Flexibilität durch kombinierte Design Pattern

Carsten Weise

BRICS*, Universität Aalborg, Dänemark, Email: cweise@cs.auc.dk

Zusammenfassung In graphischen Benutzerschnittstellen und vielen anderen Anwendungen tritt häufig das Problem auf, daß dasselbe Objekt auf verschiedene Arten dargestellt werden soll. Dies führt dazu, daß konzeptuell gleiche Klassen in verschiedenen Versionen implementiert werden müssen. Anhand einer einfachen Fallstudie wird vorgeführt, wie diese Problematik durch geeignete Kombination von Design Pattern gelöst werden kann. Hervorragende Merkmale des vorgestellten Lösungswegs sind Einfachheit, Erweiterbarkeit und Robustheit: er kann in bereits bestehende Implementierungen nachträglich eingefügt werden, ohne mehr als die betroffene Klasse zu ändern, und es lassen sich auch im Nachhinein neue und sogar neuartig strukturierte Darstellungsarten hinzufügen.

1 Einleitung

Ein typisches Problem vieler objektorientierter Anwendungen ist die Notwendigkeit der Darstellung von Objekten derselben Klasse auf verschiedene Arten. Einfache Beispiele sind hierbei z.B. im Bereich graphischer Anwendungen die zwei- bzw. dreidimensionale Darstellung desselben Körpers, oder die Ausgabe eines Bildes sowohl in Farbe als auch in Schwarz-Weiß. Beispiele in nicht-graphischen Anwendungen sind z.B. die Darstellung komplexer Zahlen als kartesische oder Polarkoordinaten, oder die Implementierung einer Liste durch ein Array bzw. eine Zeigerstruktur.

Eine elegante Lösung eines solchen Problems sollte mindestens zwei Kriterien gerecht werden:

- saubere Trennung der verschiedenen Darstellungsarten,
- größtmögliches Ausnutzen ihrer Gemeinsamkeiten, um Code-Duplikation zu vermeiden

Sind dies die einzigen Anforderungen, so existiert eine einfache Lösung: die Trennung erreicht man durch Realisierung jeder Darstellungsform in einer eigenen Klasse, und eine allen Klassen gemeinsame Basisklasse verhindert Code-Replikation. Somit wird bei diesem Ansatz "eine Klasse durch viele Klassen implementiert". Aus Gründen der Klarheit unterscheiden wir im folgenden wenn nötig zwischen dem zu implementierenden *Konzept* (in obigen Beispielen: Körper,

* BRICS: Basic Research in Computer Science, Centre of the Danish National Research Foundation

Bild, Zahl, Liste) und den das Konzept realisierenden *Darstellungsarten* (oben z.B. eine Klasse für zwei- und eine für die dreidimensionale Darstellung).

In der Praxis werden in der Regel weitere Anforderungen an die Realisierung gestellt. Oft wird eine natürliche Implementierung der einzelnen Darstellungsformen selbst wieder von bereits bestehenden Klassen abgeleitet werden können. In Kombination mit der einfachen Lösung muß dann Mehrfachvererbung verwendet werden. Dies ist jedoch in vielen objektorientierten Sprachen nicht möglich, oder es soll darauf aus konzeptionellen Gründen verzichtet werden.

Ebenso ist zu erwarten, daß sich die Notwendigkeit verschiedener Darstellungsformen eines Konzepts erst sehr viel später im Lebenszyklus einer Anwendung zeigt, und nicht bereits während des Programmdesigns. Dann soll das Ersetzen der bisher verwendeten Klasse durch die neue, komplexere Implementierung so reibungslos wie möglich vonstatten gehen. Dies bedeutet insbesondere, daß die restliche Implementierung so wenig wie möglich verändert werden soll, und die neue Realisierung die alte Klasse soweit wie möglich nachahmt. Dies wird jedoch problematisch, wenn eine der Darstellungsformen sehr stark vom ursprünglichen Aussehen der Originalklasse abweicht.

In diesem Artikel wird eine Methode zur Implementierung eines Konzepts mit verschiedenen Darstellungsformen vorgestellt, die alle obigen Anforderungen erfüllt. Die Methode greift dabei auf verschiedene einfache Formen von Design Pattern ([GHJV94]) zurück. Solche Design Pattern sind Muster für die Lösung bekannter, wiederholt auftretender Problematiken mittels objektorientierter Hierarchien. Die in der Methode verwendeten Design Pattern sind unter den Namen Envelope-Letter ([Cop92]), Handle-Body ([Str93]) oder Bridge ([GHJV94]) und Abstract-Factory ([GHJV94]) bekannt. Die Kombination dieser Design Pattern stellt dann ebenfalls ein Design Pattern dar. Durch Verwenden der Methode wird die Anwendung flexibler, denn die Wahl der Darstellungsmöglichkeit eines Objektes kann nun frei in der Anwendung erfolgen. Die Methode wird anhand eines Fallbeispieles konkretisiert. Das Fallbeispiel ist die Erweiterung einer graphischen Oberfläche eines Simulators für Echtzeitsysteme um neue Darstellungsformen.

Im nächsten folgenden Abschnitt gehen wir kurz auf die Philosophie der Design Pattern und auf die drei erwähnten Design Pattern ein. Danach wird die Problematik des Fallbeispiels erläutert. Die folgenden drei Abschnitte erläutern die alte System-Architektur, sowie in zwei einfachen Schritten die Transformation in eine neue, flexiblere Architektur mittels der vorzustellenden Methode. Der Artikel schließt mit einer Diskussion des Ansatzes und einem Ausblick auf weitere Forschung.

2 Design Pattern

Design Pattern sind die Beschreibungen von Mustern für den Aufbau von Klassenhierarchien für wiederkehrende Implementierungsprobleme ([GHJV94]). Im

weiteren werden hier die Pattern Handle-Body, Letter-Envelope und Abstract Factory verwendet, die kurz erläutert werden sollen.

Beim Body-Handle ([Str93], auch als Bridge[GHJV94] bekannt) wird die Implementierung eines Konzepts in eine Handle- und eine Body-Klasse aufgeteilt. Die Body-Klasse ist die vor dem Anwender verborgene Realisierung der Objekte. Zugriff auf die Objekte hat der Anwender nur über die Handle-Klasse, die Referenzen auf die Body-Klasse verwaltet. Das Muster erlaubt die einfache Trennung von Schnittstelle und Implementierung, und kann z.B. verwendet werden, um mehrfaches Abspeichern der gleichen Datenstruktur zu umgehen.

Das Letter-Envelope Pattern ([Cop92]) erweitert diese Idee um die Möglichkeit verschiedener Darstellungsformen für den Body. Dabei ist sogar der Wechsel der Darstellungsart während der Lebenszeit des Objektes möglich. In [Cop92] findet sich ein sehr elegantes Beispiel zu Implementierung komplexer Zahlen.

Die Abstract Factory ([GHJV94]) erlaubt das Herstellen eines Objektes durch ein Factory Objekt anstelle eines Konstruktoraufrufs. Dieses Muster ist z.B. geeignet, wenn der Konstruktoraufruf zu komplex für die übliche Schnittstelle wird, oder wenn der exakte Typ des zu erzeugenden Objekts erst zur Laufzeit ermittelt werden kann.

3 UPPAALs graphische Oberfläche

Unser Fallbeispiel ist die Erweiterung der graphischen Oberfläche eines Simulators für Echtzeitsysteme. Der Simulator ist Teil des Verifikationswerkzeuges UPPAAL ([LPY97]), das zur Darstellung der Echtzeitsysteme einen Formalismus namens *Timed Automata* (Zeitautomaten, [AD94]) verwendet. Dabei werden die Echtzeitsysteme durch eine Menge autonomer Prozessen dargestellt. Ein Prozeß wird hierbei durch einen endlichen Automaten (Finite State Machine) mit Uhren zum Messen des Zeitverlaufs beschrieben. Für die Visualisierung eines Simulationslaufs werden alle beteiligten Prozesse als gerichtete Graphen abgebildet. Der momentane Prozeßzustand wird durch Einfärben des aktiven Knoten bzw. der aktiven Zustandsübergänge (Kanten) dargestellt. Die beiden Prozesse aus den Abb. 4 bis 6 befinden sich in ihrem Anfangszustand, d.i. der jeweilige Knoten in der linken oberen Ecke.

Die graphische Oberfläche von UPPAAL wird derzeit neu implementiert. Als Implementierungssprache wurde Java gewählt, um eine plattformunabhängige Graphik verwenden zu können, und um in Zukunft die Dienste des Verifikationswerkzeuges auch über das World Wide Web zur Verfügung zu stellen.

Bei der Neu-Implementierung stellte sich die Frage nach der geeigneten Darstellungsform der Prozeßmenge. Der erste Prototyp der Java-Implementierung verwendete pro Prozeß ein eigenes Fenster. Obwohl als Verbesserung der alten Oberfläche gedacht – bei der alle Prozesse im selben Fenster dargestellt wurden – zeigten sich im Gebrauch neben den Vor- auch viele Nachteile.

Daraus entstand die Idee, entweder zur alten Darstellungsform zurückzukehren, oder zu hoffen, daß man durch Verwenden der "internal Frames" aus Swing

die Vorteile des alten und des neuen Formats verknüpft werden können. Somit standen nun drei Darstellungsmöglichkeiten zur Auswahl:

(a) alle Prozesse in einem Fenster,
(b) pro Prozeß ein eigenes, autonomes Fenster,
(c) pro Prozeß ein Fenster innerhalb eines gemeinsamen, übergeordneten Fensters.

Es wurde beschlossen, alle drei Möglichkeiten zu implementieren, sodaß der Benutzer dann die für ihn geeigneteste auswählen kann. Die Implementierung soll dabei möglichst viel des existierenden Codes des Prototyps verwenden, und möglichst wenige Änderungen im restlichen Programm nach sich ziehen. Die folgende Schritte beschreiben das Vorgehen, um dieses Ziel zu erreichen. Ausgangspunkt war dabei der bereits existierende Prototyp.

4 Die alte Architektur

Der erste Prototyp von UPPAALs neuer graphischer Oberfläche stellte alle Prozesse in eigenen, unabhängigen Fenstern dar. Abbildung 1 gibt einen Überblick über den für uns wesentlichen Teil der Klassenhierarchie.[1]

Gesteuert wir die Simulation von einem Objekt der Klasse `SimulatorWindow`, die von `JFrame` abgeleitet ist. UPPAAL kann prinzipiell mehrere verschiedene Simulation gleichzeitig fahren.

Ein `SimulatorWindow` erzeugt für jeden Prozeß ein `PDWindow`[2]. Diese Klasse ist ebenfalls von `JFrame` abgeleitet, und erzeugt ein von `SimulatorWindow` unabhängiges Fenster. Jedes `PDWindow` wiederum erzeugt ein Objekt vom Typ `PDMain`, das die wesentliche Funktionalität des Prozeßfensters implementiert. Ein `PDMain` Objekt verwendet zwei Hilfsobjekte vom Type `PDControl` und `PDCanvas`. Ersteres ist zuständig für Benutzeraktionen im Prozeßfenster, und implementiert die Schnittstellen `MouseListener` und `MouseMotionListener`. Es enthält ein Objekt vom Type `Process`, das alle für einen Prozeß nötige Informationen speichert.

Die Klasse `PDCanvas` ist zuständig für das Zeichnen der Prozeßdarstellung im `PDWindow`. Sie ist von `JComponent` abgeleitet und hat Zugriff auf ein Objekt `GraphicsContext`, das alle wichtigen Informationen und Methoden enthält, um die einzelnen, zu einem Prozeß gehörenden Objekte zeichnen zu können. `GraphicsContext` verwendet zum Abwickeln aller Zeichenoperationen das zu `PDCanvas` gehörige `Graphics` Objekt.

[1] Alle Abbildungen benutzen eine OMT/UML-ähnliche Darstellung ([UML98]): Klassen sind Rechtecke, die im obersten Kästchen den Klassennamen sowie Vererbungs- und Schnittstellenbeziehung enthalten. Darunter werden die Typen der in einem Objekt enthaltenen, relevanten Teilobjekte aufgelistet. Pfeile beschreiben Relationen zwischen den Klassen. Namen in Schreibmaschinenschrift (Teletype) sind in `Java` vordefinierte Klassen.

[2] PD = Process Display

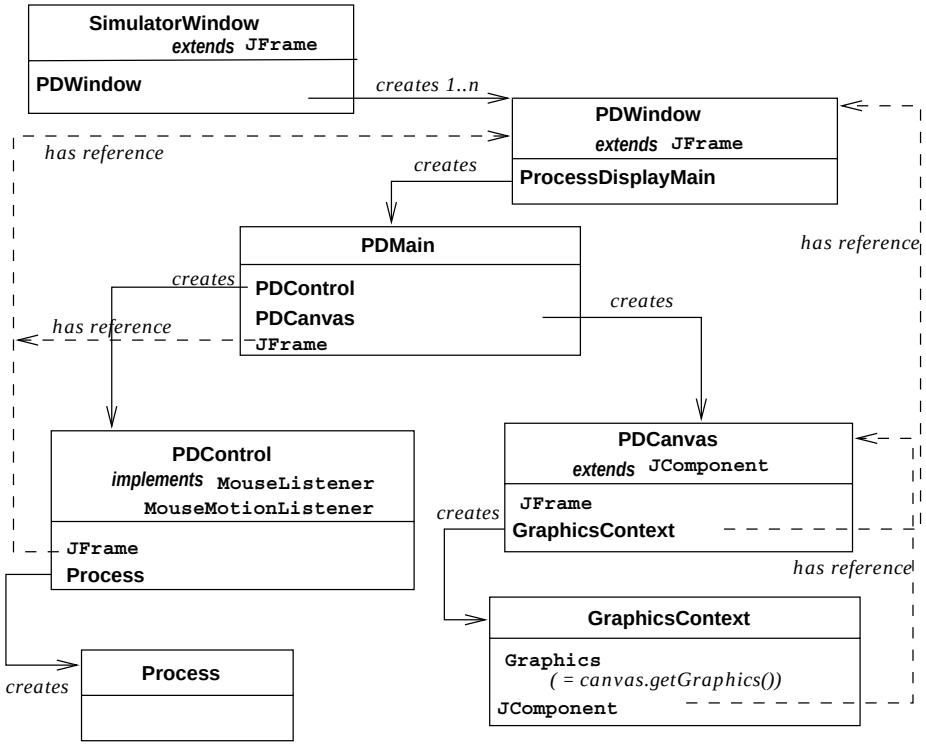

Abbildung1. Alte Systemarchitektur

Die verschiedenen Aufgaben sind bereits fein auf verschiedene Klassen verteilt, um gegenüber Änderungen flexibel zu sein. Die Modellierung ist jedoch nicht fein genug für unser Problem.

Die Klasse `PDWindow`, der unsere Aufmerksamkeit gilt, ist also in ein komplexes Geflecht eingebunden. Man beachte insbesondere, daß sie wie viele andere Klassen der Hierarchie eine Referenz auf sich selbst an direkt oder indirekt erzeugte Objekte weiterreicht. Bei der Erweiterung ist dafür Sorge zu tragen, daß diese Referenzen von den anderen Objekten wie bisher weiter benutzt werden können.

5 Erweiterung der Darstellungsformen

Zum besseren Verständnis werden in diesem Abschnitt zuerst drei unabhängige Lösungen für die Darstellungsformen (a), (b) und (c) aus Abschnitt 3 skizziert. Fall (b) bedarf keiner weiteren Erläuterung, denn er wird bereits durch die im letzten Abschnitt beschriebene Architektur gelöst. Fall (c) kann im wesentlichen

durch Ableiten der Klasse `PDWindow` von `JInternalFrame` erreicht werden. Da `JFrame` und `JInternalFrame` voneinander abweichende Attribute und Schnittstellen haben, sind zusätzlich einfache Änderungen in den anderen Klassen notwendig.

Fall (a) ist hingegen sehr viel komplizierter, da man hier die eins-zu-eins Beziehung zwischen Prozessen auf der konzeptionellen Seite und den Fenster (d.h. `JFrame` oder `JInternalFrame`) auf der Implementierungsseite verliert. Für die Klasse `PDWindow` hat diese keine weitreichenden Konsequenzen, denn sie kann statt auf einen eigenen `JFrame` auf den der Klasse `SimulatorWindow` zurückgreifen. Das eigentliche Problem tritt in der Klasse `PDCanvas` auf, denn statt einem Canvas pro Prozeß wird nun nur noch ein Canvas pro Simulation gebraucht. Der Übergang von der eins-zu-eins Beziehung zwischen `PDCanvas` und der `JComponent` (als Realisierung des Canvas) zu einer viele-zu-eins Beziehung kann wie folgt vollzogen werden.

Statt `PDCanvas` von einer `JComponent` abzuleiten, wird in der Klasse eine statische Liste angelegt, in die die erzeugten Canvas eingetragen werden. Ein `PDCanvas` kennt zum einen dem ihn zugehörigen Canvas (aus der Liste) und delegiert alle entsprechenden Aufrufe dorthin, und zum anderen kennt er seine Anfangsposition relativ zu diesem Canvas. Die einzige Änderung außerhalb von `PDCanvas` ist die Erweiterung der Klasse `GraphicsContext` um die Möglichkeit, ihr diese Position mitzuteilen.

Die statische Canvas-Liste speichert nicht nur den Canvas, sondern auch den dazugehörigen `JFrame`. Da der Konstruktor der Klasse `PDCanvas` den dazugehörigen `JFrame` erhält, kann somit leicht festgestellt werden, ob es bereits einen entsprechenden Canvas gibt oder ein neuer erzeugt werden muß. Pro Simulation wird solcher Canvas gebraucht. Da schon aus Geschwindigkeitsgründen kaum mehr als zehn Simulationen parallel gefahren werden können, wurde die Liste ganz direkt als Array fester Länge (momentan: 100 Elemente) implementiert. In komplexer gelegenen Fällen wird man jedoch auf ein Bridge oder Proxy Pattern (vgl. [GHJV94]) zurückgreifen müssen, bei dem eine neu zu definierende Canvas-Klasse vollständig in `PDCanvas` verborgen wird, und die `PDCanvas` Objekte sich dann die zugehörigen Canvas-Objekte teilen.

6 Die neue Architektur

Um in der Anwendung die Flexibilität bei der Auswahl der Darstellungsart zu erhöhen, werden die Klassen `PDWindow` und `PDCanvas` nach dem selben Muster erweitert. Bei diesen Mustern wird die Originalklasse durch eine Klasse ersetzt, die nur die Schnittstelle der Darstellungsformen und deren Gemeinsamkeiten enthält, sowie eine Referenz auf eine Darstellung des Objekts. Die Darstellungsarten selbst werden durch verschiedene, lokale (also für den Anwender unsichtbare) Klassen implementiert. Als Typ der Referenz kann man in `Java` ganz allgemein `Object` wählen, aber i.a. auch ein Interface definieren, das von allen Darstellungsformen implementiert werden muß.

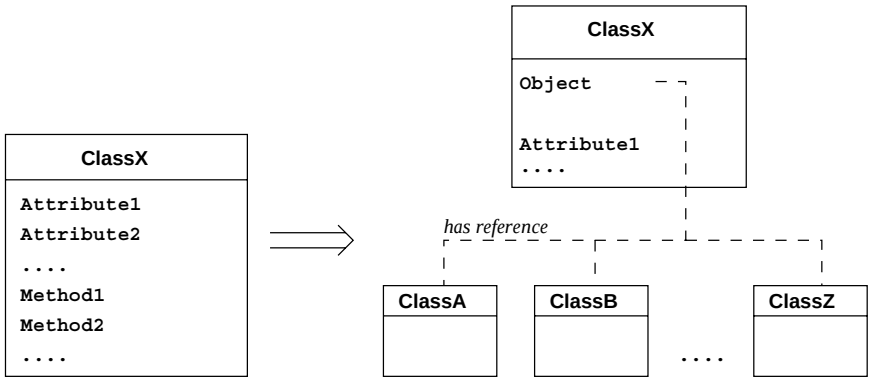

Abbildung2. Muster der Ersetzung

Abbildung 6 stellt dieses allgemeine Vorgehen dar. Die neue Version der ursprünglichen Klasse übernimmt nun die Rolle der Schnittstelle zur Außenwelt. Sie verwaltet die Referenz auf eine der Darstellungsformen und delegiert alle Anfragen, die sie nicht selbst beantworten kann. Da diese Klasse auch das Wissen über sämtliche Darstellungsarten besitzt, kann sie auf Aufruf flexibel reagieren. So kann es z.B. vorkommen, daß eine Methode nur an einige der Darstellungsarten weitergereicht werden kann, während für andere die Schnittstellenklasse eine Sonderbehandlung vornimmt. Das geschieht im Fallbeispiel z.B. da ein `JInternalFrame` nicht alle Methoden eines `JFrame` besitzt.

In unserem Anwendungsbeispiel sind die Darstellungsformen der Klasse `PDWindow` die Klassen `PDArea` (Fall (a)), `PDFrame` (Fall (b)) und `PDInternalFrame` (Fall (c)). Für die Klasse `PDCanvas` werden die Darstellungen durch die Klassen `PDCArea` (Fall (a)) und `PDCWindow` (Fall (b) und (c)) realisiert. Die Implementierungsdetails dieser Klassen wurden im vorangehenden Abschnitt bereits erläutert. Abbildung 3 zeigt die Klassenhierarchien, durch die die Klassen `PDWindow` und `PDCanvas` ersetzt werden.

Dem Konstruktor der neuen Version von `PDWindow` muß explizit mitgeteilt werden, welche Darstellungsart gewünscht wird. Dies mag in anderen Fallbeispielen implizit geschehen.

An die Klasse `PDMain` und alle weiteren wird nun eine Referenz vom Typ `PDWindow` weitergereicht. Die restliche Implementierung sieht somit von der gewählten Darstellungsform gar nichts, und braucht auch nicht geändert werden. Problematisch wird dies jedoch beim Erzeugen der `PDCanvas` Objekte, denn die dort zu wählende Darstellungsform hängt von der für `PDWindow` gewählten ab. Deshalb werden `PDCanvas` Objekte nicht durch einen Konstruktor erzeugt, sondern über eine Methode `PDWindow`. Mit Hilfe dieses Ansatzes können also auch komplexere Verzahnungen zwischen den Darstellungsarten verschiedener Konzepte bewältigt werden.

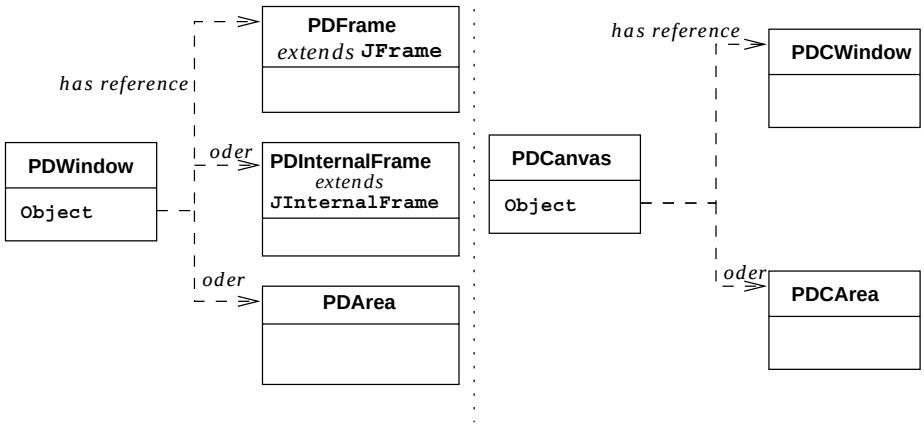

Abbildung3. Die neue Klassenhierarchie für `PDWindow` und `PDCanvas`

Das allgemeine Muster ist ein Mischtyp aus Handle-Body und Letter-Envelope. Wie bei Handle-Body werden Darstellungsart und Zugriff getrennt, und wie bei Letter-Envelope sind mehrere Darstellungsarten möglich. Allerdings ist im Gegensatz zum Letter-Envelope keine Konversion zwischen den Darstellungsformen vorgesehen.

Die Idee, Objekte vom Typ `PDCanvas` durch Objekte vom Typ `PDWindow` erzeugen zu lassen, ist eine einfache Version einer Abstract Factory. Bei der Realisierung als Abstract Factory tritt ein geringes Problem auf: `PDWindow` ruft zum Erzeugen eines `PDCanvas` dessen Konstruktor auf. Leider ist dadurch der Konstruktor aber öffentlich, wodurch "Mißbrauch" möglich wird. Es gibt leider keine einfache Möglichkeit, dies zu verhindern, ohne die bestehende Klassenhierarchie zu verändern.

Die Abb. 4 bis 6 zeigen Screenshots der drei verschiedenen Darstellungsformen für jeweils die selben zwei Prozesse.

7 Abschließendes

Im vorliegenden Artikel wurde gezeigt, wie man durch Kombination bekannter Design Pattern die Flexibilität beim Wechsel zwischen Darstellungsformen in einer graphischen Java Anwendung erhöhen kann. Der Ansatz ist weder auf graphische Anwendungen noch auf Java als Programmiersprache beschränkt, sondern allgemein in der objektorientierten Programmierung verwendbar.

Die zentrale Idee des Ansatzes ist das Zerlegen des Konzeptes in eine Klasse für die Schnittstelle und in mehrere Klassen für die verschiedenen Darstellungsarten. Diese Klassen werden dann wie beim Handle-Body/Bridge-Pattern zusammengefügt.

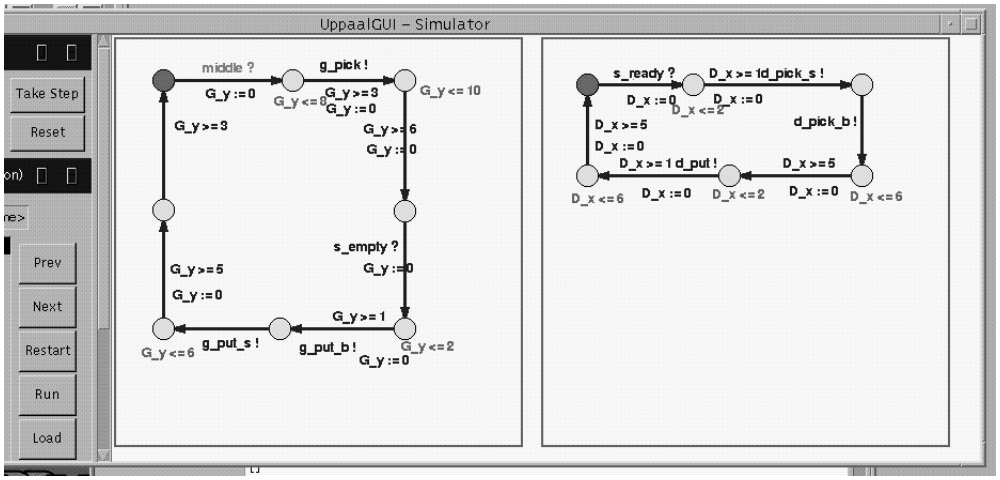

Abbildung4. Zwei Prozesse in einem gemeinsame Fenster

Es wurde gezeigt, daß Paradigmen wie Interfaces und Einfachvererbung für die Realisierung alleine nicht ausreichen. Unser Ansatz hat außerdem den Vorteil, insbesondere auch nachträgliche Erweiterungen ohne Verletzen der bestehenden Klassenhierarchie zu ermöglichen.

Verzahnung zwischen den Darstellungsformen verschiedener Konzepte konnten durch eine einfache Version des Abstract Factory Pattern gelöst werden. Ebenso wurde gezeigt, wie sich auch Darstellungsarten, die sich nicht direkt mit der bestehenden Klassenhierarchie vertragen, implementiert werden können. Der Ansatz hilft somit bei der Lösung folgender Problemarten:

- Ersetzen einer einzigen Darstellung eines Konzepts durch eine Familie von Darstellungen,
- Verzahnung verschiedener Darstellungsfamilien über mehrere Ebenen,
- Ersetzen einer eins-zu-eins Beziehung von Realisierung und Verwendung durch eine viele-zu-eins Beziehung.

Alle drei Problemarten lassen sich auf einfache Art unter zuhilfenahme passender Design Pattern leicht lösen. Insbesondere sollten sich auch komplexere Situationen mit demselben Ansatz lösen lassen, wobei dann komplexere Ausformungen der verwendeten Design Pattern zu erwarten sind. Es ist zu hoffen, daß andere Java Programmierer von dieser Erfahrung profitieren können. Eine Java Lösung kann elegant durch Java Reflections und JavaBeans unterstützt werden. Gerade JavaBeans sollten sich als hilfreich erweisen, da i.a. die Implementierung der Darstellungsklassen auch in ihren Unterschieden noch große Ähnlichkeit aufweisen werden.

Die herausragenden Eigenschaften des Ansatzes sind Einfachheit, Erweiterbarkeit und Robustheit. Für einen erfahrenen Programmierer sollte die Umset-

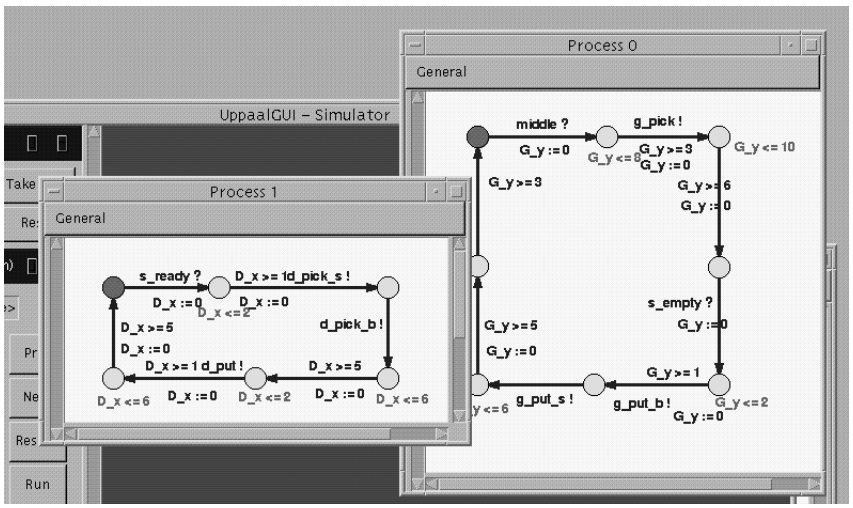

Abbildung5. Zwei Prozesse in unabhängigen Fenstern

zung des Ansatzes problemlos sein. Der Ansatz erlaubt die spätere Erweiterung um zusätzliche Darstellungsarten. Da der Ansatz die ursprüngliche Klassenhierarchie erhält, brauchen Änderungen nur sehr lokal durchgeführt zu werden. Dadurch ist der Ansatz robust: außerhalb der betroffenen Klassen sind fast keinerlei Änderungen nötig, sodaß die Anwendung auch nach den Änderungen in gewohnter Weise funktionieren sollte.

Die Entwicklung unseres Verifkationswerkzeuges ist auf dem Weg zu einem recht flexiblen und universell einsetzbaren Werkzeug, daß z.B. auch einen Paradigmenwechsel in der Prozeßdarstellung erlaubt. Auch bei solchen Problemen arbeiten wir mit ähnlichen Lösungsansätzen.

Für die Zukunft überlegen wir, `PDWindow` und `PDCanvas` vollständig als Envelope-Letter zu implementieren, sodaß sogar ein Wechsel der Darstellung während des Simulationslaufs möglich ist.

Danksagung. Der Autor bedankt sich bei Carsten Lindholst und Peter Lindstrøm für die Prototyp-Implementierung der graphischen Oberfläche, sowie bei ungenannten Gutachtern für zwei wertvolle Hinweise.

Literatur

[AD94] R. Alur, D.L. Dill. *A Theory of Timed Automata.* in: Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison Wesley, Reading, MA, 1992.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1994.

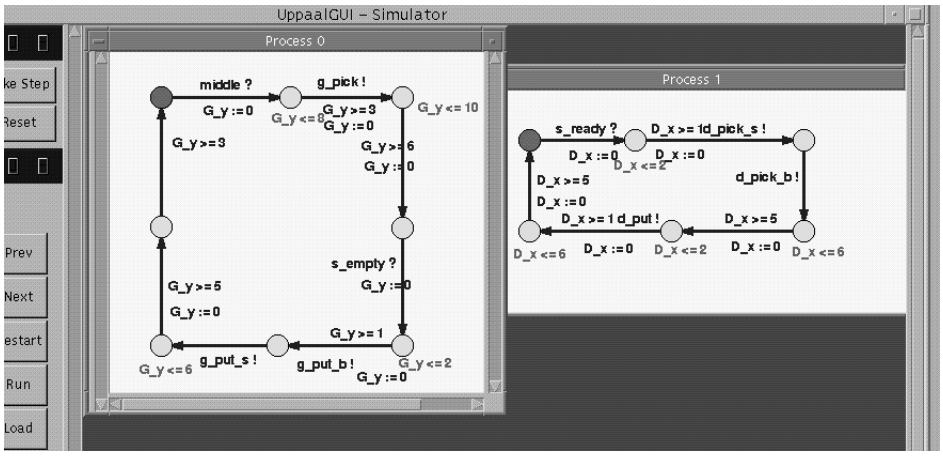

Abbildung6. Zwei Prozesse als `JInternalFrame`

[LPY97] Kim G. Larsen, Paul Pettersson, Wang Yi. UPPAAL *in a Nutshell*. International Journal on Software Tools for Technology Transfer No.1+2, Springer–Verlag, 1997.

[Str93] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, MA, 1993.

[UML98] Online Dokumentation zu UML/OMT im WWW:
`http://www.rational.com/uml/resources.html`