BOSCH

Schwerpunktseminar Mechanismen

Eine Kooperation der

Fachhochschule Gießen
Fachbereich Mathematik, Naturwissenschaften und Informatik
Wiesenstraße 14
35390 Gießen

und

Robert Bosch GmbH
FV/SLD
Eschborner Landstraße 130-132
60489 Frankfurt

im Sommersemester 2002

Kontakt: Claudia.Fritsch@de.bosch.com

Vorwort

Im Sommersemester 2002 haben die Professoren Dr. Burkhardt Renz und Dr. Wolfgang Henrich von der FH Gießen und die Robert Bosch GmbH in Frankfurt gemeinsam ein Schwerpunktseminar „Mechanismen“ durchgeführt. Bosch lieferte die Themen aus dem Umfeld „Multimedia im Auto“, die Studenten erarbeiteten und präsentierten „Mechanismen“: Lösungen für Architektur, Entwurf und Codierung von Softwaresystemen. Dieses Buch enthält die Ausarbeitungen der Seminarthemen.

Umfeld. Wir betrachten hier das Umfeld „Multimedia im Auto“. Multimedia im Auto umfasst Unterhaltung (Radio, CD, DVD, Video), Information (Navigation, Internet, Organizer-Funktionen) und Kommunikation (Telefon, SMS, E-Mail, Not- und Pannenruf). Uns interessieren hier Systeme, die diese Funktionen (oder einen Teil davon) den Autoinsassen zur Verfügung stellen.

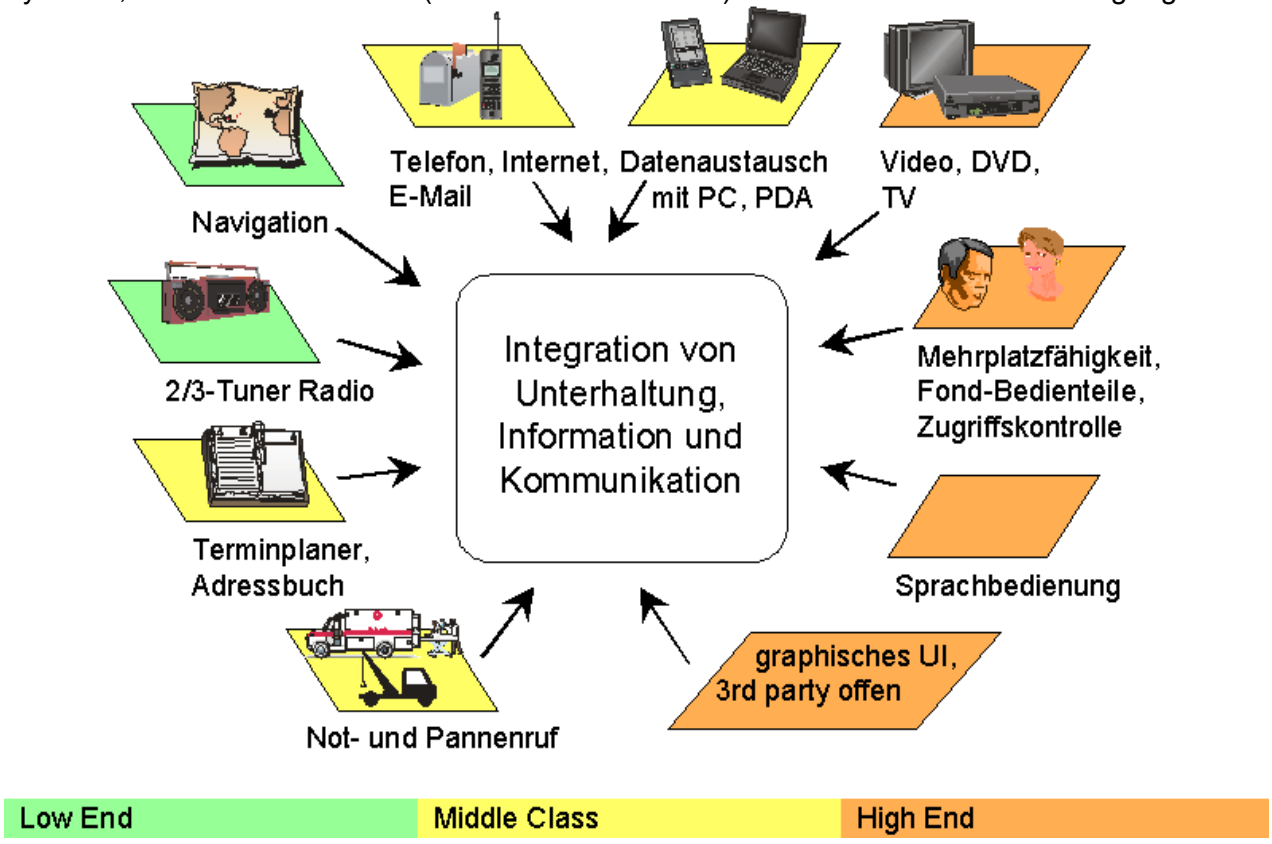


Abbildung: Multimedia im Auto: Unterhaltung, Information, Kommunikation..

Besondere Anforderungen an die Softwareentwicklung. U.a. sind folgende Aspekte wesentlich:

1. Dem Fahrer muss die Bedienung eines solchen Systems so einfach wie möglich gemacht werden, es muss hochkomfortabel sein, denn: das Fahren kommt zuerst.

2. Funktionalität des Systems, die Kontakt nach außen braucht – z.B. Telefon, Navigation, Internet – geht vom fahrenden Auto aus nur durch die Luft.

3. Die Entwicklung solcher Systeme ist zeitaufwendig und teuer und fordert deshalb eine lange Lebensdauer großer Teile der Software und damit eine gewisse Unempfindlichkeit, z.B. gegenüber Änderungen an der Hardware und der Betriebssystemsoftware. Ebenso müssen die spezifischen Wünsche der Automobilhersteller berücksichtigt werden.

4. Um die Zuverlässigkeit und Korrektheit eines solch komplexen Systems sicherzustellen, braucht man Test- und Fehleranalyseverfahren. Soll das System multi-threaded programmiert werden, so stellt dies besondere Ansprüche an die Code-Richtlinien.

Mechanismen. Um diesen Anforderungen gerecht werden zu können, hat man sich eine Reihe von Mechanismen überlegt. Sie nennen interessante Ansätze zur Lösung der Probleme. Es fehlte jedoch eine allgemein verständliche, ausführliche und einheitlich strukturierte Beschreibung der Mechanismen. Dies zu erarbeiten war Inhalt des Seminars.

Mechanismenpäckchen. Meist sind zwei oder mehr Mechanismen zu einem Päckchen zusammengefasst. Dies hat folgende Gründe:

- Mitunter tragen mehrere Mechanismen gemeinsam zur Lösung eines Problems bei. Hier war es interessant, die Mechanismen nicht nur einzeln zu betrachten, sondern auch ihr Zusammenwirken darzustellen.
- Anderswo gibt es mehrere Mechanismen, die jeder für sich das Problem lösen würden. Welchen sollte man nehmen? Wovon hängt diese Entscheidung ab? Hier war es interessant, die Mechanismen einander gegenüberzustellen und ihre jeweiligen Vor- und Nachteile abzuwägen.

Inhalt

Ausarbeitungen aus dem Seminar:

Graphische Benutzeroberflächen

Florian Hoffmann:	Dynamic Look and Feel Switching
	Visitor
Adam Kreuschner:	Vergleich zwischen Java Swing und Windows Forms

Austauschbarkeit von Diensten und Verhalten

Heiner Engelhardt:	Abstract Factory
	Builder
Knut Enners:	Strategy
Reza Rashidi:	Decorator

Multithreading

Felix Guntrum:	Einführung in das Programmieren mit Threads
Dominik Sacher:	Scoped Locking C++ Idiom
	Strategized Locking
Kurt Rosenberg:	Thread-Safe Interface
	Double-Checked Locking Optimization

Dynamic Look & Feel Switching

Die nötigen Techniken

Eine Ausarbeitung

von

Florian Hoffmann

Matrikel Nr. 639426

Fachhochschule
Gießen-Friedberg

Wofür Dynamic Look & Feel Switching?

Jeder hat die Schlagworte Dynamic Look & Feel Switching oder damit zusammenhängende wie „Skins“ schon gehört und bestimmt auch schon genutzt, vielleicht ohne es zu wissen oder darüber nachzudenken.

Viele Programme bieten heutzutage die Möglichkeit ihre Benutzeroberfläche den eigenen Bedürfnisse (un)sinnvoll anzupassen.
Unsinnvoll meiner Meinung nach in solchen Fällen wo es auf Kosten der Performance möglich ist „seine“ Oberfläche zu gestalten ohne jedoch einen wirklichen Nutzen daraus zu ziehen.
Sinnvoll wäre es dann dieses Feature an und abschaltbar zu machen wie es bei einigen solcher Funktionen in den neueren Betriebssystemen auch möglich ist.

Doch was für einen Nutzen könnte man überhaupt aus solch einem Feature ziehen? Wo könnte es nützlich sein die GUI (Graphic User Interface) zur Laufzeit eines Programmes komplett auf den Kopf zu stellen oder kleine Veränderungen durchzuführen?
Oder ist dieses Look & Feel Switching nur eine nette Spielerei für Individualisten?

Stellen wir uns folgende Situation vor:
Jemand ist viel mit dem PKW unterwegs, fährt sowohl tags als auch nachts.
Dabei ist er auf ein Navigationssystem angewiesen und will dieses sowohl nachts als auch tagsüber lesen und bedienen können.
Manche Farben sind durch die Eigenschaften des menschlichen Auges besser im hellen, manche besser im dunkeln zu erkennen. Dazu kommt noch die Leuchtstärke des Displays. Eine Veränderung des Displays wäre in dieser Situation sehr von Vorteil.

Kommt dieser Autofahrer nun in die Nähe eines Stau oder fährt mit ungeminderter Geschwindigkeit auf eine Abzweigung zu der er droht zu verpassen (oder ist sogar schon vorbeigefahren) so könnte ihm sein Display zusätzlich zu eventuellen akustischen Warnsignalen auch noch mit optischen Warnsignalen anzeigen das etwas nicht stimmt und er sich neu orientieren muss.

Optische Warnsignale können auch unter anderen Bedingungen sinnvoll sein:
Der Akku eines mobilen Gerätes kommt in einen kritischen Ladezustand...
Eine Fertigungsanlage oder gar ein Kraftwerk kommen in einen kritischen Betriebszustand der sofortiges Eingreifen erfordert...
Hier kann der Benutzer, bzw. ein „Aufseher“, am am Display durch ein Umschalten der GUI in einen Farbzustand der Gefahr signalisiert (bei Menschen gewöhnlich die Farbe rot) versetzt und eventuell ein angepasstes Menü für diese Situation bereitgestellt werden.

Loggt sich ein Benutzer mit gesonderten Rechten in ein Programm ein (Beispielsweise Administrator Rechte), so kann er über ein anderes Layout darauf hingewiesen werden das er auch viel kaputt machen kann und daher vorsichtig sein muss mit dem was er tut.
Man könnte ihm Aufgrund der gesonderten Nutzungsrechte auch komplett andere Menü Optionen bzw. erweiterte Menüs zur Verfügung stellen auf die sonst kein Zugriff ist.

Ein Nutzer könnte ein Programm regelmäßig zu mehreren verschiedenen Tätigkeiten nutzen.
Beispielsweise nutzt er es Montag bis Donnerstag nur zur Dateneingabe. Freitag nutzt er es dann nur um die eingegebenen Daten auszuwerten.
Für das Auswerten werden vermutlich andere Funktionen öfter gebraucht als für das Eingeben.
Damit die jeweiligen Funktionen schneller erreichbar sind und der Mann so schneller arbeiten kann könnte er zwischen zwei unterschiedlichen Layouts switchen.

Ein letztes Beispiel dessen Auswirkung nicht unterschätzt werden darf:
Das einrichten einer individualisierten Arbeitsumgebung.
Der Anwender fühlt sich auf seinem eigenen Layout zu Hause, ist dadurch eventuell motivierter und arbeitet dadurch effektiver.

All diese Beispiele können durch das Einsetzen von Dynamic Look & Feel Switching gemeistert werden.
Somit dürften wir nun genug Motivation haben uns näher damit zu befassen.
Es ist wohl nicht nur eine Spielerei für Individualisten und für Lustige Effekte.

Was wollen wir erreichen?

Zuerst betrachten wir nun näher, was für Anforderungen wir erfüllen müssen um Dynamic Look & Feel Switching zu ermöglichen.
Danach werden wir einen besseren Blick dafür haben was und wie wir es umsetzen können.

Wir wollen das Look & Feel, also das komplette Aussehen und Verhalten der GUI zur Laufzeit eines Programms ändern können.

Dazu gehören:

a) Inhalt, Aussehen, Platzierung einzelner Elemente, also folglich:
1)Vordergrund (Graphik, Schriftart, Farbe, Größe, Ausrichtung,...)
2)Hintergrund (Füllhalter, Füllfeder, Graphik,...)
3)Form (Rechteck, Kreis, individuelle Form, eventuelle animierte Elemente,...)
4)Platzierung auf der GUI bzw., in anderen Elementen

b) Zusammenfassen von Elementen zu Gruppen

c) Auswahl vorgefertigter Layouts

d) Möglichkeit die Layouts zu erweitern (auch durch eigene)

e) a) bis d) müssen zur Laufzeit eines Programms ohne Neustart möglich sein

Dies ist eine kleine Liste mit der wir so ziemlich alles abgedeckt haben was erforderlich ist.
Die Platzierung der Elemente sollte möglichst frei wählbar sein.
Die Möglichkeit einer absoluten Positionierung sollten wir nur wählen wenn es nicht anders geht oder wir einen Layout Manager zur Verfügung haben der die Koordinaten und Größen neu berechnet wenn sich etwas an der Umgebung ändert (z.B. Größenordnung des umgebenden Fensters) um unliebsamen Effekten vorzubeugen.
Da wir diese Möglichkeit immer im Auge haben müssen (es wird ja unter Umständen auch unter verschiedenen Auflösungen gearbeitet) sollten wir also prinzipiell, soweit vorhanden, einen Layout Manager vorziehen.
Je mehr Möglichkeiten der Einflussnahme wir haben möchten um so mächtiger und komplexer muss der genutzte Layout Manager sein.

Desweiteren müssen wir darauf achten die in den Elementen enthaltenen relevanten Daten nicht zu verlieren.
Ändern wir z.B. das Look & Feel einer Eingabemaske so wäre es für den Nutzer ärgerlich wenn die schon eingegebenen Daten verloren gehen.

Nötige Datenstrukturen und Techniken

Da wir jetzt festgestellt haben was wir erreichen wollen können wir uns um die Umsetzung Gedanken machen und was dafür erforderlich ist.

Mir sind zwei prinzipielle Möglichkeiten hierzu in den Sinn gekommen:

Zum einen eine Möglichkeit das ganze in einer rein Prozeduralen, nicht objektorientierten, Umgebung umzusetzen.
Diese Methode nenne ich mal die „Klassische Methode“.

Die andere Methode arbeitet in einer objektorientierten Umgebung.
Hier gibt es eine Unterteilung.
Zum einen kann die Möglichkeit existieren die Graphik Objekte direkt zu manipulieren, nennen wir es die „einfache objektorientierte Methode“.
Es kann aber auch sein das man die Objekte nicht direkt verändern kann und diese neu erzeugen muss. Nennen wir es die „erweiterte Objektorientierte Methode“.

Die „klassische Methode“

In einer rein prozeduralen Umgebung können wir auf die mittlerweile schon sehr gewohnten Fenster und standardisierten Bedienelemente mit all ihren mitgegebenen Eigenschaften nicht zurückgreifen.
Wir müssen uns etwas anderes überlegen:

Wir können unsere GUI aus zwei getrennten Teilen zusammensetzen.
Zum einen aus der sichtbaren GUI, also das was auf dem Display dargestellt wird, die allerdings nur eine einfache Graphik ist.
Zum anderen aus der Funktionalen GUI, die nicht sichtbar ist aber die die kompletten Funktionen der GUI zur Verfügung stellt.

Die sichtbare GUI stellen wir uns der Einfachheit halber als eine einzige Graphik vor. Tatsächlich jedoch setzen wir sie aus einzelnen Graphiken zu einer Graphik zusammen:

Jedes Bedienelement ist eine eigene Graphik, die entweder als solche vorliegt und in die sichtbare GUI eingefügt wird oder aber mit Hilfe von Zeichenoperationen dort gezeichnet wird.

Sehr wichtig ist hier die Positionierung der „Bedienelemente“ (genauer: Welche Fläche sie belegen).
Denn wir brauchen diese Koordinaten später für den funktionalen Teil der GUI.
Wir müssen hier mit absoluter Positionierung arbeiten, da wir keinen Layout Manager haben.
D.h.wir müssen uns selber darum kümmern unter welcher Auflösung wir arbeiten und unsere GUI entsprechend anpassen.
Dies geschieht indem die einzelnen Graphiken in verschiedenen Größen vorliegen (oder auf die gebrauchte Größe umgerechnet werden), bzw.die Zeichenoperationen anhand der Auflösung berechnen wie sie zeichnen.

Wollen wir nun das Layout ändern, so setzen wir die sichtbare GUI neu aus Graphiken zusammen oder zeichnen neue um das gewünschte Bild zu erhalten.
Man könnte auch so die Farbpalette ändern das ein anderes Layout entsteht. Dazu sind aber sehr sorgfältige Überlegungen bei der Farbauswahl zu treffen um brauchbare Ergebnisse zu erzielen.

Die funktionale GUI wird nun folgendermaßen aufgebaut:

Wir benötigen eine abfrage Routine für Maus und Tastatur.
Wird eine Maustaste gedrückt, so werden die Mauskoordinaten abgefragt. Liegen die Mauskoordinaten innerhalb der Koordinaten eines Bedienelementes, so wird die entsprechende Funktion aufgerufen.

Bei jeder Veränderung der sichtbaren GUI die Größe oder Position eines der „Bedienelemente“ betrifft ist die Koordinatenabfrage der funktionalen GUI entsprechend anzupassen.

Die „objektorientierte Methode“

Arbeiten wir in einer objektorientierten Umgebung in der auch Bedienelemente als Objekte zur Verfügung stehen und diese mit eigenen Ereignishandlern und Zeichenmethoden ausgerüstet sind, so haben wir den Vorteil uns nur um das Aussehen und die Positionierung der Objekte kümmern zu müssen.

Je nachdem ob die Objekte Funktionen bieten ihr Aussehen und ihre Position zur Laufzeit zu ändern oder ob dies nur bei ihrer Erzeugung möglich ist haben wir zwei Möglichkeiten:

Die einfachere von beiden ist wenn wir über Funktionen Aussehen und Position zur Laufzeit ändern können:
Mittels dieser Funktionen nehmen wir nun bei jedem Ändern des Look & Feel Einfluss auf die Objekte und passen diese unseren Vorstellungen an. Um den Rest lassen wir uns den Objekthandler und den verwendeten Layout Manager kümmern.

Stellt uns die Umgebung diese Einflussnahme auf die Objekte nicht zur Verfügung so könnten wir die Objekte dementsprechend erweitern das sie diese Funktionalität bieten, was allerdings sehr aufwendig würde da eine große Zahl an Regelungen beachtet werden muss.
Es bietet sich eine andere Lösung an:

Wir erschaffen im Hintergrund die GUI komplett neu und zwar so wie sie nach dem Switchen aussehen soll.
anschließend kopieren wir alle relevanten Daten der alten GUI Objekte in die neuen (z.B. Inhalt einer Eingabemaske), zerstören die alte GUI und wechseln zu neuen GUI.
Hierfür muss auch der Hauptframe ein Objekt sein, da sonst bei dessen Zerstörung das Programm beendet würde und unser neuer Frame gleich mit zerstört würde.

Vor- und Nachteile der Methoden

Unsere „klassische Methode“ mag zwar umständlich, veraltet und schwer zu Programmieren sein aufgrund der vielen Koordinaten die man berücksichtigen muss, doch sie ist sehr ressourcenschonend.

Die „einfache objektorientierte Methode“ ist am einfachsten zu Programmieren, bietet je nach Umgebung keine bis alle Features die wir brauchen, ist aber sehr Resourcen fressend.
Dies liegt an dem internen hohen Verwaltungsaufwand für die Objekte in der Graphischen Umgebung.

Die „erweiterte objektorientierte Methode“, bei welcher wir die GUI neu schaffen und die alte zerstören, hat die selben Eigenschaften wie die „Einfache objektorientierte Methode“. Je nach Umgebung vielleicht ein wenig weniger Verwaltungsaufwand.
Hinzu kommt noch während des switchen zwischen zwei Look & Feel ein sehr rechenintensiver Moment hinzu, bedingt durch das neu Erstellen der gesamten Objekte.

Speichern der Look & Feels

Da wir unsere Look & Feels nicht auf wenige Vorgaben beschränken wollen sondern es erweiterbar und flexibel halten wollen, empfiehlt es sich nicht diese direkt im Programm zu implementieren.
Besser ist wenn wir im Programm eine Schnittstelle schaffen die Look & Feel Beschreibungen einlesen und interpretieren kann.
Dazu benötigen wir eine Beschreibungssprache die Programm extern alles relevante für ein Look & Feel beschreiben kann sowie ein passendes Format um diese zu speichern.
Programm intern muss dieses Format gelesen werden können so dass der Interpreter dann anfangen kann die Beschreibung abzuarbeiten.

Am einfachsten lässt sich so eine Beschreibung wohl in einer Textdatei speichern.
Wählt man die Beschreibungssprache jetzt noch so das sie leicht verständlich und schreib/lesbar ist, so können wir einfach durch das editieren der Textdatei neue Look & Feels generieren.

Enthalten sein müssen alle notwendigen Beschreibungen für alle Objekte .
Man könnte Objekte auch zu Gruppen zusammenfassen, so dass man nicht alle einzeln beschreiben sondern Einstellungen für die ganze Gruppe machen kann.

Das Entwurfsmuster Visitor

Eine Ausarbeitung

von

Florian F. Hoffmann

Matrikel Nr. 639426

Fachhochschule
Gießen-Friedberg

Ursprung und Zweck von Entwurfsmustern

Der Ursprung, bzw. das Entstehen, der ersten Entwurfsmuster liegt mehrere hundert Jahre zurück. Sie entwickelten sich im Bereich der Architektur.

Ausschlaggebend hierzu war vermutlich das Auftreten immer wieder gleicher Problematiken und Konstruktionsprinzipien um diese zu lösen.

Beachtet werden mussten und müssen bei solchen Planungen Zweckmäßigkeit, Ästhetik, Farbgestaltung, perspektivischer Eindruck, Harmonie mit der Umwelt, Lichteinfall, Wärmehaushalt, Einsatz von Materialien, Wege der Bewohner, usw.

Doch auch Lösungen für schwierige Probleme waren zum Teil ein wohlgehütetes Geheimnis, das nur innerhalb von Gilden und Logen weitergegeben wurde.
Vorstellbar ist das sich innerhalb dieser die ersten Entwurfsmuster bildeten um diese Geheimnisse zu Hüten und zu Pflegen.

Ein gutes Beispiel hierfür dürfte die Freimaurer Gilde sein, ein Zusammenschluss von Steinmetzen/Freimaurern, bekannt durch Ihre sehr hohen Fertigkeiten z.B. bei dem Bau von Kirchen.
Die Freimaurer waren aufgrund dieser Fertigkeiten mit besonderen Befugnissen in ganz Europa ausgestattet, so war es ihnen auch gestattet frei durch Europa zu Reisen was damals nicht selbstverständlich war.

Durch die verschiedenen Sprachen war es vermutlich noch mehr als mit einer gemeinsamen Sprache erforderlich eine „Fachsprache“ zu entwickeln mit der man sich verständigte.
Zusammengefasst wurden die meisten dieser ursprünglichen Regeln von Christopher Alexander in seinem Werk „A Pattern Language“.

Die Idee der Entwurfsmuster wurde von Gamma, Helm, Johnson und Vlissides aufgegriffen und in ihrem Buch „Design Patterns“ das erste mal für den Bereich der Informatik verwendet.
Der Gedanke liegt nicht fern, wenn man bedenkt das man es ja mit einer ähnlichen Problematik wie in der Architektur zu tun hat.
Es ist zwar jedes Problem ein anderes, doch sie sind nicht so verschieden das man die Erfahrungen aus anderen nicht dafür nutzen könnte, wenn auch leicht abgewandelt.

Allgemeine Aussagen zu Entwurfsmustern

Die oben schon genannte Grundidee der Entwurfsmuster liegt darin Erfahrungen aus schon gelösten Problematiken auf andere anzuwenden, also auch auf die Lösungen erfahrener Entwickler zurückgreifen zu können. Ebenso geht es darum eine gemeinsame Sprache zu haben, mit der man über die Probleme und die Lösungstaktiken reden kann.

Ein Entwurfsmuster könnte man eigentlich als „Sprachunabhängige Beschreibung der Lösung immer wiederkehrender Problematiken“ beschreiben.

Zur Lösung des Problems wird eine Mikroarchitektur und ihr Zusammenspiel genauestens beschrieben, d.h.die Klassen, Verteilung der Aufgaben und das Zusammenspiel zwischen Klassen und Objekten.

Da nicht jedes Problem gleich beschaffen ist müssen diese Musterlösungen sehr allgemein gehalten sein, flexibel um an das Problem angepasst und entsprechend erweitert werden zu können.
Selten treffen wir nur auf ein Problem, d.h.die Muster sollten auch gut miteinander kombiniert werden können.

Um komplexe Probleme zu lösen versucht man durch ein intensives zusammenspielen mehrerer Klassen die Teilaufgaben erledigen die Komplexität zu senken.
Meist ist eine Klasse nur für eine einzelne Aufgabe zuständig.
Man bevorzugt Delegation gegenüber Vererbung.

Das führt zu leichtgewichtigen Klassen, die sehr eng miteinander zusammenarbeiten. Gruppen von ihnen könnte man auch als einen Baustein betrachten und nicht als viele, so eng ist die Zusammenarbeit.

Desweiteren achtet man auf eine Trennung zwischen Schnittstelle und Implementierung, versucht die Objektkomponenten zu kapseln und nur über Methoden den Zugriff auf sie zu gestatten.
Dies ist nicht unüblich bei komplexeren Strukturen... man will dem Programmierer nicht die Möglichkeit geben direkt auf sie zuzugreifen sondern nur über abgesicherte Methoden deren Funktion gewährleistet ist.

Das Entwurfsmuster Visitor

Ausgangspunkt für dieses Muster ist eine große Datenstruktur die auf die verschiedensten Arten durchlaufen und abgearbeitet werden muss.
Die Funktionen hierfür ließen sich ohne größere Probleme in den Klassen der Datenstruktur unterbringen. Doch diese werden dadurch sehr schnell unübersichtlich und für jede neue Funktionalität müssten alle Klassen geändert werden.

Die Überlegung die hinter dem Visitor Muster steckt ist nun die Funktionalität komplett von der Datenstruktur zu trennen, zusammengehörende Funktionen zusammenzufassen und diese dann von der Datenstruktur aufrufen zu lassen.

Die zusammengefassten Funktionen bilden die Visitor-Objekte, die aufrufenden Methoden in der Datenstruktur werden akzeptierende Methoden genannt.
Für jede Funktion auf jeder Datenstruktur existiert ein konkreter Visitor, sowie ein abstrakter Visitor der allen gemein ist. Die konkreten Visitore überschreiben, je nachdem welcher gebraucht wird, dann die Visitor Methode des abstrakten Visitor.

Ruft nun eine konkrete Datenstruktur einen ihrer konkreteten Visitore auf, so wird zunächst der abstrakte Visitor aufgerufen, dann dessen Visitor Methode von dem zutreffenden konkreten Visitor überschrieben und dieser „besucht“ dann die Datenstruktur.
Der konkrete Visitor arbeitet dann die Datenstruktur ab und liefert das gesuchte Ergebnis an die akzeptierende Methode zurück die es dann an die aufrufende Funktion durchreicht.

Wie dieses Zusammenspiel funktioniert ist auf den Abbildungen der nächsten Seite sehr schön zu erkennen:
In Abbildung 1. sieht man wie eine Funktion (Client) auf Elemente einer Objektstruktur zugreift, dies kann geschehen indem er direkt den passenden Visitor anspricht und auf die Obektstruktur loslässt oder aber er ruft ihn über die Objektstruktur selber auf, bzw. lässt ihn von dieser aufrufen.

In Abbildung 2. sieht man nochmal das Zusammenspiel der einzelnen Klassen und Elemente.

Das Visitor Pattern

- Struktur

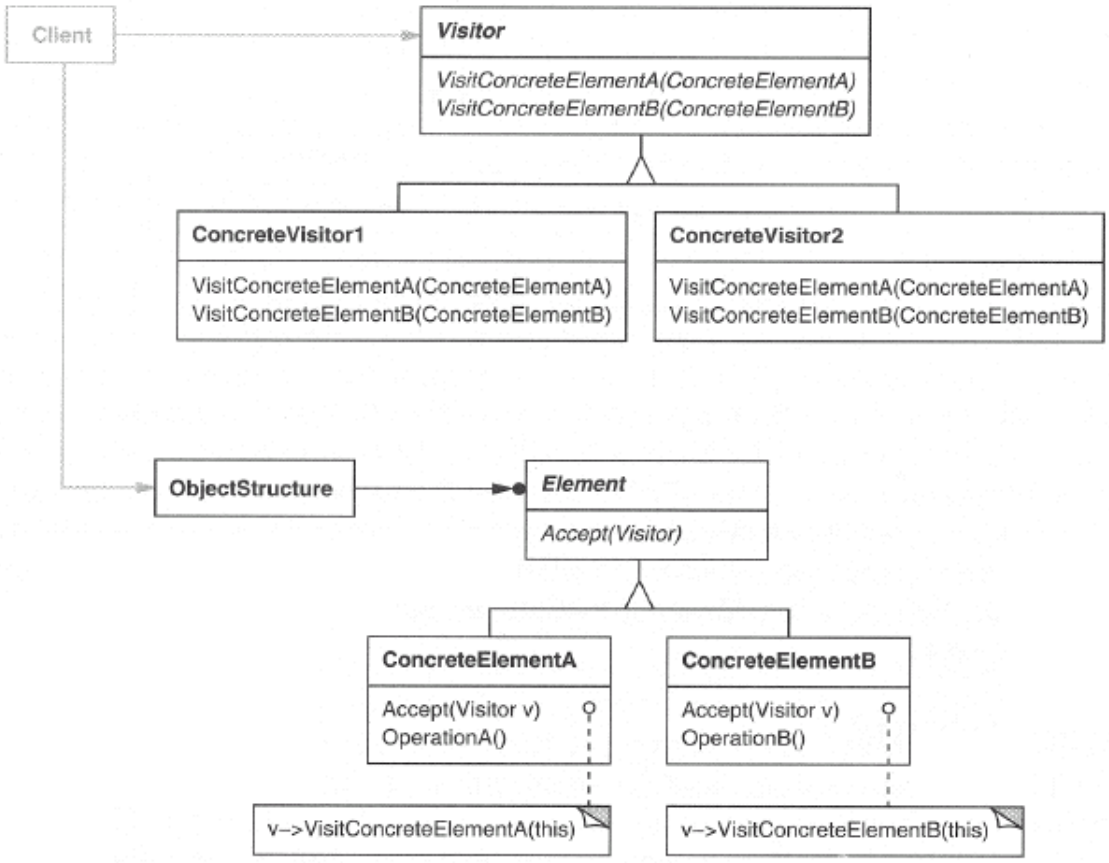


Abbildung 1.

- Zusammenarbeit

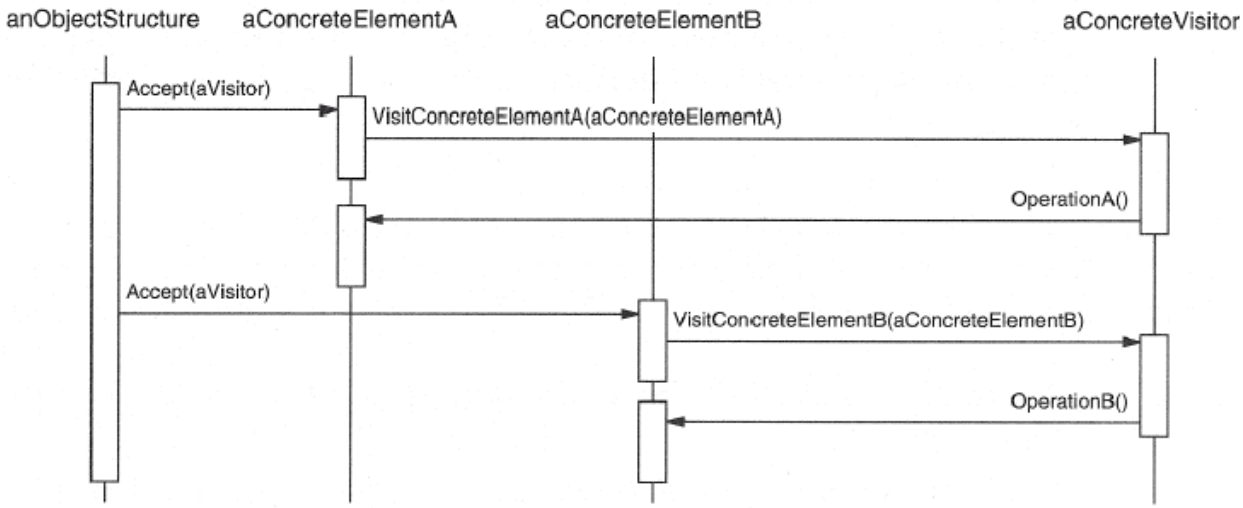


Abbildung 2.

Konsequenzen durch Anwendung des Visitor Musters

Durch das Zusammenfassen der zusammengehörigen Operationen in den Visitoren wird die Programmstruktur sehr übersichtlich. Dies bringt sowohl in der ersten Entwicklung wie auch in der weiteren Pflege des Programms Vorteile: Der Quellcode ist durch die bessere Übersichtlichkeit besser zu lesen und zu verstehen, man weiß eher was welche Funktion macht, dadurch auch bei der Programmierung weniger fehleranfällig.

Zudem ist es nun sehr leicht neue Operationen für eine Datenstruktur hinzuzufügen. Die Datenstruktur muss hierfür nicht verändert werden, es wird einfach ein neuer passender Visitor hinzugefügt.

Ein weiterer Vorteil: Ein Visitor der eine Datenstruktur durchläuft kann während dem Arbeiten Zustände der beteiligten Objekte sammeln und auswerten. Hierzu ist keine weitere Parameter Übergabe erforderlich.

Doch leider hat dieses Entwurfsmuster auch Nachteile, zumindest in bestimmten Situationen.

Wollen wir die Objektklassen der Datenstruktur verändern, oder neue hinzufügen, so müssen wir alle Visitor Methoden umschreiben. Das ist ein sehr großer Aufwand.
Daher sollte man sich vor dem Einsatz des Visitor Musters die folgende Frage stellen:
„Werden häufiger neue Operationen oder neue Klassen eingeführt?“
Bei ersterem ist ein Einsatz des Visitor Musters sehr empfehlenswert, da wie oben beschrieben das Einfügen neuer Operationen sehr einfach ist.
Bei letzterem sollte man besser auf den Einsatz des Visitor Musters verzichten und andere Lösungen ins Auge fassen um sich unnötige Arbeit zu ersparen.

Ein weiterer Nachteil ist die Verletzung der Kapselung der Datenelemente der Datenstruktur.
Um den Visitor Methoden zu ermöglichen auf diesen zu arbeiten müssen wir sie „freigeben“, d.h.der direkte Zugriff auf sie muss durch den Visitor möglich sein.
Normalerweise werden Datenobjekte gekapselt, ein direkter Zugriff auf sie ist nur über Methoden ihrer Klasse möglich welche üblicherweise gut abgesichert sind so das die Daten gegen Fehler geschützt sind.
Da unser Visitor sich aber in einer ganz anderen Klasse befindet und die Zugriffsmethoden in ihm implementiert sind ist dies nicht möglich.
Dies ist erstmal nicht schlimm, solange man bei der Implementierung der Visitore darauf achtet was man auf der Datenstruktur tut und die Zugriffsmethoden dort gut absichert/implementiert.
Ein Programmierer könnte nun allerdings, je nachdem ob es die Programmiersprache zulässt die Datenstruktur nur für die Visitore zugänglich zu machen oder nicht, auch direkt auf die Datenstruktur zugreifen.
Eigentlich ein schlechter Programmierstil die Datenstruktur so offen zu lassen, doch für das Anwenden des Visitor Musters unerlässlich.

Implementierung des Visitor Musters

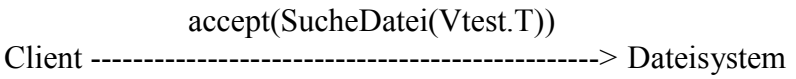
Implementieren wir das Visitor Muster, so besitzt jede Datenstruktur eine (abstrakte) Visitor Klasse. Diese Klasse besitzt für jede Klasse der Datenstruktur eine Visit Methode die den Typ der Datenstruktur als Parameter enthält.
Die konkreten Visitor Klassen (Subklassen) redefinieren(überschreiben) die Visit Methoden um ihre Funktionalität zu realisieren.

Die Datenstruktur enthält eine aufrufende Funktion die den entsprechenden konkreten Visitor über den (abstrakten) Visitor aufruft.

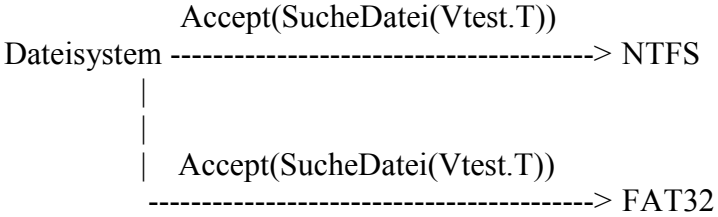
Nehmen wir ein Beispiel um uns die Arbeitsweise des Visitor Musters einmal zu verdeutlichen:

Wir haben ein Betriebssystem das auf unterschiedlichen Dateisystemen arbeitet (NTFS und FAT32).
Wir wollen nun diese Dateisysteme durchsuchen nach einer Datei Vtest.T

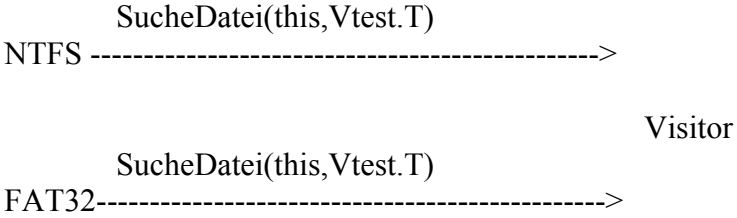
Die Dateisysteme sind unsere Datenstruktur, unser Suchprogramm der Visitor.
Zunächst stellen wir eine Anfrage an die Datenstruktur und was sie uns ausgeben soll, hierbei teilen wir der Datenstruktur mit welchen Visitor wir benutzen wollen und welche Parameter wir dem Visitor geben:



Die Datenstruktur Dateisystem gibt diese Anfrage nun an die tatsächlichen Dateisystem weiter:



Diese rufen nun den Visitor auf:



Der (abstrakte) Visitor „sieht“ welcher Objekttyp übergeben wird und delegiert nun an die entsprechenden konkreten Visitor Objekte, welche dann auf die Datenstruktur zugreifen, diese bearbeiten und dann die Ergebnisse zurückliefern.
In unserem Beispiel wird die Datei gefunden und der Visitor hat sich einige Zustände gemerkt, nämlich wie lange er dafür gebraucht hat und wie viele Dateien er durchlaufen hat bevor er sie gefunden hat.

Insbesondere dann wenn es darum geht Baum- oder ähnliche Strukturen zu durchlaufen ist ein Visitor sehr von Vorteil.
Eine der bekanntesten Anwendungen in der das Muster genutzt wird ist der Compiler des JDK ab Version 1.3

Verwendete Literatur:

1)
Go To Java 2, zweite Auflage

2)
Skript zu „Seminar Fortgeschrittene Programmiermethoden / Iterator und Visitor“, Markus Heintel und Anton Röckreisen, Fachhochschule München

3)
Skript zu „Vorlesung Softwaretechnologie WS 2001“ Universität Bonn

Die verwendeten Zeichnungen (Abbildung 1. und 2.) wurden aus 3) entnommen.

Vergleich zwischen Java Swing und Windows Forms

Ausarbeitung zum großen Seminar im SS02 [1] [2]

Adam Kreuschner, Matr.Nr.: 637091

19 Juni 2002

[1] Professoren: Prof. Dr. Renz, Prof. Dr. Henrich
[2] Erstellt mit LaTeX

Inhaltsverzeichnis

1 Einleitung

Die Programmierung von grafischen Oberflächen ist heutzutage eine der wichtigsten Aspekte beim Erstellen von Schnittstellen zwischen dem zugrundeligendem System und dem Benutzer. Die Anforderungen an die aktuellen GUI-Bibliotheken und Frameworks sind dabei sehr vielfältig. Im Folgenden wird **Java Swing**, die GUI-Bibliothek des JDK's von Sun Microsystems, und **Windows Forms**, die Klassen-Sammlung zur GUI-Progrmmierung des .NET Frameworks von Microsoft vergliechen. Das Ziel ist es zu klären inwiefern sich die beiden Technologien ähneln oder bei Welchen Ansätzen sie sich unterscheiden und welche Vorteile bzw. Nachteile sie mit sich bringen. Beide Klassen-Bibliotheken sind jeweils Bestandteil eines grossen Frameworks, mit dem sie direkt oder indirekt interagieren. Deshalb bietet dieses Dokument auch einen Blick auf die jeweils zugrundeliegenden Plattformen.

2 Sun Microsystems Java Swing

2.1 Die Designziele

Java wurde von Anfang an mit dem Anspruch entwickelt, ein vielseitiges, aber einfach zu bedienendes System für die Gestaltung grafischer Oberflächen zur Verfügung zu stellen. Das Resultat dieser Bemühungen steht seit dem JDK 1.0 als Grafikbibliothek unter dem Namen Abstract Windowing Toolkit (AWT) zur Verfügung. Kurz nachdem mit dem JDK 1.0 AWT eingeführt wurde, wurde bei SUN begonnen, über Möglichkeiten nachzudenken, wie man die Grafikausgabe von Java-Anwendungen verbessern könnte, denn folgende Eigenschaften des AWT wurden als nachteilig angesehen:

- Alle Fenster- und Dialogelemente des AWT werden von dem darunterliegenden Betriebssystem zur Verfügung gestellt, was es sehr schwierig machte, plattformübergreifend ein einheitliches Look&Feel zu realisieren. Der Anwender spürt bei AWT Anwendungen unmittelbar die Eigenarten jedes einzelnen Fenstersystems.
- Die Abhängigkeit von den betriebssystemspezifischen Komponenten führte zu erhebliche Portierungsprobleme des JDK. Das bedeutete teilweise einen großen Aufwand für die Angleichung des Verhalten der Komponenten auf den unterschiedlichen Plattformen.
- AWT bietet nur eine Grundmenge an Dialogelementen, mit denen sich aufwendige grafische Benutzeroberflächen nicht oder nur mit sehr viel Zusatzaufwand realisieren lassen.

Die als Swing in Java 2 eingeführte Bibliothek der grafischen Komponenten, ist eine interessante Alternative, zu den bis dahin verwendeten standard AWT-Komponenten. Sie bringt als sinnvolle Ergänzung der AWT-Elemente viele Vorteile mit sich. Das Hauptziel des Swing Projektes war es, eine erweiterbare GUI-Bibliothek zu erstellen, um den Programmierern die Möglichkeit zu bieten, mächtige Java Frontends für Applikationen schnell erstellen zu können. Deshalb wurden von Anfang an folgende Designziele festgesetzt:

- Implementierung erfolgt in Java um Plattformunabhängigkeit und einfache Wartung zu erreichen
- Bereitstellung einer einzigen API, die multiples Look&Feel beherrscht, so dass Programmierer und Benutzer nicht von einem einzigen, plattformabhängigem Look&Feel abhängig sind.
- Die Möglichkeit modellgesteuerte Programmierung zu betreiben
- Einhaltung der Java Beans Designprinzipien, um angemessenes Verhalten der Komponenten in Entwicklungsumgebungen und Builder Werkzeugen zu garantieren.
- Einhaltung der Kompatibilität mit den AWT APIs, um die Wissensbasis über AWT nutzen zu können und die Portierung zu vereinfachen.

Dem stehen relativ geringfügige Geschwindigkeitsverluste gegenüber, die hauptsächlich aus der eigenständigen Modellierung der Verhaltensweisen der Elemente resultiert. Die Swing-Bibliothek erlaubt, bei Verständnis der internen Arbeitsweise und der Einhaltung einiger Regeln, das Mischen von AWT und Swing- Komponenten innerhalb einer Container-Klasse.

2.2 Die Architektur

2.2.1 Java 2 JDK 1.4

Das Java 2 Standard Edition JDK (J2SE) besteht aus dem Java Compiler und der Runtime-Umgebung sowie einer umfangreichen Bibliothek von mehr als 1500 Klassen. Dadurch steht dem Entwickler die Standard-API für die Anwendungsentwicklung zur Verfügung. Das Hauptziel der Java Umgebung ist die Plattformenabhängigkeit, so dass die Funktionen der Basisbibliothek, beispielsweise für Dateizugriff, Netzwerkprogrammierung und Threads zusammen eine betriebssystemunabhängige Schnittstelle bilden.

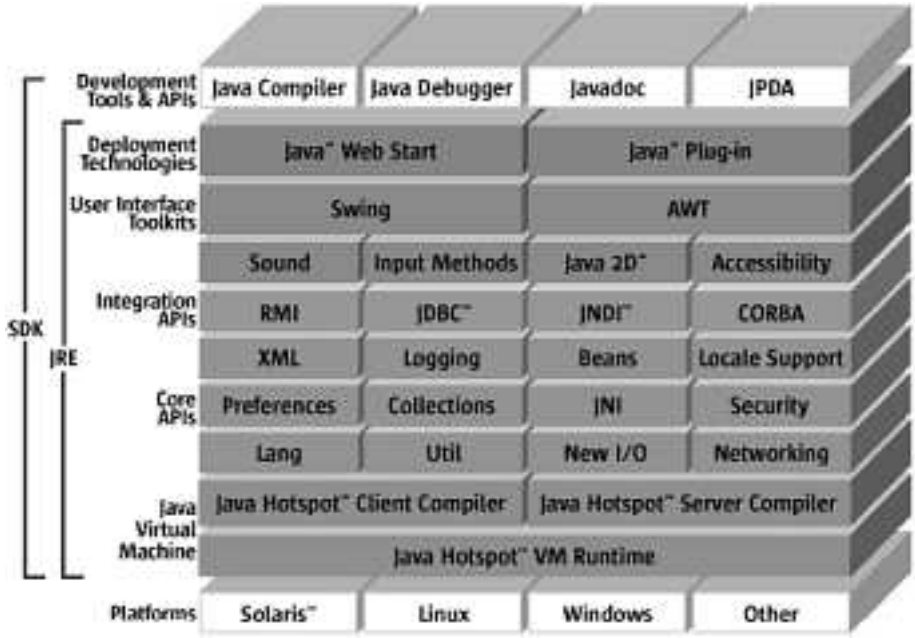


Abbildung 1: Die Java JDK 1.4 Architektur

Die Lösung, unabhängig von der Hardware Architektur zu sein, liegt in dem binären Code Format, das der Java-Compiler erzeugt. Ist die Java Runtime-Umgebung für eine Plattform verfügbar, kann jede, in Java geschriebene Applikation ohne Neukompilierung oder Anpassung, von dieser ausgeführt werden. Der Java Compiler erzeugt somit keinen Maschinencode, im Sinne von Hardware Instruktionen, sondern einen Maschinen unabhängigen Bytecode für die virtuelle Maschine (VM). Die VM wir durch die Runtime-Umgebung und dem Just in time Compiler (JIT) repräsentiert. Der Java Bytecode wurde entwickelt, um leicht auf jeder gängigen Maschine interpretiert oder aus Performanz Gründen, dynamisch in echten Maschinencode übersetzt werden zu können.

Der Garbage Collector ist dafür zuständig, Resourcen, die von nicht mehr verwendeten Objekten belegt werden, wieder freizugeben.

Das in der Standard Edition enthaltene Swing-Framework eignet sich für die Erstel-

lung plattformübergreifender GUI-Applikationen.

Die Java Beans stellen ein Komponentenmodell ähnlich dem ActiveX von Microsoft zur Verfügung; solche Software-Komponenten lassen sich in grafischen IDEs einfach zu neuen Anwendungen "zusammenstöpseln".

Für verteilte Systeme steht mit Remote Method Invocation (RMI) ein Mechanismus für entfernte Funktionsaufrufe zur Verfügung. Java-Applikationen können mittels RMI über das Netzwerk auf Objekte einer anderen Virtual Machine zugreifen. Zusätlich enthält J2SE eine Corba-Implementierung samt ORB. Das Java Naming and Directory Interface (JNDI) erlaubt den Zugriff auf Verzeichnisdienste wie LDAP.

Mit JDBC steht eine herstellerunabhängige Datenbankschnittstelle, die mit jedem Datenbankserver, für den ein entsprechender Java-Treiber verfügbar ist, kommunizieren kann.

2.2.2 Java Swing

Die Architektur von Swing hat ihre Wurzeln in dem *Model-View-Controller* (MVC) Muster, das aus der Zeit der Programmiersprache SmallTalk stammt. Die MVC Architektur verlangt, dass die grafikbasierte Applikation in drei Teile aufgespalten ist:

- **Das Modell** *(engl. model)*
  Repräsentiert die Daten der Applikation.
- **Die Ansicht** *(engl. view)*
  Die visuelle Repräsentation der Daten.
- **Die Steuerungskomponente** *(engl. controller)*
  Nimmt die Eingaben des Benutzers über die Ansicht, also die grafische Oberfläche der Applikation, entgegen, und übersetzt sie in Änderungen an dem Modell, also den Daten der Applikation. Ein Mechanismus zur Benachrichtigung über die Änderungen sichert dabei die Konsistenz zwischen der Benutzerschnittstelle und dem Modell.

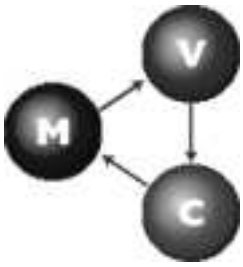


Abbildung 2: Das MVC Muster

Die Verwendung des MVC-Musters war der logische Schritt bei der Implementierung von Swing, schließlich erlaubt dies die ersten drei Desingziele im Rahmen der letzteren zwei zu verwirklichen. Tatsächlich folgte der erste Prototyp von Swing dem

MVC-Muster Prinzip. Jede Komponente hatte ihr eigenes Modell und delegierte ihre Look&Feel Implementierung an separate Ansicht- und Steuerungskomponenten-Objekte.

Schnell stellte sich jedoch heraus, dass diese Trennung im praktischen Einsatz Probleme mit sich bringt. Die Sicht und die Steuerung einer Komponente müssen miteinander fest gekoppelt sein, sonst ist es schwierig eine allgemeine Steuerungskomponente zu implementieren, die keine genauen Angaben über die Ansicht hat. Das MVC-Muster unterbindet diesen Sachverhalt jedoch. Daraufhin wurden diese beiden Entitäten in einem Objekt zusammengefasst. Daraus resultierte die endgültige Swing Architektur, die im folgenden Diagramm dargestellt ist:

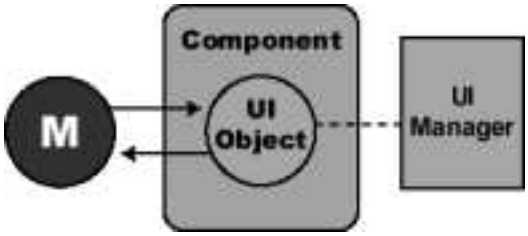


Abbildung 3: separable model architecture

Das Diagramm zeigt auf, dass die Swing Architektur sich sehr frei nach dem MVC-Muster richtet, dieses jedoch nicht streng befolgt. In der Welt von Swing wird dieses neue *quasi-MVC Muster* oft als "*trennbare Modell Architektur*" (engl. separable model architecture) bezeichnet. Die trennbare Modell Architektur von Swing behandelt das Modell einer Komponente als ein separates Element, wie es auch beim MVC-Muster der Fall ist, jedoch wird die Sicht und die Steuerungseinheit jeder Komponente in einem einzelnen User-Interface Objekt zusammengefasst.

Das User Interface-Objekt wird im Zusammenhang mit dem "anhängbarem" Look&Feel *(engl. pluggable look&feel)* oft auch als *UI delegate* (zu Deutsch: Benutzerschnittstellen Delegation) bezeichnet. Man sollte beachten, dass die Verantwortlichkeit für die Steuerungskomponente und die Sicht bei einer allgemeinen Swing-Komponente wie z.B. JButton, JTree usw. liegt. Diese Komponente delegiert die Look&Feel spezifischen Aspekte der Verantwortlichkeit an das User-Interface Objekt *(UI delegate)*, das von dem gerade verwendeten Look&Feel zur Verfügung gestellt wird.

3 Microsoft Windows Forms

3.1 Die Designziele

Windows Forms ist Bestandteil des .NET Frameworks von Microsoft. Es ist eine Grundstruktur zum Erstellen von Windows-Clientanwendungen, die die Common Language Runtime des Frameworks verwenden. Die folgenden Designziele schaffen eine Übersicht über die gebotenen Möglichkeiten bei der Programmierung von grafischen Oberflächen mit Windows Forms:

- **Einfachheit und Leistungsfähigkeit**
  Windows Forms ist ein Programmiermodell für die Entwicklung von Windows-Anwendungen, das den einfachen Aufbau des Programmiermodells von Visual Basic 6.0 mit der Leistungsfähigkeit und Flexibilität der .NET Common Language Runtime verbindet.

- **Verringerte Gesamtkosten für den Besitzer**
  Es werden die Features zur Versionserstellung und Weitergabe der Common Language Runtime genutzt, um auf lange Sicht die Weitergabekosten zu verringern und die Zuverlässigkeit von Anwendungen zu verbessern. Auf diese Weise werden die Verwaltungskosten für in Windows Forms geschriebene Anwendungen beträchtlich reduziert.

- **Architektur für Steuerelemente**
  Windows Forms bietet eine Architektur für Steuerelemente und Steuerelementcontainer, die auf der konkreten Implementierung des Steuerelements und der Containerklassen beruht. Dadurch werden Probleme beim Zusammenwirken von Steuerelementcontainern erheblich verringert.

- **Sicherheit**
  Es werden alle Sicherheitsfeatures der Common Language Runtime unterstützt. Das heißt, dass mit Windows Forms von im Browser ausgeführten, nicht vertrauenswürdigen Steuerelementen bis zu vollständig vertrauenswürdigen Anwendungen, die auf der Festplatte eines Benutzers installiert sind, alles implementiert werden kann.

- **Unterstützung für XML-Webdienste**
  Windows Forms bietet volle Unterstützung für schnelle und unkomplizierte Verbindungen mit XML-Webdiensten.

- **Umfangreiche Grafikfunktionen**
  Windows Forms ist eine der ersten lieferbaren Komponenten für die Arbeit mit GDI+. Bei GDI+ handelt es sich um eine neue Version von Windows GDI (Graphical Device Interface), die Alphablending, Strukturpinsel, erweiterte Transformationen, RTF (Rich Text Format) und andere Features unterstützt.

- **Flexible Steuerelemente**
  Windows Forms bietet eine große Auswahl an Steuerelementen, die u. a. sämtliche von Windows angebotenen Steuerelemente mit einschließt. Diese Steuerelemente bieten auch neue Features, wie etwa zweidimensionale Darstellung von Schaltflächen, Optionsfeldern und Kontrollkästchen.

- **Datenerkennung**
  Windows Forms bietet volle Unterstützung für das ADO.NET-Datenmodell. Dadurch lassen sich in den Applikationen Datenbindungen zwischen Steuerelemnten und verschiedenen Datenquellen realisieren.
- **Unterstützung für ActiveX-Steuerelemente**
  Es wir auch volle Unterstützung für ActiveX-Steuerelemente geboten. Man kann ohne Weiteres ActiveX-Steuerelemente in eine Windows Forms-Anwendung aufnehmen, aber auch ein Windows Forms-Steuerelement als ActiveX-Steuerelement benutzen.
- **Lizenzierung**
  Windows Forms nutzt das erweiterte Lizenzierungsmodell der Common Language Runtime.
- **Eingabehilfen**
  Windows Forms-Steuerelemente implementieren die von MSAA (Microsoft Active Accessibility) definierten Schnittstellen, die das Erstellen von Anwendungen erleichtern, die Eingabehilfen (z. B. Wizards oder Sprachausgabe) unterstützen.
- **Unterstützung zur Entwurfszeit**
  Windows Forms verwendet die Features für Metadaten und Komponentenmodelle der Common Language Runtime, um Personen, die Steuerelemente verwenden oder implementieren, zur Entwurfszeit umfassende Unterstützung zu bieten.

3.2 Die Architektur

3.2.1 Das .Net Framework

Das .NET Framework ist eine Entwicklungsplattform für Anwendungen. Das Fundament bildet die Common Language Runtime (CLR). Code, der unter ihrer Regie läuft, wird “Managed Code“ genannt. So werden Aktionen wie das Anlegen von Objekten oder Methodenaufrufe nicht direkt ausgeführt, sondern von der Runtime-Umgebung erledigt. Dadurch ist die Möglichkeit gegeben, dass die Runtime zusätzliche Dienste, wie Versions- und Sicherheitsüberprüfungen durchführt.

Die Compiler des Frameworks erzeugen daher keinen nativen Code mehr, sondern übersetzen vielmehr den Quelltext in eine Zwischensprache, die bei Bedarf vom JIT (Just in time Compiler) zu nativem Code kompiliert und dann unter Aufsicht der Runtime ausgeführt wird. Diese mit *Microsoft Intermediate Language* (MSIL) bezeichnete Zwischensprache ist komplett dokumentiert und offengelegt, deshalb stehen Implementierungen weiterer Compiler durch Drittanbieter alle Weg offen. Microsoft selbst liefert das .NET Framework derzeit mit Visual Basic.NET -, Visual C++ - und C# - Compilern aus, wobei der C++ - Compiler nach wie vor in der Lage ist nativen Assembler Code zu erzeugen. Offensichtlich ist Dreh- und Angelpunkt der Runtime die Sprachintegration. Ob man nun COBOL, Pascal, C# oder Visual Basic benutzt, ist egal - solange der Compiler MSIL-Code erzeugt.

Auch hier ist der Garbage Collector verfügbar. Wie auch bei Java ist dieser dafür zuständig, Resourcen, die von nicht mehr verwendeten Objekten belegt werden, wieder freizugeben.

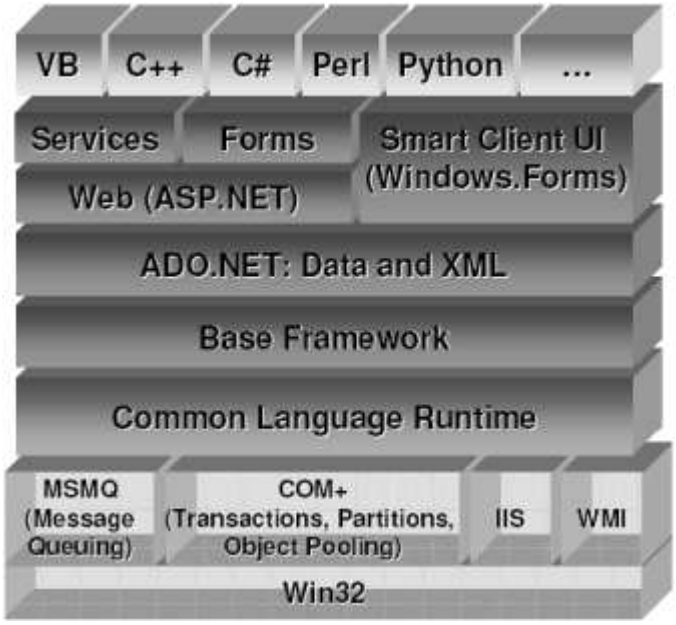


Abbildung 4: Die .Net Framework-Architektur

Das .NET Framework basiert nicht auf COM, vielmehr beschreiben sich die Komponenten unter .NET selbst. Dazu werden entsprechende Metadaten vom .NET-Compiler direkt in die Komponente geschrieben. Das Pflegen einer IDL-Datei parallel zum Quellcode wir dadurch überflüssig, und beim Verteilen einer Komponente braucht man keine Type-Library und keine Header-Dateien mehr mitzuliefern. Komponenten werden installiert, indem man sie einfach auf den Zielrechner kopiert-sie müssen sich nicht mehr in die Registry eintragen. Aber auch wenn .NET die COM-Schnittstelle eigentlich nicht mehr benötigt, arbeitet das Framework nahtlos mit COM-Komponenten zusammen. Man kann COM-Komponenten aus .NET-Komponenten heraus benutzen und umgekehrt.

Unter .NET ist es darüber hinaus auch möglich, mehrere Versionen derselben Komponente auf dem Rechner zu halten, wodurch auch die "DLL-Hölle" der Vergangenheit angehört. Programme, DLLs und Komponenten sind bei .NET unter dem Begriff "Package" zusammengefasst. Jedes Package enthält in seinen Metadaten auch Versionsinformationen, und die Runtime prüft beim Aufruf eines Packages, ob seine Version zum Client-Programm passt.

3.2.2 Windows Forms

Windows Forms unterstützen kein *"anhängbares"* Look&Feel, so präsentieren sich die Applikationen dem Benutzer in dem bereits bekannten Windows Aussehen. Dementsprechend ist die Architektur von Windows Forms einfach gehalten, denn im wesentlichen ist die Aufgabe der grafischen Benutzerschnittstelle Daten der Applikation zu repräsentieren, Datenwerte vom Benutzer entgegen zu nehmen sowie auf vom Benutzer ausgelöste Ereignisse zu reagieren. Diesem Sachverhalt entspricht das *Beobachter-*

Muster (engl. observer), das bei Windows Forms herangezogen wird. Dieses Muster ermöglicht es auf Änderungen des internen Zustand des beobachteten Objekts zu reagieren.

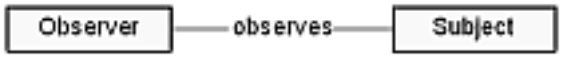


Abbildung 5: Das Observer-Muster

Das Muster besteht demnach aus den folgenden zwei Komponenten:

- **Beobachter** *(engl. observer)* Das Objekt, das den Zustand eines (oder mehrerer) anderen Objektes überwacht.
- **Subjekt** *(engl. subject)* Das vom Observer zu beobachtende Objekt, das die Daten der Applikation hält.

Dieses Muster ähnelt dem Model-View-Controller Pattern, denn das Subjekt gleicht der Modell-Komponente (engl. model) und das Beobachter Objekt ähnelt der Ansicht (engl. view) in MVC. Der Unterschied liegt in der Tatsache, dass der Beobachter die Ansicht und die Steuerungskomponente in sich vereinigt. Auf Windows Forms bezogen ist das Beobachter-Objekt also das Fenster zusammen mit allen grafischen Komponenten und den dazugehörigen Event - Handlern.

Das Muster besagt zwar, dass der Beobachter das Subjekt beobachtet, jedoch ist das, bezogen auf die Implementierung dieses Musters, eigentlich eine falsche Benennung des wirklichen Sachverhalts. In Wirklichkeit bekundet der Beobachter das Interesse über die inneren Änderungen des Subjekts informiert zu werden, indem es sich bei ihm registriert.

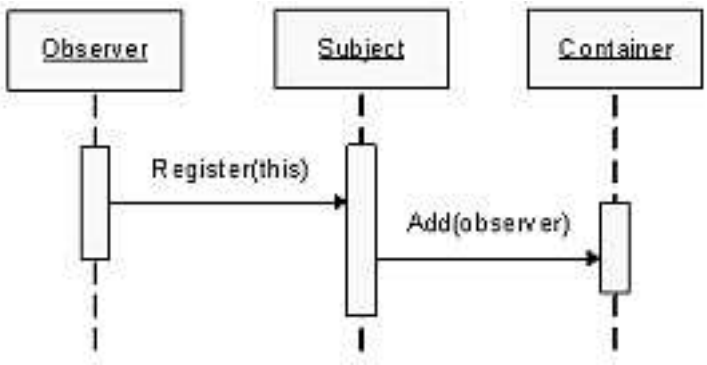


Abbildung 6: Die Registrierung des Observers

Die Registrierung erfolgt indem der Beobachter die Referenz auf sich selbst als Parameter der Register()-Methode übergibt. Diese Referenz wird jedoch nicht von Subjekt gespeichert (etwa in einer Membervariable), sondern an das Container-Objekt des Subjekts delegiert. Die Container-Klasse aller GUI-Elemente von Windows Forms ist die

Klasse *System.Windows.Forms.Control*. Analog dazu erfolgt die Kündigung der Registrierung.

Das Subjekt überprüft seinen Zustand regelmäßig und wenn eine Änderung festgestellt wird, leitet es die Benachrichtigungs-Sequenz in die Wege. Dabei werden die Referenzen aller registrierter Beobachter vom Container geholt und dann werden die Observer über die Änderung des inneren Zustands benachrichtigt. Diesen Sachverhalt spiegelt das folgende Diagramm wieder:

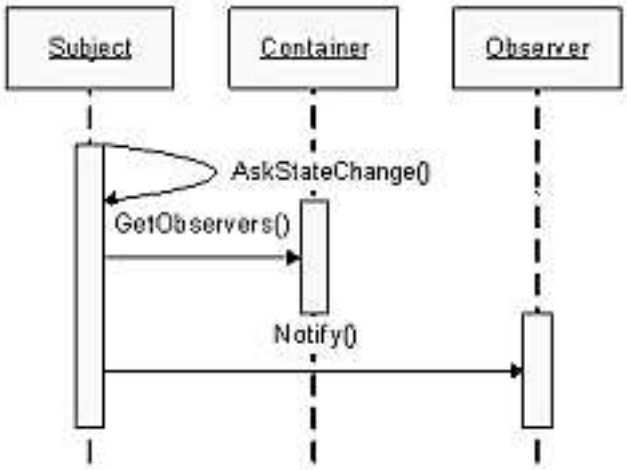


Abbildung 7: Benachrichtigungs-Sequenz

Auch wenn das .NET-Framework den Anspruch erhebt, plattformunabhängig zu sein, so ist Windows Forms tatsächlich die einzige Komponente des Frameworks, die nur von Windows Betriebsystemen unterstützt wird. Der Grund dafür liegt auf der Hand, denn Windows Forms Anwendungen greifen auf die Win32-API zu um die grafischen Elemente darzustellen. Die angepeilte Sprachintegration trifft jedoch auch auf diesen Teil des Frameworks zu - Ob man nun COBOL, Pascal, C# , C++ oder Visual Basic benutzt, wenn ein Compiler zur Verfügung steht, der MSIL-Code erzeugt, ist man in der Lage Windows Forms Anwendungen auch in dieser Programmiersprache zu Implementieren.

4 Anwendungsprogrammiermodell

4.1 Java Swing

Eine Swing Anwendung besteht meistens mindestens aus den vier folgenden Komponenten:

- **Rahmen** *(engl. frame)*
  Es ist das Hauptfenster, das durch die Klasse *JFrame* repräsentiert wird.
- **Panel**
  Diese Komponente wir manchmal auch *pane* genannt und wird durch die Klasse *JPanel* repräsentiert.
- **Button** Dem liegt die Klasse *JButton* zugrunde
- **Label** Es ist ein Objekt der Klasse *JLabel*

Ein Objekt der Klasse *JFrame* ist ein *Top-Level-Container* und bietet deshalb hauptsächlich Platz, um darin weitere Swing Komponenten zu zeichnen. *JFrame* wird herangezogen, um Hauptfenster zu erstellen. Ein weiterer Top-Level-Container wird durch die Klasse *JDialog* zur Verfügung gestellt, und ist als Rahmen für Dialog-Fenster vorgesehen.
Ein *JPanel-Objekt* ist ein *Zwischen-Container* (engl. intermediate container), der lediglich zur Vereinfachung der Positionierung von *atomaren Komponenten* (engl. atomic components) benutzt wird. Erweiterte *Zwischen-Container* wie Panel Flächen mit Scrollbalken *(JScrollPane)* und Reiter-Panels *(JTabbedPane)* spielen dabei eher eine visuelle bzw. interaktive Rolle.
Atomare Komponenten wie *JButton* oder *JLabel* sind Elemente von Swing, die keinen weiteren Swing Componenten aufnehmen können. Solche Komponenten erben von der Klasse *JComponent* und werden im Allgemeinen als Steuerelemente (engl. controls) bezeichnet. *atomaren Komponenten* existieren als unabhängige Entitäten, die dem Benutzer Informationen präsentieren oder von ihm entgegennehmen. Swing bietet viele solcher Komponenten wie Combo Boxen *(JComboBox)*, Text-Felder *(JTextField)* und Tabellen *(JTable)*.

4.1.1 Frame und Root Pane

Sogar das einfachste Swing Programm hat mehrere Ebenen in seiner Hierarchie. Die Wurzel der Hierarchie ist immer ein Top-Level-Container, denn dieser bietet den untergeordneten Swing Komponenten Fläche, auf der sie gezeichnet werden. Sinngemäß ist das eine Instanz der Klasse *JFrame*. Dialog-Fenster innerhalb der Applikation haben ein Objekt der Klasse *JDialog* als Wurzel ihrer Hierarchie.
Jeder Top-Level-Container beinhaltet indirekt einen Zwischen-Container, der unter dem Namen *content pane* bekannt ist. Dieser Container beinhaltet, direkt oder indirekt, alle visuellen Komponenten der grafischen Oberfläche der Anwendung. Die einzige Ausnahme bildet die Menüleiste der Anwendung - diese wird außerhalb des *content pane* an einer dafür vorgesehenen Stelle gehalten. *Content pane* ist tatsächlich nur ein Element von vier, die zusammengesetzt das *JRootPane* bilden. Generell instanziert man kein Objekt der Klasse *JRootPane*, vielmehr enthält jede Instanz eines Top-Level-Containers auch eine Instanz von *JRootPane*.

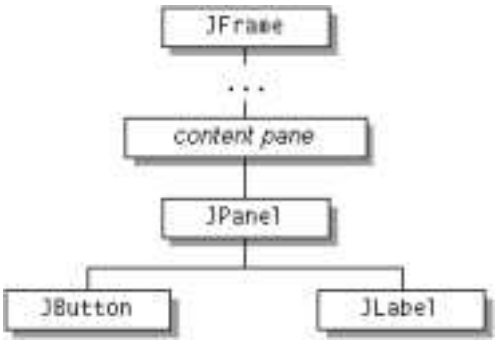


Abbildung 8: hierarchischer Aufbau eines Swing Hauptfensters

Die Weiteren drei Elemente von *JRootPane* sind:

- **Glass pane**
  Dieses Element ist standardmäßig versteckt. Wenn man es sichtbar macht, verhält es sich wie eine Glasscheibe über allen Teilen des *JRootPane*. In der Regel ist es durchsichtig, außer man implementiert die paint()-Methode so, dass diese etwas zeichnet. Darüberhinaus kann man auf der gesamten Fläche Ereignisse, die für den *JRootPane* bestimmt sind abfangen.
- **Layered pane**
  Innerhalb dieses Elemnts wird die Menü Leiste und *content pane* platziert. Es ist ein Container mit "Tiefe", so dass überlappende Elemente wie z.B. Popup Menüs im Vordergrund anderer Komponenten dargestellt werden können.
- **Optionale Menu Leiste**
  Der Platz für das Menü des Containers. Wenn der Container eine Menüleiste enthalten soll, wird diese mit der Methode *setJMenuBar* der richtigen Stelle des Containers zugewiesen.

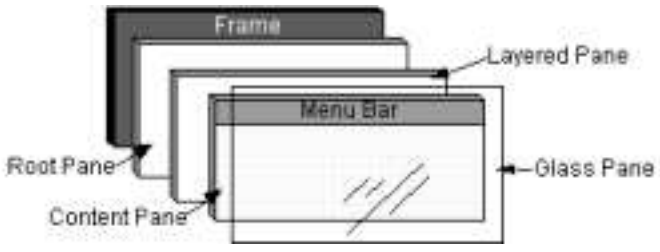


Abbildung 9: Aufbau von JRootPane

4.1.2 Steuerelemente

Java Swing stellt dem Programmierer alle üblichen Steuerelemente zur verfügung. Um eine Komponente dem Top-Level-Container hinzuzufügen, benutzt man eine Variante

der *add()* Methode die uns von einem Container-Objekt zur Verfügung gestellt wird. Diese Methode verlangt mindestens ein Argument, und zwar die Komponente die dem Container hinzugefügt werden soll. Manchmal ist es nötig weitere Parameter zu bestimmen, um etwa Layout Informationen zu übermitteln. Die Steuerelemente können dem *content pane* direkt oder indirekt hinzugefügt werden. Ist *content pane* der Haupt-Container enthält er die Steuerelemente direkt, hält dieser jedoch einen weiteren Container, wie etwa eine Instanz von *JPanel*, werden die *atomaren Komponenten* diesem hinzugefügt.

4.1.3 Layout Management

Swing bietet die Möglichkeit Layout Manager zu verwenden, um die Komponenten innerhalb eines Containers anzuordnen. Dies garantiert, dass die Anordnung dieser Komponenten auch nach Veränderung der Hauptfenster Größe proportional erhalten bleibt. Die Klasse *JPanel* bietet einen Container für Swing Komonenten, der Layout Management beherrscht. Dazu ersetzt man den *content pane* Container durch eine Instanz von *JPanel*, bestimmt seinen *Layout Manager* und fügt ihm dann die Komponenten unter Angabe von Layout-Informationen hinzu. Jeder Layout Manager implementiert seine eigene Logik bezüglich der optimalen Anordnung der Komponenten, wobei Swing folgende Layout Manager zur Verfügung stellt:

- **FlowLayout**
  Ordnet Dialogelemente nebeneinander in einer Zeile an. Wenn keine weiteren Elemente in die Zeile passen, wird mit der nächsten Zeile fortgefahren.
- **GridLayout**
  Ordnet die Dialogelemente in einem rechteckigen Gitter an, dessen Zeilen- und Spaltenzahl beim Erstellen des Layoutmanagers angegeben wird.
- **BorderLayout**
  Verteilt die Dialogelemente nach Vorgabe des Programms auf die vier Randbereiche und den Mittelbereich des Fensters.
- **CardLayout**
  Ist in der Lage, mehrere Unterdialoge in einem Fenster unterzubringen und jeweils einen davon auf Anforderung des Programms anzuzeigen.
- **GridBagLayout**
  Ist ein komplexer Layoutmanager, der die Fähigkeiten von GridLayout erweitert und es ermöglicht, mit Hilfe von Bedingungsobjekten sehr komplexe Layouts zu erzeugen

4.1.4 Ereignisbehandlung

Bei der Verarbeitung des Nachrichtenverkehrs sind zwei verschiedene Arten von Objekten beteiligt. Die Ereignisquellen (Event Sources) sind die Auslöser der Nachrichten. Eine Ereignisquelle kann beispielsweise ein Button sein, der auf einen Mausklick reagiert. Die Reaktion auf diese Nachrichten erfolgt in den speziellen Ereignisempfängern (EventListeners) - den Objekten, die das zum Ereignis passende Empfänger - Interface implementieren. Damit ein Ereignisempfänger die Nachrichten einer bestimmten Ereignisquelle erhält, muß er sich bei dieser registrieren. Dieses Kommunikationsmodell nennt sich *Delegation Event Model* oder *Delegation Based Event Handling* und wurde

mit der Version 1.1 des JDK eingeführt. Das Modell der Ereignisbehandlung ist Bestandteil des AWT und wird bei Swing Applikationen herangezogen. Dieses Modell liefert mehrere Möglichkeiten die Event-Handler zu implementieren:

- Die Fensterklasse implementiert die erforderlichen EventListener-Interfaces, stellt die erforderlichen Callback-Methoden zur Verfügung und registriert sich selbst bei den Ereignisquellen.
- In der Fensterklasse werden lokale oder anonyme Klassen definiert, die einen EventListener implementieren oder sich aus einer Adapterklasse ableiten, wobei eine Adapterklasse ein Interface mit mehreren Methoden implementiert und es somit abgeleiteten Klassen erlaubt, nur noch die Methoden zu überlagern, die tatsächlich von Interesse sind.
- GUI-Code und Ereignisbehandlung werden vollkommen getrennt und auf unterschiedliche Klassen verteilt.
- In der Komponentenklasse werden die Methoden überlagert, die für das Empfangen und Verteilen der Nachrichten erforderlich sind.

Alle diese Möglichkeiten sind je nach Anwendungsfall unterschiedlich gut oder schlecht geeignet, es bleibt also immer dem Programmierer abzuwiegen, welche die beste Methode für seine Anwendung ist.

4.2 Windows Forms

Das Anwendungsprogrammiermodell für Windows Forms (auch WinForms genannt) setzt sich in der Hauptsache aus Formularen, Steuerelementen und ihren Ereignissen zusammen. Die Architektur des Windows Forms Namespace ist sehr einfach - sie beinhaltet *Steuerelemente* (engl. controls) und *Container*. Dies ähnelt dem Java JFC Modell, in dem Container durch die Klassen *JFrame*, *JWindow*, *JPanel* usw. und Steuerelemente durch Klassen wie *JButton*, *JCheckbox*, *JLabel* usw. repräsentiert werden. Fast alle UI - Klassen des Windows Forms Namespace erben von der Klasse *System.Windows.Forms.Control*, das heisst, dass alles was man in einer WinForms Anwendung sieht, ein Steuerelement *(engl. control)* ist. Wenn ein Steuerelement in der Lage ist andere Steuerelemente aufzunehmen, dann ist es ein Container. Die grafische Oberfläche der WinForms Anwendung besteht aus einem "Formular" (engl. form), das als der Haupt-Container agiert. Dieser kann dann weitere Container und Steuerelemente beinhalten.

4.2.1 Formulare

Die *Form*-Klasse von Windows Forms repräsentiert alle in einer Anwendung angezeigten Fenster. Mit Hilfe der *BorderStyle*-Eigenschaft der *Form*-Klasse ist es möglich Standard- und Toolfenster sowie rahmenlose und unverankerte Fenster zu erstellen. Des Weiteren kann man mit der *Form*-Klasse modale Fenster erstellen, z. B. Dialogfelder. Durch Festlegen der *MDIContainer*-Eigenschaft der *Form*-Klasse ist es möglich einen speziellen Typ von Formular zu erstellen, das MDI-Formular. Der Clientbereich eines MDI-Formulars kann dann andere Formulare, so genannte untergeordnete MDI-Formulare, enthalten. Die *Form*-Klasse bietet integrierte Unterstützung für die Verarbeitung von Tastatureingaben und für das Durchführen von Bildläufen in einem Formular. Beim Entwerfen der Benutzeroberfläche für die Anwendung erstellt man i. d. R. eine Klasse, die von Form abgeleitet ist. Man legt dann Eigenschaften fest, erstell *Event*-Handler und fügt dem Formular Steuerelemente sowie Programmierlogik hinzu.

4.2.2 Steuerelemente

Alle einem Formular hinzugefügten Komponenten werden als Steuerelemente bezeichnet. Windows Forms enthält alle Steuerelemente, die unter Windows üblich sind, sowie einige neue Steuerelemente wie z.B. *DataGrid*, das zur Anzeige von ADO.NET Daten dient. Wenn man Steuerelemente verwenden, legt man i. d. R. Eigenschaften fest, mit denen man das Aussehen und Verhalten der Steuerelemente anpassen kann. So fügt man zum Beispiel dem Formular ein Button-Steuerelement hinzu und legt dessen Eigenschaften *Size* (Größe) und *Location* (Position) fest. Die Reihenfolge in der man die Eigenschaften eines Steuerelements bestimmt ist egal, sobald dessen Instanz vorhanden ist, kann man auch seine Eigenschaften bestimmen.

4.2.3 Layout Management

Windows Forms bietet drei Möglichkeiten zum Steuern des Layouts von Formularen:

- **Verankern**
  Wenn ein Steuerelement am Rand seines Containers verankert ist, bleibt der Abstand zwischen dem Steuerelement und dem angegebenen Rand beim Ändern der Größe des Containers konstant. Ein Steuerelement kann an einer beliebigen

Kombination von Rändern des Containers verankert werden. Wenn das Steuerelement an einander gegenüberliegenden Rändern des Containers verankert ist, wird seine Größe beim Ändern der Größe des Containers angepasst.
Wenn man z. B. ein TextBox-Steuerelement am linken und am rechten Rand eines Formulars verankert, wird bei einer Änderung der Größe des Formulars die Breite des TextBox-Steuerelements geändert.

- **Andocken**
  Wenn ein Steuerelement am Rand seines Containers angedockt ist, bleibt es bei einer Größenänderung des Containers mit diesem Rand verbunden. Im Windows-Explorer ist z. B. das TreeView-Steuerelement am linken Rand des Fensters angedockt, und das ListView-Steuerelement ist am rechten Rand des Fensters angedockt. Wenn an einem Rand mehrere Steuerelemente angedockt sind, ist das erste am Rand des Containers angedockt, und die übrigen sind so dicht wie möglich am Rand des Containers angedockt, ohne dass sie sich dabei überlagern. Dieses entspricht dem *BorderLayout* aus Java Swing.

- **Benutzerdefiniert**
  Es ist auch möglich einen eigenen Layout-Manager mit dem Layout-Ereignis zu implementieren. Das Layout-Ereignis wird von der Control-Klasse zur Verfügung gestellt und wird immer dann ausgelöst, wenn ein Ereignis eintritt, das für das Steuerelement das Anzeigen untergeordneter Steuerelemente bewirkt. Dazu gehören u. a. das Resize-Ereignis, das Ein- und Ausblenden sowie das Hinzufügen und Entfernen von untergeordneten Steuerelementen. Mit diesem Ereignis sollte sämtliche benutzerdefinierte Layoutlogik ausgeführt werden.

4.2.4 Ereignisbehandlung

Das Windows Forms-Programmiermodell basiert auf Ereignissen. Wenn der Zustand eines Steuerelements geändert wird, z. B. beim Klicken auf eine Schaltfläche, wird ein Ereignis ausgelöst. Damit ein Ereignis behandelt werden kann, wird von der Anwendung eine Ereignisbehandlungsmethode (kurz: Event-Handler) für das Ereignis registriert. In C# wird der Event-Handler mit Hilfe des += - Operators registriert. Dabei können für ein Ereignis auch mehrere Event-Handler registriert werden. Eine Ereignisbehandlungsmethode wird nur für ein spezifisches Ereignis eines bestimmten Steuerelements aufgerufen. Auf diese Weise wird vermieden, dass eine einzelne Methode alle Ereignisse für alle Steuerelemente des Formulars behandelt. Außerdem werden die Lesbarkeit und Verwaltung von Code durch dieses Feature erleichtert. Da die Windows Forms-Ereignisarchitektur auf Delegates basiert, sind die Ereignisbehandlungsmethoden darüber hinaus typsicher und können als private deklariert werden. Aufgrund dieser Funktion kann der Compiler Abweichungen bei den Methodensignaturen bereits zur Kompilierzeit feststellen. Die öffentliche Schnittstelle der *Form*-Klasse bleibt übersichtlich und wird nicht durch öffentliche Ereignisbehandlungsmethoden überladen.

Ereignisklassen

Alle Ereignisse werden durch zwei Klassen unterstützt:

- **Die EventHandler-Delegateklasse**
  Hiermit wird die Ereignisbehandlungsmethode registriert wird. Die Signatur von EventHandler gibt die Signatur der Ereignisbehandlungsmethode vor.

- **Die EventArgs-Klasse**
  Enthält Informationen über das ausgelöste Ereignis.

Die Signatur von *EventHandler* sieht vor, dass das erste Argument einen Verweis auf das Objekt enthält, welches das Ereignis ausgelöst hat (eine Instanz von *Object*), und dass das zweite Argument die Informationen über das Ereignis enthält (eine Instanz von *EventArgs*). Jede Ereignisbehandlungsmethode für das Click-Ereignis muss daher die folgende Signatur besitzen:

```
<access> void <name>(Object sender, EventArgs evArgs)
```

Viele Ereignisse verwenden die generische *EventHandler*-Klasse und die generische *EventArgs*-Klasse. Bei einigen Ereignissen sind aber zusätzliche Informationen erforderlich, die sich speziell auf den Typ des ausgelösten Ereignisses beziehen. Mausbewegungsereignisse enthalten z. B. Informationen zur Position des Mauszeigers oder zu den Maustasten. Diese Ereignisse definieren eigene Klassen, die von der *EventHandler*-Klasse und *EventArgs*-Klasse erben müssen. Das MouseDown-Ereignis verwendet z. B. die *MouseEventHandler*-Klasse und die *MouseEventArgs*-Klasse.

Man kann mehrere Ereignisse durch denselben Event-Handler behandeln lassen. Dazu wird dieselbe Methode für mehrere Ereignisse registriert. Wenn man für mehrere Ereignisse dieselbe Ereignisbehandlungsroutine verwenden, kann man dann anhand des Parameters *sender* feststellen, welches Steuerelement ein Ereignis ausgelöst hat.

4.2.5 Deterministische Lebensdauer und Freigabe

Im .NET Framework-Klassenmodell steht die *Dispose*-Methode zur Verfügung. Sie wird von der *Component*-Klasse bereitgestellt. Die *Dispose*-Methode wird aufgerufen, wenn eine Komponente nicht länger erforderlich ist. So ruft beim Schließen eines Formulars Windows Forms die *Dispose*-Methode für das Formular und alle darin enthaltenen Steuerelemente auf. Dadurch können große Ressourcen zeitgerecht freigegeben und Verweise auf andere Objekte entfernt werden, so dass diese von der Garbage Collection verarbeitet werden können.

5 Ein Beispielprogramm

Das folgende Programm soll die in den vorherigen Kapiteln besprochenen Themen demonstrieren. Es wird ein Fester erstellt, das einen Button und einen Label enthält. Der Text des Labels dient dazu, die Anzahl der Klicks auf den Button anzuzeigen. Klickt man auf den Button wird die Anzahl der getätigten Klicks hochgezählt und angezeigt. Für diese Funktionalität ist die für das Click-Ereignis des Buttons registrierte Ereignisbehandlungsroutine *onClick_button_1* zuständig. Sie inkrementiert beim Aufruf eine statische Integer-Variable und verändert dementsprechend den Text des Labels. Das Programm liegt in zwei Versionen vor: als Windows Forms Anwendung und als Swing Applikation, um einen direkten Vergleich machen zu können. Besonderheiten der einzelnen Implementierungen werden unter den jeweiligen Abschnitten besprochen.

5.1 Java Swing

Das folgende Listing beinhaltet den Sourcecode des beschriebenen Beispielprogramms in Java unter Verwendung von Swing. Um die Komponenten (Button und Label) anzuordnen, wir das *BorderLayout* verwendet. Zusätzlich wurde die Ereignisbehandlungsroutine so angepasst, dass beim Klicken auf den Button auch das Look&Feel zur Laufzeit geändert wird:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloJavaSwingFrame extends JFrame {
  private JPanel contentPane;
  private BorderLayout borderLayout_1 = new BorderLayout();
  private JButton button_1 = new JButton();
  private JLabel label_1 = new JLabel();
  private static int clicksCount = 0;

  //Construct the frame
  public HelloJavaSwingFrame() {

    //basic settings of the top-level-container (JFrame)
    this.setSize(new Dimension(300, 100));
    this.setTitle("Hello Java Swing!");

    //set the content pane as instance of JPanel
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout_1);

    //button settings
    button_1.setText("Click me!");
    button_1.addMouseListener(new java.awt.event.MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        onClick_button_1(e);
      }
    });

    //label settings
    label_1.setFont(new java.awt.Font("Dialog", 0, 16));
```

```
    label_1.setHorizontalAlignment(SwingConstants.CENTER);
    label_1.setHorizontalTextPosition(SwingConstants.CENTER);
    label_1.setText("Clicks on button: 0");

    //add components to the content pane
    contentPane.add(button_1,  BorderLayout.SOUTH);
    contentPane.add(label_1,  BorderLayout.NORTH);
  }

  //event handler
  void onClick_button_1(MouseEvent e) {
    label_1.setText("Clicks on button: " + (++clicksCount));

    //switch the look\&feel
    try {
      if( (clicksCount % 2) == 0) {
        UIManager.setLookAndFeel(
          new javax.swing.plaf.metal.MetalLookAndFeel()
          );
        SwingUtilities.updateComponentTreeUI(this);
      }
      if( (clicksCount % 3) == 0) {
        UIManager.setLookAndFeel(
          new com.sun.java.swing.plaf.motif.MotifLookAndFeel()
          );
        SwingUtilities.updateComponentTreeUI(this);
      }
      else if( (clicksCount % 5) == 0) {
        UIManager.setLookAndFeel(
          new com.sun.java.swing.plaf.windows.WindowsLookAndFeel()
          );
        SwingUtilities.updateComponentTreeUI(this);
      }
    }
    catch(Exception exp) {}
  }

  //the main method
  public static void main(String[] args) {
    //set the system specific look&feel if existing
    try {
      UIManager.setLookAndFeel(
        UIManager.getSystemLookAndFeelClassName()
        );
    }
    catch(Exception e) {
      e.printStackTrace();
    }

    //create the frame and show it
    HelloJavaSwingFrame frame = new HelloJavaSwingFrame();
    frame.setVisible(true);
  }
}
```

Wie sich die Anwendung dem Benutzer präsentiert, kann dem folgenden Screenshot entnommen werden:

Abbildung 10: Das Beispielprogramm mit dem Windows Look&Feel

5.2 Windows Forms in C#

Das folgende Listing beinhaltet den Sourcecode des beschriebenen Beispielprogramms in C# unter Verwendung von Windows Forms. Um die Komponenten (Button und Label) anzuordnen, wird auch hier *BorderLayout* verwendet. Windows Forms bieten keine Unterstützung für umschaltbares Look&Feel wie es bei Java Swing Programmen der Fall ist. WinForms bieten jedoch die Möglichkeit das komplette Fenster durchsichtig zu machen, deshalb wurde die Ereignisbehandlungsroutine so angepasst, dass beim Klicken auf den Button auch die Transparenz des gesamten Fensters zur Laufzeit verändert wird. Dabei ist jedoch zu beachten, dass dieses Feature erst ab Microsoft Windows 2000 unterstützt wird. Das Programm ist trotzdem auf allen Systemen ab Windows 98 lauffähig, auf denen die .NET Laufzeit-Umgebung installiert ist.

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class HelloWinFormsForm : Form {

  private Button button_1 = null;
  private Label label_1 = null;
  private static int clickCount = 0;

  public static int Main(string[] args) {
    Application.Run(new HelloWinFormsForm());
    return 0;
  }

  public HelloWinFormsForm() {

    //basic settings  of the form container
    this.Text = "Hello Windows Forms";
    this.ClientSize = new Size(300, 100);

    //controls on the form
    button_1 = new Button();
    label_1 = new Label();
```

```
    //button settings
    button_1.Dock = DockStyle.Bottom;
    button_1.Height = 40;
    button_1.Text = "Click Me!";

    //label settings
    label_1.Dock = DockStyle.Top;
    label_1.Font = new Font(
                   "Arial", 15.75F,
                   System.Drawing.FontStyle.Regular,
                   System.Drawing.GraphicsUnit.Point,
                   ((System.Byte)(0)) );
    label_1.TextAlign = ContentAlignment.MiddleCenter;
    label_1.Text = "Clicks on button: 0";

    //register the eventhandler for the click event of button_1
    button_1.Click += new System.EventHandler(onClick_button_1);

    //add controls to the form container
    this.Controls.Add(button_1);
    this.Controls.Add(label_1);
  }
  //eventhandler call-back method
  public void onClick_button_1(Object sender, EventArgs e) {
    label_1.Text = "Clicks on button: " + (++clickCount);
    if(this.Opacity == 0.25) {
      this.Opacity = 1.00;
    }
    else {
      this.Opacity -= 0.125;
    }
  }
}
```

Wie sich die Anwendung dem Benutzer präsentiert, kann dem folgenden Screenshot entnommen werden:

Abbildung 11: Das WinForms Beispielprogramm ohne Transparenz

6 Ein Experiment

Microsoft bietet für alle Versionen des **Visual Studio .NET** ein Zusatztool an, mit dem es möglich ist Java Sourcecode in C# Quelltext zu konvertieren. Der **Microsoft Java Language Conversion Assistant** liegt im Moment als Beta 2 Version vor, und kann frei unter http://msdn.microsoft.com/ downgeloadet werden. Microsoft behauptet, es sei das schnellste Tool, um Java Anwendungen nach C# zu konvertieren. Weitere Details liegen jedoch nicht vor. Deswegen habe ich mich entschlossen dieses Tool kurz zu testen.

6.1 Konvertierung einer Java Konsolenanwendung nach C#

Und schon wieder muss das gute, alte "Hello World" herhalten. Die Erwartung, dass ein solch einfaches Programm für den **Microsoft Java Language Conversion Assistant** kein Problem sein sollte, wird erfüllt. Der folgende Eingabe Java Quelltext:

```
import java.io.*;

public class HelloToConvert {
  public static void main(String[] args) {
  System.out.println("Hello, world!");
  }
}
```

wurde von dem Assistent korrekt nach C# konvertiert - das resultierende Programm war lauffähig und hat sich zur Laufzeit genauso verhalten, wie das ursprüngliche Java-Programm. Der Quelltext sieht nach der Konvertierung folgendermaßen aus:

```
using System;

public class HelloToConvert {
  [STAThread]
  public static void Main(System.String[] args)
  {
    System.Console.Out.WriteLine("Hello, world!");
  }
}
```

6.2 Konvertierung einer Java Swing Anwendung nach C#

Microsoft sagt nichts darüber, ob der benannte Assistent nur zur reinen Syntax Konvertierung von Java nach C# benutzt werden kann, oder auch fähig ist eine Java Anwendung unter Berücksichtigung von Komponenten des .NET Frameworks wie z.B. Windows Forms zu konvertieren. Deshalb habe ich mich entschlossen, die Konvertierungs-Prozedur mit einer einfachen Swing-Applikation durchzuführen. Als Eingabe Sourcecode, habe ich das Swing-Beispielprogramm aus dem Abschnitt [5.1] verwendet (allerdings ohne dem Source-Block, der für das Umschalten des Look&Feel zuständig ist), in der Hoffnung, dass der **Microsoft Java Language Conversion Assistant** meine Swing Applikation zu einer Windows Forms Anwendung konvertiert. Mir war bewusst, dass Windows Forms nicht alle aus Swing bekannten Elemente zur Verfügung stellt, schon alleine wenn es um das Layout Management geht, ist Windows Forms nicht so üppig ausgestattet wie Swing. Aber schließlich war meine Swing Anwendung

nicht von Komplexität geprägt, und nutzte auch das in Windows Forms verfügbare *BorderLayout* um die Komponenten anzuordnen. Leider wurden meine Erwartungen nicht ganz erfüllt:

- Alle Swing Konstrukte in dem Eingabe Quelltext wurden nicht etwa in entsprechende Windows Forms Konstrukte konvertiert, sondern so belassen, wie sie waren. So erbt die Klasse *HelloJavaSwingFrame* auch in dem konvertierten Quelltext von *JFrame* und nicht etwa von *Form* auch alle anderen Swing Komponenten wie *JPanel, JButton, JLabel* und *BorderLayout* wurden als unbekannte Elemente behandelt und ignoriert.
- Alle primitiven Java Konstrukte wurden, wie bereits bei der Konvertierung der Konsolenanwendung, korrekt nach C# portiert. Die Code Fragmente, die dem Konvertierungsassistenten fremd waren, wurden so belassen, wie sie in der Eingabedatei vorzufinden sind.
- Erstaunlich ist allerdings, dass der Assistent trotzdem ein Visual Studio Projekt angelegt hat, und die Quelltext Fragmente, die nicht konvertiert werden konnten, kommentiert wurden. Die Kommentare beinhalten, da wo das Konvertierungsproblem bekannt ist einen Link auf die dazugehörige Hilfe - Seitemit der Beschreibung des Problems und einer Hilfestellung, und bei kritischen Fehlern eine Empfehlung den Quelltext manuell zu portieren.

Folgender Quelltext ist nur ein Ausschnitt aus der Ausgabedatei:

```
using System;

public class HelloJavaSwingFrame:JFrame {

...

  private JPanel contentPane;
  //UPGRADE_ISSUE: Class 'java.awt.BorderLayout' was not converted.
  //'ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword=...
  //UPGRADE_NOTE: The initialization of  'borderLayout_1' was moved
  //to method 'InitBlock'.
  //'ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword=...
  private BorderLayout borderLayout_1;
  //UPGRADE_NOTE: The initialization of  'button_1' was
  //moved to method 'InitBlock'.
  //'ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword="jlca1005"'
  private JButton button_1;
  //UPGRADE_NOTE: The initialization of  'label_1' was
  //moved to method 'InitBlock'.
  //'ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword="jlca1005"'
  private JLabel label_1;
  private static int clicksCount = 0;

  //Construct the frame
  public HelloJavaSwingFrame()
  {
    InitBlock();

    //basic settings of the top-level-container (JFrame)
```

```
    this.Size = new System.Drawing.Size(300, 100);
    this.Title = "Hello Java Swing!";

    //set the content pane as instance of JPanel
    contentPane = (JPanel) this.ContentPane;
    contentPane.Layout = borderLayout_1;

    //button settings
    button_1.Text = "Click me!";
    button_1.addMouseListener(new AnonymousClassMouseAdapter(this));

    //label settings
    //UPGRADE_NOTE: If the given Font Name does not exist, a default
    //Font instance is created.
    //'ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword="jlca1075"'
    label_1.Font = new System.Drawing.Font("Dialog",
                                           16,
                                          (System.Drawing.FontStyle) 0);
    label_1.HorizontalAlignment = SwingConstants.CENTER;
    label_1.HorizontalTextPosition = SwingConstants.CENTER;
    label_1.Text = "Clicks on button: 0";

    //add components to the content pane
    //UPGRADE_ISSUE: Field 'java.awt.BorderLayout.SOUTH'
    //was not converted.
    //'ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword=...
    contentPane.add(button_1, BorderLayout.SOUTH);
    //UPGRADE_ISSUE: Field 'java.awt.BorderLayout.NORTH'
    //was not converted.
    //'ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword=...
    contentPane.add(label_1, BorderLayout.NORTH);
  }
  ...
}
```

Das komplette Projekt ist auf dieser CD im Unterverzeichnis:
[CDROMROOT]:/Source/Experiment/HelloSwingConverted/ zu finden.

Fazit

Der **Microsoft Java Language Conversion Assistant** scheint nur bei primitiven Java Konstrukten richtig zu funktionieren, zumindest in der vorliegenden Beta 2 Version. Die Konvertierung von Swing basierten Anwendungen wird derzeit scheinbar nicht unterstützt, da der Assistent jedoch bei schwierigen Aufgaben die Konvertierung nicht abbricht sondern trotzdem eine teilweise Portierung vornimmt und zu den von ihm nicht lösbaren Aufgaben Hilfestellung bietet, ist es durchaus zu überlegen, ob sein Einsatz bei Portierungsaufgaben doch nicht sinnvoll wäre. Eine vollkommene Lösung bietet der Assistent von Microsoft jedoch nicht.

7 Fazit

Ob es nun **Swing** oder **Windows Forms** ist, hinter beiden besprochenen Bibliotheken zur Erstellung von grafischen Oberflächen steckt jeweils ein gutes, solides Konzept. **Swing** und **Windows Forms** sind sich in dem Programmiermodell sehr ähnlich und man stellt fest, dass auch die zugrundeliegenden Frameworks sehr ähnliche, wenn nicht die gleichen, Konzepte verfolgen. Wie es fast immer der Fall ist, haben beide ihre Vor- und Nachteile. Die Vorteile von Java Swing sind vor allem:

- Die Plattformunabhängigkeit, die durch das JDK geboten wird. Java Swing Anwendungen sind auf jedem Betriebssystem lauffähig, für das es eine Implementierung der Java 2 Runtime zur Verfügung steht.
- Java Swing bietet darüberhinaus plattformunabhängies, umschaltbares Look&Feel.
- Swing ist ansonsten eine sehr umfangreiche Bibliothek, die sich dazu einfach einsetzen lässt.

Swing bleibt aber von offensichtlichen Nachteilen nicht verschont. Zu diesen zählen vor allem:

- Die Performance zur Laufzeit ist einer der Aspekte, der je nach Anwendungsfall gegen den Einsatz von Java und der Swing Bibliothek sprechen kann. Es ist aber auch bereits seit Langem ein beliebter Streitpunkt. Viele behaupten, Java sei zu langsam um die Sprache und ihre Klassen-Bibliothek professionell einsetzten zu können, andere behaupten, Java Programme seien kaum langsamer als C++ Anwendungen. Die Wahrheit liegt wie immer irgendwo dazwischen.
- Die Abhängigkeit von einer einzigen Programmiersprache ist spätestens seit der Einführung des .NET Frameworks zu einem negativen Aspekt geworden.

Für der Einsatz von Windows Forms und dem .NET Framework bei der Entwicklung von Anwendungen mit grafischer Benutzerschittstelle sprechen vor allem folgende Aspekte:

- Die Sprachintegration des .NET Frameworks wirkt sich auf die Flexibilität von Windows Forms aus. Dies bedeutet auch, dass alte Anwendungen, die auch noch vielleicht in einer anderen Programmiersprache als C++ oder Visual Basic geschrieben wurden, sich leicht auf die neue Technik des Frameworks von Microsoft portieren lassen.
- Windows Forms bietet eine viel leichtere Umgebung zur Programmierung von grafischen Oberflächen als es die MFC-Bibliothek oder die Win32-API tut. Verwendet man dann auch noch die neue Java - ähnliche Programmiersprache C#, kommt man schnell an das erwünschte Ziel.
- Zwar verfolgt das .NET Framework mit MSIL, einer Laufzeit-Umgebung und Garbage Collection sehr ähnliche Design Konzepte wie Java, jedoch sind Windows Forms Anwendungen schneller zur Laufzeit als Java Swing Anwendungen. Die Lösung dafür liegt in der Tatsache, dass der plattformunabhängige MSIL-Code nicht von einer Virtuellen Maschine interprätiert wird, sondern zum nativen Binär-Code compiliert und dann ausgeführt wird. Dadurch lässt sich die Anwendung genau für das System und seine Architektur compilieren, auf dem sie gerade ausgeführt werden soll.

Gegen den Einsatz von Windows Forms sprechen die Folgenden Punkte:

- Zwar erhebt das .NET Framework den Anspruch plattformunabhängig zu sein, jedoch gibt es zur Zeit keine vollständige Implementierung des Frameworks für ein anderes Betriebssystem als Microsoft Windows. Die GNU Organisation hat zwar ein Projekt gestartet, der sich damit beschäftigt, das .NET Framework auch für Linux/Unix Systeme zur Verfügung zu stellen, jedoch befindet sich das Projekt noch in Arbeit und ist noch nicht vollständig abgeschlossen (siehe http://www.gnu.org/projects/dotgnu/index.html). Auch Ximian hat mit **Mono** eine Initiative ins Leben gerufen, die das Ziel hat das .NET Framework auch unter Linux/Unix zur Verfügung zu stellen. Jedoch ist auch diese Implementierung noch nicht vollständig (siehe http://www.go-mono.net/index.html). Die Arbeit der beiden Projekte geht jedoch schnell voran und hat bereits erste Früchte erbracht, so dass dieser nachteilige Aspekt in der Zukunft wegfallen kann.

- Der zur Zeit größte Nachteil von Windows Forms ist, dass es die einzige Komponente des Frameworks ist, die nicht vom Betriebssystem unabhängig ist. WinForms Anwendungen sind nur auf Microsoft Windows Systemen lauffähig, weil die Windows Forms Bibliothek auf die Win32-API zugreift. Das Mono Projekt hat sich jedoch auch die Portierung des System.Windows.Forms Namespace zum Ziel gemacht. So soll es in der Zukunft möglich sein auch Windows Forms Anwendungen auf Linux/Unix Systemen auszuführen. Um dies zu ermöglichen werden die Gnome- und GTK-Bibliotheken herangezogen. Also könnte auch dieser Nachteil bald der Vergangenheit angehören.

- Windows Forms bietet kein "anhängbares" Look&Feel, womit sich das Aussehen und das "Feel" von Anwendungen verändern ließe.

Wie man sieht haben beide Bibliotheken ihre Vor- und Nachteile, die je nach Anwendungsfall mehr oder weniger ins Gewicht fallen.

8 Quellenangaben

1. ".Net Framework Essentials", Thuan Thai & Hoang Q. Lam, O'Reilly 2001, ISBN: 0-596-00165-7
2. "Handbuch der Java-Programmierung", Guido Krüger, Addison-Wesley 2002, ISBN 3-8273-1949-8
3. Microsoft Developer Network, http://msdn.microsoft.com
4. Sun Microsystems Java Homepage, http://java.sun.com

Seminar Mechanismen

Thema: Entwurfsmuster
AbstractFactory & Builder

Ausarbeitung[1] zum grossen Seminar bei
Prof. Dr. Henrich und Prof. Dr. Renz
zusammen mit der Firma Bosch.
FH Giessen-Friedberg

Heiner Engelhardt (Matr. Nr.: 636791)

25. Juni 2002

[1] Erstellt mit LaTeX

Inhaltsverzeichnis

1 Klassifizierung der Muster

Die Entwurfsmuster *AbstractFactory* und *Builder* gehören zur Familie der objektbasierten Erzeugungsmuster.
Erzeugungsmuster dienen zum Erzeugen von Objekten und helfen den Erzeugungsprozess (wie die Objekte erzeugt werden) zu verstecken.
Objektbasiert bedeutet, dass Klassen nicht durch Vererbung sondern durch Delegation[1] erweitert werden.
Weiterhin helfen Erzeugungsmuster, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt oder repräsentiert werden.

[1] Eine Nachricht wird von einem Objekt nicht selbst interpretiert sondern an ein anderes Objekt weitergeleitet.

2 Vorstellung (AbstractFactory / AbstrakteFabrik)

2.1 Zweck

Erich Gamma schreibt in [1]:

> ”Biete eine Schnittstelle zum Erzeugen von Familien verwandter Objekte, ohne ihre konkreten Klassen zu benennen.”

Dies bedeutet, dass Objekte wenn sie erzeugt werden nicht mit dem Klassennamen benannt werden, sondern eine Schnittstelle existiert, mit Hilfe derer die Objekte erzeugt werden.

2.2 Problem und Lösung

Um das Muster genauer zu erleutern und um es besser zu verstehen, wird ein Beispiel verwendet, welches uns durch die gesamte Vorstellung führt:

Ziel: Implementierung einer Anwendung, bei der es möglich ist, die Geometrie von Widgets zu verändern, d.h. es soll möglich sein, Buttons oder Dialoge in dreieckiger, viereckiger oder runder Form darzustellen.

Mögliche Lösung: Möglich wäre, alle Widgetklassen auszuprogrammieren und die Erzeugung dieser auf die Anwendung zu verteilen.

Problem: Schwierigkeiten und grössere Umstände treten dann auf, wenn die Geometrie der Widgets zur Laufzeit geändert werden soll, da die Erzeugung dieser unterschiedlichen Widgets auf der gesamten Anwendung verteilt ist.

Bessere Lösung: Besser wäre es, eine abstrakte WidgetFabrik zu definieren, die eine Schnittstelle zur Erzeugung aller benötigten Arten von Widgets bietet. Beim Erzeugen der Widgets wollen wir nur diese Schnittstelle verwenden und nicht die Widgetklassen konkret ansprechen.

Abbildung 1 zeigt die Struktur der Anwendung zum verändern der Geometrie von Widgets. Zunächst haben wir konkrete Klassen, welche die geometrischen Widgets grundlegend implementieren (*KreisDialog, ViereckDialog, KreisButton, ViereckButton*). Für diese Klassen definieren wir jeweils eine Oberklasse (*Dialog, Button*). Weiterhin definieren wir eine abstrakte Klasse *WidgetFabrik*, welche Operationen zum Erzeugen von Widgets bietet ($ErzeugeDialog()$, $ErzeugeButton()$). Die Klasse *WidgetFabrik* hat nun zwei Unterklassen (*ViereckWidgetFabrik*, *KreisWidgetFabrik*), welche die Operationen der abstrakten Klasse implementieren. Die Methode $ErzeugeButton()$ der Klasse *KreisWidgetFabrik* gibt einen konkreten *KreisButton* zurück, die Methode $ErzeugeDialog()$ der Klasse *ViereckWidgetFabrik* einen konkreten Dialog in viereckiger Form.

Um nun auch Dialoge und Buttons in verschiedenen geometrischen Formen erzeugen zu können, benötigen wir eine weitere Klasse (*Manager*). Die Klasse *Manager* verwendet nun lediglich die Schnittstellen (*WidgetFabrik*, *Dialog*, *Button*), um Dialoge und Buttons zu erzeugen, kennt aber die Klassen nicht, welche die konkreten geometrischen Widgets (*KreisDialog*, *ViereckButton*) implementieren.

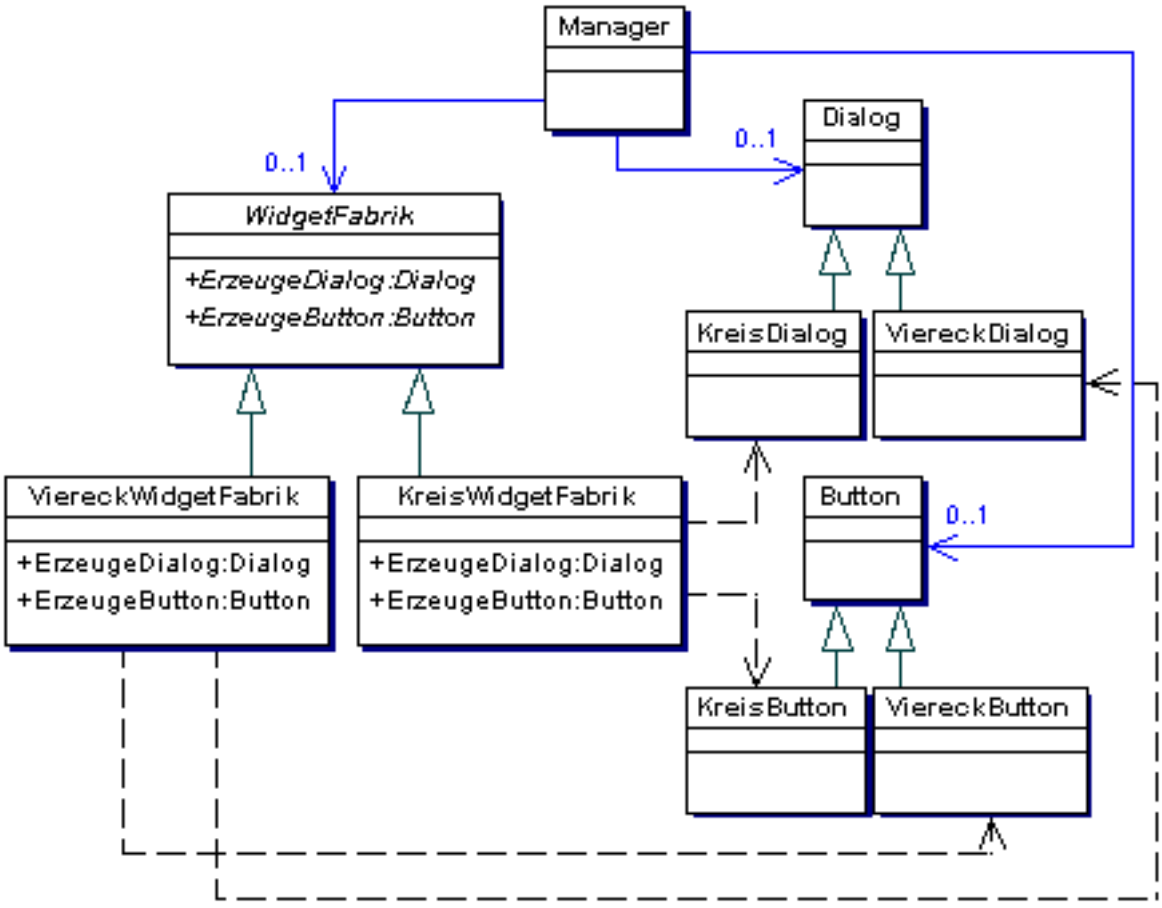


Abbildung 1: Struktur der Anwendung zum Verändern der Geometrie von Widgets

2.3 Struktur

Die allgemeine Struktur des Musters zeigt Abbildung 2. Es existiert eine Klasse *AbstrakteFabrik*, welche Methoden zum Erzeugen von konkreten Produkten bietet. Unterklassen der abstrakten Fabrik, die konkreten Fabriken (*KonkreteFabrik*1, *KonkreteFabrik*2) implementieren nun diese Methoden, indem sie konkrete Produkte (*ProduktA*1, *ProduktB*2) erzeugen und sie zurückgeben. Der *Klient* erzeugt die konkreten Produkte ausschliesslich über die Schnittstellen der abstrakten Fabrik und der abstrakten Produkte.

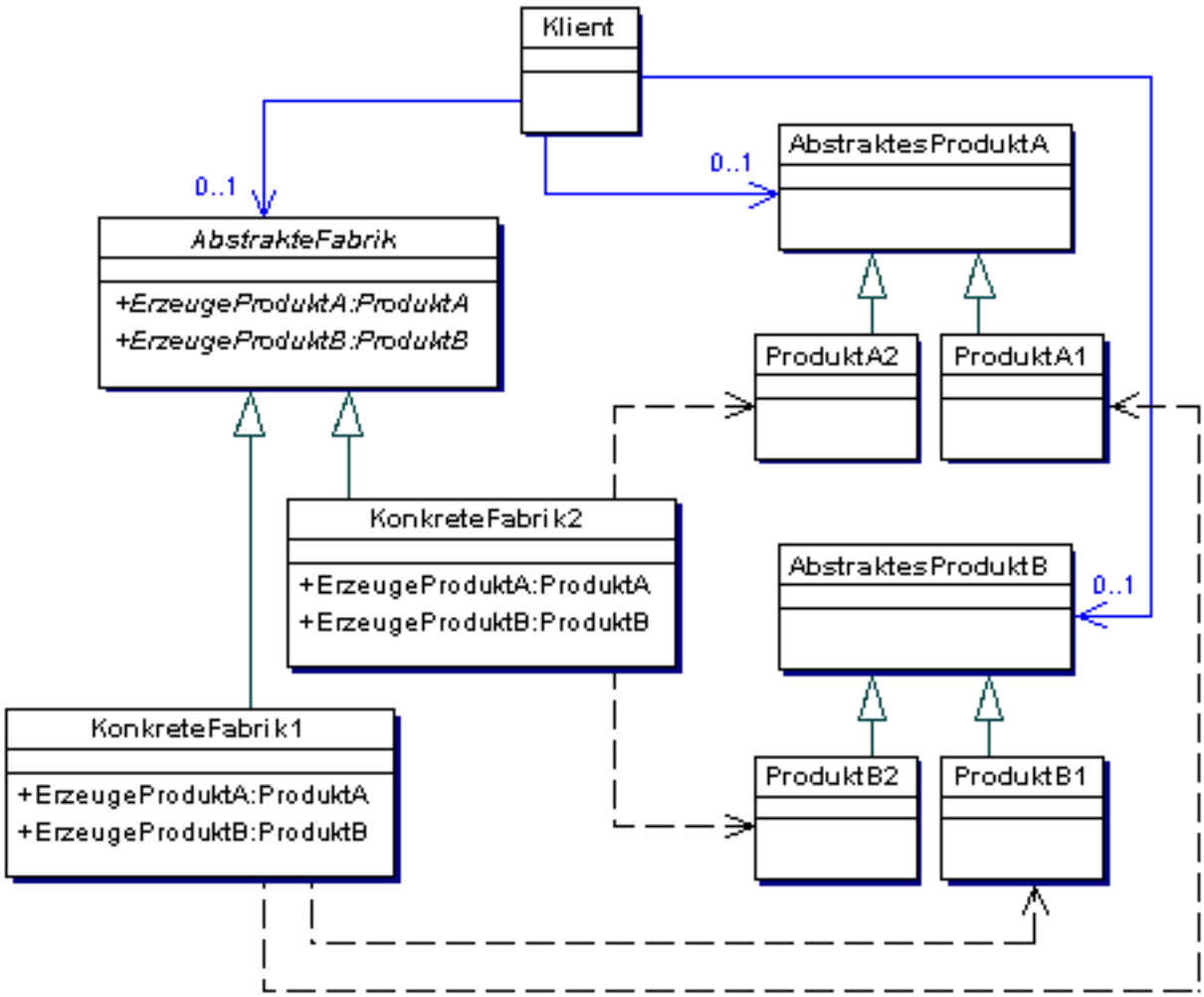


Abbildung 2: Allgemeine Struktur des Musters *AbstrakteFabrik*

2.4 Teilnehmer

AbstrakteFabrik: Schnittstelle für Operationen um Produktobjekte zu erzeugen (*WidgetFabrik*).

KonkreteFabrik: Implementierung der Operationen, um konkrete Produktobjekte zu erzeugen (*KreisWidgetFabrik*, *ViereckWidgetFabrik*).

AbstraktesProdukt: Schnittstelle für einen bestimmten Produkttyp (*Dialog*, *Button*).

KonkretesProdukt: Implementiert die Produktoperation, d.h. hier ist festgelegt wie die konkreten Widgets (*KreisDialog*, *ViereckButton*) gezeichnet werden.

Klient: Verwendet nur die abstrakten Schnittstellen von *AbstrakteFabrik* und *AbstraktesProdukt* (*Manager*).

2.5 Interaktion (Dynamik des Musters)

Üblicherweise wird nur ein einziges Objekt einer konkreten Fabrik zur Laufzeit erzeugt. Um nun verschiedene Produktobjekte erzeugen zu können, sollten die Klienten mit unterschiedlichen konkreten Fabriken konfiguriert werden.
In Abbildung 3 wird aus einem Kontext heraus ein Objekt einer konkreten Fabrik erzeugt. Weiterhin wird ein Klient erzeugt, welcher mit dem Objekt der konkreten Fabrik konfiguriert wird, d.h. er bekommt die konkrete Fabrik als Parameter übergeben. Nun werden im Klientenobjekt alle Methoden der *AbstrakteFabrik* Schnittstelle aufgerufen. Da nun aber der Klient mit einer konkreten Fabrik konfiguriert wurde, und diese konkrete Fabrik alle Methoden der abstrakten Fabrik implementiert, werden nun die Methoden der konkreten Fabrik aufgerufen, um konkrete Produktobjekte zu erzeugen.

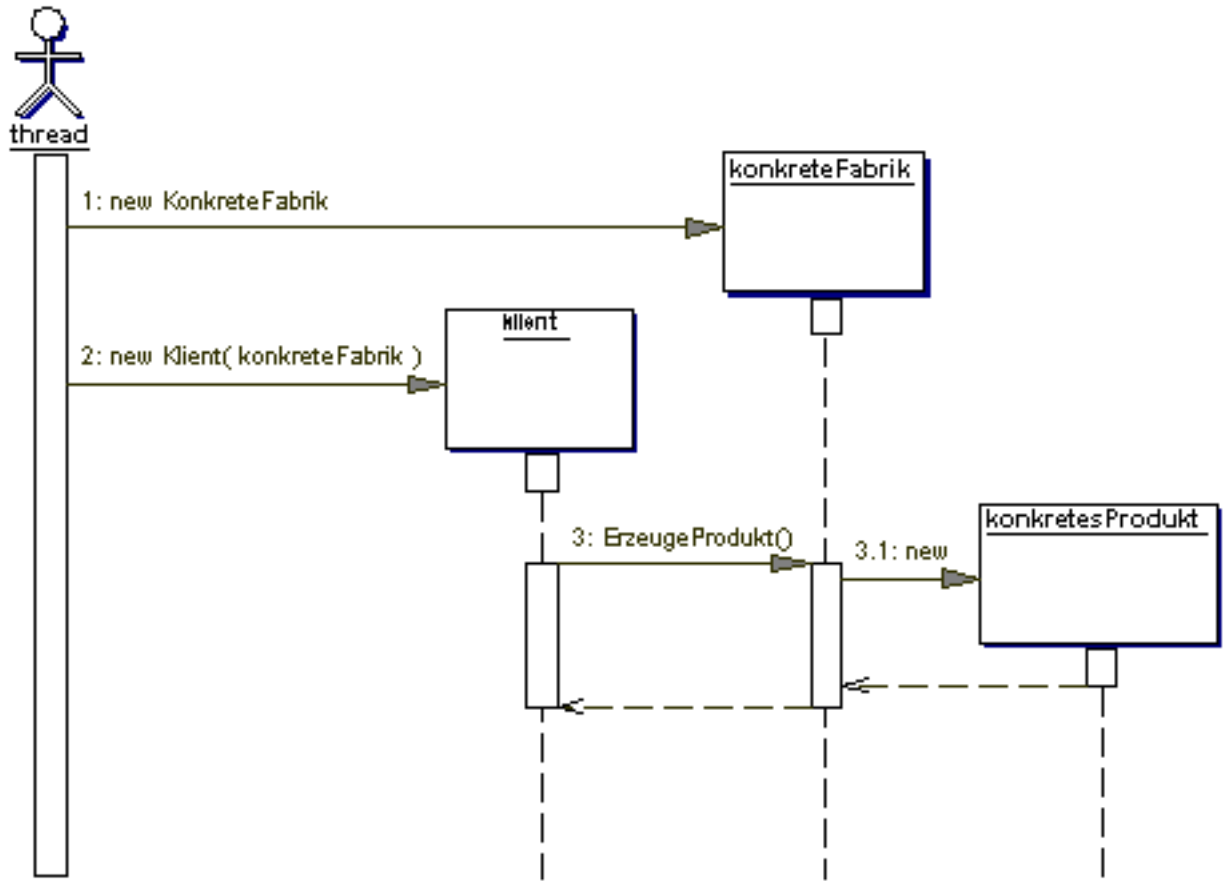


Abbildung 3: Dynamik des Musters

2.6 Implementierung

Nachdem nun die Struktur und die Dynamik des Musters vorgestellt wurde, soll hier nun grob gezeigt werden, wie man das einführende Beispiel (Geometrische Widgets) in C++ implementieren kann.

Zunächst schreiben wir eine abstrakte Klasse *WidgetFabrik*, die zwei virtuelle Methoden zum Erzeugen von Widgets enthält. Diese Klasse ist die *AbstrakteFabrik* und dient als Schnittstelle.

```
class WidgetFabrik {
    ...
    virtual Dialog* ErzeugeDialog() = 0;
    virtual Button* ErzeugeButton() = 0;
};
```

Nun brauchen wir weiter eine konkrete Fabrik *KreisWidgetFabrik*. Diese Klasse erbt von der abstrakten Fabrik. Hier werden auch die Methoden zum Erzeugen der Widgets implementiert.

```
class KreisWidgetFabrik : public WidgetFabrik {
    ...
    virtual Dialog* ErzeugeDialog() {
        return new KreisDialog;
    }
    virtual Button* ErzeugeButton() {
        return new KreisButton;
    }
};
```

Um nun Widgets zu erzeugen, deren Klassennamen wir aber nicht benennen wollen, brauchen wir einen Klienten in Form der Klasse *Manager*. In dieser Klasse kann man sehr gut sehen, dass lediglich die Schnittstellen verwendet werden, um konkrete Produkte, die Widgets, zu erzeugen. Der Klient, die Klasse *Manager*, bekommt im Konstruktor ein Objekt vom Typ *WidgetFabrik*. Da die konkreten Fabriken wegen der Vererbung auch Widget Fabriken sind, ist die Klasse *Manager* also unabhängig von der konkreten Fabrik die ihr übergeben wird.

```
class Manager {
    Dialog* m_dialog;
    Button* m_button;
    ...
    Manager( WidgetFabrik* fabrik ) {
        m_dialog = fabrik->ErzeugeDialog();
        m_button = fabrik->ErzeugeButton();
    }
    void Show() {
        m_dialog->Paint();
        m_button->Paint();
    }
};
```

Jetzt wollen wir noch betrachten, wie aus einem Kontext heraus eine konkrete Fabrik und ein Klient erzeugt werden, und wie diese interagieren . (siehe auch 2.5 Interaktion)

```
...
WidgetFabrik* fabrik = new KreisWidgetFabrik;

Manager* mgr = new Manager( fabrik );
mgr->Show();
...
```

2.7 Anwendbarkeit

Das Entwurfsmuster *AbstrakteFabrik* kann verwendet werden, wenn

- ein System unabhängig davon sein soll, wie seine Objekte erzeugt oder repräsentiert werden, da lediglich die Schnittstellen zum Erzeugen der Objekte verwendet werden.
- ein System mit einer von mehreren Produktfamilien konfiguriert werden soll. Stellen wir uns eine Anwendung vor, die verschiedene Look&Feels unterstützen soll. Jedes Look&Feel wäre dann eine Produktfamilie, die austauschbaer sein soll.
- man eine Klassenbibliothek anbieten möchte, von der nur die Schnittstellen, jedoch nicht die Implementierung offenliegen soll.

2.8 Vor- und Nachteile

Das *AbstrakteFabrik* Muster hat folgende Vor- und Nachteile:

- Vorteile
  - Isolation von konkreten Klassen. Die Struktur der Produktinhalte erscheinen nicht im Klientencode, und somit ist die Anwendung von den konkreten Produkten isoliert.
  - Einfacher Austausch von Produktfamilien. Da eine konkrete Fabrik nur einmal in der Anwendung erscheint, ist sie somit auch sehr leicht austauschbar.
  - Konsistenzsicherung unter Produkten. Betrachten wir wieder unser Beispiel. Man möchte, wenn man kreisförmige Widgets verwendet auch sicherstellen, dass *alle* verwendeten Widgets kreisförmig sind. Dies ist im Muster *AbstrakteFabrik* sichergestellt, denn die konkreten Fabriken erzeugen nur solche Widgets, die zu genau einer Produktfamilie gehören.

- Nachteile
  - Schwierige Unterstützung neuer Produkte. Wenn zu einer bestehenden Anwendung ein neues Produkt hinzu kommen soll, in unserem Beispiel eine Scrollbar, dann müssen diverse Änderungen und Erweiterungen in der Anwendung vorgenommen werden. Es muss zunächst ein neue Schnittstelle für Scrollbars definiert werden. Für diese Schnittstelle gibt es nun für jede Widgetform konkrete Unterklassen, welche die Operationen implementieren, wie runde und eckige Scrollbars gezeichnet werden sollen.

2.9 Bekannte Verwendung

Das Entwurfsmuster *AbstrakteFabrik* findet Anwendung in der Programmiersprache JAVA. Die abstrakte Klasse *java.awt.Toolkit* dient als *AbstrakteFabrik*. Diese Klasse enthält Methoden zum erzeugen von Widgets.

```
java.awt.Toolkit
    -> ButtonPeer createButton( Button b )
    -> DialogPeer createDialog( Dialog d )
    -> ...
```

Die Schnittstellen der konkreten Produkte befinden sich im Paket *java.awt.peer.*∗, hier sind zwei aufgelistet.

```
java.awt.peer.ButtonPeer
    -> void paint( Graphics g )
    -> void setLabel( String s )
    -> ...

java.awt.peer.DialogPeer
    -> void paint( Graphics g )
    -> void setTitle( String s )
    -> ...
```

Abhängig davon, auf welcher Platfrom man sich befindet, sei es *Windows*, *Unix*, oder *Macintosh*, existieren konkrete Fabriken für die jeweiligen Platformen. D.h. wenn man sich momentan auf einem *Unix* System befindet und ein JAVA Programm läuft, dann existiert auch eine konkrete Fabrik von *Toolkit*, nämlich das *UnixToolkit*. In Abbildung 4 wird dies etwas näher verdeutlicht.

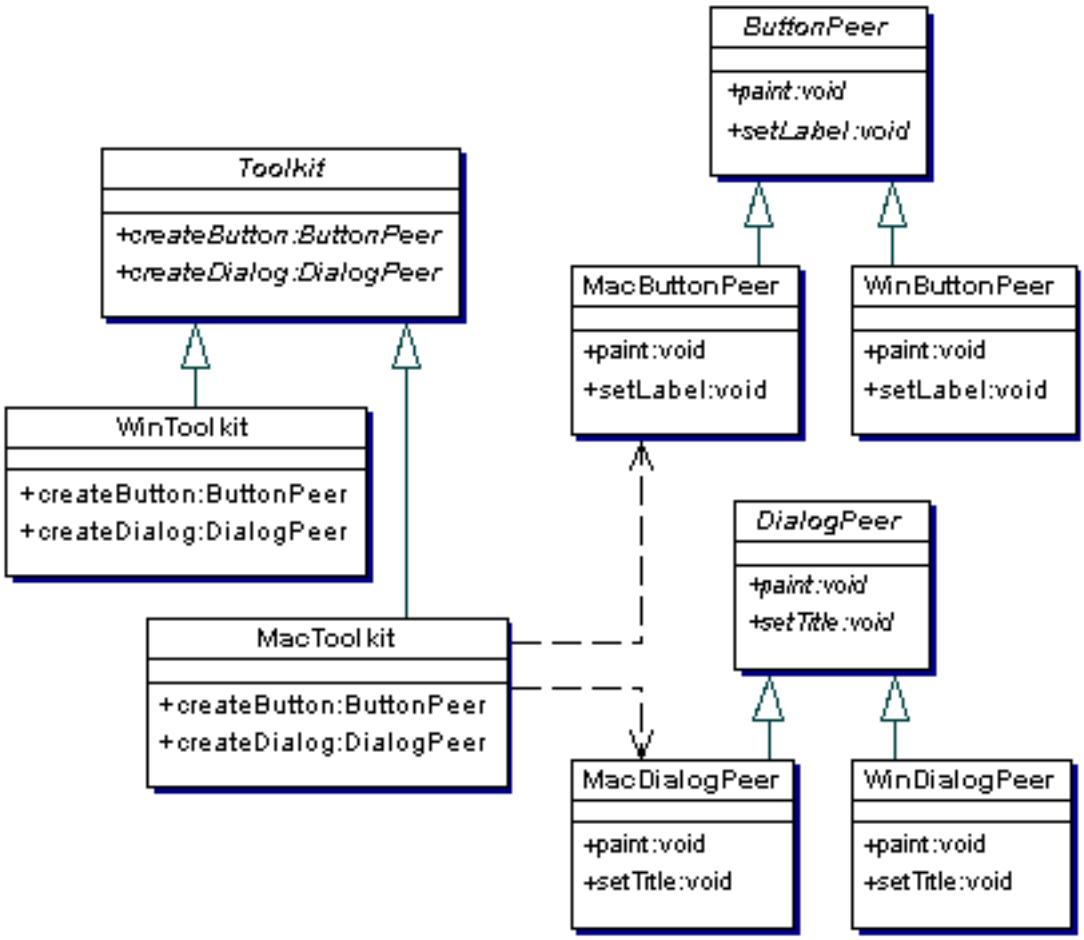


Abbildung 4: Verwendung der abstrakten Fabrik in JAVA

2.10 Verwandte Muster

- Abstrakte Fabriken werden oft durch Fabrikmethoden[2] implementiert.
- Konkrete Fabriken werden häufig als Singletons[3] realisiert, da man meistens genau ein Exemplar einer konkreten Fabrik benötigt.

[2]Man bietet eine Schnittstelle zum Erzeugen von Objekten, lässt aber deren Unterklassen entscheiden von welcher Klasse die zu erzeugenden Objekte sind

[3]Stelle sicher, das im System genau ein Objekt einer Klasse existiert und biete einen globalen Zugriff darauf

3 Beispielprogramm (AbstractFactory / AbstrakteFabrik)

Als Beispiel, ein Programm, bei dem es möglich sein soll, Widgets in verschiedenen Formen anzeigen zu lassen. Abbildung 5 zeigt einen Screenshot des laufenden Programms.

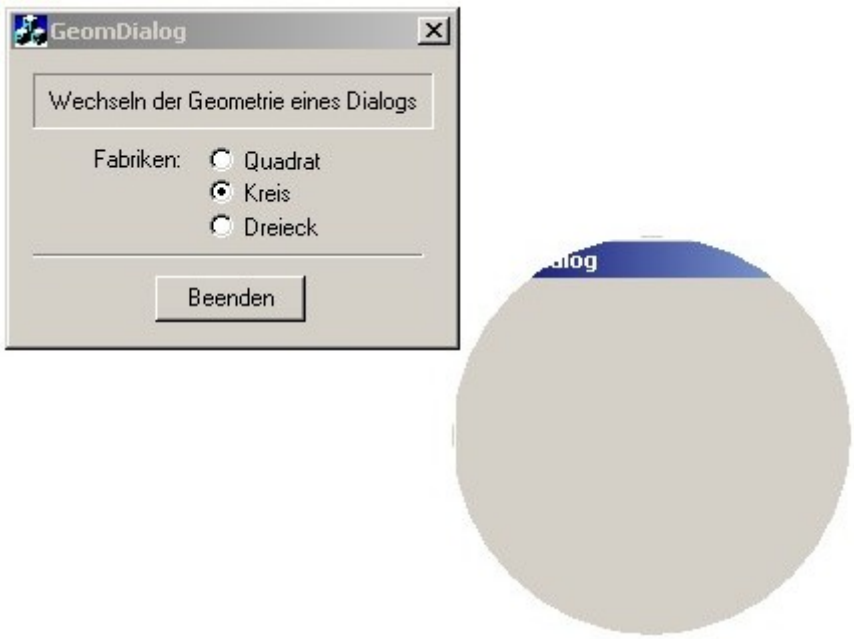


Abbildung 5: Screenshot der Beispielimplementierung

4 Vorstellung (Builder / Erbauer)

4.1 Zweck

Erich Gamma schreibt in [1]:

> ”Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.”

Es sollen also komplexe Objekte erzeugt werden. Da haben wir zum einen die Konstruktion, d.h. wie ein Objekt erzeugt wird und zum anderen die Repräsentation, sprich was dieses komplexe Objekt uns bietet, beispielsweise die Funktionalität des Objektes. Wir wollen nun immer mit dem selben Konstruktionsprozess, verschiedene Repräsentationen dieser komplexen Objekte erzielen. Um dies zu erreichen, müssen wir die Konstruktion der Objekte von der Repräsentation trennen.

4.2 Problem und Lösung

Wie im vorigen Muster, soll auch dieses Muster anhand eines Beispiels nähergebracht werden:

Ziel: Implementierung einer Simulation zum Bauen verschiedener Fahrzeuge (Fahrrad, Motorrad, Auto).

Mögliche Lösung: Ausprogrammieren aller benötigten Operationen jedes Fahrzeuges und diese dann in der Applikation verwenden.

Problem: Die Werkstatt ist abhängig vom Bau der Fahrzeuge, bei neu zu bauenden Fahrzeugen muss auch sie verändert werden. Man will aber laut Definition des Musters die Konstruktion von der Repräsentation trennen, somit muss die Werkstatt unabhängig vom zu bauendem Fahrzeug bleiben, denn sie erzeugt ja die Objekte, sprich die Fahrzeuge.

Bessere Lösung: Besser wäre es eine Schnittstelle zu definieren, die Operationen zum Bau von Teilen aller möglichen Fahrzeuge bietet.

Abbildung 6 zeigt, wie diese Anwendung mit dem *Erbauer* Muster realisiert ist. Als Schnittstelle definieren wir die Klasse *FahrzeugBauer*, welche Funktionen zum Bauen von Fahrzeugteilen anbietet (*BaueRahmen*(), *BaueRaeder*(), *BaueMotor*() und *BaueTueren*()). Weiterhin definieren wird drei Unterklassen *FahrradBauer*, *MotorradBauer* und *AutoBauer*, welche alle Funktionen der Oberklasse überschreiben, die sie zum Bau des Fahrzeugs benötigen. Weiter ist auch noch jeweils eine Funktion *GibFahrzeug*() zum Zurückgeben des Fahrzeugs vorhanden. Der *FahrradBauer* muss also nur die Funktionen *BaueRahmen*() und *BaueRaeder*() überschreiben, denn

einen Motor und Türen hat ein Fahrrad ja nicht. Wir stellen fest, dass die Oberklasse die Vereinigungsmenge der Funktionen besitzen muss, die ihre Unterklassen besitzen. Dieses ganze Konstrukt, d.h. die Oberklasse *FahrzeugBauer* und ihre Unterklassen nennt man den *Erbauer*. Die Klasse *Werkstatt* kommt nun ins Spiel, um konkrete Fahrzeuge zu bauen. Sie wird mit einem Objekt von *FahrzeugBauer* konfiguriert und ruft dann in ihrer *Konstruiere*() Funktion alle Methoden der Klasse *FahrzeugBauer* auf. Natürlich wird nicht direkt die Klasse *FahrzeugBauer* instanziiert, sondern deren Unterklassen welche aber wegen der Vererbung auch *FahrzeugBauer* sind. Die Klasse *Werkstatt* ist also unabhängig vom Typ der *FahrzeugBauer* Objekte, denn sie ruft wie gesagt nur die Funktionen der Oberklasse auf, welche je nach Fahrzeugart (Fahrrad, Motorrad oder Auto) überschrieben worden sind.

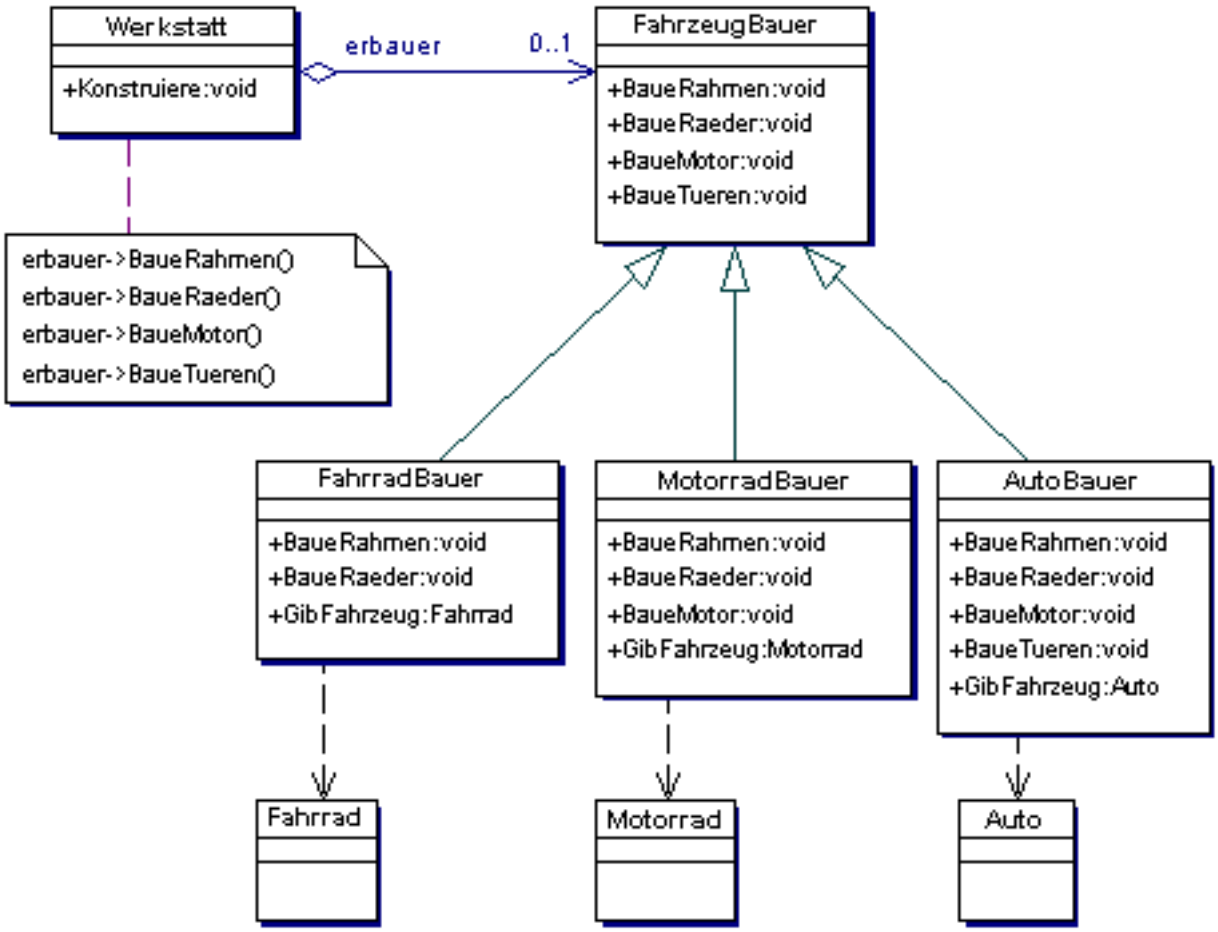


Abbildung 6: Struktur der Beispielimplementierung *FahrzeugBau*

4.3 Struktur

Abbildung 7 zeigt die allgemeine Struktur des Musters. Die Klasse *Erbauer* bietet Methoden zum Bau verschiedener Teile eines Produktes. Der konkrete Erbauer überschreibt die Methoden der Oberklasse die er benötigt, um ein Produkt zu bauen. Weiter gibt es im konkreten Erbauer eine Funktion *GibErgebnis*(), um das fertige Produkt zurückzugeben. Der Direktor wird mit einem *Erbauer* konfiguriert und ruft dann alle Methoden der Schnittstelle zum Bau von Produktteilen auf.

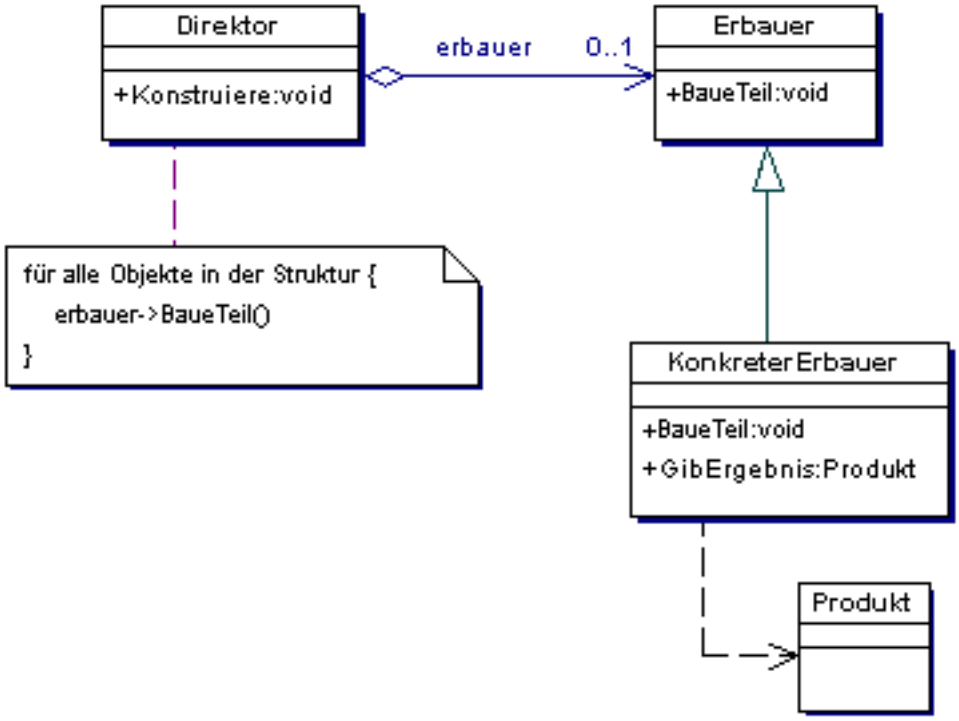


Abbildung 7: Allgemeine Struktur des Erbauer Musters

4.4 Teilnehmer

Erbauer: Schnittstelle zum Erzeugen von Teilen eines Produktobjektes (*FahrzeugBauer*).

KonkreterErbauer: Implementiert die Erbauerschnittstelle und konstruiert Teile des Produkts (*FahrradBauer*, *MotorradBauer*, *AutoBauer*). Bietet auch eine Schnittstelle zum Zurückgeben des Produkts (*GibFahrzeug*()).

Direktor: Verwendet die Erbauerschnittstelle um Objekte zu konstruieren (*Werkstatt*).

Produkt: Repräsentiert das gerade konstruierte komplexe Objekt (*Fahrrad*, *Auto*).

4.5 Interaktion (Dynamik des Musters)

Nun wollen wir das dynamische Verhalten des Musters betrachten. Es wird zuerst ein Objekt eines konkreten Erbauers angelegt. Danach wird die Klasse *Direktor* instanziiert, welcher dann das Objekt des konkreten Erbauers als Parameter mitgegeben wird, d.h. das *Direktor* Objekt wird mit einer konkreten Fabrik konfiguriert. Sind beide Objekte erzeugt, wird die *Konstruiere*() Methode des Direktors aufgerufen, in welcher nun alle Methoden der Schnittstelle *Erbauer* aufgerufen werden, um Teile eines Produktes zu konstruieren. Da der Direktor mit einem konkreten Erbauer Objekt konfiguriert wird, alle konkreten Erbauer aber wegen der Vererbung auch *Erbauer* sind, ist der Direktor also unabhängig und verwendet immer den selben Konstruktionsprozess, um verschiedene komplexe Objekte zu konstruieren.

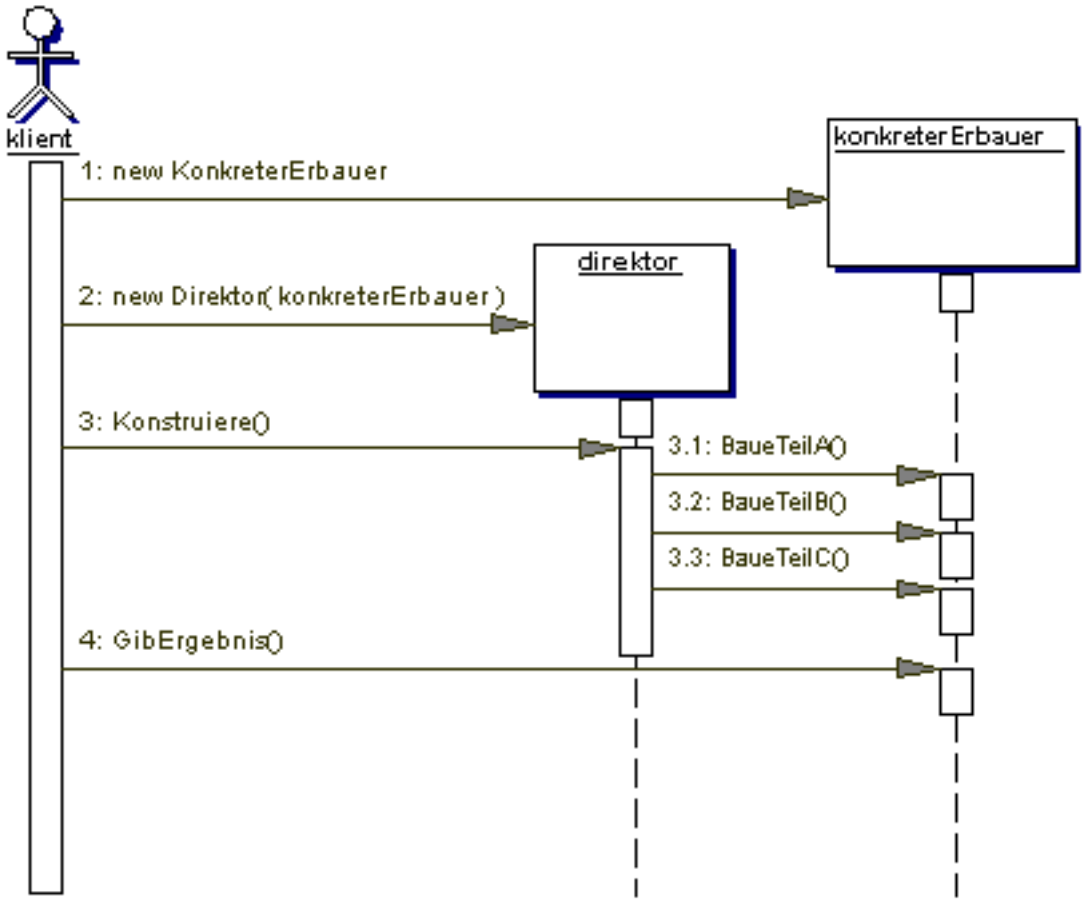


Abbildung 8: Dynamischer Verlauf des Musters *Builder*

4.6 Implementierung

Als *Erbauer* Klasse wählen wir die Klasse *FahrzeugBauer*, welche Funktionen zum Erzeugen von Teilen eines Produktes enthält. Diese Funktionen sind leer implementiert.

```
class FahrzeugBauer {
    ...
    virtual void BaueRahmen() {}
    virtual void BaueRaeder() {}
    virtual void BaueMotor() {}
    virtual void BaueTueren() {}
};
```

Als konkreter Erbauer wird hier nur der *MotorradBauer* aufgeführt. Dieser konkrete Erbauer erbt vom *FahrzeugBauer* und überschreibt die Methoden, welche für den Bau diese Produktes (Motorrad) relevant sind.

```
class MotorradBauer : public FahrzeugBauer{
    ...
    virtual void BaueRahmen() {
        SetzeRahmen( 1 );
    }
    virtual void BaueRaeder() {
        SetzeRaeder( 2 );
    }
    virtual void BaueMotor() {
        SetzeMotor( 1 );
    }
};
```

Den *Direktor* implementieren wir als Klasse *Werkstatt*, welche eine Methode *KonstruiereFahrzeug*() besitzt, in der alle Methoden der Klasse *FahrzeugBauer* aufgerufen werden. Wenn im konkreten Erbauer nicht alle Methoden der Oberklasse überschrieben werden, werden die jeweiligen leer implementierten Methoden der Basisklasse aufgerufen. Die Klasse *Werkstatt* wird mit einem Objekt vom Typ *FahrzeugBauer* konfiguriert, d.h. der Kunstruktor der Klasse *Werkstatt* bekommt ein Objekt vom Typ *FahrzeugBauer* übergeben. Unabhängig vom Typ des Objektes wird immer der selbe Konstruktionsprozess verwendet, um dann aber abhängig vom Typ verschiedene Fahrzeuge schrittweise zu konstruieren. Die Werkstatt ist also unabhängig von der Repräsentation der Objekte, verwendet aber immer den selben Konstruktionsprozess, um diese zu erzeugen.

```
class Werkstatt {
    ...
    Werkstatt( FahrzeugBauer* erbauer ) {
        m_erbauer = erbauer;
    }
    void KonstruiereFahrzeug() {
        m_erbauer->BaueRahmen();
        m_erbauer->BaueRaeder();
        m_erbauer->BaueMotor();
        m_erbauer->BaueTueren();
    }
};
```

4.7 Anwendbarkeit

Das Builder Muster kann verwendet werden wenn:

- der Konstruktionsprozess verschiedene Repräsentationen des zu konstruierenden Objekts erlauben muss.
- der Algorithmus zum Erzeugen eines komplexen Objekts unabhängig von den Teilen sein soll aus denen es besteht und wie sie zusammengesetzt werden, in unserem Beispiel also die Methode *KonstruiereFahrzeug*(). Der Algorithmus beeinhaltet die Funktionsaufrufe, um die einzelnen Teile der Fahrzeuge zu konstruieren. Dieser Algorithmus ist natürlich auch unabhängig davon aus welchen Teilen das Fahrzeug besteht, denn es wird immer wieder derselbe verwendet, um verschieden Fahrzeuge zu konstruieren.

4.8 Vor- und Nachteile

- Vorteile
  - Variation der internen Repräsentation eines Produktes.
  - Isolierung des Codes zur Konstruktion und Repräsentation: Der Klient weiss nichts über die interne Struktur des Produktes, da nur die Schnittstelle verwendet wird.
  - Genauere Steuerung des Konstruktionsprozesses: In unserem Beispiel wäre es natürlich möglich, unter gewissen Umständen, den Bau des Motors vor den Bau der Räder zu ziehen. Somit ist wie gesagt eine genauere Steuerung möglich.

- Nachteile
  - Wenige Übereinstimmungen in der Schnittstelle bei zu unterschiedlichen konkreten Erbauern. Es muss die Vereinigunsmenge der Funktionen aller Unterklassen gebildet werden. Wenn die Schnittmenge aller Funktionen nun sehr gering oder gar leer ist, müssen alle Funktionen aller Unterklassen in der Oberklasse aufgeführt werden, was diese Klasse dann natürlich sprengen könnte.

4.9 Bekannte Verwendung

Eine bekannte Verwendung des *Builder* Musters ist das JDBC API von JAVA. Als *Erbauer* dient die Klasse *Driver*.

```
java.sql.Driver
    -> Connection connect( String s, Properties p )
```

Wenn man sich nun z.B. einen JDBC Treiber für die mySQL Datenbank aus dem Internet runterlädt, findet man in dieser Paketstruktur auch eine Klasse *Driver*, welche die Klasse *java.sql.Driver* implementiert.

```
org.gjt.mm.mysql.Driver
    -> Connection connect( String s, Properties p )
```

Als *Direktor* dieser Struktur, dient die Klasse *DriverManager*.

```
java.sql.DriverManager
    -> Connection getConnection( String s )
```

Der *DriverManager* wird mit einem Objekt vom Typ *Driver* konfiguriert, enthält unter anderem die Methode *getConnection*() welche die *connect*() Methode des konfigurierten Objekts aufruft, um eine Verbindung zur Datenbank herzustellen. Das entstandene Produkt ist die *Connection*.

In Abbildung 9 wird dies näher veranschaulicht.

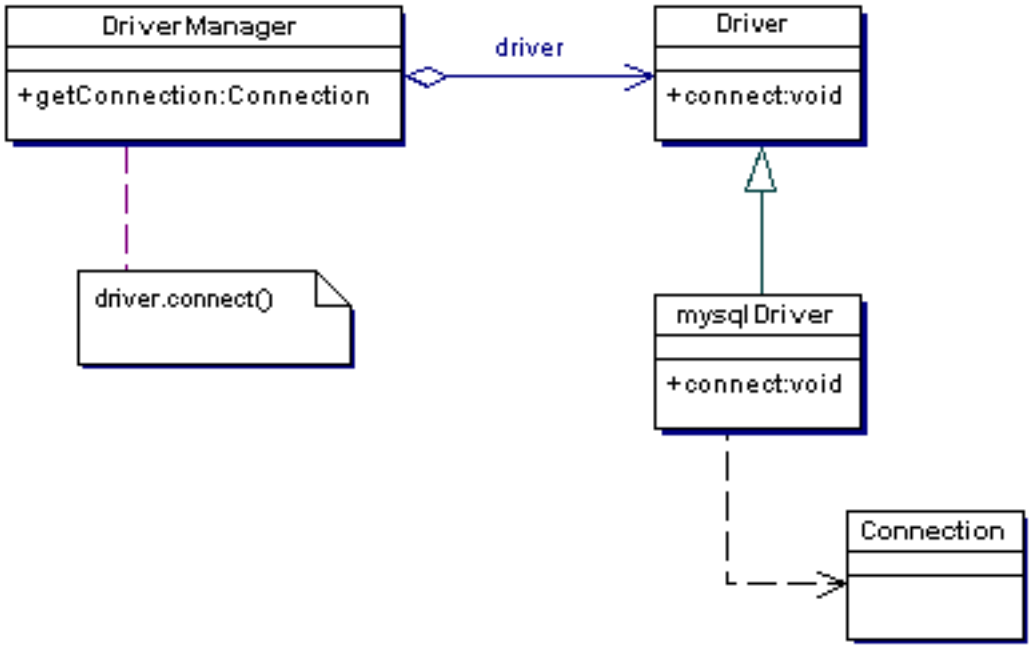


Abbildung 9: Verwendung des Builder Musters in JDBC

4.10 Verwandte Muster

- Ähnlichkeiten zum *AbstrakteFabrik* Muster
  - Es werden ebenfalls komplexe Objekte konstruiert.
- Unterschiede zum *AbstrakteFabrik* Muster
  - Der Konstruktionsprozess läuft beim *Erbauer* Muster schrittweise ab.
  - Der *Erbauer* gibt das Objekt als letzten Schritt zurück, die *AbstrakteFabrik* unmittelbar.

5 Beispielprogramm (Builder / Erbauer)

Zum Abschluss des Musters, ein Screenshot der Anwendung FahrzeugBau.

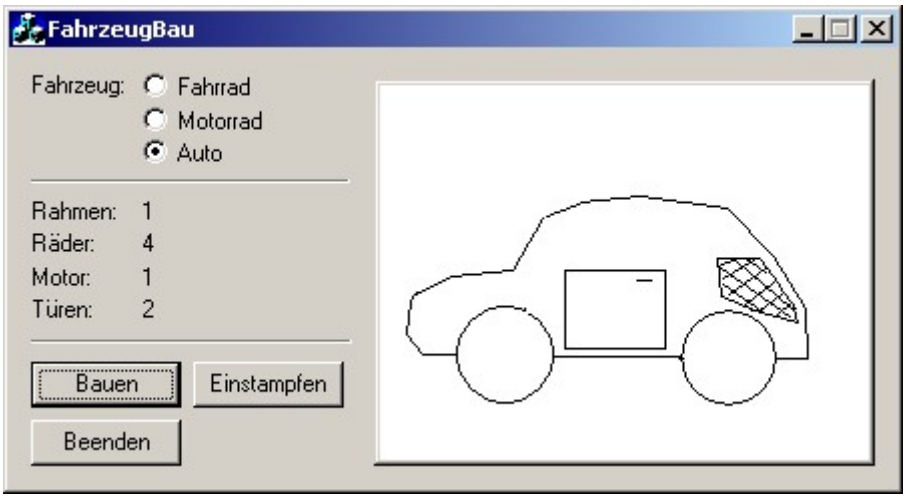


Abbildung 10: Screenshot der Beispielimplementierung

6 Fazit

Als Fazit möchte ich gerne nochmal aufzeigen, wann man konkret welches Muster verwenden kann.

Beide möglich: Wenn komplexe Objekte erzeugt werden sollen, ihre Konstruktion aber verborgen bleiben soll.

AbstrakteFabrik: Wenn ein System mit einer von mehreren Produktfamilien (Look&Feel) konfiguriert werden soll.

Erbauer: Wenn der Ablauf, wie komplexe Objekte erzeugt werden, unabhängig von ihren Teilen und derer Repräsentation sein soll.

7 Quellen

1. Entwurfsmuster
   Gamma, Helm, Johnson, Vlissides
   Addison-Wesley 2001

2. Gang of Four Design Patterns
   http://www.tml.hut.fi/ pnr/Tik-76.278/gof/html/

3. The Design Patterns Java Companion
   http://www.patterndepot.com/put/8/JavaPatterns.htm

4. Patterns in Java AWT
   http://www.soberit.hut.fi/tik-76.278/group6/awtpat.html

5. Design Patterns
   http://www.dofactory.com/patterns/patterns.asp

Strategy Pattern

Knut Enners

29. Juli 2002

Seminar Entwurfsmuster Sommersemester 2002

Inhaltsverzeichnis

1 Entwurfsmuster Strategy

1.1 Einleitung und Definition

Ein Standardproblem der Softwareentwicklung ist das Adaptieren des Verhaltens einer Anwendung, abhängig aus Benutzereingaben oder der Umgebung. So führt in einem Textverarbeitungsprogramm die Auswahl von verschiedenen Umbruchsystemen (z.B. Blocksatz oder linksbündig) zu unterschiedlichem Aussehen des Textes. Ein Beispiel für ein umgebungsgesteuertes System wäre ein Verkehrsleitsystem, das abhängig vom Füllungsgrad der Strassen arbeitet. Mit VERHALTEN wird in diesem Zusammenhang die Art bezeichnet, wie eine Anwendung auf Einflüsse reagiert (Sicht des Benutzers). Der Programmierer realisiert die verschiedenen Verhaltensweisen durch STRATEGIEN oder ALGORITHMEN, die diese implementieren. Die Applikation entscheidet sich abhängig vom gewünschten Verhalten für eine dieser Strategien. Ein Entwurfsmuster, das sich mit diesem Problem beschäftigt, ist STRATEGY[1]:

"*Definiere eine Familie von Algorithmen, kapsele jeden einzelnen von ihnen und mache sie austauschbar.*" [Gamma95, object behavioral]

An einem Fallbeispiel wird das Prinzip des Musters erklärt, näher beleuchtet und Gründe für seine Anwendung diskutiert. Gewählt wurde ein Routenplaner, der verschieden Strategien verfolgt, um einen Weg zwischen zwei Orten zu ermitteln.

1.2 Prinzip

Zunächst wird die gemeinsame Schnittstelle der Algorithmen in einer Klasse definiert; dies geschieht üblicherweise durch Definitionen der benötigten Konstruktoren und Funktionen mit leerem Rumpf. Im Muster wird diese Klasse als *Strategy* bezeichnet. Die Algorithmen werden in den *Concrete-Strategy* Klassen

[1] auch Policy genannt

definiert, die von *Strategy* abgeleitet werden. Die *Context* Klasse verwaltet die benötigten Daten und hält eine Referenz auf eine konkrete Strategie[2]. Insbesondere sollte der Kontext eine Methode zur Verfügung stellen, diese Referenz zu setzen[3]; man spricht auch von der Konfiguration des Kontexts mit einer konkreten Strategie.

Die Algorithmen werden genutzt, indem der *Client* (nicht in der Abb.) ein Objekt des Kontexts erzeugt, ggf. mit weiteren Daten initialisiert und mit einem Objekt einer konkreten Strategie konfiguriert. Über den Kontext kann der Client nun auf die Algorithmen zugreifen. Eine genauere Betrachtung dieses Musters anhand eines Beispiels soll dieses Konzept verdeutlichen.

1.3 Anwendung

Aufgabe Es soll ein Navigationssystem entwickelt werden, das zwischen gegebenen Orten Routen berechnen kann (siehe Abb. 1.2). Das Programm soll dabei Wünsche des Anwenders berücksichtigen, wie z.B.

[2] i.F. werden die deutschen Begriffe verwendet

[3] eine Standard Strategie sollte bei der Erzeugung eines Kontext-Objekts dem Konstruktor übergeben werden

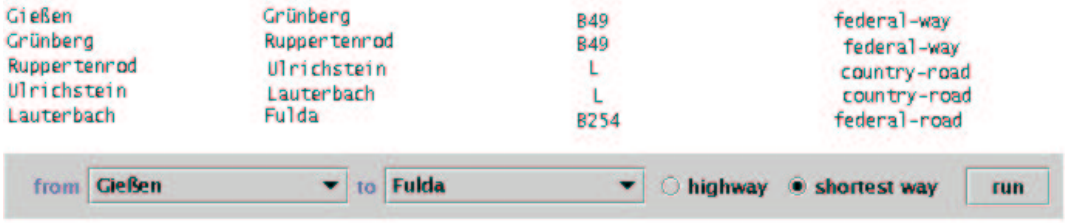
Abbildung 1.1: allgemeines Muster

„kürzester Strecke" oder „schnellste Verbindung".

Implementierung Die Klasse „Navigator" (siehe Abb.) implementiert die Schnittstelle zu den verschiedenen Strategien zur Routenberechnung. In diesem Fall eine Methode „Route berechneRoute(Ort von, Ort nach)", die Start- und Zielort übergeben bekommt und die gewünschte Route zurückgibt. Die konkreten Strategien, die von ihr abgeleitet sind, implementieren die Algorithmen, in denen die Route tatsächlich berechnet wird.

Der Kontext „Routenplaner" bietet eine Schnittstelle auf die Daten, die die Strategien benötigen: die Strassenkarte. An dieser Stelle interessieren weniger die Datenstrukturen, die die

Abbildung 1.2: Screenshot der Anwendung



Orte mit ihren Verbindungen darstellen. Es wurde ein einfaches Modell gewählt (siehe Abb. 2), dessen Objekte nach dem Programmstart mittles *Castor*[4] aus einer XML Datei vom Klienten „unmarshalled" werden.

Der Klient „Navigationssystem" (nicht in der Abb.) sorgt für die Interaktion mit dem Benutzer (realisiert mit Java Swing) und initialisiert den Routenplaner mit den benötigten Daten. Je nachdem welchen Routentyp der Benutzer wünscht, wird der Routenplaner mit der entsprechenden konkreten Strategie konfiguriert. Der Aufruf „berechne Route" seitens des Benutzers führt dazu, dass das Navigationssystem-Objekt die entsprechende Methode via dem Routenplaner „Route berechneRoute(Ort von, Ort nach)" mit den ausgewählten Orten aufruft und das Ergebnis, die gewünschte Route, darstellt.

[4] siehe castor.exolab.org für weitere Informationen

1.4 Anwendbarkeit

1. Viele Klassen unterscheiden sich nur in ihrem Verhalten.
2. Eine Klasse implementiert viele Verhaltensweisen.
3. Es werden verschiedene Versionen von Algorithmen benötigt.
4. Daten, die der Algorithmus verwendet, sollen vor dem Klienten geheimgehalten werden.
5. Der Algorithmus muss keinen Zustand über mehrere Aufrufe hinweg speichern.
6. Klient braucht kein Wissen über Implementierungsdetails.
7. Keine starke Variation der Parameter der konkreten Strategien.

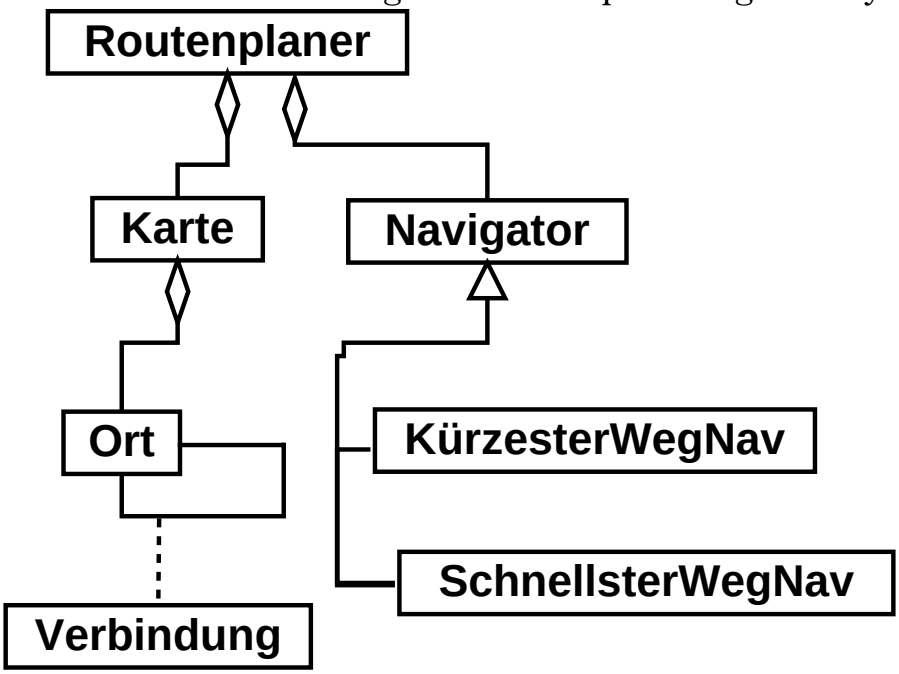
- Alternativ könnte man auf die Schnittstellenklasse verzichten und die konkreten Strategien direkt benutzen (Punkt 1). Dies hätte zur Folge, dass die

Abbildung 1.3: Ausschnitt der Implementierung

```
class Routenplaner {
   private Karte karte;
   private Navigator nav;
   ...
   public void konfiguriere(Navigator navigator) {
      nav = navigator;
   }
...
}

class Navigationssystem
      extends JFrame implements ActionListener {
   ...
   public void actionPerformed(ActionEvent e) {
      if(e.getActionCommand().equals("run")) {
         ...
         //Benutzer hat Option 'kuerzester
         //Weg' gewaehlt
         if(kurzerWeg.isSelected())
            routenPlaner.konfiguriere(
                  new KuerzesterWegNavigator());
      ...
}
```

Abbildung 1.4: Fallbeispiel Navigationssystem



Schnittstellen schwerer zu warten wären, da prinzipiell jede Implementierung der einzelnen Algorithmen eine andere zur Verfügung stellen könnte. Schwerer wiegt jedoch, dass der Kontext alle Strategien kennen müsste und eine weitere Strategie folglich zum Neukompilieren des Kontexts führt.

- In zweiten Fall müsste man mit einem weiteren Parameter das gewünschte Verhalten identifizieren und z.B. mit einer komplexen switch-case Konstruktion dann zu den einzelnen Implementierungen verzweigen. Sind viele Verhaltensweisen vonnöten würde dies zu einer grossen Klasse führen. Weiterhin bleibt ein Fehler bei diesem Parameter während der Kompilierzeit unentdeckt.
- Ein häufiger Anwendungsfall ist das Konfigurieren einer Applikation bezüglich ihres Einsatzes mit verschiedenen Versionen eines Algorithmus. Zum Beispiel könnte man eine schlankere Version in einem embedded System in Kraftfahrzeugen einsetzen als auf einer PC Distribution.
- Die Kapselung der benötigten Daten im Kontext erleichtert die Wartung und Portierbarkeit (Punkt 4).
- (Punkt 5) Sollte es nicht mög-

lich sein, Zustände im Kontext zu verwalten, kann man nicht mehr vom Entwurfsmuster *Strategy* sprechen. Man sollte auch bedenken, dass es ein Indikator für ein schlechtes Design ist, wenn Algorithmen über mehrere Aufrufe hinweg Zustände speichern müssen.

- Wenn die Strategien mit stark variierenden Parametern aufgerufen werden müssen, führt dies zu einer langen Parameterliste der allgemeinen Schnittstelle. Die einzelnen Strategien verwenden allerdings nur wenige der übergebenen Werte - ein Einsatz des Strategy-Entwurfsmusters ist dann fraglich, denn der Overhead, den ein Muster mit sich bringt, sollte den Nutzen nicht übersteigen. Weiterhin ist die Gefahr gross, dass die allgemeine Schnittstelle bei neuen Strategien wiederum erweitert werden muss! Wann immer diese Gefahr besteht, sollte dieses Entwurfsmuster nicht verwendet werden, da dann alle beteiligten Klassen angepasst werden müssten.

1.5 Bekannte Anwendungsbeispiele

- InterViews Textformatierungssystem
- Compiler: RTL Register Allokierungsstrategien
- ET++ SwapsManager Engine[5]
- Java Servlets aus Sicht des Application Server

Anmerkungen zum letzten Beispiel: Diese Antwort bekam ich in [NEWS] auf die Frage nach bekannten Anwendungsfällen. Die Idee ist, dass in der Java Servlet Spezifikation eine Klasse Servlet für die allgemeine Schittstelle definiert wird, von der aber nie direkt Instanzen erzeugt werden (*Strategy).* Der Anwendungsprogrammierer implementiert mit seinen von Servlet abgeleiteten Klassen die konkreten Strategien und die JSPs dienen als *Client.*

1.6 Variationen

Variationen betreffen im Allgemeinen Implementierungsdetails, wie z.B. kann der Kontext an die Strategie

[5]Diese Anwendungsfälle konnte ich nicht durch Codeinspektion nachvollziehen

übergeben werden, anstatt dass diese über die Kontextschnittstelle auf relevante Daten zugreift (nur Referenz auf Kontext). Ein umfassende Analyse müsste an dieser Stelle unter anderem auf Aspekte der Programmiersprachen eingehen, da Entscheidungen wie letztere z.B. davon abhängig sind, wie Objekte übergeben werden (Kopie, Referenz, Zeiger). Eine derartige Analyse würde den Rahmen dieses Seminars sprengen.

Die Strategien müssen nicht immer - wie bisher beschrieben - dynamisch austauschbar sein, auch eine statische Bindung findet Anwendung. In diesem Fall spricht man von einer Konfiguration des Systems.

1.7 Verwandte Muster

Proxy Verschiedene SERVICES werden bei der gemeinsamen Schnittstelle PROXY registriert, die dann der Client via Proxy nutzen kann. Der Unterschied besteht darin, dass der Proxy im Gegensatz zur STRATEGY aktiv[6] ist: Er wählt einen Service, „mapped" benötigte Informationen passend zu dessen Schnittstelle, nimmt an der Stelle des Clients den Service in Anspruch und gibt Ergebnisse an diesen zurück. Insbesondere ist es nicht eine ungewollte Ausnahme, wenn sich die Schittstelle zu den Services ändert; das Mapping der Anfragen auf die Schnittstelle der Services ist vielmehr eine zentrale Aufgabe des Proxy.

[6] offensichtlicher Unterschied: Es gibt Proxy Instanzen, aber keine von Strategy !

1.8 Fazit

Anmerkung Die Beschreibung dieses Entwurfsmuster in [Bosch01, Kap. 2.6] kann ich nicht nachvollziehen: *„By implementing the strategy pattern the client[7] need not decide which external service has to be used. But this pattern includes more intelligence than the Chain of Responsibility Pattern, because it is provided with a dedicated knowledge-base system and uses heuristic rules based on experience and guesses."* Dieses Entwurfsmuster erlaubt es dem Client, dynamisch zwischen verschiedenen Algorithmen zu wechseln, es wird aber keine Aussage darüber gemacht, nach welchen Kriterien diese ausgewählt werden. Diese Aufgabe bleibt dem Client selbst überlassen, ob er sie nun durch Benutzerinteraktion oder mittels heuristischer Regeln

[7] An dieser Stelle ist wohl der User als Client gemeint, nicht zu verwechseln mit dem member Client des Musters.

löst ist nicht für das Muster relevant.

Bewertung Solange keine Änderungen an der Algorithmenschnittstelle zu erwarten ist, stellt dieses Muster ein einfaches und effektives Konzept zur Behandlung des besprochenen Problems dar. Möglicherweise wird man das Muster selten in „Reinform" antreffen, wohl aber die dahinterstehende Grundidee.

Literaturverzeichnis

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Design Patterns; Addison Wesley, 1995

[Vlissides98] John Vlissides; Pattern Hatching; Addison Wesley, 1998

[Bosch01] Claudia Fritsch; Arbeitsunterlage Schwerpunktseminar Mechanismen; Bosch, 2001

[NEWS] comp.software.patterns; Newsgroup

[IBM] John Vlissides (Maintainer); Research Software Patterns; IBM Watson Research Center; www.research.ibm.com/designpatterns

Abbildungsverzeichnis

Seminar Entwurfsmuster SS2002

Decorator

Reza Rashidi
3. Juli 2002

Inhaltsverzeichnis

Zusammenfassung

Trotz vielfältiger Problemstellungen in der Welt der Software stoßen wir doch häufig auf ähnliche Strukturen, die in den Lösungen dieser Probleme auszumachen sind. Nach den Vorbildern in der gegenständlichen Welt der Architektur haben sich Softwareingenieure auf die Suche nach Mustern begeben, die das Verständnis und somit die Lösung ihrer Probleme erleichtern.

In diesem Text befasse ich mich mit dem Entwurfsmuster Decorator. Die Basis meiner Recherche bildet das klassische Buch von Design Patterns [1] . Auch im Internet fand ich viele Verweise auf dieses Werk, das nicht nur zum Lesen, sondern zur Arbeit in kleinen Gruppen empfohlen wird [1].

Ich richte mich bei der Abhandlung dieses Musters weitgehend an der bewährten Struktur des Buches, da ich sie als übersichtlich und einheitlich empfunden habe. Ein weiterer Vorteil dieses Vorgehens sehe ich in der Möglichkeit, die in der Vortragsreihe vorgestellten Muster besser zu verstehen und leichter miteinander vergleichen zu können.

[1] Es gibt verschiedene Working Groups, die sich intensiv mit dem Buch auseinandersetzen. http://www.industriallogic.com/papers/learning.html

1 Was sind Muster?

Im Wörterbuch Wahrig findet man folgende Beschreibung:

> Vorlage, Modell; Vorbild, Vollkommenes in seiner Art;<Gramm.> Beispielwort od. -satz, Paradigma;

Der Ursprung von Entwurfsmustern liegt in der Arbeit vom Architekten Cristopher Alexander in den achziger Jahren. Er schrieb damals zwei Bücher, *A Pattern Language* und *A Timeless Way of Building*, die zusammen mit Beispielen seine Logik des Dokumentierens von Mustern beschreiben.
Danach wurde es still um Design Patterns bis 1987. Seither erschienen viele Bücher und Präsentationen, verfasst von Experten wie Grady Booch, Richard Helm, Erich Gamma und Kent Beck. Im weiteren Verlauf gab es bis 1995 viele Zeitschriften, die Artikel mit direktem oder indirektem Bezug zu Mustern herausgaben. 1995 erschien *Design Patterns* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides.
Was bedeutet aber Entwurfsmuster in unserem Kontext, also wenn wir von Software reden? Die Definiton leuchtet besser ein, wenn wir wissen, was alles dazu gehört. Sie finden im nächsten Abschnitt die Definition von Entwurfsmustern.

1.1 Dokumentation von Mustern

Das Schreiben eines Aufsatzes sollte definierten Regeln folgen , so auch die Dokumentation von Entwurfsmustern. Allgemein muß eine solche Dokumentation folgendes beachten:

- Die Motivation oder der Kontext, in dem das Problem auftritt
- Vorbedingungen, die bei der Auswahl eines Musters erfüllt sein müssen
- Die Beschreibung der Programmstruktur, die das Muster definiert
- Konsequenzen aus dem Gebrauch des Musters, Vor- und Nachteile
- Beispiele

Für die Welt der Software gilt dann folgende Definition:

> Ein Muster beschreibt ein bestimmtes, in dem bestimmten Entwurfskontext häufig vorkommendes Problem und liefert ein erprobtes Schema zu seiner Lösung. Dieses Lösungsschema beschreibt die beteiligten Komponenten, welche Zuständigkeiten sie haben, wie sie miteinander verbunden sind und wie diese Komponenten zusammenarbeiten.

Ein Muster umfaßt demnach drei wesentliche Merkmale:

- Kontext: Der Kontext liefert eine Beschreibung der Situation, in der das Problem auftritt.
- Problem: Es gibt eine Beschreibung des Problems, das immer in einem bestimmten Kontext wiederholt auftaucht.
- Lösung: Dieser Teil der Musterbeschreibung zeigt, wie das Problem gelöst werden kann.

1.2 Decorator

Es gibt insgesamt drei Arten von Entwurfsmustern:

- Erzeugende Muster
- Strukturelle Muster
- Verhaltensmuster

Das Buch Design Patterns zittierend, gehört das Muster Decorator (Dekorierer) zu den strukturellen Mustern. Man findet Decorator in anderen Quellen unter Verhaltensmuster.

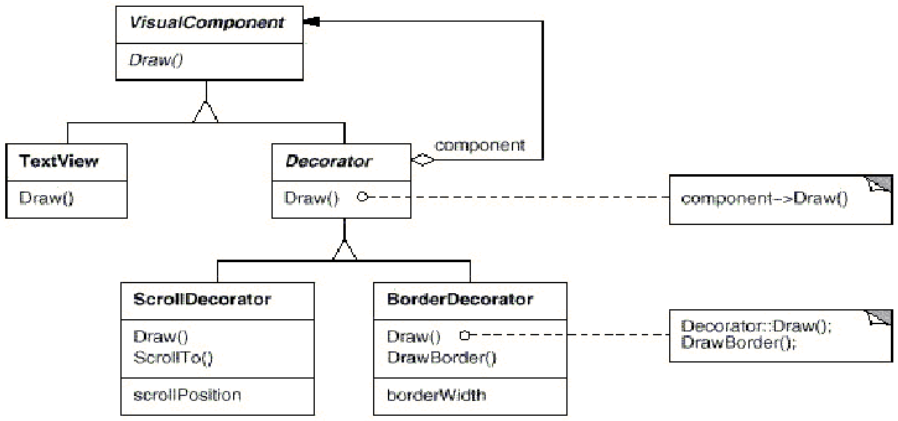


Abbildung 1: TextView mit Decorators

1.2.1 Zweck

Wenn es darum geht, ohne Ableitung und eine Vielzahl von Klassen die Verantwortlichkeiten eines Objekts dynamisch zu ändern, kommt der Decorator zum Einsatz. Dieses Muster ist auch als Wrapper bekannt. Wie wir in 1.2.8 sehen werden, hüllen Decorator Objekte das dekorierte Objekt ein, so dass Methoden der dekorierten Komponente weiterhin verfügbar sind und es für Clients keinen Unterschied macht, ob sie mit einem dekorierten oder nicht dekorierten Objekt arbeiten. Daher werden sie auch als durchsichtige Hülle bezeichnet.

1.2.2 Motivation

Manchmal möchte man einzelnen Objekten zusätzliche Funktionalität geben, ohne dass die ganze Klasse davon betroffen ist. Beispielsweise sollte eine graphische Benutzeroberfläche eines Editors fähig sein, Komponenten wie Rand oder Verhalten wie Bildlaufleiste zur Verfügung zu stellen. In diesem Beispiel ist es TextView, das Text-Anzeigeprogramm, das um neue Features erweitert werden soll.
In diesem Kontext scheint die Vererbung als erste Möglichkeit geeignet. So werden alle Unterklassen den Rand von der Oberklasse vererben, aber diese Erweiterung ist statisch. Ein Client kann nicht über den Zeitpunkt und über die Art der Erweiterung entscheiden.
Eine noch flexiblere Vorgehensweise ist das Kapseln der Komponente in ein anderes Objekt, das den Rand erzeugt. Dieses kapselnde Objekt heißt Decorator. Ein Decorator Objekt paßt sich der Schnittstelle der Komponente an, so dass er für Benutzer der Komponente transparent ist. Der Decorator leitet Serviceanfragen an die Komponente weiter und führt eventuell zusätzliche Aktionen durch (den Rand zeichnen), vor bzw. nach der Weiterleitung des Requests. Durch die Transparenz ist man in der Lage, Decorators in Mengen einzusetzen und so eine unbegrenzte Zahl von Verantwortlichkeiten zu ermöglichen.
Ein Objekt von TextView zeigt Text in einem Fenster an. TextView hat ursprünglich keinen Rand und keine Bildlaufleiste, da man diese nicht immer benötigt. Bei Bedarf kann man ein Objekt des BorderDecorator und/oder ScrollDecorator zu TextView zusammenzustellen und dadurch das gewünschte Ergebnis erreichen. Folgendes Diagramm in Abbildung 1 zeigt, wie ein TextView Objekt um einen Rand und um einen Scrollbar ergänzt werden kann.
Die Klassen ScrollDecorator und BorderDecorator stammen von Decorator ab. Das ist eine abstrakte Klasse für visuelle Komponenten, die sie dekorieren.
VisualComponent ist eine abstrakte Klasse für visuelle Objekte. Sie beschreibt ihre Schnittstelle und wie sie auf Ereignisse reagieren. Man sieht, wie die Klasse Decorator Zeichen-Anforderungen an ihre Komponenten weiterleitet und wie Unterklassen von Decorator diese Operationen erben.
Unterklassen von Decorator können beliebig Operationen für bestimmte Funktionalität hinzuaddie-

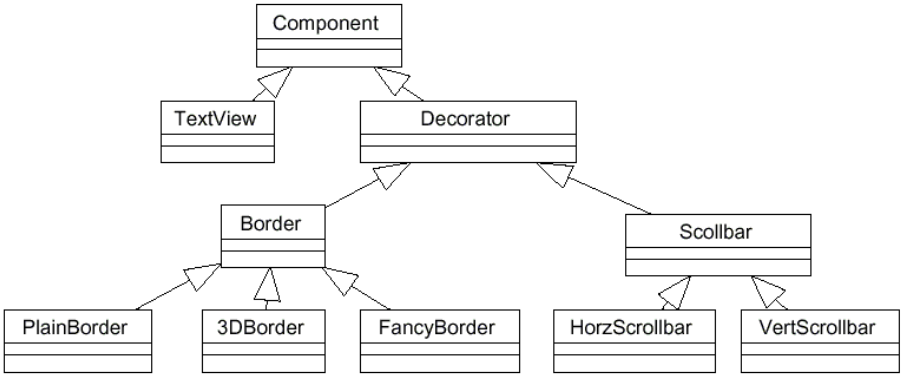


Abbildung 2: Vererbung endet in Variantenexplosion

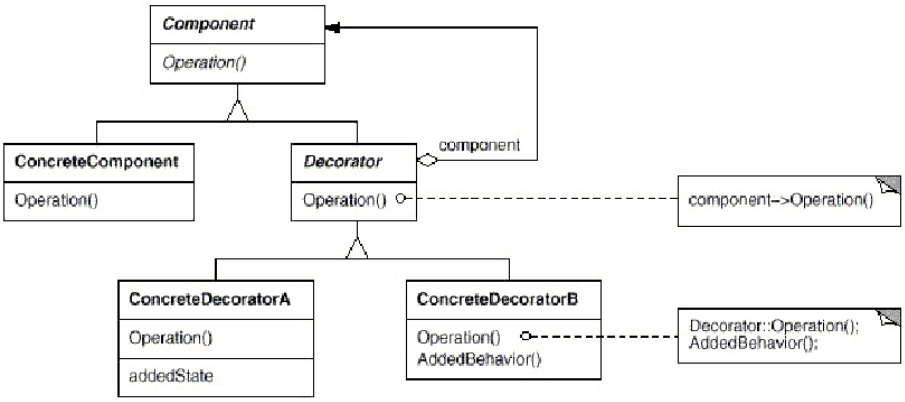


Abbildung 3: Struktur von Entwurfsmuster Decorator

ren. Der wichtige Aspekt bei diesem Muster ist, dass Decorators überall dort auftreten können, wo ein Objekt von VisualComponent erscheinen kann. Auf diese Weise könnte Benutzer dekorierte und nichtdekorierte Objekt nicht unterscheiden und sind daher unabhängig von Dekoratoren.

1.2.3 Anwendung

Decorators werden in folgenden Fällen eingesetzt:

- Einzelne Objekte bekommen dynamisch und transparent neue Verantwortlichkeiten, ohne andere Objekte zu tangieren.
- Wenn entbehrliche Zuständigkeiten eine Kernfunktionalität ergänzen sollen.
- Wenn die Erweiterung durch Vererbung unpraktisch ist. Es gibt Fälle, in denen die Erweiterung durch Vererbung zu einer Fülle von Unterklassen führt. Oder wenn die Klassendefinition versteckt ist bzw. nicht zum Ableiten zur Verfügung steht.

1.2.4 Struktur

Abbildung 3 zeigt die zentrale Rolle von *Decorator*, die diesem Muster Transparenz verleiht. *Component* definiert die notwendige gemeinsame Interface für Objekte der *ConcreteComponent* und potentielle *ConcreteDecorator*s.

1.2.5 Teilnehmer

Component
Component definiert die Schnittstelle für Objekte, denen dynamisch Verantwortlichkeiten auferlegt werden können (VisualComponent).
Decorator
Er besitzt einen Verweis auf ein Component-Objekt und definiert eine Schnittstelle, die der Schnittstelle von Component entspricht.
ConcreteDecorator
Es erweitert die Verantwortlichkeiten von Component (BorderDecorator, ScrollDecorator).

1.2.6 Zusammenarbeit

Decorator leitet Anfragen an sein Component-Objekt weiter. Eventuell führt er noch Aktionen vor bzw. Nach der Übermittlung der Nachfrage.

1.2.7 Konsequenzen

Nun kommen wir zu Vor- und Nachteilen, die mit dem Einsatz dieses Musters einhergehen:

1. Wir erreichen mehr Flexibilität als mit Vererbung. Das Decorator Muster bietet eine flexiblere Art, Objekten mehr Verantwortlichkeiten zu geben als mit statischer (mehrfacher) Vererbung. Mit Decorators kann zusätzliche Funktionalität hinzuaddiert bzw. weggenommen werden, indem man sie einfach zur Laufzeit dazunimmt bzw. entfernt. Vererbung hingegen geht durch Erzeugen einer neuen Klasse für jede weitere Funktionalität (z. B. BorderedScrolableTextView, BorderedTextView). Dies erhöht die Anzahl der Klassen und steigert die Komplexität des Systems. Ferner kann man durch unterschiedliche Decorator Klassen für eine bestimmte Component Klasse Verantwortlichkeiten miteinander mischen und Anpassungen vornehmen.

2. Wir vermeiden hoch entwickelte Klassen in den oberen Stufen der Hierarchie. Decorators bieten eine "pay-as-you-go" Lösung. Statt alle möglichen Eigenschaften in einer komplexen, maßgeschneiderten Klasse zu packen, kann man eine einfache Klasse definieren, der man schrittweise durch Decorator-Objekte mit neuen Features bestückt. Die Funktionalität wird so aus einfachen Stücken zusammengestellt. Anwendungen profitieren davon, indem sie nur für das bezahlen, was sie in Anspruch genommen haben. Es ist auch leicht möglich, neue Arten von Decorators zu definieren, unabhängig von Klassen der Objekte, die sie erweitern sollen. Dies sogar für unvorhergesehene Erweiterungen. Das Erweitern einer komplexen Klasse jedoch bringt die Gefahr mit sich, Einzelheiten preiszugeben, die mit beabsichtigten Erweiterungen nichts zu tun haben.

3. Ein Decorator und sein Component sind nicht identisch. Ein Decorator verhält sich wie eine transparente Hülle. Betrachtet man die Objekt-Identität, so ist ein dekoriertes Component nicht identisch zum Component selbst. Also man sollte sich nie auf Objekt-Identität verlassen, wenn man Decorators einsetzt.

4. Unschar kleiner Objekte. Ein Design, das Decorators benutzt, endet in ein System, das aus vielen kleinen Objekten besteht, die alle sehr ähnlich sind. Die Objekte unterscheiden sich in der Art, wie sie miteinander verbunden sind, nicht in ihren Klassen oder in den Werten ihrer Variablen. Obwohl diese Systeme von Fachkundigen leicht anzupassen sind, erweisen sie sich als schwer verständlich.

1.2.8 Implementierung

Bei dem Einsatz des Decorator Musters sollte man einige Punke beachten:

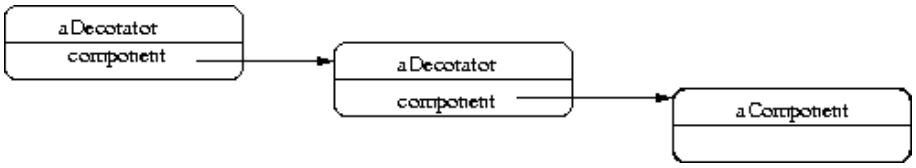


Abbildung 4: Komponente kümmert sich nicht um Decorators

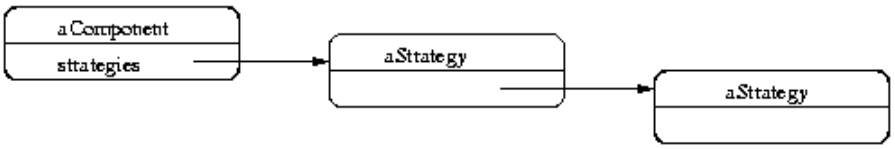


Abbildung 5: Strategies werden von Komponente verwaltet

1. Anpassung der Schnittstellen. Die Schnittstelle eines Decorator-Objektes muss der des Components gleich sein, das er dekoriert. Dementsprechend müssen Klassen von ConcreteDecorator von einer gemeinsamen Klasse abstammen (zumindest in C++).

2. Auslassen der abstrakten Decorator Klasse. Das Definieren einer abstrakten Decorator Klasse ist nicht nötig, wenn man nur eine Verantwortlichkeit addieren will. Das ist meist dann der Fall, wenn man es mit einer vorhandenen Klassen-Hierarchie zu tun hat und keine neue entwerfen will. Im letzteren Fall kann man die Verantwortlichkeit des Decorators - Er leitet Requests weiter an Component - zum ConcreteDecorator geben.

3. Component Klassen schmal halten. Um eine einheitliche Schnittstelle zu gewährleisten, müssen Components und Decorators von einer gemeinsamen Component Klasse stammen. Es ist wichtig, diese gemeinsame Klasse klein zu halten, d. h. sie hat nur die Aufgabe, eine Interface zu definieren und keine Daten zu sammeln. Die Definition der Datenstrukturen sollten die Unterklassen übernehmen, sonst würde die Komplexität der Component Klasse ihren Einsatz in höherer Anzahl zu schwer machen. Andererseits würde das Aufladen des Component mit vielen Funktionalitäten die Wahrscheinlichkeit erhöhen, dass Unterklassen für nicht benutzte Funktionalität zahlen müssen.

4. Den Schein statt die Eingeweide eines Objekts verändern. Wir stellen uns einen Decorator vor als eine Haut auf einem Objekt, die das Verhalten des Objekts ändert. Eine Alternative besteht darin, das Innere des Objekts zu verändern. Das ist die Art, wie das Strategy Muster vorgeht. Startegies sind bessere Alternativen in Fällen, wo die Component Klasse sehr schwergewichtig ist, so dass das Anwenden des Decorator Musters zu kostenträchtig wird. Beim Startegy Muster leitet das Component-Objekt einige seiner Verhalten an ein spezielles Strategy-Objekt. Dieses Muster lässt uns die Funktionalität eines Components ändern, indem das Startegy-Objekt ersetzt wird. Beispielsweise können wir unterschiedliche Bordertypen unterstützen, indem Component das Zeichnen der Border einem separaten Border-Objekt überlässt. Das Border-Objekt ist ein Strategy-Objekt, das das Border-Zeichnen kapselt. Durch Erhöhung der Anzahl von Strategies erreichen wir den gleichen Effekt wie durch rekursive Verschachtelung von Decorators. Da das Decorator Muster ein Component nur von Aussen ändert, muss der Component nichts über seine Decorators wissen, d. h. Decorators sind für den Component unsichtbar 4.

Bei Strategies weiss der Component über mögliche Erweiterungen. Daher muss ein Component seine Strategies kennen und pflegen 5.
Die strategy-basierte Lösung setzt Änderungen des Component voraus, um neue Erweiterungen zu ermöglichen. Auf der anderen Seite kann aber ein Strategy seine spezialisierte Schnittstelle haben, während die Schnittstelle des Decorators der des Components entspricht. Ein Strategy, zuständig

für das Zeichnen des Randes, muss nur die Schnittstelle zum Rand-Zeichnen definieren (DrawBorder, GetWidth), was zugleich bedeutet, dass das Strategy schmal sein kann, auch wenn die Component Klasse komplex ist.

1.2.9 Beispiel

Ein in C# implementiertes Beispiel 3 soll die Struktur dieses Musters und die Kollaboration der darin vorkommenden Teilnehmer veranschaulichen.
Wir haben gefordert, dass Erweiterungen von Objekten für deren Clients transparent und dynamisch sein sollen. Um die Transparenz zu erreichen, müssen die Komponente und die Decorators in ihrer Schnittstelle gleich sein. Das setzt voraus, dass sie von einer gemeinsamen Klasse abgeleitet sind. Diese Klasse definiert die Schnittstelle, die Basisoperation, die die Komponente garantieren muss.
Die konkrete Komponente erbt und implementiert diese Operation. Die abstrakte Klasse Decorator definiert die Beziehung zwischen der konkreten Komponente und den abzuleitenden konkreten Decorators.

```
public void SetComponent( Component component ){
    this.component = component;
  }
```

Diese Zeile in SetComponent bewirkt, dass zukünftige Decorators wissen, welches Objekt dekoriert werden soll.
Die konkreten Decorators werden von Decorator abgeleitet. Sie definieren die zusätzliche Funktionalität, die zu der eigentlichen Aufgabe der konkreten Komponente addiert werden kann.
Wenn eine Anforderung nun von einem Client an eine konkrete Komponente gerichtet wird, werden Decorators, die über dieselbe Schnittstelle verfügen, aktiv. Decorators rufen hierbei stets die Basisoperation der Komponente auf und leisten noch ihren eigenen Beitrag dazu bei.

```
  base.Operation();
  // eigene Verantwortlichkeit von Decorator
```

Nun kann ein Client, im Beispiel das Object c, von zwei Decorators begleitet werden. Man beachte, wie d1 c dekoriert und wie diese selbst von d2 eingehüllt wird.

```
  ConcreteComponent c = new ConcreteComponent();
  ConcreteDecoratorA d1 = new ConcreteDecoratorA();
  ConcreteDecoratorB d2 = new ConcreteDecoratorB();
```

Die Ausgabe dieses einfachen Beispiels zeigt, dass die Methode Operation von c aufgerufen wird. Dazu erhält der Client noch die Dienste von d1 und d2.

```
  ConcreteComponent.operation()
  ConcreteDecoratorA.operation()
  ConcreteDecoratorB.operation()
```

1.2.10 Bekannte Einsätze

Viele Werkzeuge objekt-orientierter Benutzer-Schnitstellen benutzen Decorators, um Widgets graphische Verzierungen zu verpassen. Etwas exotischere Anwendungen von Decorators sind Debugging-Glyph von InterViews und PassivityWrapper von ParcPlace Smalltalk. Ein DebuggerGlyph gibt vor und nach einer Weiterleitung von Layout-Anforderungen an seine Component Informationen aus. Diese Trace-Information kann genutzt werden, um bei komplexen Kompositionen das Layout-Verhalten von Objekten zu analysieren und zu debuggen. PassivityWrapper kann die Interaktion des Benutzers mit dem Component ein- und ausschalten.

Das Decorator Muster ist keinesfalls auf graphische Benutzerschnittstellen beschränkt. Folgendes Beispiel soll dies veranschaulichen.
Ströme sind eine fundamentale Abstraktion in vielen Ein/Ausgabe-Bausteinen. Ein Strom ist verantwortlich für den Transport von Daten von einer Datenquelle zu einer Datensenke.
In Java wird unterschieden zwischen Strömen für

- Byte-Daten: InputStream/OutputStream und davon abgeleitete Klassen
- Character-Daten: Reader/Writer und davon abgeleitete Klassen

Dies ist notwendig, da char-Werte dem 16-bit Unicode-Zeichensatz angehören und im allgemeinen mehr als ein Byte benötigen. Im wesentlichen bieten Reader/Writer Klassen allerdings die gleichen Methoden wie InputStream/OutputStream Klassen. Nur dort, wo Byte-Strom Methoden auf Bytes oder Byte-Feldern (byte, byte[]) operieren, arbeiten die entsprechenden Character-Strom-Methoden mit Zeichen, Zeichenfeldern und Zeichenketten (char, char[], String). Die Klassen InputStreamReader und OutputStreamWriter wandeln Character-Daten in Byte-Daten: InputStreamReader liest von einem InputStream und wandelt Byte-Daten in Character-Daten, ein OutputStreamWriter schreibt in einen OutputStream und wandelt dabei Character-Daten in Byte-Daten.
Weiterhin werden die Ströme nach der Art der Datenquellen/-senken unterschieden:

- Datei: FileInputStream, FileOutputStream, FileReader, FileWriter
- Arbeitsspeicher:
  - Zeichenkette: StringReader, StringWriter
  - Zeichenfeld: CharArrayReader, CharArrayWriter
  - Byte-Feld: ByteArrayInputStream, ByteArrayOutputStream
- Pipe: PipedInputStream, PipedOutputStream, PipedReader, PipedWriter

Je nach Verarbeitungsart während des Datentransports werden Ströme wie folgt kategorisiert:

- puffern: BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream
- filtern: FilterReader, FilterInputStream, FilterWriter, FilterOutputStream
- konvertieren zwischen:
  - Byte und Character: InputStreamReader, OutputStreamWriter,
  - Byte und Objekt: ObjectInputStream, ObjectOutputStream
  - Byte und primitiven Java Datentypen: DataInputStream, DataOutputStream
- konkatenieren: SequenceInputStream
- Zeilen zählen: LineNumberReader
- vorausschauen: PushbackReader, PushbackInputStream
- formatieren: PrintWriter

Das bei java.io Paket eingesetztes Entwurfsmuster stellt Decorator. Das Muster unterscheidet zwischen konkreten und dekorierenden Objekten. Eine konkretes Objekt stellt eine unverzichtbare Basisoperation bereit. Ein dekorierendes Objekt ergänzt ein anderes Objekt um eine bestimmte zusätzliche Operation oder einen weiteren Zustand. Dabei spielt es keine Rolle, ob das so dekorierte (umhüllte) Objekt konkret oder gar selbst dekorierend (umhüllend) ist. Ein dekorierendes Objekt stellt die Basisoperation ebenfalls bereit, realisiert diese Operation aber nicht selbst, sondern beauftragt im allgemeinen das umhüllte Objekt mit der Erledigung dieser Operation. Folgender Quelltext zeigt die Verschachtellung von Decorators in Java.

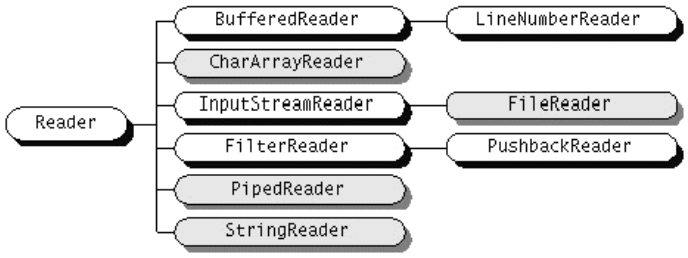


Abbildung 6: Klassenhierarchie von Reader

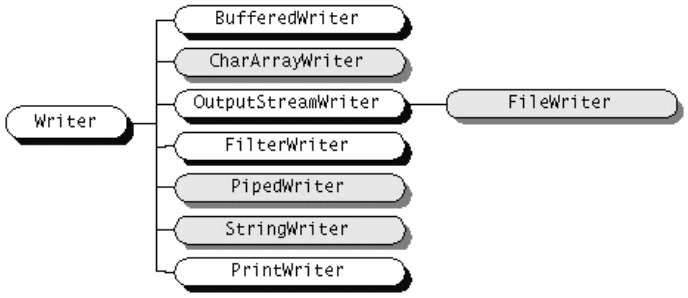


Abbildung 7: Klassenhierarchie von Writer

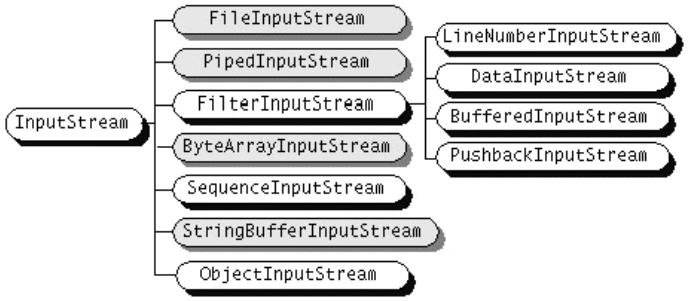


Abbildung 8: Klassenhierarchie von InputStream

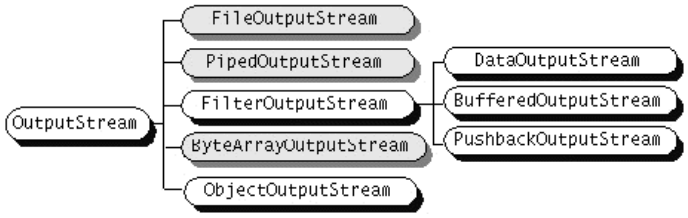


Abbildung 9: Klassenhierarchie von OutputStream

```
public class JavaIO {
    public static void main(String[] args) {
        FileInputStream is = new FileInputStream("file.txt");
        BufferedInputStream bin = new BufferedInputStream(is);
        DataInputStream dbin = new DataInputStream(bin);
        PushbackInputStream pbdbin = new PushbackInputStream(dbin);
    }
 }
```

1.2.11 Verwandte Muster

Adapter
Ein Decorator ändert die Verantwortlichkeiten eines Objekts, aber ein Adapter gibt einem Objekt eine neue Schnittstelle.
Composite
Ein Decorator ist eine degenerierte Composite, da er nur aus einer Komponente besteht.
Strategy
Ein Decorator ändert das Gesicht eines Objekts, ein Strategy aber die Eingeweide.

1.3 Missverständnisse

Nach anfänglicher Lektüre von Design Patterns oder verwandter Literatur könnte man voreilig folgendes schliessen:

- *Die Nutzung von Mustern könne mechanisch erfolgen.* Tatsache ist jedoch, dass man dabei fast immer die Entwurfsmuster an die konkreten Anforderungen anpassen muss, damit es im konkreten Fall funktioniert.

- *Entwurfsmuster würden erfunden.* Die Geschichte von Entwurfsmustern und das gegenständliche Vorbild, die Architektur, beweisen das Gegenteil. Aus unzähligen konkreten Erfahrungen haben sich häufig vorkommende Strukturen in den Lösungen herauskristalisiert. Erst die Abstrahierung dieser Strukturen hat zu Entwurfsmustern geführt.

- *Jegliche Software-Gebiete seien im Hinblick auf Muster vollständig erforscht.* Einige spezielle Gebiete wie objektorientierter Entwurf, die Programmierung von Benutzeroberflächen und verteilten Anvendungen gehören dazu. Das entspricht jedoch nicht der Wahrheit, denn noch immer gibt es Gebiete, die noch nicht flächendeckend durch Muster beschrieben worden sind. Beispiele hierfür sind Sicherungssysteme, Parallelität, numerische Berechnungen und fehlertolerante Systeme. Für die Zukunft gilt es, diese Lücken zu schließen.

- *Die Anwendung eines Entwurfsmusters verbessere automatisch eine Applikation.* Ein begründeter Einsatz von Entwurfsmustern bringt mehr Vorteile als Systeme mit Mustern unnötig zu belasten[2].

- *Man könne mit einem einzigen Entwurfsmuster auch eine komplexere, anspruchsvollere Anwendung realisieren.* Die Realität von Anwendungen jedoch belegt den geeigneten Einsatz von mehreren Entwurfsmustern, die auch in geeigneter Weise miteinander kombiniert worden sind.

Betrachtet man die junge Geschichte von Entwurfsmustern, stellt man fest:

- Sie waren ein Erfolg, da man sie direkt anwenden konnte

- Muster sind Werkzeuge, aber keine Regeln

[2] "Patterns over-enthusiasm has resulted in over-engineered designs"... "Simplicity is the most important architectural quality" Erich Gamma

- Der grösste Beitrag von Patterns ist die Bereitstellung eines Mechanismus zur Kommunikation von Design. Dadurch sind Experten in der Lage, sich über Softwareproblemen auszutauschen.

2 Ausblick

Aufgrund vielseitiger Vorteile, die Entwurfsmuster mit sich bringen, wächst zwar ständig die Gemeinde der Nutzer von Patterns. Es ist jedoch zu prüfen, ob Design Patterns in der realen Welt der Softwareentwickler tatsächlich eingesetzt und konsequent bis zur Fertigstellung der Software verfolgt werden.
Die Weiterentwicklung und Dokumentation von Design Patterns basiert zudem auf der Arbeit von Entwicklern und Designern, die ihre Erfahrungen in koordinierter, einheitlicher Form zur Filterung elaborierter bzw. neuer Muster zur Verfügung stellen.

3 Anhang A

Quelltext aus http://www.dofactory.com.

```
using System;

// "Component"

abstract class Component
{
  // Methods
  abstract public void Operation();
}

// "ConcreteComponent"

class ConcreteComponent : Component
{
  // Methods
  override public void Operation()
  {
    Console.WriteLine("ConcreteComponent.Operation()");
  }
}

// "Decorator"

abstract class Decorator : Component
{
  // Fields
  protected Component component;

  // Methods
  public void SetComponent( Component component )
  {
    this.component = component;
  }

  override public void Operation()
  {
    if( component != null )
      component.Operation();
  }
}

// "ConcreteDecoratorA"

class ConcreteDecoratorA : Decorator
{
  // Fields
  private string addedState;

  // Methods
```

```
  override public void Operation()
  {
    base.Operation();
    addedState = "new state";
    Console.WriteLine("ConcreteDecoratorA.Operation()");
  }
}

// "ConcreteDecoratorB"

class ConcreteDecoratorB : Decorator
{
  // Methods
  override public void Operation()
  {
    base.Operation();
    AddedBehavior();
    Console.WriteLine("ConcreteDecoratorB.Operation()");
  }

  void AddedBehavior()
  {
  }
}

/// <summary>
///   Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create ConcreteComponent and two Decorators
    ConcreteComponent c = new ConcreteComponent();
    ConcreteDecoratorA d1 = new ConcreteDecoratorA();
    ConcreteDecoratorB d2 = new ConcreteDecoratorB();

    // Link decorators
    d1.SetComponent( c );
    d2.SetComponent( d1 );

    d2.Operation();

  }
}
```

4 Anhang B

Mit Decorator können Objekte derselben Klasse unterschiedliches Laufzeitverhalten zeigen. Diesen Effekt ereicht man mit Decorator, indem man die Technik des Hüllens bzw. des Ankettens verwendet(wrapping, chaining).
In diesem Beispiel ist ein Mitarbeiter an der Fachhochschule zunächst ein Professor. Wenn er im Verlauf seiner Karriere andere Rollen spielen soll, können verschiedene Kombinationen dieser Rollen vorkommen. Für jede dieser Varianten eine Klasse zu definieren wäre nicht flexibel. Decorator liefert die bessere Lösung.
Hier gibt es die konkreten Klassen *Professor*, *Betreuer* und *Administrator*, die Rollen repräsentieren. Die abstrakte Klasse *Mitarbeiter* definiert die gemeinsame Schnittstelle der Dekorierer und der Dekorierten. Die Klasse *MaDecorator* spielt aber die Hauptrolle. Die Variable dekorierteMitarbeiter, die die Aggregation darstellt, fungiert als Delegation. Bei Substitution im Polymorphismus kann sie jeder Unterklasse von Mitarbeiter angehören sowie MaDecorator. Da alle diese Klassen die Methode *gehalt()* überschreiben, kann ein Mitarbeiter beliebig dekoriert werden. Dazu wird in Application.java zunächst ein Professor erzeugt, der dann auch andere Rollen übernehmen kann.
In Analogie zur Abbildung 3 ist die Klasse *Mitarbeiter* die *Component*, die dekoriert werden kann. *Mitarbeiter* entspricht der *ConcreteComponent*, *MaDecorator* der *Decorator* und *Betreuer* und *Administrator* den *ConcreteDecorator*s.

```
public class Application{

    public static void main(String[] args){
        FH fh = new FH(" Giessen-Friedberg");
        fh.beschaeftige(new Professor(4711));
        fh.beschaeftige(new Professor(0815));
        fh.displayMitarbeiter();
        System.out.println("Die Gesamtkosten betragen  " + fh.kosten() + "EURO");

        Mitarbeiter ma = fh.suche(4711);
        fh.entlasse(ma);
        fh.beschaeftige(new Betreuer(ma));
        fh.displayMitarbeiter();
        System.out.println("Die Gesamtkosten betragen  " + fh.kosten() + "EURO");

        ma = fh.suche(0815);
        fh.entlasse(ma);
        fh.beschaeftige(new Administrator(new Betreuer(ma)));
        fh.displayMitarbeiter();
        System.out.println("Die Gesamtkosten betragen " + fh.kosten() + "EURO");
    }
}

import java.util.Vector;

public class FH {
    public FH(String einName){
                Name = einName;
                dieMitarbeiter = new Vector();
    }

    public double kosten() {
                double gesamtKosten = 0.0;
```

```
                for(int i = 0; i < dieMitarbeiter.size();i++)
                     gesamtKosten += ((Mitarbeiter) dieMitarbeiter.elementAt(i)).gehalt();
                return gesamtKosten;
    }

    public void displayMitarbeiter() {
                System.out.println();
                System.out.println("Mitarbeiterliste fuer FH" + Name);
                for(int i = 0; i < dieMitarbeiter.size();i++){
                    ((Mitarbeiter) dieMitarbeiter.elementAt(i)).zeige();
                    System.out.println();
                }
    }

    public void beschaeftige(Mitarbeiter einMitarbeiter){
                dieMitarbeiter.addElement(einMitarbeiter);
    }

    public Mitarbeiter suche(int eineNummer) {
                for(int i = 0; i < dieMitarbeiter.size();i++){
                    Mitarbeiter ma = (Mitarbeiter) dieMitarbeiter.elementAt(i);
                    if(eineNummer == ma.Nummer)
                         return ma;
                }
                return null;
    }

    public void entlasse(Mitarbeiter einMitarbeiter) {
                dieMitarbeiter.removeElement(einMitarbeiter);
    }

    private String Name;
    private Vector dieMitarbeiter;
}

abstract public class Mitarbeiter{

    public Mitarbeiter(){
                Nummer = 0;
    }

    public Mitarbeiter(int eineNummer){
                Nummer = eineNummer;
    }

    public double gehalt(){
                return GrundGehalt;
    }

    public void zeige(){
                System.out.print("Mitarbeiter " + Nummer + " ist ein ");
    }
```

```
    public final int Nummer;
    protected final double GrundGehalt = 100.0;
}

abstract public class MaDecorator extends Mitarbeiter{

    public MaDecorator(Mitarbeiter einMitarbeiter){
                dekorierteMarbeiter = einMitarbeiter;
    }

    public double gehalt(){
                return dekorierteMarbeiter.gehalt();
    }

    public void zeige(){
                dekorierteMarbeiter.zeige();
    }

    private Mitarbeiter dekorierteMarbeiter;
}

public class Professor extends Mitarbeiter{

    public Professor(int eineNummer){
                super(eineNummer);
    }

    public double gehalt(){
                return super.gehalt();
    }

    public void zeige(){
                super.zeige();
                System.out.print("Professor");
    }
}

public class Betreuer extends MaDecorator {

    public Betreuer(Mitarbeiter einMitarbeiter){
                super(einMitarbeiter);
    }

    public double gehalt(){
                return super.gehalt() + (super.gehalt() * Faktor);
    }

    public void zeige(){
                super.zeige();
                System.out.print(" und ein Betreuer");
    }

    private final double Faktor = 0.1;
}
```

```
public class Administrator extends MaDecorator {

    public Administrator(Mitarbeiter einMitarbeiter){
                super(einMitarbeiter);
    }

    public double gehalt(){
                return super.gehalt() + (super.gehalt() * Faktor);
    }

    public void zeige(){
                super.zeige();
                System.out.print(" und ein Administrator");
    }

    private final double Faktor = 0.2;
}
```

Dieses kleine Programm erzeut folgende Ausgabe. Die Professoren mit den IDs 4711 und 0815 bekommen zusätzliche Verantwortlichkeiten, die sich in ihrem Gehalt widerspiegeln.

```
Mitarbeiterliste fuer FH Giessen-Friedberg
Mitarbeiter 4711 ist ein Professor
Mitarbeiter 0815 ist ein Professor
Die Gesamtkosten betragen  200.0 EURO

Mitarbeiterliste fuer FH Giessen-Friedberg
Mitarbeiter 0815 ist ein Professor
Mitarbeiter 4711 ist ein Professor und ein Betreuer
Die Gesamtkosten betragen  210.0 EURO

Mitarbeiterliste fuer FH Giessen-Friedberg
Mitarbeiter 4711 ist ein Professor und ein Betreuer
Mitarbeiter 0815 ist ein Professor und ein Betreuer und ein Administrator
Die Gesamtkosten betragen 242.0 EURO
```

5 Anhang C

Hier finden Sie die Internet-Adressen, die im Zusammenhang mit Entwurfsmustern als empfehlenswert gelten.

- http://web.mit.edu/
- http://www.rational.com/
- http://www.dofactory.com/
- http://www.industriallogic.com/

Literatur

[1] **Erich Gamma, Richard Helm,Ralph Johnson, and John Vlissides**, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, October 1994 2

[2] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal**, *Pattern-orientierte Software-Architektur*, Addison Wesley, Januar 1998

Erstellt in LaTeX.

Einführung in das Multithreading

Seminar Mechanismen SS2002

Felix Guntrum

Fachhochschule Gießen-Friedberg
Fachbereich MNI

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Writing multithreaded applications is like working with high-powered tools that have sharp blades. If you know the capabilities and limitations of your tools, respect their power, and know how to use them correctly, you can create great products. If you´re ignorant of their power or the proper way to use them, you can waste a lot of time and material trying to build your product .. [COH,S.11]

Aaron Cohen, Mike Woodring

Eine treffendere Einleitung in die Materie Multithreading kann es kaum gegeben. Es wird vorallem für die anknüpfenden Beiträge zu dem Thema nötig sein, sich die Grundlagen noch einmal zurück ins Gedächnis zu rufen. Gerade Nebenläufige Applikationen erfordern höchste Sorgfalt bei der Programmierung. Fehler sind oft nur schwer zu finden und treten meist nicht reproduzierbar auf. Ziel des Vortrages und der Ausarbeitung ist es also, den interessierten Leser erneut für das Thema zu sensibilisieren und Techniken für sicheres Programmieren mit Threads vorzustellen.

Zu beachten ist, dass diese Abhandlung keinesfalls einen Anspruch auf Vollständigkeit besitzt und lediglich eine Einleitung in die Thematik bieten kann. Das Studium weiterführender Literatur ist für den Programmierer nebenläufiger Anwendungen unausweichlich, gerade auf Betriebssystem- und Programmiersprachenspezifische Details kann hier nicht immer eingegangen werden.

Ich hoffe dennoch eine gute Auswahl der Themen und Beispiele getroffen zu haben und wünsche viel Freude beim Lesen der Lektüre.

Kapitel 2

Vom Prozess zum Thread

Vorraussetzung für das Verständnis des Themas Multithreading ist das Konzept des Prozesses. Daher folgt zunächst ein kurzer Abriss des Prozessmodells.

Im Laufe der Zeit sind die Ansprüche der Benutzer an Computersysteme stark gewachsen. Die ersten Maschinen arbeiteten im Batchbetrieb und mussten in aufwendiger Weise von einem Operator mit Informationen gefüttert werden. Programme wurden in Lochkarten gestanzt und nacheinander im sogenannten Batchbetrieb abgearbeitet. Eingriffe in den Programmverlauf waren nicht möglich.

Mit dem Aufkommen von ersten Betriebssystemen war es dem Anwender mittels Terminals möglich, mit dem Programm zu interagieren. Um eine ähnlich gute Auslastung wie beim Batchbetrieb dieser teuren Computeranlagen zu erreichen, wurden die Betriebsysteme um weitere Funktionalitäten ergänzt (z.B.):

- Multiuserbetrieb Das Betriebsystem muss dafür sorgen, dass jeder dieser Benutzer den Eindruck erhält, er habe exklusiven Zugriff auf das System. Dazu werden die Benutzer gegeneinandeder abgeschottet und die Rechenzeit wird unter ihnen aufgeteilt.
- Multitasking Von Multitasking gibt es eine Reihe von Varianten, die alle gemein haben, dass mehrere Handlungsstränge gleichzeitig im Speicher sind und im Wechsel abgearbeitet werden können.

Abbildung 2.1 verdeutlicht, wie die funktionale Anforderung an das Betriebssystem technisch realisiert wird.

2.1 Definition und Struktur von Prozessen

Aus Sicht des Prozessors ist ein Prozess also die Menge aller notwendigen Informatonen und Datenstrukturen, die benötigt werden, um ein Programm ablaufen zu lassen.

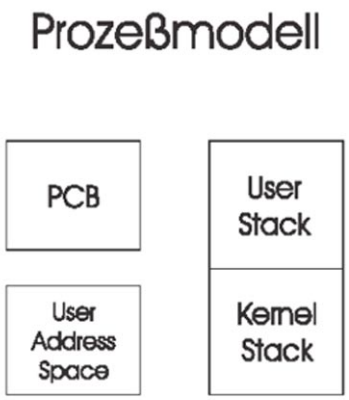


Abbildung 2.1: Prozessmodell

Im Detail gehören zu einem Prozess

- der eigentliche Programmcode, der zur Ausführung in den Hauptspeicher geladen wird
- Dateideskriptoren sowie alle weiteren Referenzen auf Systemresourcen, die der Prozess vom Betriebssystem im Laufeseines Lebens alloziiert
- Stack und Heap
- Kontrollinformationen, die das Betriebssystem für jeden Prozess verwaltet; Beispiele sind Prozess ID, Zuordnung zu einem Benutzer, Prioritäten

Das Prozesskonzept schafft die Grundlage für eine komfortable Systemnutzung. Viele Benutzer können gleichzeitig mit mehreren Programmen arbeiten und vorhandene Ressourcen effizient nutzen. Es ermöglicht überhaupt erst den Betrieb von grafischen Oberflächen, die die Ergonomie des Systems erhöhen, dessen Einarbeitungszeit verkürzen und so erst den Computer als Arbeitsmittel für einen breiten Personenkreis zugänglich machen.

2.2 Weitere Entwicklung

Die immense Leistungszunahme der Computersysteme erlaubte immer komplexere Programme. Es wurde nötig, deren Arbeit auf mehrere Prozesse aufzuteilen, um z.B. zu verhindern, dass der Prozessor auf eine Benutzereingabe wartet, obwohl er schon eine Datei kopieren könnte. Das Prozessmodell, das eigentlich für das Abschotten von Programmen zwischeneinander entworfen wurde, stösst in diesem Zusammenhang an seine Grenzen:

- Fehler in Anwendungscode auf Kosten der Systemeffizienz zu verhindern oder zu behandeln, ist nicht die Aufgabe des Betriebssystems. [LETSCH,S.18]
- Interprozesskommunikation führt zu zahlreichen Kontextwechseln; Rechenleistung und Ressourcen des Systems werden nicht optimal genutzt.
- Prozesserzeugung ist aufwendig und benötigt auf einem PIII Linuxsystem in etwa 20.000 CPU Zyklen.

2.3 Leichtgewichtiger Prozess

Um zeitaufwendige Kontextwechsel zwischen Prozessen zu ersparen braucht man die Möglichkeit mehrere Handlungsstränge innerhalb eines Prozesses zu vereinen. Weiterhin können sich diese Handlungsstränge verschiedene Kontrollinformationen und Ressourcen teilen - eine Abschottung untereinander kann innerhalb einer Anwendung auf ein Minimum reduziert werden. Aus diesem Gedanken heraus enstand das Konzept des Threads, auch „leichtgewichtigr Prozess“ genannt.

Kapitel 3

Threads

Definition:

A thread, sometimes called a lightweight process, is a basic unit of CPU utilization; it comprises a thread ID, a programm counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. [SILBERS,5.1]

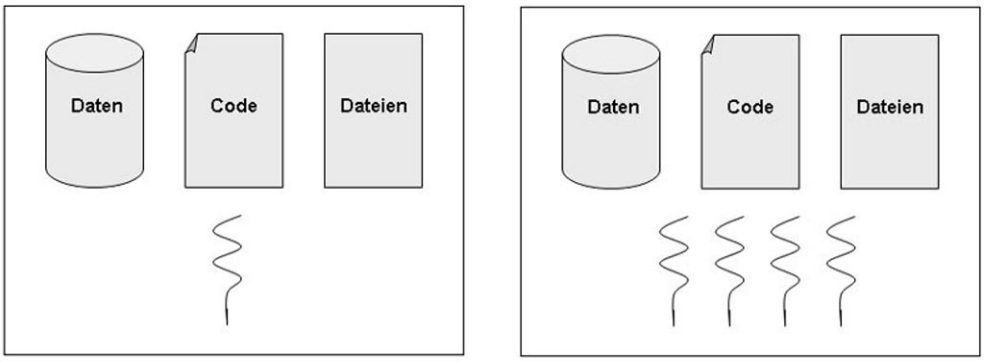


Abbildung 3.1: Single vs. Multithreading

Im folgenden Kapitel wird auf den Aufbau und die Struktur von Threads eingegangen; Varianten der Implementierung werden diskutiert und erläutert.

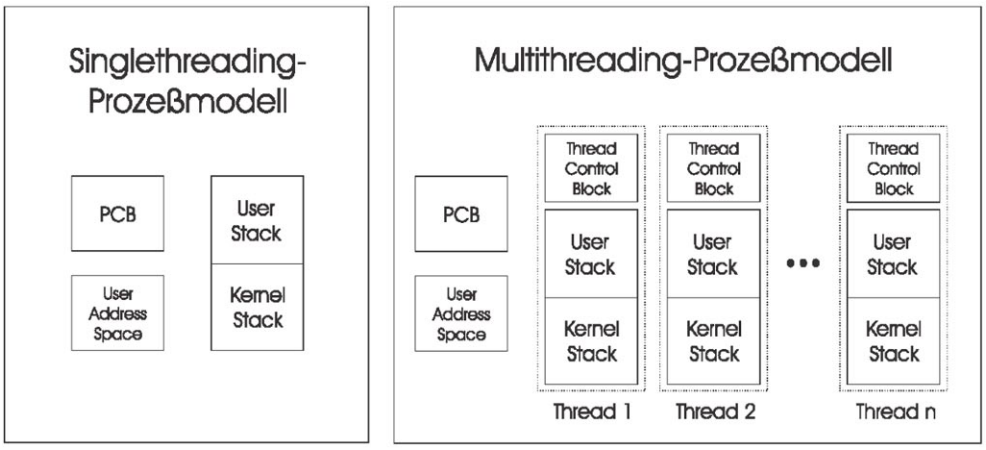


Abbildung 3.2: Die Modelle im Vergleich

3.1 Struktur von Threads

Ein Thread nutzt eigene Ressourcen und teilt sich soweit möglich verschieden Informatinen mit den anderen Threads desselben Prozesses.

Eigene Ressourcen

- Thread ID
- Stack
- Registersatz
- Programmzähler
- Zustandsinformationen (bereit, laufend, blockiert)

Gemeinsame Ressourcen

- Prozess-Kontroll-Block (PCB, „Maschinenzustand“)
- Adressraum des Prozesses (Daten und Programmcode)
- Systemressourcen, wie z.B. offene Dateien und Signale

3.2 Zusammenfassung der Vorteile

Besseres Antwortverhalten - Programme können weiterlaufen, selbst wenn ein Teil von ihnen blockiert ist [1].

Ressoucenteilung - Threads teilen sich den Speicher und die Ressourcen des zugehörigen Prozesses, die Anwendung hat nun mehrere Handlungsstränge innerhalb des selben Adressraums. Zeitraubende Kontextwechsel und aufwendige Prozesserzeugung ist nicht mehr notwendig.

Nutzung von Multiprozessorsystemen - Mehrere Threads eines Prozesses können echtparallel ausgeführt werden. Auf diese Weise können einzelne Programme den Rechenvorteil des Systems ausnutzen, ohne den Overhead von mehreren Prozessen in Kauf nehmen zu müssen.

3.3 Implementierung und Varianten von Threads

Prinzipiell unterscheidet man zwischen drei Varianten von Threads:

- User-Level-Threads, die mit einer Bibliothek realisiert werden
- Kernel-Threads, d.h. Thread Funktionalität des Betriebssystems
- Kombination aus User-Level- und Kernel-Threads
- Threads als Bestandteil der Programmiersprache

Alle vorgestellten Thread Modelle haben ihre spezifischen Vor- und Nachteile, die sie für verschieden Problemstellungen mehr oder weniger qualifizieren. Zunächst werden die Modelle im einzelnen vorgestellt und anschliessend auf diesen Aspekt hin diskutiert.

3.3.1 User-Level-Threads

Bei User-Level-Threads werden mehrere sog. „user threads“ auf einen Prozess abgebildet, weshalb dieses Modell auch als „Many-to-One Model“ zu finden ist. Üblicherweise bedient man sich einer Bibliothek, wie z.B. POSIX, die einem eine API zur

[1] Unterschiede in den versch. Varianten

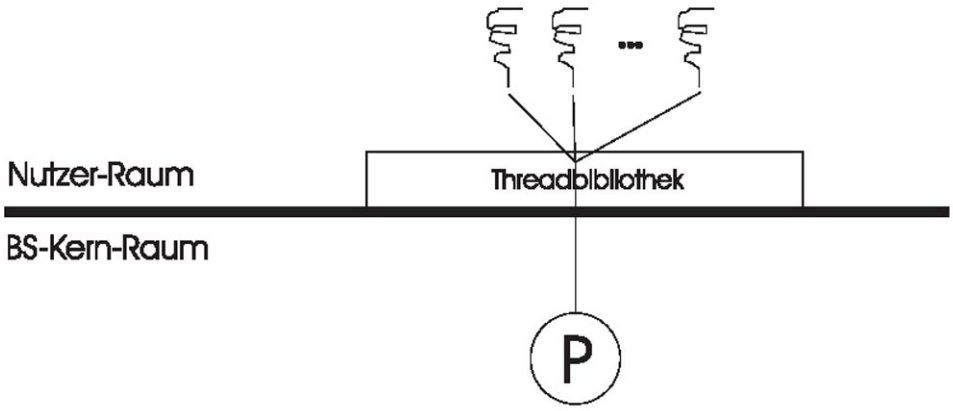


Abbildung 3.3: User Level Threads

Threadverwaltung anbieten. Zu den wichtigsten Bestandteile einer solchen Bibliothek gehören:

- Operationen zum erzeugen und terminieren von Threads
- Kommunikationsmechanismen
- Scheduling Mechanismen
- Synchronisationsmechanismen [siehe Kapitel 4.1]

3.3.2 Kernel-Level-Threads

Implementing kernel threads normally requires the reorganisation of essential parts of the OS. The scheduler, for example, no longer allocates processor time to processes but directly to individual threads. Every executed program consists of one or more threads which together form a task. A task is, therefore, like a container for threads which may be distributed across several CPUs. [SCHM,S.140]

In einer reinen Kernel-Level-Thread Umgebung wird das Threadmanagement vom BS-Kern durchgeführt. Diese Technik wird von fast allen Betriebsystemen unterstützt, variiert aber in der Realisierung. Ihnen gemein ist, dass der Programmierer über eine API Threads alloziieren kann (User Threads) und das das Betriebssystem diese auf sog. „kernel threads“ abbildet. Der Unterschied besteht zumeist darin, wie abgebildet wird: Solaris verwendet die in Kapitel 3.3.3 erläuterte Mischform und WindowsNT bildet einen User-Thread auf einen Kernel-Thread ab.

Listing 3.1: User Level Threads

```
void * betrag_von_x(void * parameter) {
    int ergebnis;
    int * x = (int *)parameter;
    ergebnis = (* x>=0)?*x:-*x;
    printf("'Ergebnis: %d\n"',ergebnis);
}

int main(){

    int ergebnis, argument = -8;
    pthread_t threadID;
    int fehlerNr;

    fehlerNr = pthread_create(&threadID,    //Thread ID
                             0,             //Thread-Attribute (0=Standartwerte)
                             betrag_von_x,  //Auszufuehrende Funktion
                             (void *)(&argument) //Argumente dieser Funktion
                             );
    return fehlerNr;
}
```

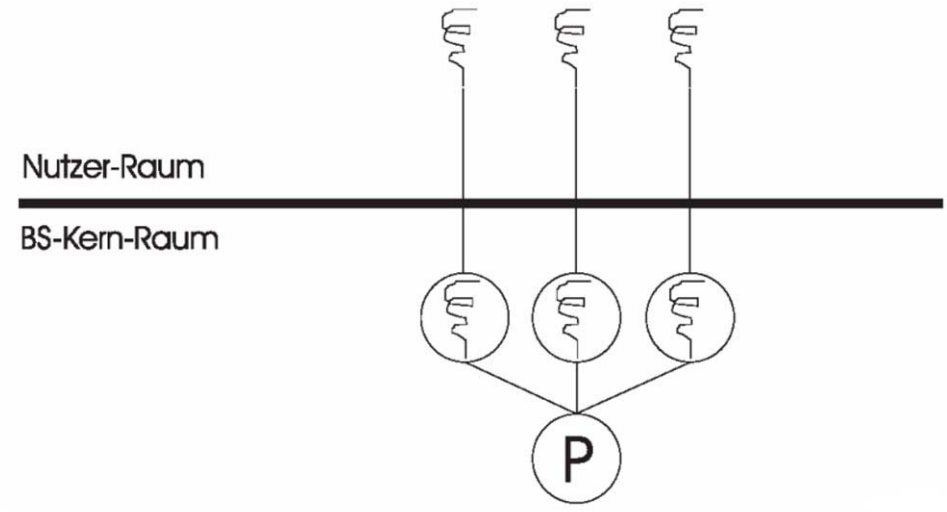


Abbildung 3.4: Kernel Level Threads

Vorstellen werden wir diese Technik am Beispiel von Linux. Hier kann der Programmierer aus einer Auswahl bestimmen, welche Ressourcen sich die Threads teilen sollen und welche nicht [2]. Folglich erhält man einen reinen Thread, wenn er sich alles teilen soll und einen Prozess, wenn alle angegebenen Resourcen exklusiv genutzt werden sollen.

Tatsächlich ist es unkomfortabel und fehleranfällig, „clone“ direkt zur Erzeugung von Kernel Threads zu benutzen. Zum einen ändert sich die Schnittstelle von Kernelversion zu Kernelversion häufig, zum anderen möge man z.B. bedenken, was passieren kann, wenn der Stackspeicher nicht alloziiert werden kann und sich dann mehrere Threads einen Stack teilen. Tatsächlich wird „clone“ verwendet, um die „pthreads“-Bibliothek von Linux zu implementieren.

3.3.3 Mischkonzepte

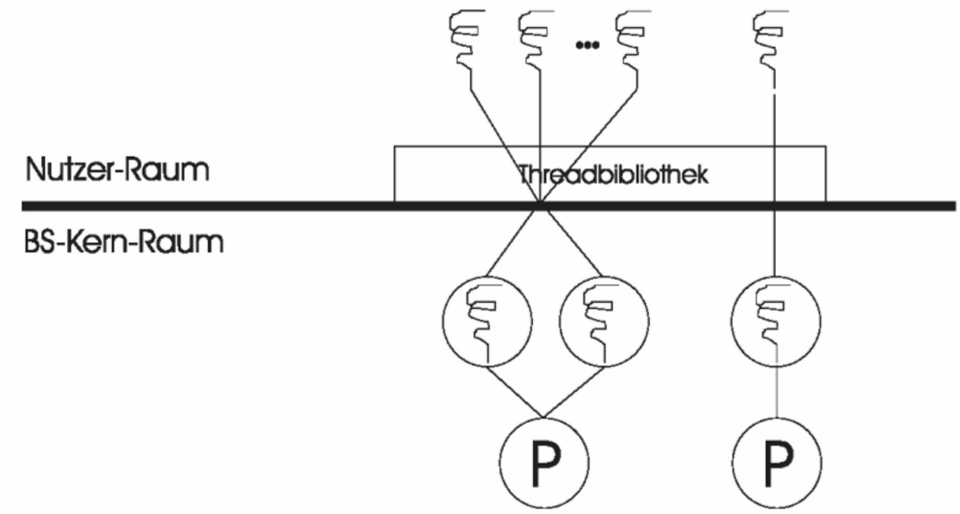


Abbildung 3.5: Mischkonzepte

Die Idee ist dabei, User-Level-Threads auf eine definierte Anzahl von Kernel Threads abzubilden, weshalb diese Technik auch Many-to-Many Model genannt wird. Die Anzahl an Kernel Threads, die einer Anwendung eingeräumt werden, kann je nach Priorität der Anwendung und vorhandenen Resourcen variieren, der Anwendungsprogrammierer hat aber keinen Einfluss darauf.

[2]Kernel Funktion „clone“; siehe man pages und Codelisting bzw. in Datei „process.c“ des Linux Kernels 2.4.3 ab Zeile 690

Listing 3.2: Kernel Level Threads

```
#include <sched.h> #include <stdio.h>

int betrag_von_x(void * parameter) {
    int ergebnis;
    int * x = (int *)parameter;
    ergebnis =(* x>=0)?*x:-*x;
    printf("'Ergebnis: %d\n"',ergebnis);

    return 0;
}

int main(){

    int ergebnis,fehlerNr;
    int  share_flags ;
    int argument;

    //Achtung: man kann auch NULL uebergeben, dann wird aber auch der Stack geteilt !!!
    void* child_stack = malloc(64000);      //Stack üfr  thread
    argument = -8;                          //Argument "' "'

    share_flags = CLONE_FS                  //FS Info teilen
        | CLONE_FILES                       //Dateizugriff  teilen
        | CLONE_SIGHAND                     //Signalbehandlungsroutinen teilen
        | CLONE_PID                         //ab V.2.4: gleiche  Prozess ID
        | CLONE_VM                          //gemeinsamer Speicherberreich
                                            //Achtung: erst ab Version 2.4:
        | CLONE_THREAD;                     //gehoert zum selben Task

    fehlerNr = clone( betrag_von_x,         //Funktion des Threads
                child_stack ,               // alloziierter  Stack
                share_flags ,               //was wird geteilt
                (void*) &argument);         //Argument der Thread Funktion

    return fehlerNr;
}
```

Listing 3.3: Threadbeispiel in Java

```
class ThreadBeispiel extends Thread {
    public void run() {
        System.out.println("'Hallo␣lieber␣Benutzer!"');
    }
}

public class Verwender {
    public static void main(String args[]) {
        ThreadBeispiel bsp = new ThreadBeispiel();

        bsp.start ();
    }
}
```

3.3.4 Threads auf Sprachebene

Threads, als reguläre Kontrollstruktur einer Programmiersprache haben eine lange Tradition. Beispielsweise existstierte schon vor 30 Jahren mit ADA eine Sprache mit eigener Threadunterstützung.

Prominentestes Beispiel heute ist wohl JAVA; die Programmierschnitstelle ist in Klassen und Interfaces gekapselt.

Prinzipiell hat man zwei Möglichkeiten, Threads in JAVA zu programmieren:

(1) Man definiert zunächst eine Unterklasse von Thread und überschreibt die Methode run(). Ein neuer Thread wird in dem Moment erzeugt, wenn ein Objekt dieser Klasse erzeugt und dessen Methode start() aufgerufen wird.

(2) Die zweite Methode bedient sich des Strategy Musters: Eine definierte Klasse bindet das Interface Runnable ein und überschreibt die geerbte Methode run(). Um nun einen Thread zu erzeugen wird eine Referenz auf ein Objekt dieser Klasse in einer Runnable Instanz gespeichert. Nun erzeugt man ein Thread Objekt und übergibt dabei dem Konstruktor die Runnable Instanz. Mit diesem Objekt kann nun ein Thread erzeugt werden, indem die start() Methode aufgerufen wird.

3.3.5 Bewertung

Bibliotheken, die Threads auf Benutzerebene realisieren, haben gegenüber Betriebssystemthreads in einigen Bereichen Vorteiele in der Performance: Bei der Erzeugung ist kein Kontextwechsel in den privilegierten Modus vonnöten und die Verwaltungsstruk-

Listing 3.4: Threadbeispiel2 in Java

```
class ThreadBeispiel2 implements Runnable {
    public void run() {
        System.out.println("'Hallo␣lieber␣Benutzer!"');
    }
}

public class Verwender2 {
    public static void main(String args[]) {
        Runnable bspRunner = new ThreadBeispiel2();
        Thread thread = new Thread(bspRunner);

        thread.start ();
    }
}
```

turen sind im Allgemeinen schlanker. Wechsel zwischen den Threads sind ebenfalls nicht mit Kontextwechseln verbunden, sondern stellen nur Programmsprünge dar.

Allerdings stellt sich das Problem, dass ein blockierender Systemaufruf zu einer Blockierung aller Threads desselben Tasks führt. Abhilfe kann dadurch geschaffen werden, einen blockierenden Systemaufruf in einen nicht blockierenden Systemaufruf transformiert. Ist ein Ausgabegerät beispielsweise gerade besetzt, wird diese Ressource später nochmal angefragt anstatt zu blockieren.

Der Hauptnachteil besteht darin, dass keine Multiprozzesorsysteme unterstützt werden: Da alle Threads auf einen Prozess abgebildet werden, ohne dass das Betriebssystem von deren Existenz weiss, können die Threads auch nicht auf mehrere Prozessoren verteilt werden.

Das Many-to-Many Modell versucht die Vorteile von Threads auf Benutzerebene mit Betriebssystem Threads zu vereinen. Die zugeteilten Kernel Threads verhindern ein Blockieren des kompletten Tasks und lassen sich auf mehrere Prozessoren verteilen; die grössere Anzahl an Benutzer Threads geniesst jedoch deren Vorteile.

Kapitel 4

Scheduling

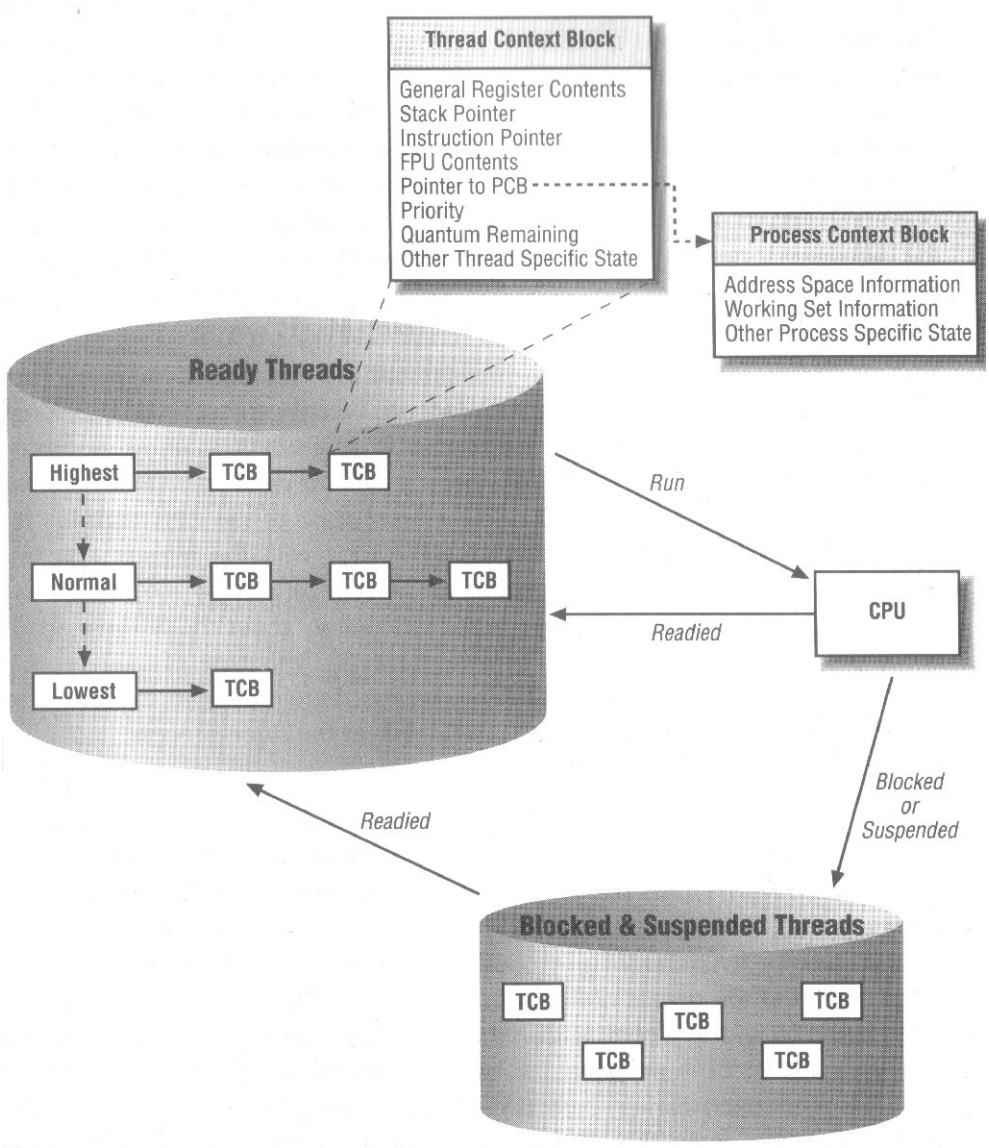
Sobald ein Thread ausgeführt wird, besitzt einen der drei folgenden Zustände:

Running: Ein Thread mit dem Zustand „running“ befindet sich in Ausführung in der CPU. Diesen Status kann immer nur ein Thread pro CPU haben

Ready: Ein Thread der sich nicht in Ausführung befindet, aber darauf wartet Rechenzeit zugeteilt zu bekommen. Anhand der Scheduling-Strategie wird entschieden welcher Thread als nächstes den Prozessor zugeteilt bekommt.

Blockiert: Wenn ein Thread während seiner Ausführung an einen Punkt gelangt wo er nicht weiter arbeiten kann wird er blockiert. Dies geschieht z.B. wenn ein Thread versucht sich einen Mutex anzueignen der schon von einem anderen Thread benutzt wird. (später mehr dazu)

Suspendierte Threads sind ebenfalls nicht laufbereit und verbrauchen genau wie die blockierten Objekte keine CPU-Zeit. Threads werden beispielsweise bei einen Systemaufruf für eine kurze Zeit vom BS suspendiert. Durch Aufruf der sleep-Methode kann sich ein Thread aber auch selbst in den Zustand suspendiert versetzen.



Quelle: [COH]

Abbildung 4.1: Scheduling von Threads

Kapitel 5

Syncronisation

Einer der wichtigsten Vorteile von Threads, die Ressourcenteilung, bringt ein Problem mit sich: Bei dem gemeinsamen Zugriff auf Daten muss deren Konsistenz gewahrt werden. Dieses Problem gehöhrt zu den Standardproblemen der Informationsverarbeitung und wird „Consumer-Producer Problem“ genannt (siehe Codelisting).

Das Problem besteht darin, dass die Operation „count++“ von der CPU nicht atomar -also in einem nicht unterbrechenbaren Zyklus- ausgeführt wird, sondern aus mehreren Schritten besteht. Dateninkonsistenz tritt dann auf, wenn einem Thread A die Kontrolle über die CPU zwischen diesen Bearbeitungsschritten entzogen wird, einem anderen Thread B zugeteilt wird und dieser die Variable verändert. Kommt Thread A wieder an die Reihe, wird die Operation fortgeführt unte der Annahme eines falschen Wertes. Die Situation, in der mehrere Threads schreibend auf gemeinsame Daten zugreifen und die Reihenfolge der Zugriffe eine Rolle spielt, nennt man „race condition“. Das Stück Code eines Threads, in dem auf gemeinsame Daten zugegriffen wird, nennt man „critical section“ (kritischer Abschnitt).

Der holländische Mathematiker Dikstra formulierte schon um 1960 folgende Forderungen, die erfüllt werden müssen, um obiges Problem zu lösen:

- keine zwei Threads dürfen gleichzeitig in ihrem kritischen Abschnitt sein (Mutual Exclusion)
- kein Thread darf ausserhalb eines kritischen Abschnitts einen anderen oder sich selbst blockieren (siehe Deadlocks)
- wartet ein Thread auf Zutritt in einen kritischen Abschnitt, muss ihm dieser auch irgendwann gewährt werden.

Listing 5.1: User Level Threads

```
// Produzent
while(zaehler == PUFFER_GROESSE)
    ; // warte

//produzieren
++zaehler; puffer[p] = objekt;
p = (p + 1) % PUFFER_GROESSE;

// ------------------------------

// Konsument
while(zaehler == 0)
    ; // warte

//konsumieren
--count; objekt = puffer[k]; item = puffer[k];
k = (k + 1) % PUFFER_GROESSE;
```

- es dürfen keine Annahmen zur Abarbeitungsgeschwindigkeit, zur Anzahl der Threads oder Prozessoren gemacht werden.
- ein Thread darf sich nur für eine endliche Zeit in seinem kritischen Abschnitt aufhalten

Softwarelösungen, die diese Forderungen erfüllen sind der „Peterson Algorithmus“, der allerdings nur für zwei Threads geeignet ist sowie seine Erweiterung für mehere Threads, der „Bakery Algorithmus“.

Mit Hilfe eines Hardware gestüzten Ansatzes lässt sich das Problem mit einem sogenannten Mutex realisiern: Die Hardware muss den Wert einer Variablen in einer atomaren Operation lesen und auf einen neuen Wert setzen können („test-and-set“). Der Mutex enthält eine Variable die als Schloss dient. Bevor ein Thread seinen kritischen Abschnitt betritt, überprüft er, ob das Schloss offen ist und -falls dies zutrifft- verschliesst es (atomare Operation). Solange die Schlossvariable gesetzt ist, muss er warten.

Eine Variante von dieser Technik ist die Semaphore, die für mehrere gleichwertige Ressourcen geeignet ist: Der Wert der Semaphore stellt die Anzahl der freien Ressourcen dar. Bevor ein Thread seinen kritischen Abschnitt betritt, überprüft er, ob sie grösser Null ist und erniedrigt sie um eins, falls dies zutrifft (atomare Operation). Nach dem Verlassen seines kritischen Abschnitts gibt er die belegte Ressource wieder frei, indem er sie wieder um eins erhöht.

Kann ein Thread nicht in seinen kritischen Abschnitt, kann er entweder in einer Endlosschleife warten, bis die Eintrittsbedingung erfüllt ist („spin lock“ oder „aktives Warten“) oder er gibt die Kontrolle über die CPU auf. Dazu sendet er ein Signal, dass ihn in eine Warteschlange einreiht; verlässt ein Thread seinen kritischen Abschnitt sendet er ein Signal an die Verwaltugsinstanz der Warteschlange, die den nächsten Thread wieder aktiviert.

Die vorgestellten Techniken haben den Nachteil, dass sie von der richtigen Anwendug des Programmierers abhängen: Er kann vergessen, die Bedingung für das Eintreten in den kritischen Abschnitt zu überprüfen oder den Eintritt in diesen wieder zu öffnen. Solche Fehler sind vor allem deshalb schwer zu finden, da sie nicht ohne weiteres reproduzierbar sind. Um dieses Problem zu umgehen wird der kritische Abschnitt in einem sogenannten Monitor gekapselt, von dem die Threads dessen Ausführung anfordern können. Der Monitor bedient immer nur einen Prozess zur selben Zeit und reiht etwaige weiteren Threads in eine Warteschlange ein.

Kapitel 6

Deadlock

Durch ungeschickte Verwendung von Synchronisationsmitteln kann ein System in den Zustand der Verklemmung fallen, dieser Zustand wird im allgemeinen auch als Deadlock bezeichnet. Ein Deadlock liegt vor, wenn zwei Threads um die selbe Ressource konkurrieren aber sich gegenseitig daran hindern, auf diese Ressource zuzugreifen. Da beide Threads auf die Freigabe des Betriebsmittels durch den jeweils anderen warten, kommt das System zum Stillstand.

Wir betrachten im folgenden ein Beispiel, in dem zwei Threads um den exclusiven Zugriff auf eine Datei und einen Drucker konkurrieren.

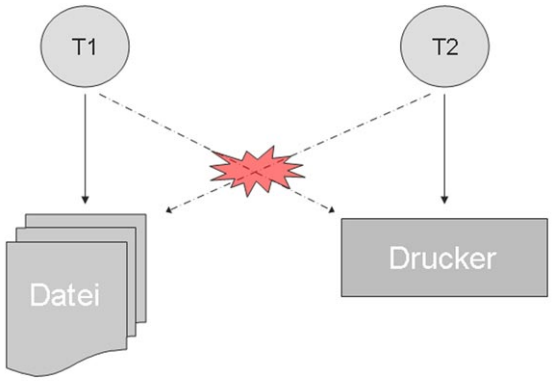


Abbildung 6.1: Deadlocksituation

Thread T1 fordert den exklusiven Zugriff auf die Datei an und Thread T2 den Drucker. Beiden Anforderungen wird entsprochen.

Nun fordert Thread T1 den Drucker an, ohne aber die Datei freizugeben und Thread

T2 verlangt dagegen nach der Datei - jedoch ohne den Drucker abzugeben. Die Folge ist eine gegenseitige Blockierung der beiden Threads.

6.1 Bedingungen für eine Verklemmung

Wir definieren folgende Bedingungen:

- **Exclusive Nutzung, Mutual exlusion:** Mindestens ein Betriebsmittel ist plattformunabhängigen exklusiv reserviert.
- **Wartebedingung, Hold and wait:** Es werden verfügbare Ressourcen reserviert, während auf zusätzliche Betriebsmittel gewartet wird.
- **Nichtentziehbarkeit, No preemption:** Einem Thread können reservierte Ressourcen nicht zwangsweise entzogen werden, sondern er muss sie explizit freigeben.
- **Geschlossene Kette, Circular wait:** Eine geschlossene Kette existiert, wenn jeder auf ein Betriebsmittel wartet, dass durch den nächsten in der Kette gehalten wird.

Wenn sich eine geschlossene Kette nicht lösen lässt, kommt es zu einem Deadlock. Dies ist der Fall, wenn die ersten drei Bedingungen gegeben sind.

Wir stellen fest: Alle vier Fälle zusammen sind also notwendige und hinreichende Bedingung für einen Deadlock.

6.2 Grafische Darstellung von Deadlocks

Mit gerichteten Grafen kann das Problem der Ressourcenzuordnung betrachtet werden. Abbildung 6.2 veranschaulicht dies. Die Kreise sind die Threads, die Rechtecke die Ressourcen. Eine Kante von einer Ressource zu einem Thread stellt eine erfolgte Allozierung der Ressource durch den Thread dar. Eine Kante von einem Thread zu einer Ressource bedeutet die Anforderung der Ressource durch den Prozess. Eine Ressource mit mehreren verfügbaren Instanzen wird durch die entsprechende Anzahl von Punkten innerhalb der Ressource abgebildet.

Nun gelten folgende Aussagen:

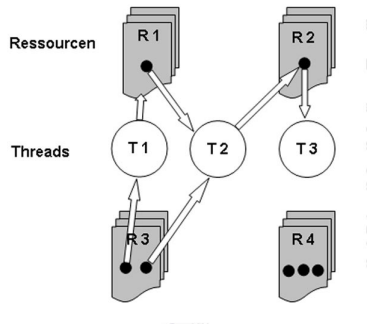


Abbildung 6.2: Darstellung über gerichtete Graphen

- Wenn ein Graf keinen Zyklus enthällt, liegt auch kein Deadlock vor. (footnote Zyklus: gerichteter, geschlossener Kreis)
- Falls der Graf einen einen Zyklus enthällt und jede Ressouce nur über eine Instanz verfügt, liegt ein Deadlock vor.
- Enthällt der Graf einen Zyklus und die Ressourcen im Zyklus verfügen über mehrere Instanzen, so kann ein Deadlock vorliegen, weitere Bedingungen sind notwendig.

Zu Einem Deadlock kommt es, wenn Thread T3 nun eine weitere Ressource anfordert, dargestellt in Abbildung 6.3.

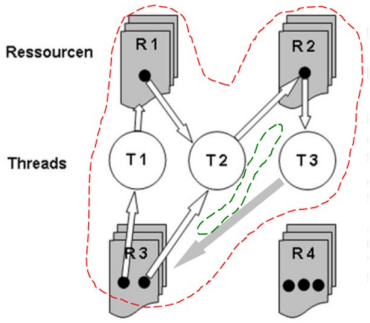


Abbildung 6.3: Deadlock Situation

Es entstehen zwei minimale Zyklen.

(1) T1 - R1 - T2 - R2 - T3- R3 - T1

(2) T2 - R2 - T3 - R3 - T2.

6.3 Behandlung von Verklemmungen

Grundsätzlich gibt es drei verschiedene Möglichkeiten, mit Verklemmungen umzugehen.

Erste Möglichkeit ist die Verwendung von Verfahren, die Deadlocks verhindern oder vermeiden. Zur Verhinderung muss das Eintreten von den oben genannten Bedingungen für einen Deadlock ausgeschlossen werden. Zur Vermeidung ist das Mitführen und Auswerten von zusätzlichen Informationen über die Ressourcen und deren Verwendung notwendig. Ressourcen werden dann nicht zugesprochen, falls die Gefahr eines Deadlocks besteht, bzw. nur zugewiesen, wenn ein sicherer Zustand garantiert werden kann.

Nachteile dieser Verfahren sind ganz klar die schlechte Ressourcenausnutzung, Wartezeiten, die sequentialisierung des Programmverlaufs und eventuell kommt es sogar zu Starvation [1].

Zweitens können wir Deadlocks zulassen, den Programmverlauf beobachten und den Zustand erst lösen, falls er eingetreten ist. Dazu muss periodisch auf Deadlocks untersucht werden. Bei Ressourcen mit einer Instanz wird bei einer Allozierung der Graph auf Zyklen untersucht, bei Ressourcen mit mehreren Instanzen findet der Bankers Algorithmus Anwendung.

Eine dritte Alternative ist die schlichte Ignorierung des Problems falls der Aufwand zur Behandlung von Deadlocks in unkritischen Anwendungen nicht gerechtfertigt ist. Viele Betriebssysteme, wie auch Unix, verwenden diese Methode zumindest auf Prozessebene.

[1] „Verhungern“ - endloses warten auf Zuteilung von Rechenzeit

Kapitel 7

Threads und Objekte

Apartment Threading ist ein threading-Model welches von Microsoft im Zusammenhang mit dem Component Object Model (COM) benutzt wird.

Auch COM Objekte können von mehreren Threads innerhalb eines Prozesses genutzt werden.

Die Begriffe „Single-threadet Apartment“ und „Multi-Threadet Apartment“ sind ein konzeptionelles Framework zur Beschreibung der Beziehung zwischen Threads und COM Objekten und von Konkurenzsituationen der Objekte untereinander

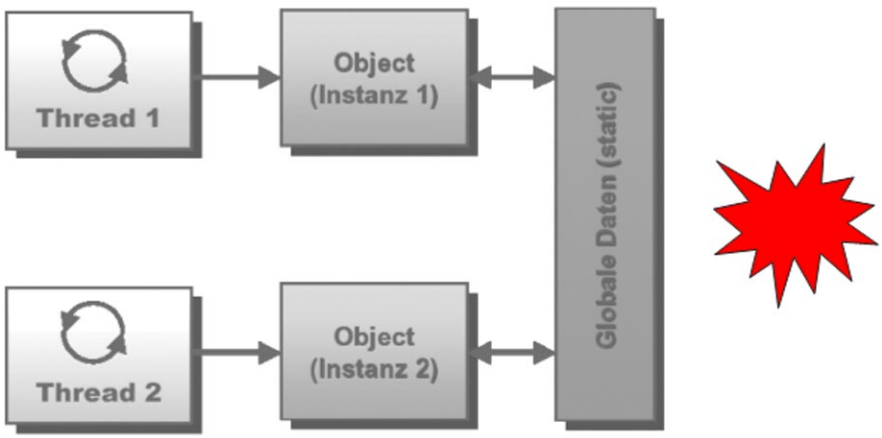


Abbildung 7.1: Threads und Objekte

In der betrachteten Grafik erzeugen zwei Threads innerhalb eines Prozesses das gleiche

Objekt. Es kommt zu Konkurenzsituationen wenn die Methoden beider Objektinstanzen auf die gleichen globalen Daten zugreifen.

Die Lösung Microsofts besteht darin, Threads expliziten Objektmengen, den sog. Appartments, zuzuordnen. Der Zugriff auf die COM Objekte wird nicht mehr ungeschützt zugelassen und je nach Anforderung serialisiert (marshaling).

Im wesentlichen kann man sagen: In diesen Appartment werden Objektgruppen zusammengefasst, die ähnliche Konkurenzkriterien haben.

Laufen also zwei Komponenten zusammen in einem Apartment, dann haben sie die gleichen Anforderungen, wie sie von Threads behandelt werden möchten.

In Windows wird zwischen einem Single Thread Apartment (STA) und dem Multi Thread Apartment (MTA) unterschieden.

STA: Objekte innerhalb dieses Apartments können sicher sein, das immer nur ein Thread auf sie zugreift. Befinden sich mehrere Objekte innerhalb des Apartments, so kann auch immer nur auf ein Objekt zur gleichen Zeit zugegriffen werden. Im STA befinden sich also Objekte, die nicht Thread-save programmiert sind.

Komponenten, die einem MTA zugeordnet werden müssen Thread-save programmiert sein.

Weiterhin unterstützt das Modell die sichere Zeigerübergabe und den Synchronisierten Datenaustausch zwischen Objekten und Threads.

Dieses Modell ist meiner Meinung nach vor allem im Hinblick auf die Wiederverwendung von Objekten eine wertvolle Technik, denn Objekte die nicht Thread-save programmiert wurden, können nun ohne Probleme weiterhin eingesetzt werden. Zwar ist innerhalb eines STA keine Parallelität mehr möglich, aber in vielen Fällen ist dies sicher eine sinnvolle Alternative.

Literaturverzeichnis

[LETSCH] Prof. Dr. Thomas Letschert
Unterlagen zur Vorlesung Betriebssysteme I
http://homepages.fh-giessen.de/ hg51/BS-I/
FH Gießen-Friedberg

[JAEG] Prof. Dr. Michael Jäger
Skript zur Vorlesung Betriebssysteme I
http://homepages.fh-giessen.de/ hg52/lv/bs1/skripten/bs1skript/pdf/bs1skript.pdf
FH Gießen-Friedberg

[SILBERS] Avi Silberschatz, Peter Baer Galvin, Greg Gagne
Applied Operating System Concepts
Wiley, 2000

[STROUS] Bjarne Stroustrup
Die C++ Programmiersprache
Addison-Wesley, 1998

[COH] Aaron Cohen Mike Woodring
Win32 Multithreaded Programming
O'Reilly, 1998

[SCHM] Jürgen Schmidt, c't 13/98, S.140, Double Play

FH GIESSEN-FRIEDBERG
FACHBEREICH MNI

Scoped-Locking-Idiom
Strategized-Locking-Muster

Seminar Mechanismen SS2002

Dominik Sacher

Vortragsdatum 21.06.2002
Robert Bosch GmbH
Abt. FV/SLD
Frankfurt/Main

Inhaltsverzeichnis

1 Einleitung

1.1 Überblick

> "Each problem that I solved became a rule which
> served afterwards to solve other problems."
> *Rene Descartes (1596-1650),*
> *"Discours de la Methode"*

Entwurfsmuster - Architekturmuster - Architekturstil - Idiom - Mechanismus

Diese teilweise synonym gebrauchten Begriffe finden sich heute im Vokabular fast jedes Softwareentwicklers. Schlagwörter wie *Singleton*, *Observer* oder *Abstrakte Fabrik* müssen in Fachgesprächen vorkommen, um ihm den entsprechend wissenschaftlichen Anstrich zu geben. Muster sind quasi "in aller Munde". Eine Einordnung der oben aufgeführten Begriffe werde ich im Rahmen dieser Ausarbeitung trotzdem nicht bieten, das Anliegen dieser Ausarbeitung ist darauf begrenzt, dem Leser den Unterschied zwischen "Muster" und "Idiom" zu verdeutlichen. Das folgende Kapitel beschäftigt sich mit den entsprechenden Definitionen. In Kapitel 2 ab Seite 5 wird das C++-Scoped-Locking-Idiom einschließlich einer Beispielimplementierung vorgestellt. Dem Strategized-Locking-Muster ist Kapitel 3 ab Seite 15 gewidmet. Eine Kombination beider Mechanismen wird ebenfalls präsentiert. Die Zusammenfassung der meiner Ansicht nach relevantesten Erkenntnissen findet in Kapitel 4 ab Seite 25 statt. Abschließen werde ich diese Ausarbeitung mit den obligatorischen Literaturhinweisen und einem Glossar. Grundlegende Kenntnisse über die Programmierung nebenläufiger Systeme und die Probleme die dort auftreten, setze ich als Grundlagenwissen voraus. Gegebenenfalls hilft ein Blick in den Glossar, [JÄG] oder [SILB], um die Begrifflichkeiten zu klären. Alle Code-Beispiele sind in C++ geschrieben. Auch dies setze ich zum Verständnis der Materie voraus. In [STROU1] sind sämtliche hier verwendeten C++-Konstrukte erklärt. Zum Abschluss der Einleitung will ich noch auf diverse Formalia hinweisen: Quellcode ist `verbatim` gedruckt, Zitate sind *kursiv* und als solche kenntlich gemacht. Quellenangaben stehen in [eckigen] Klammern und verweisen auf den Anhang (Kapitel 5.2 ab Seite 28). Auf Anglizismen wurde weitestgehend verzichtet, einige Begriffe sind aber entweder schlecht übersetzbar ("Strategized-Locking"), oder die deutsche Übersetzung erschwert das Verständnis (Framework = "Rahmenwerk").

1.2 Muster vs. Idiom

Laut [POSA1, S.8] *beschreibt ein (Softwarearchitektur-)Muster ein bestimmtes, in einem speziellen Entwurfskontext häufig auftretendes Entwurfsproblem und präsentiert ein erprobtes generisches Schema zu seiner Lösung. Dieses Lösungsschema spezifiziert die beteiligten Komponenten, ihre jeweiligen Zuständigkeiten, ihre Beziehungen untereinander und ihre Kooperationsweise.*
Weiterhin[POSA1, S.13] *beschreibt ein (Entwurfs-)Muster eine häufig auftretende Struktur ... , die ein allgemeines Entwurfsproblem in einem bestimmten Kontext löst.* Man könnte an dieser Stelle sicherlich etliche Zitate aus [GOF1], [ALEX], [POSA2] oder aus mehr oder weniger zweifelhaften Quellen des Internets anführen. An diesem Punkt reichen die genannten aber vollkommen aus. Sinn und Zweck der Definition habe ich bereits im Überblick beschrieben. Zentrale Eigenschaft eines Musters ist das Anbieten einer abstrakten, sprachen- und plattformunabhängigen Lösung eines Problems in einem Kontext. Das Strategized-Locking-Muster wird im übernächsten Kapitel besprochen.
Was macht dann aber ein Idiom aus? In [POSA1, S.14] liest man folgende Definition: *Ein Idiom ist ein für eine bestimmte Programmiersprache spezifisches Muster auf einer niedrigen Abstraktionsebene. Es beschreibt, wie man bestimmte Aspekte von Komponenten oder den Beziehungen zwischen ihnen mit den Mitteln der Programmiersprache implementieren kann.*
$\iota\delta\iota os$ (gr., sprich: “idios”) bedeutet *eigen*. Ich würde ein Idiom demzufolge als Eigenheit (einer Sprache) oder auch als Trick oder Kniff bezeichnen. Idiome nutzen gezielt Eigenschaften (Eigenheiten) einer Sprache aus, um relativ konkrete Lösungen für Probleme in einem Kontext anzubieten. Das im folgenden Kapitel behandelte C++-Scoped-Locking-Idiom macht sich die Speicherverwaltung des C++ - Laufzeitsystems zunutze, um einfacher zu wartende, wiederverwendbare und fehlertolerantere (also einfach qualitativ hochwertigere) Software zu erstellen[1].

[1] Es ist sicherlich sinnvoll, sich mit wohldefinierten Qualitätskriterien für Software zu beschäftigen; ein Einstieg bietet [RENZ].

2 Das C++-Scoped-Locking-Idiom

> "C makes it easy to shoot yourself in
> the foot. C++ makes it harder, but when
> you do, it blows away your whole leg."
> *Dr. Bjarne Stroustroup (* 1950)*

2.1 Problemstellung

Da ein Muster (Idiom) immer eine Lösung eines Problems in einem Kontext repräsentiert, beginne ich mit einer Problemstellung, die durch den Einsatz des C++ - Scoped-Locking-Idioms gelöst werden kann.
Es gilt, ein klassisches Synchronisationsproblem zu behandeln. Der nebenläufige Zugriff mehrerer Threads auf eine gemeinsam benutzte Ressource muss synchronisiert werden, um deterministisches Verhalten gewährleisten zu können. Der Ad-Hoc[2]-Ansatz wäre, den kritischen Abschnitt mit einem Mutex zu schützen, um Nebenläufigkeit beim Zugriff auf gemeinsam benutze Ressourcen (z.B. globale Variablen) zu verhindern. Der Code hierfür könnte wie folgt aussehen:

```
/*
 * Funktion, die eine globale Variable (globalVar)
 * veraendert und vor Nebenlaeufigkeit geschuetzt ist
 */
bool CriticalFunction(){
  globalMutex.acquire();      //Akquirieren
  globalVar = (42 - pi) * getSolarSystem();
  globalMutex.release();      //Freigeben
  return true;
}
```

In der Funktion `CriticalFunction()` wird die globale Variable `globalVar` manipuliert. Der Mutex `globalMutex` wird vor dem Beginn der Manipulation akquiriert[3] (d.h. in diesem Fall "gesperrt") und nach der Manipulation wieder freigegeben. Diese Lösung sollte an sich funktionieren.
Was passiert aber mit dem Mutex, wenn innerhalb des kritischen Abschnitts (also während der Mutex gesperrt ist) eine Exception auftritt? Vielleicht wirft die aktuelle Version der (Bibliotheks-)Funktion `getSolarSystem()` keine Exception, wohl aber eine zukünftige! Was passiert weiterhin mit dem Mutex, wenn innerhalb des kritischen Abschnitts eines der berühmt-berüchtigten C-Konstrukte, wie `goto`, `break`, `return` oder `continue` benutzt werden?

[2] Ad-hoc: lat. "auf der Stelle", "sofort", aber auch: "provisorisch"
[3] lat. acquirere: herbeisuchen

Im Quellcode könnte dies in etwa so aussehen:

```
/*
 * Funktion, die eine globale Variable (globalVar)
 * veraendert und (durch einen MUTEX geschuetzt)
 * nebenlaeufig aufgerufen werden kann
 */
bool CriticalFunction(){
  globalMutex.acquire();  //Akquirieren
  globalVar = (42 - pi) * getSolarSystem();
      if( globalVar != 23 )
          return false;   //Ausprung aus der Funktion
  globalMutex.release();  //Freigeben
  return true;
}
```

Durch das Einbringen des Abbruchkriteriums `if(globalVar != 23)` wurde ein Aussprungpunkt aus der Funktion implementiert; der Mutex wird **nicht** freigegeben. Ein anderer Thread könnte nun nie den kritischen Abschnitt betreten und würde ewig warten, da er den Mutex selber nicht akquirieren kann. Somit hätten wir eine **Verklemmung**[4] produziert.
Obigem einfachen Beispiel mangelt es an Komplexität und an Zeilenanzahl. Man stelle sich nur vor, zwischen Akquirieren und Freigeben des Mutex´ befinden sich mehrere Bildschirmseiten Quellcode! Sollte also der kritischer Abschnitt durch eine der oben beschriebenen Art und Weisen verlassen werden, ist es sehr wahrscheinlich, dass der Mutex nicht mehr freigegeben wird (und somit "gesperrt" bleibt). Bei komplexem oder unübersichtlichem Code ist es extrem aufwändig wenn nicht sogar fast unmöglich, jede mögliche "Aussprungmöglichkeit" zu erkennen und entsprechend mit dem Freigeben des Mutex´ zu behandeln. Aus dieser Misere hilft uns das Scoped-Locking-Idiom.

[4] in der Literatur findet sich häufig das englische Äquivalent "Deadlock"

2.2 Vorstellung des Idioms

"Das C++-Idiom Scoped Locking garantiert, dass der Eintritt in einen Anweisungsblock automatisch eine neue Sperre akquiriert und dass das Verlassen des Blocks diese Sperre automatisch wieder freigibt - unabhängig davon, wie der Block verlassen wird!" [POSA2, S.359]
Diese Essenz des Idioms scheint genau unser Problem zu beantworten. Egal, ob wir unseren kritischen Abschnitt korrekt oder inkorrekt verlassen, muss unser Mutex (oder eine beliebige andere Sperr-Komponente) freigegeben werden, um den ordnungsgemäßen Programmablauf zu garantieren. Als "Bauanleitung" könnte man obigen Satz beispielsweise so umsetzen: *"Implementiere eine Wächterklasse, die beim Erzeugen eine Sperre akquiriert und diese am Ende ihrer Lebenszeit wieder freigibt!"*
In C++ könnte diese Wächter-Klasse durch den folgenden Code-Auszug repräsentiert werden:

```
/*
 * MUTEX-Waechterklasse, die zur Implementierung
 * des Scoped-Locking-Idioms benutzt werden kann
 */
class Guard{
  Mutex* m_Mutex;

  Guard( Mutex* mutex ) : m_Mutex(mutex){
    m_Mutex->acquire();
  }

  ~Guard(){
    m_Mutex->release();
  }
};
```

Die Klasse `Guard` (engl. Wächter) bietet in dieser extrem kompakten Version schon die Grundfunktionalität bzw. Essenz des Scoped-Locking-Idioms. Erklären lässt sich der Kern des Idioms folgendermaßen:

- Die **Membervariable** `Mutex* m_Mutex` beinhaltet einen Pointer (eine Referenz) auf den Mutex, der den kritischen Abschnitt schützen soll.
- Im **Konstruktor** `Guard(Mutex* mutex)` wird der Mutex an unsere Wächterklasse übergeben und sofort akquiriert[5].
- Durch den Aufruf des **Destruktors** wird der Mutex implizit freigegeben.

An einem konkreten Code-Beispiel wird der daraus resultierende Nutzen klar. Ich beziehe mich wieder auf die `CriticalFunction()` aus Kapitel 2.1 auf Seite 6.

[5] der Aufruf zum Anlegen der Wächterklasse blockiert solange, bis sie den Mutex akquirieren kann

```
/*
 * Schutz vor nebenlaeufiger Manipulation einer globalen
 * Variablen (globalVar) mittels einer Waechterklasse (Guard)
 */
bool CriticalFunction(){
  Guard mutexGuard( globalMutex );
  globalVar = (42 - pi) * getSolarSystem();
      if( globalVar != 23 )
          return false;
  return true;
}
```

In dieser Version der Funktion wird am Beginn des zu schützendes Abschnitts ein Objekt der Wächterklasse `Guard` angelegt. Wie in Kapitel 2.2 auf Seite 7 zu sehen ist, wird der Mutex, der der Wächterklasse im Konstruktor übergeben wird, von ihr implizit akquiriert. Somit ist der Eintritt in den kritischen Abschnitt gesichert. Auch in dieser Version der Funktion findet sich ein Abbruchkriterium bzw. der zugehörige Aussprungpunkt (`return false`). Verlässt der Ausführungsfaden unserer Applikation die Funktion, verliert die Wächterklasse ihre "Daseinsberechtigung" und ihr Destruktor wird automatisch aufgerufen. Durch den Destruktoraufruf wird der Mutex freigegeben. **Egal auf welche Weise** der kritische Abschnitt oder die Funktion verlassen wird, wird am Ende der Lebenszeit der Wächterklasse der Mutex freigegeben. Damit hat unser Code erheblich an Einfachheit, Robustheit und Wartbarkeit gewonnen. Beliebige Abbruchkriterien oder Sprunganweisungen[6] können eingefügt werden, ungefangene Exceptions können auftreten, ohne weiteren Aufwand oder Nichtdeterminismus entstehen zu lassen.
Wichtig für das Verständnis des Idioms ist die Rolle der C++-Speicherverwaltung. Für Objekte, die mit `new` angelegt werden, wird Speicher auf dem Heap alloziert. Werden diese Objekte nicht mehr benötigt, nimmt das System das Freigeben des belegten Speichers **nicht** selbsttätig vor. Der Entwickler muss das Objekt (bzw. den Speicher in dem es liegt) explizit mittels `delete` freigeben[7].
Objekte, die ohne `new` angelegt werden, werden als lokal-existierende Objekte betrachtet und auf dem Stack angelegt. Die Speicherverwaltung dieser Objekte wird durch das C++-Laufzeitsystem erledigt, indem es den Ablauf der Lebenszeit des Objektes feststellt, dessen Destruktor aufruft den Stack-Pointer verschiebt (und somit Platz auf dem Stack schafft).

[6] "beliebig" ist an dieser Stelle nicht ganz korrekt, siehe Kapitel 2.3 auf Seite 10
[7] In JAVA erledigt die sog. Garbage-Collection diese Aufgabe automatisch

Was mit dieser Version der Wächterklasse nicht erreicht werden kann, ist das explizite Freigeben des Mutex‘. Vielleicht soll der Mutex zwischenzeitlich freigegeben und später innerhalb der Funktion noch einmal benutzt werden. Sollte der Mutex direkt freigegeben werden, wird er am Ende der Lebenszeit des Wächterobjekts erneut freigegeben, was, je nach Mutex-Implementierung, durchaus nicht-definiertes Verhalten auslösen kann. Zum besseren Verständnis des Problems folgt der entsprechende Quellcode:

```
/*
 * Schutz vor nebenlaeufiger Manipulation einer globalen
 * Variablen (globalVar) mittels einer Waechterklasse (Guard)
 */
bool CriticalFunction(){
  Guard mutexGuard( globalMutex );       //Akquirieren
  globalVar = (42 - pi) * getSolarSystem();
      if( globalVar != 23 )
          globalMutex->release();        //Freigeben
  ...
  return true;       //Ende der Lebenszeit des Waechters
}                    //-> erneutes Freigeben des Mutex'
```

Um das explizite Freigeben zu ermöglichen und jegliches undefinierte Verhalten zu beseitigen, erweitern wir unsere `Guard`-Klasse um zwei Methoden (`acquire()` und `release()`) sowie eine `bool`-Membervariable (`m_IsOwnerOfMutex`). Der Code der fertigen Wächterklasse könnte nun folgendermaßen aussehen:

```
/*
 * MUTEX-Waechterklasse, die zur Implementierung
 * des Scoped-Locking-Idioms benutzt werden kann
 * und explizites Freigeben des MUTEX erlaubt
 */
class Guard{
    Mutex* m_Mutex;
    bool   m_IsOwnerOfMutex;

    void acquire(){
        if( !m_IsOwnerOfMutex ){
          m_Mutex->acquire();
          m_IsOwnerOfMutex = true;
        }
    }
    void release(){
        if( m_IsOwnerOfMutex ){
           m_IsOwnerOfMutex = false;
           m_Mutex->release();
        }
```

```
     }
    Guard( Mutex* mutex ) :
        m_Mutex( mutex ), m_IsOwnerOfMutex( false ){
        acquire();
    }
    ~Guard(){ release(); }
};
```

Die Membervariable `m_IsOwnerOfMutex` beinhaltet den Zustand der Wächterklasse. In ihr wird die Information gespeichert, ob das aktuelle Objekt der Wächterklasse den Mutex akquiriert hat (sprich "sein Besitzer ist") oder nicht, um mehrmaliges Akquirieren oder Freigeben zu verhindern. Soll der Mutex jetzt "von Hand" gesperrt oder freigegeben werden, ist dies über die neuen Methoden `acquire()` und `release()` möglich. Die entsprechenden Methoden des Mutex' dürfen nicht mehr direkt benutzt werden[8].

2.3 Was gilt es zu beachten?

Auch das C++-Scoped-Locking-Idiom bringt einige wenige Nachteile mit sich. Bei Rekursion sollte man äußerste Vorsicht walten lassen. Ruft die Methode, die den kritischen Abschnitt beinhaltet, sich selber auf, entsteht eine Verklemmung, da ein neues Objekt der Wächterklasse den (schon akquirierten) Mutex akquirieren will und weder der alte noch der neue Rekursionsschritt weiterarbeiten können. Abhilfe schafft ein rekursiver Mutex, der beispielsweise mit dem Thread-Safe-Interface-Muster[9] implementiert werden kann.
Ein weiterer kritischer Punkt sticht sofort ins Auge. Wie verhält es sich mit der Performanz eines Systems, wenn am Beginn jedes kritischen Abschnitts ein Objekt angelegt wird, welches am Ende des Abschnitts noch dazu automatisch zerstört wird? Die Dimension dieses Problems kann man verkleinern, indem man der Wächterklasse mit möglichst wenigen oder keinen virtuellen Methoden[10] ausstattet. Für jeden Aufruf einer virtuellen Methode muss in der "Virtuellen-Methoden-Tabelle" (`vtable`) eines Objektes nach der Adresse der entsprechenden Methode gesucht werden, was zwangsläufig Performanzeinbußen zur Laufzeit mit sich bringt.
Die automatische Speicherverwaltung des C++-Laufzeitsystems funktioniert nicht (bzw. der Destruktor eines Objektes auf dem Stack wird nicht aufgerufen), wenn eine der folgenden Funktionen benutzt wird:

- `longjmp(jmp_buf env)` veranlasst einen Sprung innerhalb des Stacks
- Die Win32-Funktion `TerminateThread(Handle hThread)` beendet einen Thread ohne Destruktor-Aufrufe der Stackobjekte

[8] auf Sichtbarkeitsattribute oder Zugriffsbeschränkungen("friend") wurde bewusst verzichtet; diese verstehen sich von selbst und würden den Code nur unnötig verkomplizieren

[9] siehe [POSA2,S.383]

[10] ich zähle hier auch Destruktoren als Methoden

- Die UNIX-Funktion `thread_exit()` zeigt dasselbe Verhalten

Je nach Compiler treten bei Einsatz des Scoped-Locking-Idioms unter Umständen Warnungen zur Compilezeit auf, da ein Objekt der Wächterklasse (`Guard`) angelegt wird, welches später nicht mehr referenziert oder benutzt wird. Mögliche Warnmeldungen lauten *"Statement has no effect..."* oder *"unused local variable"*. Bei häufigem Einsatz des Idioms kann die Suche nach richtigen Fehlern oder wichtigen Warnungen in der Compiler-Ausgabe stark erschwert werden. Eine Möglichkeit, die Abhilfe schafft, ist, die Compilerwarnungen (z.B. mittels Präprozessormakros wie `#pragma`) gezielt auszuschalten. Alternativ kann durch folgendes Makro dem Compiler eine Nutzung des Wächterobjekts vorgetäuscht werden:

```
/*Praeprozessormakro*/
#define UNUSED_ARG(arg) {(if(&arg);)}
/* ... */

/*Anlegen des Waechterobjektes*/
Guard mutexGuard( &globalMutex );

/*Vortaeuschen der expliziten Nutzung des Objektes*/
UNUSED_ARG(guard)
```

2.4 Beispielimplementierung

Als Beispiel für die Funktionsweise des Idioms habe ich eine nebenläufige MFC-Applikation erstellt, in der zwei Threads (symbolisiert durch "PacMan's") um eine gemeinsam benutzte Ressource (hier einen Slider) konkurrieren. Jeder der beiden Threads schläft eine zufällige Zeit und probiert dann den Slider in eine vorher definierte Richtung zu bewegen. Ein Screenshot der Beispielapplikation findet sich in Anhang (Kapitel 5.3 auf Seite 29). Der entsprechende Quellcode sieht folgendermaßen aus:

```
1: while( dlg->getContinueWork() ){
2:   {
3:     DS_Mutex_Guard guard(sliderMutex);
4:     PostMessage( dlg->GetSafeHwnd(),
5:                  WM_SLIDERCHANGE, ...);
6:   }
7:   sleepTime = dlg->randFunc();
8:   Sleep( sleepTime );
9:  }
```

Die zusätzliche Klammerung in den Zeilen 2 und 6 dient dazu, die Lebenszeit des Objekts der `DS_Mutex_Guard`-Klasse und damit den Zeitraum der Mutex-Sperrung zu begrenzen. Ohne diese Klammern wäre jeder Durchlauf der Schleife komplett vor

Nebenläufigkeit geschützt, also auch die Anweisungen, die den Thread lediglich schlafen legen. Semantisch macht es keinen Sinn, die `PostMessage`-Methode als kritischen Abschnitt zu bezeichnen, denn das Betriebssystem[11] kümmert sich selbst um die Thread-Sicherheit der Message-Queue. Für unsere Beispielzwecke mag dies ausreichend sein. Die Implementierung der Klasse `DS_Mutex_Guard` entspricht exakt der Klasse `Guard` im Kapitel 2.2 auf Seite 9. Mittels des Buttons mit der Aufschrift "Start (Mutex)" benutzt die Applikation einen einfachen Mutex zur Synchronisation. Durch aufeinander folgendes Drücken des "Stop"- und des "Start (Error)"-Buttons kann ein Deadlock erzeugt werden. Der linke "PacMan" (bzw. "Thread") steigt mit einem definierten Kriterium aus der Methode aus, ohne den Mutex wieder freizugeben. Der rechte Thread wartet nun bis zum Beenden der Applikation auf das Akquirieren des Mutex'. Nach erneutem Start der Applikation kann mit "Start (Guard)" gezeigt werden, wie das C++-Scoped-Locking-Idiom diese Fehlersituation hervorragend behandelt und keine Verklemmung entstehen lässt.

[11] meiner Ansicht nach besser: "der Windowmanager"

2.5 Bekannte Verwendungen

2.5.1 MFC

In den MFC findet sich eine Implementierung des Idioms. Die beiden Klassen bzw. deren Konstruktoren `CSingleLock(CSyncObject* pObject)` und `CMultiLock(CSyncObject* ppObjects[])` entsprechen jeweils exakt einer Wächterklasse, in deren Konstruktor eine Objektreferenz einer von `CSyncObject` abgeleiteten Klasse (z.B. `CMutex`) übergeben wird und dort akquiriert wird. Ebenso erfolgt die Freigabe des Objektes im Destruktor der `CSingleLock`- bzw. `CMultiLock`-Klasse.

2.5.2 Java

In Sun's JAVA findet sich das Idiom im Bytecode wieder. Der Compiler erzeugt für jede möglicherweise auftretende Exception einen Block, der immer mit `monitorexit` beendet wird, um die Sperre in jedem beliebigen (Fehler-) Fall aufzuheben.

2.5.3 .NET (C#)

In Microsoft's C# (sprich "C sharp") findet sich das Idiom ebenfalls. Das `lock`-Schlüsselwort verhindert die Nebenläufigkeit innerhalb eines Blocks. Egal, auf welche Weise der Block verlassen wird, die Sperre wird immer aufgehoben. Das Codebeispiel verdeutlicht den Sachverhalt:

```
lock( this ){         //Beginn der Sperre
  globalVar = (42 - pi) * getSolarSystem();
      if( globalVar != 23 )
        return false;
  return true;
}                     //Ende der Sperre
```

2.6 Siehe auch

Zu jedem Muster gehört auch ein "also known as". Das C++-Scoped-Locking-Idiom ist eine Spezifikation eines allgemeineren Idioms namens "Resource-Acquisition - is-Initialization", welches Dr. Bjarne Stroustrup in [STROU1] (ab Ausgabe des Jahres 2000) beschreibt. Er (als "Erfinder von C++") sieht in diesem Ansatz eine so gute Idee, dass er laut [STROU2] dieses Idiom in der neuen C++-Spezifikation an vielen Stellen in die Sprache einbringen will. Der allgemeine Ansatz findet sich auch in vielen anderen objektorientierten C++-Frameworks,indem die Akquisition einer Ressource im Konstruktor eines Objektes und die Freigabe in dessen Destruktor objektorientiert gekapselt wird (z.B. `malloc` und `calloc`). Eine alternative Bezeichnung dieses

Musters wurde von Kevlin Henney auf der EuroPLoP[12] 2000 als “Execute-Around Object” geprägt.

2.7 Das Scoped-Locking-Mini-HowTo

1. Kopple das Öffnen und Schließen einer Sperre an die Lebenszeit einer Wächterklasse.

2. Lege ein Objekt der Wächterklasse (auf dem Stack!) am Beginn des zu schützenden Bereiches an.

3. Wird der Gültigkeitsbereich des Objekts (egal auf welche Weise) verlassen, wird die Sperre automatisch geöffnet.

[12] Europäische Muster-Konferenz

3 Das Strategized-Locking-Muster

"Make everything as simple as
possible, but not simpler."
Albert Einstein (1879-1955)

3.1 Motivation

Wenn man das Strategized-Locking-Muster isoliert betrachtet, kommt schnell die Ansicht auf, dass das Muster an sich relativ naheliegend und wenig aufregend ist. Bei der Beschäftigung mit diesem Muster kommt es hier besonders auf den Kontext an, in dem das Muster entstanden ist und steht. Aus diesem Grund wird die "Motivation" für dieses Muster etwas umfangreicher ausfallen. Es ist hilfreich, wenn der Leser beim Lesen der folgenden Kapitel die Motivation immer im Hinterkopf behält und sich bewusst ist, welchen Profit das Muster bringt.

Als Umfeld für das Muster könnte die Aufgabenstellung angesehen werden, ein plattformunabhängiges Framework zu entwickeln. Es soll weiterentwickelt werden und auch an andere Entwickler verkauft werden, die durch das Framework Komfort und Arbeitsersparniss gewinnen sollen. Im Rahmen der Ausarbeitung und als Fortführung des ersten Teils liegt das Hauptaugenmerk auf den Synchronisationsmechanismen innerhalb des Frameworks. Das Framework soll also beispielsweise auf verschiedenen Betriebssystemen wie Windows NT, VxWorks und POSIX-konformen Systemen kompilierbar und lauffähig sein. Damit stellt sich das Problem, dass die systemnahen Synchronisationsmechanismen von den Plattformen[13] bereitgestellt werden und so in Implementierung und API extrem plattform-spezifisch sind. Diese grundlegenden Unterschiede betreffen daher schon die Implementierung des Frameworks.

Es gilt nun, einen Weg zu finden, der es erlaubt, je nach Plattform unterschiedliche Code-Blöcke zu kompilieren. Plattformunabhängiger Code zum Schützen eines kritischen Abschnitts könnte (mithilfe von Präprozessormakros) in etwa so aussehen:

```
/*
 * plattformunabhaengige Funktion, die eine
 * globale Variable (globalVar) manipuliert
 */
void CriticalFunc(){
  #IFDEF W32
    W32EnterCriticalSection();
  #ELIF defined VX
    Vx_CritSecStart();
  #ELSE
    POSIX_Mutex_Down();
  #ENDIF
  globalVar = ...;
```

[13] "Betriebssystem" und "Plattform" sind in diesem Kontext äquivalent

```
  ...
}
```

Durch die verschiedenen Makros (z.B. `#IFDEF`) wird unmittelbar vor dem Kompilieren festgestellt, für welche Plattform compiliert werden soll und so der entsprechende plattformspezifische Code übersetzt. Sowohl der Code des Frameworks als auch der Code der Nutzer des Frameworks besteht nun an jeder Stelle, an der Synchronisation benötigt wird, aus einem solchen kryptischen, schwer les- und wartbaren Code-Fragment. Man stelle sich nur das Hinzufügen einer weiteren Plattform vor (die Rolle des "Portierers"): an jeder Stelle, wo Synchronisationsbedarf besteht, muss eine neue Präprozessoranweiung hinzugefügt werden, sowohl im Code des Frameworks wie auch im Code der Nutzer bzw. Kunden des Frameworks.

Was macht der mehr oder weniger erfahrene Softwareentwickler in so einer Situation? Er probiert, das Problem mit Mitteln der Objektorientierung zu umgehen! Es wäre denkbar, eine Klasse als plattformunabhängige Synchronisationskomponente in das Framework einzugliedern, die exemplarisch so aussehen könnte:

```
/*
 * Klasse, die eine plattformunabhaengige Kapselung
 * eines Locking-Mechanismus repraesentiert
 */
class Lock{
  void acquire(){
    #IFDEF W32
      W32EnterCriticalSection();
    #ELSE IFDEF VX
      Vx_CritSecStart();
    #ELSE
      POSIX_Mutex_Down();
    #ENDIF
  }
  ...
};
```

Damit wären `#IFDEF` und seine kryptischen Freunde in einer Klasse gekapselt, beseitigt ist das Problem an sich aber nicht. Der Benutzer des Frameworks kann jetzt beruhigt die `Lock`-Schnittstelle benutzen, Entwickler und Portierer müssen aber nachwievor mit einem unschönen Konzept arbeiten.
Wenn wir weiterführend davon ausgehen, dass wir nicht nur einen sondern mehrere Synchronisationsmechanismen[14] für mehrere Plattformen[15] implementieren und bereitstellen wollen, bricht eine kombinatorische Explosion von `#IFDEF`s über uns her-

[14] z.B. Mutex, Semaphore, Leser-Schreiber-Sperre
[15] z.B. Windows NT, VxWorks, POSIX

ein. `N` Plattformen benötigen jeweils `M` Sperrmechanismen, also beinhaltet unser Framework `N` Klassen mit jeweils `M` `#IFDEF`s. Das Resultat ist nachwievor schwer wartbar, umständlich erweiterbar und fehleranfällig.
Ein Blick über den Tellerrand offenbart vielleicht das *Strategie-Muster*, beschrieben von Gamma, Helm, Vlissides und Johnson in [GOF1, S.373ff]. Das nächste Kapitel erklärt kurz das Strategie-Muster. Für ein tiefergehendes Verständnis möge der geneigte Leser [GOF1] konsultieren.

Dies mag als Motivation ausreichen. Im weiteren Verlauf werde ich auf die drei Rollen “Entwickler”, ‘Portierer” und “Nutzer” des Frameworks wiederholt eingehen.

3.2 Das Strategie-Entwurfsmuster in Wort und Bild

"Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar. Das Strategie-Muster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren." [GOF1, S.373]

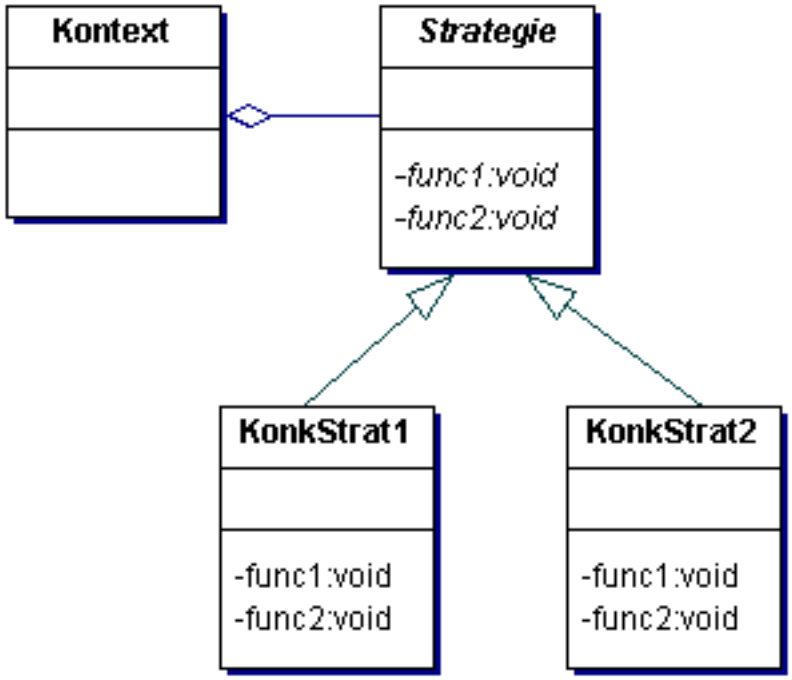


Abb. 3.2.1 Klassendiagramm des Strategie-Musters in UML-Notation

Die Struktur des Musters werde ich in nur wenigen Sätzen erläutern.

- Der Kontext ist der Klient, der eine Referenz auf die Schnittstelle "Strategie" hält und diese benutzt.
- Die abstrakte Basisklasse[16] "Strategie" stellt die gemeinsame Schnittstelle der verschiedenen Algorithmen dar.
- Die konkreten Implementierungen ("KonkStrat1", "KonkStrat2") der Basisklasse kapseln die unterschiedlichen Algorithmen.

Laut [GOF1, S.276] vermeidet das Strategie-Muster mehrfache Bedingungsanweisungen. Wie im folgenden Kapitel ersichtlich, sind die Präprozessoranweisungen die "mehrfachen Bedingungsanweisungen", die wir vermeiden wollen.

[16] in JAVA auch: Interface

3.3 Implementierung von Strategized-Locking

Zwei Möglichkeiten zur Implementierung bieten sich an. Zum einen lässt sich durch Vererbung unterschiedliches Verhalten einer Methode in verschiedenen Subklassen erreichen (“Polymorphie”). Alternativ lässt sich Strategized-Locking mittels Parametrisierung der Klienten-Klasse erreichen. Parametrisierung wird in C++ mit Templates realisiert, Java unterstützt Parametrisierung voraussichtlich ab Version 1,5. Ich werde die polymorphe Variante ausführlich darlegen, die parametrisierte unterscheidet sich vom Konzept her kaum und wird deshalb nur knapp behandelt.

3.3.1 Strategized-Locking mittels Polymorphie

Das Klassendiagramm für eine mögliche Implementierung könnte dementsprechend so aussehen:

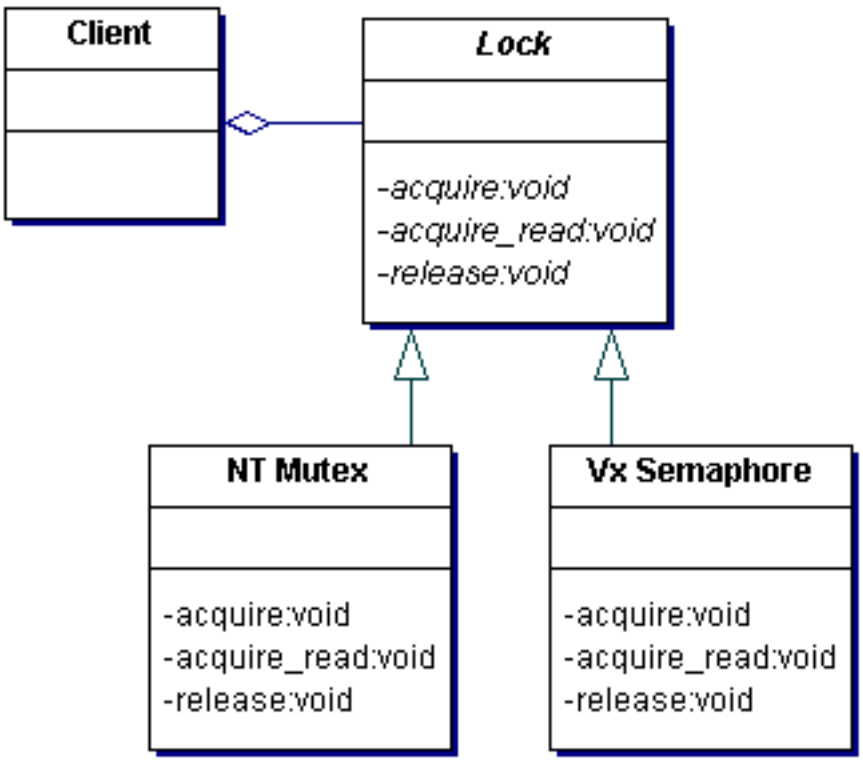


Abb. 3.3.1.1 Klassendiagramm des Strategized-Locking-Musters in UML-Notation

- Die abstrakte Basisklasse `Lock` bildet die Schnittstelle zur Nutzung der unterschiedlichen Synchronisationsmechanismen; die Methode `acquire_read()` ist für die Implementierung einer Leser-Schreiber-Sperre notwendig.
- Der `Client` hält eine Referenz auf die Schnittstelle, die er benutzen kann, um seine kritischen Abschnitte zu schützen.

- Es lassen sich beliebig viele Subklassen bzw. Implementierungen der Schnittstelle bilden, die die unterschiedlichen Locking-Strategien und die unterschiedlichen Plattformen mit deren spezifischem Code kapseln (`NT_Mutex`, `Vx_Semaphore`, ...).

Die Bedingung, welcher Mechanismus und welche Plattform verwendet werden soll, wird so ausschließlich an eine Stelle (nämlich die Objekterzeugung einer konkreten Subklasse der Schnittstelle) gebunden. Grundsätzlich brauchen weder der Entwickler noch der Nutzer des Frameworks ein einziges `#IFDEF`!
Aus Bequemlichkeitsgründen könnte man in einer Header-Datei dem Nutzer an einer Stelle die Arbeit erleichtern, indem man dort die bedingte Objektauswahl vor ihm versteckt. Dies könnte etwa so aussehen:

```
/*
 * vom Framework generierte Header-Datei
 */
#IFDEF W32
  #define PLATFORMLOCK NT_Mutex
#ELSE IFDEF VX
  #define PLATFORMLOCK Vx_Mutex
#ENDIF
```

Der Nutzer des Frameworks kann nun folgenden, eleganten und gut lesbaren, **plattformunabhängigen** Code schreiben:

```
int main( void ){
  Lock*   lock   = new PLATFORMLOCK();
  Client* client = new Client( lock );
  client->CriticalFunc();
  ...
}
```

3.3.2 Strategized-Locking mittels Parametrisierung

Da sich die Parametrisierung aus der Sicht des Musters nur wenig von der polymorphen Variante unterscheidet, folgen hier lediglich drei kommentierte Quellcodeauszüge. Der relevanteste Unterschied ist, dass bei der Parametrisierung die Implementierung der Schnittstelle zur Compilezeit festgelegt werden muss. Auf die Performanz zur Laufzeit wirkt sich dies aber positiv aus (Stichwort: `vtable`).

```
/*
 * Vom Nutzer des Frameworks entwickelte Klasse,
 * die einer plattformunabhaengigen Synchronisation bedarf
 */
template<class LOCK> class Client{
  LOCK* m_Lock;
  Client( LOCK* lock ) : m_Lock(lock) { ... }
  void CriticalFunc(){
    m_Lock->acquire();
    globalVar = ...;
    m_Lock->release();
   }
 ...
 };

/*
 * Diese Headerdatei koennte das Framework
 * dem Benutzer zur Verfügung stellen
 */
#IFDEF W32
  typedef Client<NT_Mutex>     MUTEXCLIENT
  typedef Client<NT_Semaphore> SEMAPHORECLIENT
#ELSE IFDEF VX
  typedef Client<Vx_Mutex>     MUTEXCLIENT
  typedef Client<Vx_Semaphore> SEMAPHORECLIENT
...
#ENDIF

/*
 * Der Nutzer des Frameworks koennte diesen
 * plattformunabhaengigen(!) Code schreiben
 */
int main( void ){
  MUTEXCLIENT client( &globalMutex );
  client.CriticalFunc();
  ...
}
```

3.4 Scoped-Locking in Verbindung mit Strategized-Locking

Wie der aufmerksame Leser beim Durchsehen des Quellcodes vielleicht bemerkt hat, lässt sich Strategized-Locking hervorragend mit Scoped-Locking aufwerten. So muss der Klient bzw. der Nutzer des Frameworks nicht explizit die Sperre freigeben, da der Destruktor-Aufruf der Wächterklasse dies implizit für ihn erledigt. Wir können die `Guard`-Klasse aus Kapitel 2.2 auf Seite 9 benutzen, um dies zu erreichen. Die Klasse `Client` kann erneut (z.B. im Konstruktor) mit einem `Lock`-Objekt konfiguriert werden und nutzt dies, um sich eine Wächterklasse anzulegen, die die Sperr-Logik und alle Vorteile des Scoped-Locking-Idioms kapselt. Im Klassendiagramm nimmt die Wächterklasse die Rolle des Klienten ein. Der umfangreiche Quellcode zu diesem Beispiel findet sich im Anhang (Kapitel 5.4 auf Seite 29).

3.5 Vorteile beim Einsatz des Musters

Die Vorteile werden nun aus den drei aus der Motivation bekannten Rollen beleuchtet. Für den **Entwickler und Portierer** des Frameworks ergeben sich folgende Vorteile:

- Ein deutlich geringerer Portierungsaufwand; es muss lediglich die Schnittstelle für die neue Plattform implementiert werden und in das Framework an einer definierten Stelle eingefügt werden.
- Gute Wartbarkeit wird erreicht durch das Wegfallen kryptischer und verstreuter Präprozessormakros und verteiltem plattformspezifischem Code
- Verbesserte Wiederverwendung ist meistens ein Hauptvorteil der Objektorientierung; in diesem Fall können sowohl die Synchronisationsmechanismen als auch der Code, der die Geschäftslogik des Kunden beinhaltet, an anderer Stelle weiterverwendet werden.

Für **Benutzer** des Frameworks ist vorteilhaft:

- Für ihn ist es irrelevant, welche Lockingmechanismen der zugrunde liegenden Plattform genutzt werden; es wäre sogar möglich, ihn einen "NullMutex"[17] nutzen zu lassen, um Plattformen zu unterstützen, die überhaupt keine Nebenläufigkeit anbieten.
- Es ist egal, für welche Plattform er entwickelt; einmal geschriebener Code lässt sich für verschiedene Plattformen kompilieren und einsetzen.
- Weiterhin wird dem Nutzer eine einfache Verwendung des Frameworks und damit auch der unterschiedlichen Plattformen ermöglicht; durch die hier vorliegende "Separation of Concerns"[18] kann er sich voll und ganz auf seine Geschäftslogik konzentrieren und muss sich nicht mit kryptischem, systemnahen und plattformspezifischem Code beschäftigen.

[17] siehe [POSA2], S.376

[18] engl.:"Trennung der Belange", geprägt von Dijkstra

- Die polymorphe Version erlaubt es, den Synchronisationsmechansimus zur Laufzeit auszutauschen.

3.6 Bekannte Verwendungen

3.6.1 .NET und Java

Strategized-Locking wird implizit auch in .NET und JAVA verwendet. Hier erkennt man das Muster eher am konkreten Einsatz als an einer Klassenstruktur. Beide Frameworks bzw. Runtimes bieten folgende Möglichkeiten:

- Das Programm läuft unabhängig von der verwendeten Plattform "unterhalb" der jeweiligen Runtime.
- Der Portierer muss lediglich neue Unterklassen definieren, die die Details seiner Plattform verbergen, sodass der Code der Runtime unberührt bleibt.
- Es existiert somit ein unveränderte Schnittstelle für den Entwickler und für das Framework!

3.6.2 C unter Linux

Auch hier lässt sich die Verwendung des Musters nicht in eine Klassenstruktur pressen. Der Linux-Kernel lässt sich für verschiedene Plattformen kompilieren und nutzt folglich auch die verschiedenen Synchronisationsmechanismen der Plattformen. Als Klient würde ich der Einfachheit halber den Compiler benennen, der mithilfe der Header-Datei `kernel.h` und des Verzeichnisses `src` (Source-Code-Quellen) den Kernel übersetzt. Die Header-Datei existiert genau einmal und enthält (als abstrakte Schnittstelle) die unter allen Plattformen benötigten Deklarationen der Kernelfunktionen. Für jede Plattform existiert ein eigenes Verzeichnis mit Quellen (`i386_src`, `sparc_src`, usw.). Das Makefile findet vor dem Kompilieren des Kernels heraus, für welche Plattform die Quellen kompiliert werden sollen. Darauf hin legt es einen symbolischen Link[19] von dem Verzeichnis, das die zu kompilierenden, plattformabhängigen Quellen enthält, auf das `src`-Verzeichnis. Der Compiler kann nun mithilfe des auf das richtige Verzeichnis zeigenden `src`-Verzeichnisses und der Datei `kernel.h` den korrekten Kernel übersetzen.

[19] unter Windows Verknüpfung genannt

3.7 Das Strategized-Locking-Mini-HowTo

1. Bilde eine Schnittstelle, die von den unterschiedlichen Synchronisationsmechanismen abstrahiert.

2. Komponenten nutzen ausschließlich diese Schnittstelle.

3. Kapsele die unterschiedlichen Mechanismen in konkrete Implementierungen der Schnittstelle.

4. Biete die Möglichkeit, an einer Stelle die Mechanismen aller Komponenten auszutauschen (`typedef` etc.).

4 Epilog

> “When one door closes, another opens. But we often look so regretfully upon the closed door that we don´t see the one that has opened for us.”
> *Alexander Graham Bell (1847-1922)*

Dem C++-Scoped-Locking-Idiom würde ich das Prädikat “Genial einfach und einfach genial” verleihen. Es ist sehr einfach anzuwenden und hilft, auch in der allgemeineren Form, bessere Software zu entwickeln. Obwohl das Idiom in den neueren Sprachen in seiner konkreten Form vermutlich weniger Anwendung finden wird, lässt sich das Konzept auch in andere Umgebungen einbringen (siehe “Bekannte Verwendungen” im Kapitel 2.5 auf Seite 13).
Beide Muster häufig werden implizit benutzt. Dennoch ist es sehr empfehlenswert, auch ihrem Verständnis Zeit zu widmen. Tritt bei der impliziten Nutzung eines Musters ein Problem auf, kann oft nur das tiefgreifende Verständnis des Musters weiterhelfen.
Das leicht aufkommende Vorurteil[20], viele Muster seien ähnlich, sollte man mit Misstrauen gegenüber treten. Zu einem Muster muss immer der Kontext betrachtet werden, aus dem heraus es entstanden ist. Ein Paradebeispiel bietet das Strategized-Locking-Muster bzw. das Strategie-Muster. Es ist für sich genommen eher unspektakulär und lässt sich augenscheinlich mit dem Muster “Abstrakte Fabrik” aus [GOF1] austauschen. Bei genauer Betrachtung ist dem nicht so! Beide Muster sind aus einem anderen Kontext heraus entstanden. Aus diesem Grund messe ich der Motivation des Strategized-Locking-Musters eine hohe Bedeutung zu.
“Kein Muster steht für sich” ist eine oft gebrauchte Phrase. Diese Aussage (u.a. von Richard Helm in [GOF2] zitiert). Im Rahmen dieser Ausarbeitung sollte auch klar geworden sein, dass sich Muster durchaus gewinnbringend verknüpfen lassen. Die sinnvollen, oft benutzten Kombinationen gehören genauso zum Wissen über ein Muster, wie sein Kontext.
Mit diesem kurzen Epilog endet die Ausarbeitung, deren Lesen hoffentlich lehrreich und interessant war.

[20] siehe auch [GOF2, S.5f]

5 Anhang

"I'm all in favor of keeping dangerous
weapons out of the hands of fools.
Let's start with typewriters."
Frank Lloyd Wright (1868-1959)

5.1 Glossar (alphabetisch sortiert)

- API: engl. "Application Programming Interface"; Schnittstelle, die von einer Bibliothek (s.a. Framework) dem Nutzer zur Verfügung gestellt wird
- Deadlock: siehe Verklemmung
- Exception: engl. "Ausnahme"; Datenstruktur, die bei einem Fehlerfall innerhalb einer Software-Komponente auftritt und diesen repräsentiert
- Framework: wiederverwendbare Software-Architektur
- Heap: ein Bereich im Hauptspeicher, der von der Anwendungssoftware selbst verwaltet werden kann
- kritischer Abschnitt: eine Folge von Instruktionen, die nicht nebenläufig ausgeführt werden sollte
- Leser-Schreiber-Sperre: ein Sperrmechanismus, der nebenläufigen Lese-Zugriff auf eine Ressource, aber keinen nebenläufigen Schreib-Zugriff erlaubt; ein Beispiel hierfür wäre der Zugriff auf eine Datenbank
- Linux-Kernel: essentieller Mittelpunkt des kostenlosen Linux-Betriebssystems, der die gesamte Grundfunktionalität für alle anderen Prozesse zur Verfügung stellt (siehe `http://www.linux.org`, `http://www.kernel.org`)
- Makefile: unter dem Betriebssystem UNIX sehr verbreiteter Mechanismus, um mittels eines Textfiles in bestimmter Syntax bedingtes Kompilieren zu ermöglichen
- Message-Queue: Warteschlange für Nachrichten
- MFC: "Microsoft Foundation Classes", umfangreiche Klassenbibliothek
- Mutex: engl. "Mutual Exclusion", Synchronisationsmechansimus bei dem zu jedem Zeitpunkt nur ein Ablauffaden im zu schützenden Abschnitt aktiv sein kann
- POSIX: "Portable Operating System Interface", Standardisierung der IEEE

- Runtime: Laufzeitumgebung, die Zwischencode interpretiert, zu Maschinencode kompiliert und ausführt
- Semaphore: Synchronisationsmechansimus, der einen internen Zähler verwaltet und so einer definierten Anzahl von Threads den Eintritt in den kritischen Abschnitt erlaubt; Anwendung finden Semaphoren beispielsweise bei einer Drucker-Warteschlange (“7 Druckaufträgen verteilt auf 3 Drucker”)
- STL: engl. “Standard Template Library”, Bibliotheksfunktionen und -klassen (meist für Container-Komponenten), die C++ zur Verfügung stellt
- Stack: vom Betriebssystem verwalteter Speicherbereich, der nach dem “last in first out“ - Prinzip arbeitet
- Thread: innerhalb eines Prozesses eigenständig ablaufender Programm-Code, der sich mit anderen Threads innerhalb des Prozesses Betriebsmittel teilt
- Verklemmung: gegenseitige (unbegrenzte) Blockade mehrerer Ablauffäden beim Warten auf eine Ressource
- VxWorks: Echtzeitbetriebssystem der Firma WindRiver
(siehe `http://www.vxworks.com`)

5.2 Literaturverzeichnis

[ALEX]
Christopher Alexander: *The Timeless Way of Building*, Oxford University Press, 1979

[GOF1]
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*, Addison-Wesley, 1996

[GOF2]
John Vlissides: *Entwurfsmuster anwenden*, Addison-Wesley, 1999

[JÄG]
Prof. Dr. Michael Jäger: *Skript zur Vorlesung Betriebssysteme I*, `http://homepages.fh-giessen.de/~hg52/lv/bs1/skripten/bs1skript/pdf/bs1skript.pdf` , FH Giessen-Friedberg, 2002

[POSA1]
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Patternorientierte Softwarearchitektur - Ein Pattern-System*, Addison-Wesley, 1996

[POSA2]
Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: *Patternorientierte Softwarearchitektur - Muster für nebenläufige und verteilte Objekte*, dpunkt, 2002

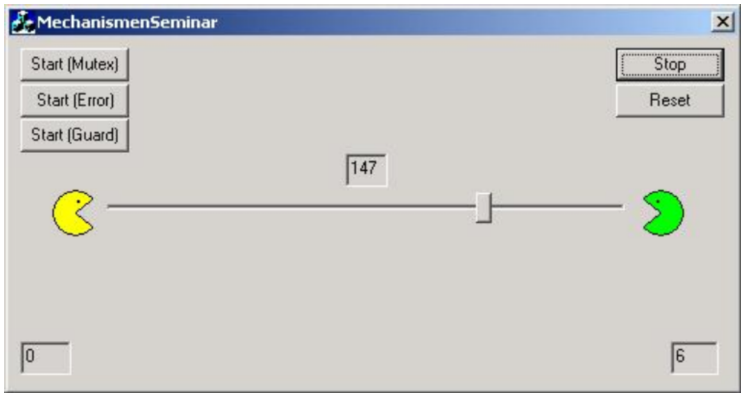
[RENZ]
Prof. Dr. Burkhardt Renz: *Folien zur Vorlesung Softwarearchitektur und Anwendungsentwicklung*, `http://homepages.fh-giessen.de/~hg11260/mat/saa-f1-print.pdf` , FH Giessen-Friedberg, 2002

[SILB]
Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Applied Operating System Concepts*, John Wiley & Sons inc., 2000

[STROU1]
Bjarne Stroustrup: *Die C++ Programmiersprache*, Addison-Wesley, 1998

[STROU2]
Bjarne Stroustrup: *Possible Directions for C++0x*, `http://www.research.att.com/~bs/C++0x_keynote.pdf`

5.3 Screenshot der Beispielapplikation des Scoped-Locking-Idioms



5.4 Pseudocode zur Kombination von Scoped- und Strategized-Locking

```
/*
 * plattformunabhaengige Waechterklasse,
 * die das Scoped-Locking-Idiom implementiert
 */
class Guard{
  Lock* m_Lock;
  bool   m_IsOwnerOfLock;

  void acquire(){
      if( !m_IsOwnerOfLock ){
        m_Lock->acquire();
        m_IsOwnerOfLock = true;
      }
  }
  void release(){
      if( m_IsOwnerOfLock ){
         m_IsOwnerOfLock = false;
         m_Lock->release();
      }
   }
  Guard( Lock* lock ) :
      m_Lock( lock ), m_IsOwnerOfLock false ){
      acquire();
  }
  ~Guard(){ release(); }
};
```

```
/*
 * abstrakte Basisklasse fuer alle Mechanismen
 */
class Lock{
  virtual void acquire()=0;
  virtual void acquire_read()=0;
  virtual void release()=0;
};

/*
 * konkrete Implementierung eines Mechanismus (Mutex)
 * fuer eine Plattform (Windows NT)
 */
class NT_Mutex : public Lock{
  void acquire(){
    W32EnterCriticalSection();
  }
  void release(){
    W32LeaveCriticalSection();
  }
  ...

/*
 * konkrete Implementierung eines Mechanismus (Semaphore)
 * fuer eine Plattform (VxWorks)
 */
class Vx_Semaphore : public Lock{
  void acquire(){
    Vx_semaphore_down();
  }
  void release(){
    Vx_semaphore_up();
  }
  ...

/*
 * Header-Datei, die das Framework dem Nutzer zur Verfügung stellt
 */
#IFDEF W32
  #define PLATFORMMUTEX     NT_Mutex
  #define PLATFORMSEMAPHORE NT_Semaphore
#ELSE IFDEF VX
  #define PLATFORMMUTEX     Vx_Mutex
  #define PLATFORMSEMAPHORE Vx_Semaphore
#ENDIF
```

```
/*
 * plattformunabhaengige Anwendungs-Klasse, vom Nutzer des Frameworks
 * entwickelt; nutzt eine Waechterklasse zur Synchronisation
 */
class Client{
  Lock* m_Lock;

  Client(){
    m_lock = new PLATFORMSEMAPHORE();
  }

  bool CriticalFunction(){
    Guard guard( m_Lock );
    globalVar = (42 - pi) * getSolarSystem();
      if( globalVar != 23 )
          return false;
    ...
    return true;
  }
};

/*
 * plattformunabhaengiger Code, vom Nutzer geschrieben
 */
int main( void ){
  Client* client = new Client();
  client->CriticalFunc();
  ...
}
```

Seminar Mechanismen

Zuständige Professoren:
Dr.rer.nat., Dipl.-Math. Wolfgang Henrich
Dr. phil. nat., Dipl.-Math. Burkhardt Renz

Zwei Muster für nebenläufige Objekte:

1. Thread-Safe Interface Muster
2. Double-Checked Locking Optimierungsmuster

Verfasser:
Kurt Rosenberg

SS02 FH - GIESSEN

Vorwort des Verfassers

Bemerkung: Der Lesbarkeit halber wurde durchweg die weibliche Form gewählt.
« Honni soit qui mal y pense ! »

Waren Sie schon mal in der Situation, etwa einem kleinem Kind, welches noch nicht schreiben aber bereits eine Zeit lang reden kann, irgendwelche grammatikalische Regeln erklären zu müssen? Gemeint ist etwas simples, wie z.B. warum man in

„ein einfaches Beispiel"

dem Wort „einfach"

die Endung „-es" anhängen muss; oder, warum eigentlich dies gar kein vollständiger Satz ist.

Egal wie die Antwort lautet, das größte Problem dabei (abgesehen davon, das Kind dazu zu bringen sich für die Erklärung tatsächlich zu interessieren) : es gibt keine gemeinsame Sprache für die Vermittlung der Information und Beschreibung des Zustände; Sie müssen fast von Null anfangen.

Ohne Begriffe wie „Adjektiv", „Nominativ", „Substantiv", „bestimmter Artikel", „fehlendes Prädikat", „Suffix", „Deklination" usw. müssen Sie der Zuhörerin z.B. klar machen, dass beim Aufbau eines vollständigen Satzes es bestimmte Regeln gibt, unter anderem eine Aktion im Satz dabei zu haben. Was eine Aktion ist ?! ...

Sicherlich ist das Ziel schneller erreicht bei einer Zuhörerin, die zumindest einige anderen korrekten Muster der Sprache kennt. Wenn diese sogar die Begriffe für die in diesen einigen Mustern vorkommenden Bestandteilen und Eigenschaften kennt ist dann die Sache umso schneller erklärt.[1] Wenn es für die einzelne verschiedene Muster eine eindeutige Benennung gibt, dann reicht sogar allein der Name, um sich zu verständigen.

Allgemein gilt, die Weitergabe von Kenntnissen über ein Thema wird nach Erkennung ,Zerlegung und Katalogisierung dessen Bestandteilen (durch eine Analyse) mäßig einfacher. Muster in der Softwareentwicklung sollen für immerwiederkehrende lösungsbedürftige Konstellationen diese wichtige Abstraktionsrolle spielen.

Muster beschreiben jeweils eine bewertete Art und Weise mit einem bestimmten Typ von Situation umzugehen, welche wiederum für sich bestimmte Eigenschaften besitzt und innerhalb eines charakteristischen Umfeldes auftritt. Dieser Umgangsform wird immer Stärken und durchaus Schwächen erweisen. Dies alles zusammen, der Zusammenhang zwischen Ausgangssituation, anzuwendende Umgangsform und Ergebnis, definiert das Muster.

---

[1] Ein Adjektiv eines neutralen Substantivs, welches mit bestimmten Artikel benutzt wird, erhält „-es" als Suffix in der Nominativdeklination.

Die zwei hierin vorgestellten Muster finden ihre Anwendungsgebiet bei der Programmierung mit mehrfachen nebenläufigen Ablauffäden. Ein Grundverständnis für dieses Thema ist Vorraussetzung. Darüber hinaus treten beide, wie es für Muster oft der Fall ist, in Zusammenhang mit weiteren Muster. Auf diese wird nur leicht eingegangen.

Tatsächlich bieten Erklärungen für sich selten eine Lösung. Die Umsetzung der aus diesen Erklärungen erkannten Möglichkeiten zur Lösung des Problems ist maßgebend für das Resultat. Aber wiederum, allein das Erkennen der Möglichkeiten ist oft schwieriger als eine danach durchzuführende Umsetzung. Bedauerlicherweise kann man dieses Können, diese Fähigkeit Lösungsmöglichkeiten zu erkennen, nicht als solches weitergeben. Es heißt, das Können an sich komme entweder von Übung oder vom Talent.

An der Wichtigkeit der Weitergabe von Erkenntnissen, an dem Anlernen von Techniken, zweifelt keiner. Gerade dies erlaubt es anderen, als nur den naturbegabten, in irgendeiner Tätigkeit besser werden zu können.
Je verständlicher die Weitergabe, umso geschmeidiger und flotter das Anlernen, desto praktikabler die Techniken. Hierin die Wichtigkeit aller Medien zur Vermittlung von Information; der Sprachen; der Muster.

Kurt Rosenberg

Das Thread-Safe Interface

Dieses ist ein simples Muster in seiner Anwendung. Ihre Umsetzung beschränkt sich auf das Einhalten einiger einfachen Richtlinien. Seine Umsetzung hat aber durchaus weitreichende Konsequenzen für die Robustheit und Performanz eines Systems.

Das Thread-Safe Interface Muster fokussiert sein bestreben in die richtige Trennung und Zuordnung der Zuständigkeiten für verschiedene Aktionen.[2]
Es versucht eine strikte Trennung verschiedener Aufgaben innerhalb einer Komponente zu erreichen. Besonders wichtig ist die Trennung der internen Aufgaben von denen der Schnittstellen nach außen.

Im folgenden C++- Code- Beispiel wird die Anwendung des Thread-Safe Interface Musters erläutert. Das Muster wird in die Ausprägung eines weiteren Musters namens „Strategized Locking"
[SSRB02] eingebettet. Das Thread-Safe Interface Muster ist an sich von der Benutzung weiterer Muster unabhängig. Tatsache ist dennoch, dass bei der Programmierung nebenläufiger Ablauffäden, sowohl das „Strategized Locking"-Muster als auch das „Scoped Locking"-Idiom [SSRB00], wesentliche Punkte der Synchronisation behandeln, die nicht außer Acht gelassen werden sollen. Daher liegt es auf der Hand die Bedeutung des Thread-Safe Interface Musters in Zusammenhang zu zeigen.

„Strategized Locking"
ist ein Muster, bei dem der gewählte Synchronisationsmechanismus für einen vor nebenläufigem Zugriff zuschützenden kritischen Abschnitt einer Komponente als Parameter übergeben wird. Auf dieser Art und Weise kann ich den Synchronisationsmechanismus abstrahieren, nicht in meiner Komponente fest Kodieren und erst beim Gebrauch der Komponente passend per Parameter übergeben.

Das „Scoped Locking"- C++ - Idiom basiert auf der Eigenschaften der Speicherverwaltung dieser Programmiersprache. Die Akquisition einer Sperre für einen kritischen Abschnitt wird beim Eintritt des Anweisungsblock, die Freigabe beim verlassen des Blocks – ungeachtet der Art und Weise wie dies geschieht -- garantiert. Objekte werden in C++ kreiert sobald ihren Konstruktor implizit oder explizit aufgerufen, und vernichtet sobald sie ihren „scope" (Gültigkeitsbereich) verlassen wird. Dieses Idiom benutzt diese Eigenschaft und Kapselt Sperrenmechanismen in Klassenobjekten. Somit wird garantiert das beim Anlegen einer Instanz solcher Sperrmechanismen, die Sperre mitkreiert wird und beim Verlassen des Gültigkeitsbereiches der Instanz, diese vernichtet wird und gleichzeitig die Sperre freigegeben wird.

---

[2] Eventuell hilfreich für den Leser ist die Vertraulichkeit mit einem weiteren Muster, welches sich mit Zuständigkeiten für Aktionen beschäftigt: das Strukturmuster „Dekorator". Dieses Muster erlaubt das dynamische Hinzufügen von Zuständigkeiten zu einem Objekt. [GoF96]

Beispiel:

```
class DateiCache {


  mutable Lock *lock_;
   public:

   // parametrisierter Konstruktor: strategized Locking
   DateiCache (Lock &l): lock_ (& l) { }


//---------------------------------


   const void *suche (const string &pfad) const{

      Waechter waechter (lock_);


      const void *datei_zeiger= checkCache(pfad);

      if (dateiZeiger == 0) {

            einfuegen (pfad);
            dateiZeiger = checkCache(pfad);
      }

      return dateiZeiger;

  }


//-----------------------------------

   void einfuegen(const string &pfad) {

      Waechter waechter( lock_);


      // restliche Implementierung
      // der Methode
        . . .
   }

}
```

Es gibt einige Probleme beim Einsatz von Instanzen dieser Klasse in der hier dargelegten Form. Übergibt man einen nicht rekursiven Mutex, könnte bei nebenläufigem Zugriff die Methode `einfuegen(...)` versuchen die Sperre zu bekommen, die bereits von der Methode `suche (...)` akquiriert wurde und man hätte eine Selbstverklemmung.

Die Übergabe eines rekursiven Mutexes, um die unmittelbar oben beschriebene Situation zu vermeiden, würde zuviel Aufwand mit sich bringen.

Die für dieses Beispiel einzige nicht zu Selbstverklemmung führende Übergabe, wäre die eines Null-Mutexes. Diese bringt aber natürlich keine Synchronisation mit sich.

Hier kommt das Thread-Safe Interface Muster ins Spiel. Verwendet man dieses bei der Implementierung von Komponenten nebenläufiger Anwendungen, welche interne Methodenaufrufe enthalten, vermeidet man durch einen entstehenden serialisierten Zugriff

die Selbstverklemmungen.

Thread-Safe Interface Muster im Einsatz:

```
class DateiCache {

  mutable Lock *lock;
  public:

  DateiCache (Lock &l): lock (& l) { }

//---------------------------------

  const void *suche (const string &pfad) const{
       Waechter waechter (lock_);

       return sucheIntern (pfad);
        (pfad);
  }
```

```
const void *sucheIntern(const string &pfad)const{

    const void *dateiZeiger= checkCache(pfad);

       if (dateiZeiger == 0) {
            einfuegenIntern(pfad);
             dateiZeiger = checkCache(pfad);
       }

       return dateiZeiger;
   }
```

```
//-----------------------------------

   void einfuegen(const string &pfad) {
      Waechter waechter( lock_);

      return einfuegenIntern
  }
}
```

```
void einfuegenIntern (const string &pfad) {

// restliche Implementierung
 // der Methode
     . . .
}
```

In dem Beispiel wird der Schnittstellenaufruf des Objektes von der tatsächlichen interne Aufgabe getrennt (Zuständigkeitstrennung und –zuordnung für die Aufgaben).

„Die einzige Aufgabe einer Schnittstellenmethode, beispielsweise einer öffentlicher Methode in C++, sollte sein, Sperren für die Komponente zu akquirieren bzw. freizugeben." [SSRB02]

Der Zustand der Komponenten wird geschützt und eine verbesserte Performanz des Systems wird erreicht, insofern dass Sperren nur dann akquiriert oder freigelassen werden bei tatsächlicher Notwendigkeit. Die Trennung der Sperrmechanismen für eine Komponente und ihrer Funktionalität vereinfacht die Struktur der Software.

Allgemeine Nachteile

Für jede Methode, die einen Schnittstellendienst darstellt muss eine neue interne Methode implementiert werden. Abgesehen vom Zuwachs an Speicherbedarf wird eine zusätzliche Indirektionsebene eingeführt.

Die Möglichkeit der Selbstverklemmung ist nicht ausgeschlossen. Interne Methode verlassen sich darauf, dass vor ihrem Aufruf alle nötigen Sperren akquiriert wurden. Sie sollen nicht versuchen selbst Sperren zu akquirieren und auch keine Schnittstellenmethoden aufrufen, denn diese akquirieren die Sperren.

Der Gewinn an Performanz kann möglicherweise wieder eingebußt werden, dadurch dass für jede Komponenteninstanz eine Sperre akquiriert wird. Der Synchronisationsaufwand fürs System kann sich mäßig erhöhen. Tritt eine Verklemmung, z.B. aus versehentlichem Programmierfehler, dann ist sie aufgrund der hohe Anzahl an Sperren schwieriger zu entdecken.

Die geläufigsten objektorientierte Programmiersprachen bieten Zugriffsschutz nur auf Klassenebene. Es ist also möglich für eine Instanz die internen Methoden einer anderen Instanz aufzurufen und somit die Akquisition der Sperre umzugehen. Daher sollen Objekte einer Klasse keine direkte Aufrufe auf interne Methoden anderen Objekte, wenn sie mit diesen Kommunizieren sollen.

Double-Checked Locking Optimierungsmuster

Das Double-Checked Locking Optimierungsmuster ist wie wir sehen werden ein Idiom. Ihre Umsetzung hängt stark von der verwendeten Speichermodell der eingesetzten Programmiersprache. Obwohl oft eingesetzt in Java, gerade das Speichermodell der zur Zeit am meisten verwendeten Versionen (IBM SDK für Java Technology Version 1.3 und Suns JDK 1.3) können dieses Muster nicht verwenden.

Das Double-Checked Locking Optimierungsmuster soll die Vermeidung unnötiges Akquirieren von Sperren gewährleisten. Typisch für die Erläuterung dieses Musters ist die Behandlung des auftretenden Problems, wenn die kanonische Form des Singleton-Musters [GoF96] in nebenläufiger Ablauffäden eingesetzt wird.

Das Singleton-Musters wird sollte dann verwendet werden, wenn man gewährleisten möchte, dass es von einer Klasse nur eine Instanz erzeugt werden kann.

Kanonische Form des Singleton-Musters:

```
class
Singleton{

public:
        static Singleton* holeDieInstanz () {
                if(Instanz_ == null){
                // Betrete kritischen Abschnitt
                Instanz_= new Singleton();
                // Verlasse kritischen Abschnitt
                }

                return Instanz_;
        }

        void methode_1();
        void methode_2();
        // weitere Methoden der Klasse

private:
        static Singleton * Instanz_ ;
        // Wird automatisch mit Null initialisiert.

}
```

Wenn man eine Instanz der Klasse erhalten möchte, benutzt man die Ruckgabe der statischen Funktion instance, etwa:

Singleton zeiger_exklusiveInstanz = Singleton::holeDieInstanz();

Das Problem hierbei ist das gleichzeitige betreten des kritischen Abschnittes von mehreren Ablauffäden. Abgesehen vom Speicherleck, ist das Verhalten des Systems unvorhersehbar und sicherlich nicht das erwünschte. Jede Rückkehr aus der Funktion, die eine weitere Initialisierung fälschlicher Weise erfolgreich gemacht hat, wird den Zeiger des bereits initialisierten Instanz für sich „klauen". Möglicherweise wurde aber bereits mit der vorherigen Instanz gearbeitet und Änderungen an den Zuständen vorgenommen. In jedem Fall ist Chaos vorgeschrieben.

Zuerst werde ich das ganze Muster in C++ behandeln, wo eine praktikable Lösung erreicht werden kann. Nachträglich werden wir sehen, dass dieses Optimierungsmuster eine Reihe von Rahmenbedingungen an Software, Hardware und Betriebssystem erfordert, was zu seiner größte Nachteil wird. Diese Abhängigkeiten macht auch den Lösungsweg für das aktuelle Javaspeichermodell unbrauchbar.

Als erstes muss der kritische Abschnitt vor nebenläufigem Zugriff geschützt werden. Wir benutzen hierfür das „scoped-locking"- Idiom[3], um eine automatische Akquisition und Freigabe der Sperre zu gewährleisten.:

```
class
Singleton{

public:
        static Singleton* holeDieInstanz () {

                Waechter<ThreadMutex> waechter <singleton_lock_>

                if(Instanz_ == null){
                // Betrete kritischen Abschnitt
                Instanz_= new Singleton();
                // Verlasse kritischen Abschnitt
                }

                return Instanz_;
        }

        void methode_1();
        void methode_2();
        // weitere Methoden der Klasse

private:
        static Singleton * Instanz_ ;
        static ThreadMutex  singleton_lock_ ;
}
```

Ist die Sperre korrekt implementiert ist nun auch der kritische Abschnitt vor nebenläufigem Zugriff geschützt. Dennoch wird eine Sperre bei jedem Aufruf auf die Methode **holeDieInstanz(...)** akquiriert, was sicherlich nach einer richtigen

[3] Auf das „scoped-locking"- Idiom wird kurz in dem Abschnitt „Thread-Safe Interface Muster" eingegangen. Eine tiefere Behandlung wird in [SSRB02] vorgenommen.

Initialisierung gar nicht mehr Notwendig wäre. Das Verschieben der Akquisition der Sperre innerhalb der if-Abfrage ist aber nicht möglich, denn dann hätte man wieder das Problem der Ausgangssituation.
Die Lösung hierfür, und auch endlich das Muster für sich, ist eine doppelte Überprüfung einzuführen. Mittels eines Flags ermittelt man den Zustand der Initialisierung vor Akquisition der Sperre. Deutet der Zustand des Flags auf eine bereits erfolgte Initialisierung hin, braucht man die Sperre gar nicht zu holen. Ist dies nicht der Fall, akquiriert man die Sperre. Nun ist es sicher, bis die Sperre freigegeben wird mischt sich kein weiterer Ablauffaden. Dennoch ist es möglich, dass in der Zeit in der die Sperre akquiriert wurde, ein anderen Ablauffaden die Initialisierung erfolgreich hinter sich gebracht hat. Also ist es notwendig ein zweites Mal den zustand des Flags zu testen. Deutet dieser immer noch auf eine noch nicht erfolgte Initialisierung, kann man diese nun bedenkenlos durchführen. Andernfalls ist eine neue Initialisierung auf keinen Fall durchzuführen. Die Freigabe der Sperre erfolgt, wie schon erläutert, beim Verlassen des Gültigkeitsbereiches automatisch. In unserem Beispiel wird der Zeiger auf die Instanz auch als Flag benutzt:

```
class
Singleton{

public:
        static Singleton* holeDieInstanz () {

          if (Instanz_ == null ){
              Waechter<ThreadMutex> waechter <singleton_lock_>

              if(Instanz_ == null){
              // Betrete kritischen Abschnitt
              Instanz_= new Singleton();
              // Verlasse kritischen Abschnitt
              }
          }

              return Instanz_;
        }

        void methode_1();
        void methode_2();
        // weitere Methoden der Klasse

private:
        static Singleton * Instanz_ ;
        static ThreadMutex  singleton_lock_ ;
}
```

Es ist klar, dass die Vermeidung von Sperrenakquisition eine große Steigerung der Performanz mit sich bringt. Zusammen mit der Vermeidung von Zugriffskonflikten, was die Verwendung des so brauchbaren Singleton-Musters möglich macht, stellen diese beide Aspekte die größten Vorteile des „double-checked" Optimierungsmusters dar.

Allgemeine Nachteile

Wie schon erwähnt, der größte Nachteil dieses Musters sind die für die Gewährleistung der korrekten Arbeitsweise auf verschiedenen Ebenen einzuhaltende Rahmenbedingungen. Auf diese wird nun eingegangen:

Bei Mehrprozessorplattformen können unerwünschte Nebeneffekte auftreten. Die Optimierung des Zwischenspeichers kann bei manchen Architekturen dazuführen, dass Lese- und Schreibeoperationen über mehrere CPU-Caches in einer anderen Reihenfolge als die geplante ausgeführt werden. Dieses Problem lässt sich mit der Verwendung von Speicherbarrieren vermeiden, welche aber CPU-spezifische Anweisungen und so auch einen weiteren Nachteil mit sich bringt. Die Verwendung eines Template-Adapters schränkt zumindest die Integration der CPU-spezifischen Anweisungen auf eine einzigen Stelle ein.

Ein weiterer Punkt: das Beschreiben und Lesen aller Bits des benutzen Flags (in unserem Fall des Zeigers zur Instanz) zur Abfrage des Zustandes muss atomar sein. Ist dies nicht der Fall, kann es dazu kommen das andere Ablauffäden einen ungültigen Flag lesen, obwohl dieser gerade gesetzt wird. In unserem Fall müsste also das Beschreiben des Speichers beim Aufruf des Konstruktors für die Instanz eine atomare Aktion bilden. Mann könnte dies umgehen, in dem man als Flag einen integralen Typ der Größe verwendet, für die die Hardware atomares Schreiben und Lesen unterstützt.

Leider tritt unter Verwendung des aktuellen Speichermodells der „Java Virtual Machine" ein ähnlicher Effekt ein, und macht das Muster unbrauchbar für die Sprache, unter der zur Zeit unterstützte Speichermodell:

Mini Java - Beispiel: (ohne double-check)

```
class Singleton {
  private static Singleton instance;
  private boolean inUse;
  private int val;

  private Singleton() {
     inUse = true;
     val = 5;
  }
}
```

Der hieraus resultierende Assembler-Code zeigt, dass der Aufruf auf den Konstruktor nicht den Speicher initialisiert (Abb. 1) Somit ist es nie gewährleistet, dass die Initialisierung der Instanz korrekt getestet werden kann. Hiermit funktioniert weder das Scope-Locking Konzept noch hilft die doppelte Prüfung auf das Flag.

```
;asm code generated for getInstance
054D20B0  mov   eax,[049388C8]    ;lade „instance" Referenz
054D20B5        test  eax,eax
054D20B7  jne   054D20D7
054D20B9  mov   eax,14C0988h
054D20BE  call  503EF8F0          ;Speicher allozieren
054D20C3  mov   [049388C8],eax    ;Zeiger speichern in „instance" Referenz
                                  ;Konstruktor noch nicht gelaufen

054D20C8  mov   ecx,dword ptr [eax]
054D20CA  mov   dword ptr [ecx],1         ;inline ctor - inUse=true;
054D20D0  mov   dword ptr [ecx+4],5       ;inline ctor - val=5;
054D20D7  mov   ebx,dword ptr ds:[49388C8h]
054D20DD  jmp   054D20B0
```

Abb. 1

Thread-Local , die Lösung?

Thread-Local Variablen erlauben es jeden Ablauffaden eine eigene Instanz der Variable zu verwalten. So ist es möglich für jeden Ablauffaden zu testen, ob dieser Ablauffaden seine Synchronisation bereits gemacht hat oder nicht. Thread-Local Variablen waren unter der Implementierung von Sun 1.2 sehr langsam und sollen in der Version 1.3 deutlich schneller sein. Was dies tatsächlich bedeutet kann ich leider nicht belegen.

Dennoch sind mit diesem Lösungsvorschlag von Alexander Terekhov (IBM - Deutschland) die bereits erwähnten Probleme für Mehrprozessorplattformen leider nicht aus der Welt:

Thread-Local Interface:

```
public class ThreadLocal {
    public Object get();
    public void set(Object newValue);
    public Object initialValue();
}
```

```
class Singleton {

  private static Singleton theInstance;
```

```
  private static final ThreadLocal perThreadInstance =
     new ThreadLocal() {
     public Object initialValue() { return createInstance(); }
  };

  public static Singleton getInstance() {
  // First check -- inside ThreadLocal.get()
        return (Singleton)perThreadInstance.get();
  }

  private static synchronized Singleton createInstance() {
    // Double check
    if (theInstance == null)
        theInstance = new Singleton();
    return theInstance;
  }
}
```

BIBLIOGRAPHIE

- **[GoF96]** ***Design Patterns: Elements of Reusable Software***
  by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
  (also known as the Gang of Four, or GoF).

- **[SSRB02]** ***Patternorientierte Software-Architektur Vol.2***
  by Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann
  (also known as the Gang of Four, or GoF).

- **Exploiting ThreadLocal to enhance scalability Level: Advanced**
  „http://www-106.ibm.com/developerworks/java/library/
  j-threads"
  Software Consultant, Quiotix Corp.
  October 2001

- **Multithreaded Programming:**
  „Can double-checked-locking be fixed?"
  Readers Letters
  Brian Goetz

- **"The 'Double-Checked Locking is Broken' Declaration":**
  "http://www.cs.umd.edu/~pugh/java/memoryModel/
  DoubleCheckedLocking.html"
  ***Signed by*: David Bacon(IBM Research) Joshua Bloch (Javasoft), Jeff**
  Bogda, Cliff Click (Hotspot JVM project), Paul Haahr, Doug Lea, Tom May,

Jan-Willem Maessen, John D. Mitchell (jGuru) Kelvin Nilsen, Bill Pugh, Emin Gun Sirer