Fujaba kann in diesem Bereich noch verbessert werden, so daß die Spezifizierung einer Jini Komponente komplett in Fujaba vorgenommen werden kann.

5.2 Fujaba und das JavaBean Modell

Im Vergleich zum Jini Komponentenmodell ist das JavaBean Komponentenmodell leichter umzusetzen. Die wesentlichen Unterschiede ergeben sich aus Erstellung der Schnittstelle und der eigentlichen Umsetzung des Modells. Jini Komponenten benötigen eine XML-Schnittstellendatei, währen bei JavaBean Komponenten die Schnittstelle allein durch die Einhaltung der Namenskonvention erstellt wird. Auch die Umsetzung des JavaBean Modells ergibt sich im wesentlichen aus der Einhaltung der Namenskonvention. So lassen sich bereits einfache Java Klassen in eine JavaBean Komponente wandeln. Die Spezifikation von JavaBean Komponenten mit Fujaba ist leider nur eingeschränkt möglich. Es sind teilweise Änderungen am generierten Quellcode erforderlich. Im folgenden werden Negativbeispiele aufgeführt, die verdeutlichen, daß Fujaba die Entwicklung von JavaBean Komponenten nicht optimal unterstützt.

Probleme entstehen bereits durch einfache Assoziationen zwischen zwei Klassen. Die Introspektion wird durch get- und set- Methoden sowie deren Sichtbarkeit gesteuert. Die Methodennamen werden durch Fujaba zwar konform zur JavaBean Konvention erzeugt, jedoch lassen sich von assoziierten Klassen die Sichtbarkeiten für die get- und set- Methoden nicht getrennt festlegen. Ein Eingriff in den generierten Quellcode ist daher erforderlich, sofern der Entwickler verschiedene Sichtbarkeiten für die Methoden benötigt. Weitere Probleme entstehen durch gebundene Attribute [11]. Nach [1] müssen für gebundene Attribute Operationen zum Verwalten von *PropertyChangeListener* Objekte implementiert werden.

- *public void addPropertyChangeListener(PropertyChangeListener x)*

- *public void removePropertyChangeListener(PropertyChangeListener x)*

Fujaba bietet während der Erstellung von Attributen keine Möglichkeit an, diese als gebundene Attribute zu erzeugen, so daß eine entsprechende *PropertyChange* Unterstützung für die Attribute erzeugt wird. Auch hier ist ein Eingriff in den Quellcode erforderlich.

Diese einfachen Beispiele machen bereits deutlich, das auch das JavaBean Modell nicht ohne weiteres mit Fujaba umgesetzt werden kann. Änderungen im generierten Quellcode bzw. die Angabe von Java Quellcode in Fujaba, widersprechen der eigentlichen Idee von Fujaba, nämlich visuell zu entwickeln. Die Entwicklung eines JavaBean-Plugins bietet sich an. Das Plugin kann Dialogfelder, wie z.B. den Attribut-Editor entsprechend ändern und die Codegenerierung anpassen. So wäre ein Eingriff in den generierten Code nicht mehr nötig.

6 Zusammenfassung

Fujaba kann zur Entwicklung von Softwaresystemen genutzt werden. Es eignet sich im Allgemeinen gut zur Änderung von Objektstrukturen sowie für spezifische Probleme, die gut durch visuelle Programmierung gelöst werden können. Das UML Klassendiagramm ist leicht erstellt und bietet dem Entwickler einen guten Überblick. Änderungen an der Softwarearchitektur können später bequem am UML Klassendiagramm vorgenommen und durch die Codegenerierung umgesetzt werden. Kleine bzw. nicht allzu komplexe Methoden können ebenfalls leicht mit dem Aktivitätsdiagramm entwickelt werden. Sind die Methoden sehr komplex, so verringert sich der Vorteil der visuellen Programmierung. Sehr große Diagramme sind nicht automatisch leicht lesbar oder leicht zu verstehen. Hier bietet es sich an, die Methoden direkt in Java zu implementieren. Ein weiterer Grund, komplexe Methoden direkt in Java zu programmieren ist, daß die Codegenerierung von Fujaba nicht gut dokumentiert ist. Probleme entstehen auch, wenn nur ein Teil eines Softwareprojektes mit Fujaba erstellt wird. Ist der Entwickler gezwungen direkt auf Methoden des generierten Quellcodes zu zugreifen, beispielsweise auf einen durch Fujaba automatisch erzeugten Iterator einer Attributmenge, so gibt die Fujaba Dokumentation keine Hinweise, wie diese bezeichnet werden. Es ist dann mühsam, im generierten Quellcode nachzusehen und nachzuvollziehen, wie diese heißen. Weiterhin findet für Java Quellcode, der im Aktivitätsdiagramm benutzt wird, keine Syntaxprüfung statt.

Fujaba unterstützt die Spezifikation und die Nutzung von Komponenten nur schlecht. Das heißt, die Entwicklung von Softwaresystemen durch Softwarekom-

ponenten ist bisher überhaupt nicht mit Fujaba möglich. Komponentenmodelle, die nicht auf der Programmiersprache Java basieren, lassen sich mit Fujaba gar nicht umsetzen. Für die restlichen Komponentenmodelle muß jeweils einzeln geprüft werden, ob und wie diese durch Fujaba unterstützt werden. Jini Komponenten können mit Fujaba spezifiziert werden, [14]. Die Kompilierung des Quellcodes und das Packen in *jar* Archive wurde manuell vorgenommen. Das JavaBean Modell kann mit Fujaba nur eingeschränkt umgesetzt werden. Eventuelle Anpassungen am Quellcode sind erforderlich. Plugins können sowohl auf den Fujaba-Editor als auch auf die Codegenrierung Einfluß nehmen. Es bietet sich daher an, daß Fujaba durch entsprechende Plugins erweitert wird, so daß beispielsweise das JavaBean Modell vollständig umgesetzt werden kann.

Literatur

[1] BALZERT, HELMUT: *Lehrbuch der Software-Technik 1*. Spektrum Akademischer Verlag, 2001. ISBN: 3-8274-0480-0.

[2] DOTOR, ALEXANDER: *Fujaba Tutorial: Creating a Mancala-Game with Fujaba*. Universität Bayreuth, Lehrstuhl für Angewandte Informatik I, 2006. `http://www.se.eecs.uni-kassel.de/~fujabawiki/index.php/User_Documentation`.

[3] FUJABA-WEBSITE. Zuletzt besucht: Januar 2007. `http://www.fujaba.de/`.

[4] MEISEN, TOBIAS: *Fujaba*. Seminararbeit, RWTH Aachen, Lehrstuhl für Informatik 3, 2005.

[5] MICROSOFT, COM. Zuletzt besucht: Dezember 2006. `http://www.microsoft.com/com/default.mspx`.

[6] OBJECTMANAGEMENTGROUP. Zuletzt besucht: Dezember 2006. `http://www.omg.org/`.

[7] SCHWICHTENBERG, HOLGER: *Markt der Softwarekomponenten und Komponentenmodelle*. Zuletzt besucht: Dezember 2006. `http://www.dotnetframework.de/komponenten/markt/default.aspx`.

[8] SCHÄFER, PROF. DR. W.: *Fujaba Dokumentation*. Universität Paderborn, Fachbereich 17, Arbeitsgruppe Softwaretechnik, 2002. `http://ai4.inf.uni-bayreuth.de/export/sites/default/Documents/ai4_de/Projects/Informatik_Wettbewerb/Schuljahr_0607/Fujaba-Doku.pdf`.

[9] SUNMICROSYSTEMS, BEANBUILDER. Zuletzt besucht: Januar 2007. `https://bean-builder.dev.java.net/`.

[10] SUNMICROSYSTEMS, ENTERPRISEJAVABEANS. Zuletzt besucht: Dezember 2006. `http://java.sun.com/products/ejb/`.

[11] SUNMICROSYSTEMS, JAVABEAN. Zuletzt besucht: Dezember 2006. `http://java.sun.com/products/javabeans/`.

[12] SUNMICROSYSTEMS, JAVABEANSPECIFICATION. Zuletzt besucht: Dezember 2006. `http://java.sun.com/products/javabeans/docs/spec.html`.

[13] SZYPERSKI, CLEMENS: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002. ISBN:0-201-74572-0.

[14] TICHY, MATTHIAS: *Durchgängige Unterstützung für Entwurf, Implementierung und Betrieb von Komponenten in offenen Softwarearchitekturen mittels UML*. Diplomarbeit, Universität Paderborn, Fachbereich 17, Arbeitsgruppe Softwaretechnik, 2002.

[15] WIKIPEDIA-EINTRAG, CASE-WERKZEUGE. Zuletzt besucht: Dezember 2006. `http://de.wikipedia.org/wiki/Computer-Aided_Software_Engineering`.

[16] WIKIPEDIA-EINTRAG, FUJABA. Zuletzt besucht: Dezember 2006. `http://de.wikipedia.org/wiki/Fujaba`.