Objektzugriffsmuster

Decorator, Proxy, Master-Slave

Seminararbeit zum Seminar “Softwaredesignpattern”

Name:	Mario Heidenreich
Matrikel:	9765183

Institut für Informatik
Universität Leipzig

Sommersemester 2009

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Kategorisierung der vorliegenden Muster 4
1.2 Begriffsklärung 4

2 Die Muster **5**

2.1 Decorator 5
2.1.1 Problemstellung 5
2.1.2 Lösung 6
2.1.3 Vor- und Nachteile 7
2.1.4 Anwendungsbeispiele 8
2.2 Proxy 8
2.2.1 Problemstellung 8
2.2.2 Lösung 9
2.2.3 Vor- und Nachteile 10
2.2.4 Anwendungsbeispiele 11
2.3 Master-Slave 11
2.3.1 Problemstellung 11
2.3.2 Lösung 12
2.3.3 Vor- und Nachteile 13
2.3.4 Anwendungsbeispiele 14

3 Abgrenzung der Muster **14**

4 Zusammenfassung **15**

5 Quellen **16**

1 Einleitung

Als Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides 1994 ihr Werk „Design Patterns - Elements of Reusable Object-Oriented Software“ zum Thema Muster in Software und Algorithmen veröffentlichten, stellte dies in der Softwareentwicklung und -produktion einen wichtigen Meilenstein dar.[1]
Probleme und Situationen die in Zusammenhang mit den damals üblichen Programmiersprachen immer wieder auftraten erhielten standardisierte Lösungen, die bereits erprobt und ausgereift waren.
Da man Muster aber im Allgemeinen nicht erfinden sondern "lediglich" entdecken kann, besteht der Verdienst der Viererband (engl: Gang of Four) darin, erstmals das vorhandene Wissen gesammelt, kategorisiert und greifbar gemacht zu haben. Die Arbeit(en) der Gang of Four waren also ein wichtiger Schritt in der Kultivierung des Softwareentwicklungprozesses und des Qualitätsmanagements.
Nach dem Vorbild dieses Standardwerkes enstanden schnell weitere Sammlungen von Mustern, insbesondere auch für domänenspezifische Probleme und für bestimmte Sprachen. Dazu zählt z.b. die Reihe "Pattern-Oriented Software Architecture" von Frank Buschmann et al., die die Sammlung der GoF aufgreift und um zahlreiche, zum Teil aber sehr spezifische Muster erweitert.
Der Gedanke auf Standardlösungen zurückzugreifen und so eine Art "Pattern-driven development" zu ermöglichen stößt allerdings bei vielen Entwicklern auch auf Kritik. Gewarnt wird dabei zum Beispiel vor "überdesignten" Programmen, die eine unnötig hohe Komplexität zu Gunsten von vernachlässigbaren Vorteilen aufweisen. Zahlreiche Muster werden mit neuen Ansätzen wie Aspektorientierter Programmierung oder Dynamischen Programmiersprachen in Frage gestellt. Das GoF Buch, dass sogar nach Ansicht von Vlissides wohl eher "Making C++ suck less" hätte heißen sollen, sammelte vorallem Muster, die fehlende Features in Sprachen wie C++ oder Java ausglichen.[2]
Diese Arbeit möchte sich deshalb nun mit einem kleinen Teil von Mustern beschäftigen, um deren Vor und -Nachteile nach heutigen Standards zu diskutieren, ihre Daseinsberechtigung anhand von aktuellen Verwendungen zu prüfen und eventuell um Alternativen zu zeigen.
Dafür werden die Muster zunächst vorgestellt, indem eine kurze Motivation gegeben wird und dann der Aufbau jedes Muster erläutert wird. Anschließend werden jeweils Vor -und Nachteile des Einsatzes des jeweiligen Musters gegeneinander abgewogen und denkbare Anwendunsszenarien geliefert.
Die hier betrachteten Muster Proxy, Decorator und Master-Slave lassen sich dabei grob in die Richtung der "Objektzugriffsmuster" einteilen. Dieser Begriff soll hier stellvertretend für alle Muster stehen, die eine "Zwischenkomponente" für einen Methodenaufruf definieren, ist aber quasi frei erfunden. So etwas wie eine Ähnlichkeitsbeziehung existiert hier insbesondere zwischen Decorator und Proxy Muster, während sich das Master-Slave Muster in Funktion und Aufbau relativ stark von den anderen beiden unterscheidet. Dies ist aber stark von der gewählten Betrach-

[1] siehe `http://www.ddj.com/joltawards/prev_bks.htm#1994`

[2] Auszug aus dem Blog von Neal Ford: `http://memeagora.blogspot.com/2007/09/ruby-matters-design-patterns-in-dynamic.html`

tungsweise und der vorliegenden Einteilung abhängig.

1.1 Kategorisierung der vorliegenden Muster

Wie bereits erwähnt existieren mehrere mögliche Arten der Kategorisierung von Entwurfsmustern. Sicherlich am häufigsten verwendet wird dabei die Einteilung nach [GOF94] in Erzeugungsmuster, Strukturmuster und Verhaltensmuster.[3]
Orthogonal dazu definierte die Viererbande noch die Unterscheidung in klassen und -objektorientierte Muster, je nachdem ob ein Muster statisch auf Klassenstrukturebene arbeitet oder dynamisch zur Laufzeit Objekte manipuliert. Proxy und Decorator fallen hierbei in die Kategorie der "Strukturmuster", die sich mit der strukturellen Anordnung von Klassen und Objekten zueinander beschäftigen und die die Gruppe der objektorientierten Muster, da sie jeweils erst zur Laufzeit an eine bestimmte Komponente gebunden werden.
Das Master-Slave Muster selbst wurde nicht in [GOF94] definiert und dementsprechend nicht in dieses Schema einsortiert. Würde man es versuchen, würde das Master-Slave Muster wohl am ehesten zu den objektorientierten Verhaltensmustern gezählt werden. Diese beschäftigen sich nach Definition "[..] nicht nur [mit] Muster von Objekten und Klassen, sondern auch die Muster der Interaktion zwischen ihnen."[4]
Das Master-Slave Muster findet allerdings Erwähnung in [POSA96] und wird dort wie Proxy und Decorator komplett anders eingeordnet. Frank Buschmann und seine Co-Autoren wählten eine verfeinerte Kategorisierung nach den Funktionen und dem Umfang der Muster.[5]
Der Proxy landet hierbei bei den "Zugriffsmustern", der Decorator in "Servicerweiterungen" und Master-Slave bei den "arbeitsverteilenden" Mustern. Dabei liegen alle drei Muster in der Einteilung als "Entwurfsmuster" zwischen den "Architekturmustern" und den "Idiomen" auf einer Abstraktionsebene.[6]

1.2 Begriffsklärung

Obwohl nun die Einteilung der Muster augenscheinlich wenige Gemeinsamkeiten offenbart hat, liegt allen drei Mustern das selbe abstrakte Prinzip zu Grunde: sie leiten Funktions -oder Serviceaufrufe einer Komponente zu einer anderen weiter und fügen extra Aufgaben zur eigentlich aufgerufenen Funktion aus.
Um einen besseren Vergleich der drei Muster zu ermöglichen, wird deshalb die Klasse oder Komponente, die eine Funktion ausruft jeweils "Client" genannt. Die Klasse oder Komponente, die eine gewünschte Funktion oder einen Service zur Verfügung stellt wird einfach "Component" genannt.
Diese Begriffe sind an die Diagramme in [GOF94] angelehnt[7], diese Arbeit beschränkt sich also ebenso wie dieses Buch auf die Anwendung der Muster auf Probleme in der Softwaretechnik (obwohl ein Muster per Definition keinem festen The-

[3] [GOF94], S.9-11
[4] [GOF94], S.221f.
[5] [POSA96], S.379ff.
[6] Ebd.
[7] siehe z.B. [GOF94], S. 177

mengebiet zugeordnet wird).
Die in dieser Arbeit diskutierten Muster beschäftigen sich nun also grundsätzlich mit Problemen, bei denen ein dirketer Zugriff des Clients auf die Component nicht erwünscht ist, inpraktikabel oder aber unmöglich ist (aufgrund von Sicherheitsbschränkungen etc.) .

2 Die Muster

2.1 Decorator

Das erste hier vorgestellte Muster stammt aus dem Buch der Vierbande [GOF94] und dient der dynamischen Funktionserweiterung eines Objekts zur Programmlaufzeit. Um ein Beispiel zur Motivation zu liefern wird an dieser Stelle ein typisches Problem aufgegriffen.[8]

2.1.1 Problemstellung

Ein Windowmanager System besitzt eine zentrale Klasse (hier Display genannt), die alle Fenster über einen Aufruf der jeweiligen paint() Methode zeichnet. Dabei wird zwischen verschiedensten Fensterarten unterschieden, die aber alle das selbe Interface implementieren. Fenster können verschiedene Rahmen, Scrollbars und Farben haben, die alle bei Aufruf von paint() gezeichnet werden sollen.
Die naive Lösung wäre nun ein Interface "Fenster", von dem alle möglichen Kombinationen von Fenstern in einer Vererbungshierarchie abgeleitet werden. Neben "ScrollbaresFenster" und "FensterMitRahmen" existieren also auch "FenterMitDickenRahmen" und "ScrollbaresFensterMitRahmen" in der Hierachie.
Es ist also ersichtlich, dass dieser Kombinationsansatz zu einer unübersichtlichen Vererbungshierachie führt, die zudem statisch und damit schwer erweiterbar wird. Zudem werden einige dieser Kombi-Klassen womöglich gar nicht während der Laufzeit gebraucht und sind damit total unnötig.[9]

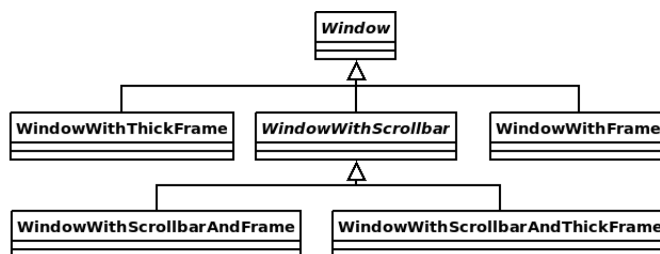

Abbildung 1: Folge des naiven Ansatzes

Wir abstrahieren das Problem nun zunächst einmal um eine möglichst allgemeingültige Lösung zu finden: die Funktionalität eines Objektes soll sich so dynamisch gestalten, dass während der Laufzeit quasi beliebiges Pre- und Postprocessing von Methodenaufrufen möglich wird und nur solche Objekte erweitert werden, bei denen

[8] vgl. [GOF94], S.175f.
[9] Ebd.

es auch wirklich nötig ist.
Dabei soll auf Vererbung weitgehend verzichtet werden, um Probleme wie die oben beschriebenen zu vermeiden und unsere Lösung sollte transparent für den Client sein. Das heißt insbesondere, dass sich am Interface der aufgerufenen Component nichts ändern darf.

2.1.2 Lösung

Die Idee unserer Lösung ist nun relativ einfach. Wir legen einen "Wrapper" um die aufgerufene Component, der alle zusätzlichen Funktionen breitstellt. Funktionsaufrufe des Clients laufen nun erst "durch" diesen Wrapper, der den Aufruf an die eigentliche Component weiterleitet und die gewünschten Zusatzaufgaben ausführt. Zu diesen Zweck erhält der "Wrapper" eine Referenz auf die ursprüngliche Component.
Der "Wrapper", oder Decorator genannt, implementiert darüberhinaus das selbe Interface wie die Component und leitet alle Interface Aufrufe des Clients über die Referenz an die Originalcomponent weiter. Das Klassendiagramm sind demnach folgendermaßen aus[10]:

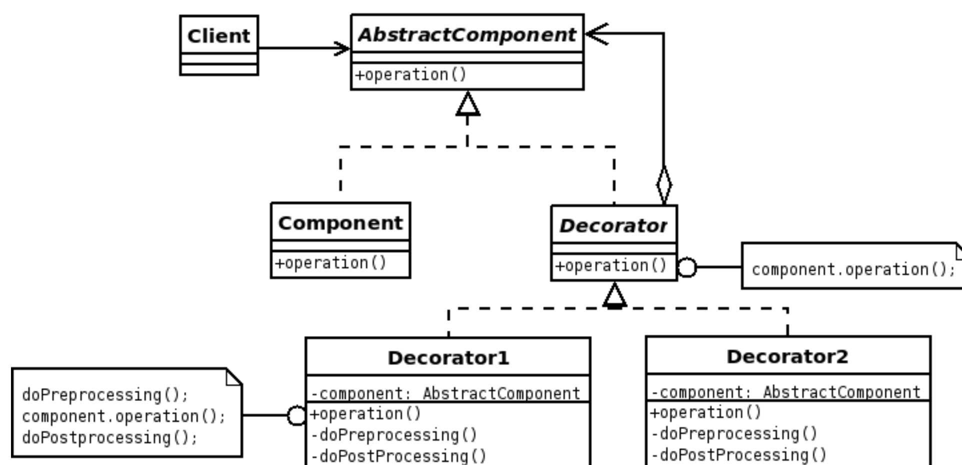

Abbildung 2: Decorator Klassendiagramm

Die eigentlichen Decorator implementieren dabei noch ein zusätzliches Interface "Decorator", während alle konkreten Objekte direkt vom Interface erben. So eine Vererbungshierarchie ermöglicht zum einen mehrere Decorator und zum anderen die Unterscheidung zwischen Decorator und konkreter Componentimplementation.
Ein Methodenaufruf eines Clients läuft nun nach folgenden Schema ab: ein Client schickt einen Methodenaufruf an das Interface der Component. Tatsächlich ruft er dabei die Methode eines Decorators auf, der zunächst eigene, private Methoden ausführt (Preprocessing) und erst dann die Interfacemethode an die eigentlich gewünschten Component weiterleitet. Diese arbeitet den Methodenaufruf ab und der Decorator führt im Anschluß eventuell wieder eigene Funktionen aus (Postprocessing).

[10] vgl. [GOF94], S.177

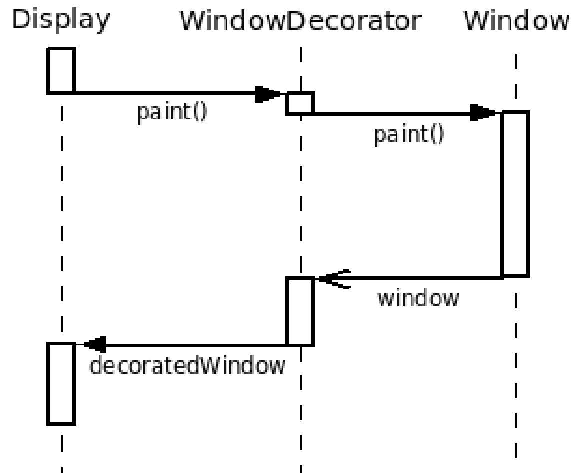

Abbildung 3: Methodenaufruf im Decorator Muster

Diese Lösung ist deshalb gut, weil sie quasi unbegrenzt skalierbar ist. Da sowohl Client als auch Decorator lediglich eine Referenz auf eine Component benötigen[11], lassen sich auch mehrere Decorators ineinander legen. Anschaulich wohl am besten vorstellbar durch mehrere Schichten von Wrappern um das eigentliche Objekt liegt. Jede einzelne "Wrapperschicht" kann durch Pre-und Postprocessing eigene Funktionen aufrufen, während der Aufruf "durch" die einzelnen Schichten zur Component und zurück zum Client geleitet wird.

2.1.3 Vor- und Nachteile

Der größte Vorteil dises "Zwiebelmodells" ist offenbar die wesentlich größere Flexibiltät gegenüber einer statischen Vererbungshierarchie[12]. Selbst nicht vorhergesehene Erweiterungen lassen sich nun fast beliebig mit vorhanden Components mischen. Neben einer beliebigen Verschachtelung unterschiedlicher Decorator ist es nun auch einfach möglich ein Objekt zweimal mit dem selben Decorator zu "schmücken". In unserem Ausgangsbeispiel wäre dies z.B. das Zeichnen eines doppelt so breiten Rahmens durch zweifaches hinzufügen des RahmenDecorators zum Fenster.
Allgemein erlaubt das Decoratorkonzept kleinere Klassen, da Funktionalität, die weniger gebraucht wird, einfach ausgelagert werden kann. Code wird somit wartbarer und die Objekte im allgemeinen kleiner. Auf der anderen Seite führen aber viele kleine Klassen auch zu vielen kleinen potentiellen Fehlerquellen und es kann durchaus schwerer werden, einen Fehler eines Methodenaufrufes nachzuvollziehen, wenn der Aufruf erst durch diverse Objekte weitergeleitet wird und vieleicht gar nicht klar

[11] vgl. [GOF94], S.179ff.
[12] [GOF94], S.178

ist, was die einzelnen Decorator noch für eigene Funktionalität ausführen.
Werden viele kleine Objekte statt eines großen verwendet, führt dies aber auch zu einem erhöhten Speicherbedarf und zu Problemen bei der Objektidentität. Ein Wrapper und die eigentlich Component haben unterschiedliche Objektidentitäten. Da es für einen Client von außen nicht sichtbar ist, ob er es nun mit der originalen Component zu tun hat oder mit einem Decorator, kann er sich nun nicht in jedem Fall auf die Objekt-IDs verlassen um zwischen zwei Components zu vergleichen [13].

2.1.4 Anwendungsbeispiele

Trotz aller Nachteile ist der Decorator (oder "Wrapper") ein weit verbreitetes Muster, dass in dieser oder ähnlicher Form (siehe auch das nächste Muster: Proxy) in diversen Implementierungen für z.B. Java oder C++ auftaucht. Zu den bekanntesten Anwendungsgebieten zählen dabei sicherlich GUI Frameworks wie SWT, die in etwa das selbe Konzept verwenden, das im Beispiel angedeutet wurde.
Zu einer anderen wichtigen Anwendung des Musters in Java gehört auch das komplette I/O Stream Api.[14] Vom Basisinterface Inputstream leiten sich neben den konkreten InputStreamm Klassen auch das Interface FilerInputStream ab. Dieses bildet im Prinzip das Basisdecorator Interface von dem sich solche Klassen wie ByteInputStream, BufferedInputStream etc. ableiten. Diese Klassen wiederum erweitern einen InputStream, dessen Referenz sie im Konstruktor erwarten, dann um bestimmte Funktionen wie puffern und encoding in ein anderes Format.

2.2 Proxy

Im weiteren Sinne fällt auch das Proxy Muster in die Abteilung "Funktionserweiterungen" eines Methodenaufrufes eines Clients zu einer Component. Dabei handelt es sich aber um Funktionen, die außerhalb des eigentlichen Aufgabenbereiches der Component liegen.[15]

2.2.1 Problemstellung

Zu solch einem Aufgabenbereich zählt zum Beispiel die Kommunikation eines Clients mit einer Component über ein Netzwerk, wenn beide Teile auf unterschiedliche Systeme verteilt werden. Würde man hier den offensichtlichsten Weg wählen, müsste man sämtliche Logik, die zur Kommunikation notwendig ist, in den Client und die Component implementieren.
Die Nachteile dieses Vorgehens liegen dabei klar auf der Hand: Veränderungen am Client oder der Component sind potentielle Fehlerquellen, da andere Teile der Software kritisch auf Änderungen im Interface reagieren können. Zudem wird mindestens die Component mit Informationen und Funktionen überladen, die für ihre eigentliche Aufgabe(n) keine Relevanz haben. Zusätzlich dazu führen wiederholte Zugriffe auf eine Component immer wieder die gleichen (kostspieligen) Operationen aus um

[13] Ebd.
[14] `http://java.sun.com/j2se/1.4.2/docs/api/java/io/package-summary.html`
[15] vgl. [GOF94], S.207f.

die Remote-Verbindung herzustellen [16].
Wir möchten also nun ganz allgemeingefasst, dynamisch Funktionalität zu einem Methodenaufruf eines Clients hinzufügen, die so nicht zum Aufgabenbereich der aufgerufenen Component gehört oder nicht gehören kann. Wichtig dabei ist, dass, wie schon beim Decorator, keine Veränderung am Client oder an der Component vorgenommen werden müssen, unser Klassendiagramm übersichtlich bleibt und die Aufrufstruktur möglichst dynamisch zu Laufzeit änderbar ist.

2.2.2 Lösung

Da wir die gewünschten Funktionen nicht *in* die Component packen können, fügen wir sie einfach in eine Zwischenschicht in Form eines zusätzlichen Objektes *vor* der Component ein. Dieses vorgeschaltete Objekt, der "Proxy", fängt dabei wieder die Methodenaufrufe des Clients ab, kümmert sich um Pre -und Postprocessing mit Hilfe zusätzlicher (privater) Methoden und leitet dann den Aufruf gegebenfalls an die vom Client gewünschte Component weiter.
Der Proxy implementiert dabei ebenfalls das selbe Interface wie die eigentliche Component, so dass es für den Client wiederum völlig transparent ist, ob er über das Interface nun die Component oder aber das vorgelagerte Objekt anspricht.[17] Der Proxy ist in der Lage Methodenaufrufe an die Component weiterzuleiten, da er sie als Referenz übergeben bekommt und ist darüber hinaus nicht nur auf eine Component beschränkt sondern kann mehrere Referenzen verwalten [18]. Im Gegensatz zum Decorator muss der Proxy aber nicht zwangsweise den Aufruf an die Component weiterleiten und deren Antwort einholen, wenn er der Meinung ist, dass es nicht notwendig ist.
Dieses Verhalten führt auch zu den alternativen Bezeichnungen "Surrogate" (Stellvertretrer) und "Ambassador" (Botschafter oder Repräsentant) für das Proxy Muster. Das Klassendigramm des Proxymusters siehtnfolgendermaßen aus:

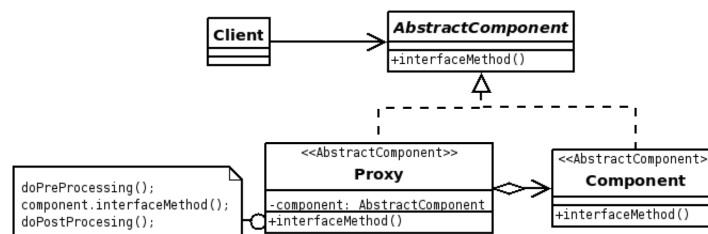

Abbildung 4: Klassendiagramm für das Proxy Muster

und lässt einen weiteren wichtigen Untrschied zum Decorator erkennen. Proxys verlangen nach der Vorstellung von [POSA96] eine Referenz auf eine *konkrete* Componentimplementierung, da sie ohnehin nicht zum Verschachteln vorgesehn sind [19]. Optimalerweise erhält eine Component also maximal einen vorgeschalteten Proxy, der sich um gewünschte Zusatzaufgaben kümmert. Die üblichsten Aufgaben eines

[16] [POSA96], S.263f.
[17] [GOF94], S.209
[18] [POSA96], S. 265
[19] [POSA96], S.266 und [GOF94], S.209

Proxys werden dabei in verschiedenen Kategorien zusammengefasst. Die Gang of Four unterteilte den Proxy deshalb in Remote-Proxy, virtuellen Proxy, Schutzproxy und Smart-Reference [20].
Ein Remote-Proxy übernimmt, wie der Name bereits suggeriert, Aufgaben im Bereich Kommunikation für Komponenten in verschiedenen Adreßräumen. Der virtuelle Proxy überprüft zunächst jeden einkommenden Methodenaufruf und versucht das erstellen und verarbeiten teurer Objektemöglichst zu verzögern, indem er diese erst dann erstellt, wenn es unumgänglich ist. In Zusammenhang mit Sicherheitsfragen eines Objekts steht der Schutzproxy, der zunächst Zugriffsberechtigungen auf das gewünschte Objekt überprüft und dann den Aufruf weiterleitet. Als letztes definiert [GOF94] die Smart-Reference, unter der alle sonstigen Zusatzaufgaben zusammengefasst werden. In "Patter-oriented System Architecture"([POSA96]) wird dieser Oberbegriff allerdings weiter in Cache Proxy, Synchronization Proxy, Counting Proxy und Firewall Proxy unterteilt[21] .
Im Allgemeinen wird (auch weil ja immer nur maximal ein Proxy pro Component verwendet wird) solch ein vorgeschaltetes Objekt mehrere der oben genannten Aufgaben übernehmen. Gängige Kombinationen sind zum Beispiel der Remote-Proxy mit integrierten Sicherheitsabfragen oder der virtuelle Proxy, der bei Bedarf auch teure Objekte und Antworten cacht[22].

2.2.3 Vor- und Nachteile

Ein allgemeiner Vorteil des Proxys ist, ähnlich wie beim Decorator, die Befreiung der Komponenten von Funktionen und Informationen und damit ein "schlankhalten" der resultierenden Objekte. Zusatzfunktionen werden nur hinzugefügt, wenn sie tatsächlich auch gebraucht werden.
Die verschiedenen Proxyarten und -verwendungen haben aber auch entsprechend weitere verschieden Vor-und Nachteile. So halten Remote und -Schutzproxy den Methodenaufruf für einen Client transparent, da für ihn keine Änderungen im Interface der Component erfolgen und somit auch keine änderungen im Aufrufablauf nötig sind. Spätere Änderungen im Kommunikationsweg oder der Sicherheitsüberprüfung bewirken dann auch keine Änderung am Client[23].
Der virtuelle und der Cache Proxy zielen darauf ab, Speicher und Rechenleistung zu sparen, indem sie unnötige und doppelte Operationen vermeiden. Dies gelingt allerdings nur, wenn die angeforderten Objekte tatsächlich gecacht werden können. Müssen sie ohnehin (fast) immer neu erzeugt oder geladen werden, weil sie z.B. immer aktuell sein müssen, bringt ein Proxy an dieser Stelle wenig Vorteile und kann mit seinen zusätzlichen Overhead sogar schlechter als eine naive Lösung sein[24].
Der Aufwand der Indirektion ist also nicht immer nötig oder förderlich, bei einem richtigen Einsatz dieses Muster überwiegen aber die Vorteile.

[20] [GOF94], S.208f.
[21] [POSA96], S.268ff.
[22] [POSA96], S. 272
[23] [POSA96], S.274
[24] [POSA96], S.275

2.2.4 Anwendungsbeispiele

Beispiele in denen Proxys große Vorteile bringen, finden sich dann auch relativ zahlreich. Am häufigsten ist dabei der Proxy sicherlich in seiner typischsten Funktion vertreten: als Kommunikationsendpunkt in einem Netzwerk. Nahezu alle Webservice Frameworks, darunter auch SOAP, RMI und CORBA setzen auf Remote-Proxys[25] um die eigentliche Businesslogik von der Kommunikationslogik zu befreien. Nebenbei liefern diese Proxys (oder "Stubs") auch die Möglichkeit das grundlegende Programmgerüst nur anhand einer Funktionsbeschreibung zu erzeugen.
Im Bereich Cache -und virtuelle Proxys ist sicherlich der Webproxy am bekanntesten. Dieser cacht Internetseiten, respektive HTTP Responses, um unnötigen Netzverkehr zu sparen und schnellere Antwortzeiten zu ermöglichen. Das gleich Prinzip findet man auch in Datenbank(mapping)systemen wie zum Beispiel Hibernate. Hier werden einmal aus der relationalen Datenbank erzeugte Objekte gecacht, um einer ständigen Wiederholung der Prozedur bei Benutzung des selben Objekts entgegenzuwirken. [26] Zudem hilft hier ein virtueller Proxy bei Collections (Listen etc.), unnötige Zugriffe zu sparen, indem die Elemente der Collection erst geladen werden, wenn *direkt* auf sie zugegriffen wird.

2.3 Master-Slave

Das Master-Slave Muster ist weitaus weniger gebräuchlich als die beiden bisher genannten, was sicherlich auch an seinen relativ speziellen Anwendungsgebieten liegt. Im Gegensatz zu Decorator und Proxy war dieses Muster auch nicht Bestandteil des ursprünglichen Werkes von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, sondern wurde erst später als solches entdeckt und beschrieben. Die hier gegebenen Informationen stammen deshalb zum größten Teil aus [POSA96], dass sich im Gegensatz zu [GOF94] auch mit verteilten Systemen beschäftigt.

2.3.1 Problemstellung

Eine typisches Problem für das Master-Slave Muster ist die Ausführung einer komplexen Berechnung in einem System mit mehreren CPUs. (Ob die Kerne dabei alle in einem Rechner stecken oder in einem Netzwerk verteilt sind, ist dabei egal). Solche Berechnungen findet man zum mittlerweile in vielen Breichen der Informatik, als Beispiel dient hier die Bildverarbeitung.
Ein aufwändiges 3D Bild mit mehreren Millionen Polygonen soll in einem solchen System gerendert werden. Würde man den Brechnungsalgorithmus als einen großen Prozess auf einer CPU auf, läge ein Großteil der Systemressourcen brach, während die Berechnung scheinbar ewig dauert. Ein besserer Ansatz wäre es nun, das Bild in einzelne, unabhängige Bereiche aufzuteilen und diese Bereiche von verschiedenen CPUs parallel berechnen zu lassen.
Das eigentlich Problem liegt nun in der Koordination der voneinander uabhängigen Teilaufgaben und Threads. Ein Client möchte einen Methodenaufruf absetzen, der

---

[25] [POSA96], S.273

[26] siehe z.B. `https://www.hibernate.org/hib_docs/v3/api/org/hibernate/proxy/map/MapProxy.html`

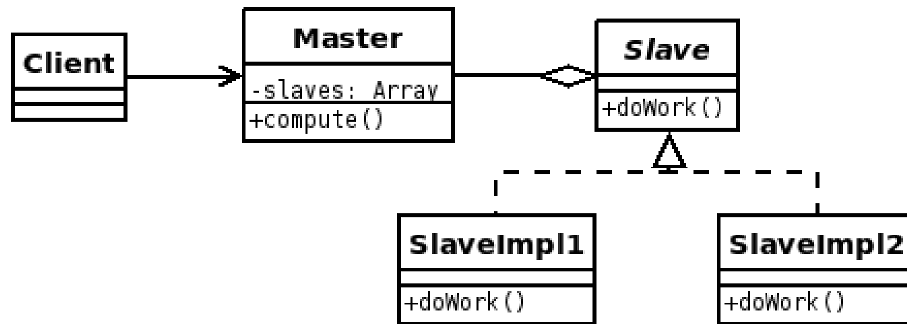

Abbildung 5: Klassendiagramm für das Master-Slave Muster

in mehrere gleichartige Teilaufgaben zerfällt, Zusatzaufgaben wie Koordination und Überwachung der Teilprozesse sind dabei aber nicht sein Aufgabengebiet und deshalb auch aus seinem Code rauszuhalten. Auf der anderen Seite ist aber eine solche zentrale Instanz notwendig, um die einzelnen Threads (oder Prozesse) zu steuern und das Gesamtergebnis zu aggregieren[27].

2.3.2 Lösung

Die zentrale Instanz wird im Master-Slave Muster nun durch den "Master" repräsentiert, die einzelnen Teilaufgaben durch eigenständige "Slaves". Sämtliche Logik zur Vergabe und Verwaltung der Aufgaben an die möglichen Threads und Prozesse wird vom "Master" implementiert. Ebenso läuft sämtliche Kommunikation von und zum Client sowie die Koordination zwischen den Slaves über den Master. Das Klassendiagramm des Master-Slave Muster verdeutlicht dies[28] Zu beachten ist hier, dass in diesem Muster alle Slaves *das selbe* Interface implementieren. Eine typische Anwendung von Nebenläufigkeit, die Verbesserung des Reaktionsverhaltens eines Programmes, lässt sich deshalb so mit dem Master-Slave Muster nicht umsetzen.
Das bevorzugte Anwendungsgebiet dieses Muster liegt nun tatsächlich in der Berechnung identischer Teilaufgaben.[29] Die parallele Berechnung der Teilaufgaben in verschiedenen Threads oder Prozessen beschleunigt die Berechnung des Gesamtergebnisses. Der Master benötigt dabei in der Regel alle Teilergebnisse seiner verwalteten Slaves um das Gesamtergebnis zu berechnen, wobei die Slaves in diesem Fall nicht nur das selbe Interface, sondern auch über eine identische Implementierung verfügen.
Verwendet man unterschiedliche Implementierungen der Slaves, kann man das Master-Slave Muster auch dazu benutzen, die Genauigkeit einer Berechnung zu erhöhen. Ein Master übermittelt dabei an verschiedenartige Slaves ("[..] at least three[..]")[30] die selben Eingabeparameter. Die Teilergebnisse der Slaves sollten nun alle die selben sein, es ergibt sich aber aufgrund der verschiedenen Berechnngen und Rundungsfehler der Soft und -hardware jeweils ein wenig anderes Ergebnis.
Die Aufgabe des Masters besteht hier nun darin, aus einer Reihe von Ungenauig-

[27] vgl. [POSA96], S.246
[28] vgl. [POSA96], S.247f.
[29] [POSA96], S.246
[30] [POSA96],S.256

keiten ein möglichst genaues Gesamtergebnis zu erzielen. Dabei stehen dem Master verschiedene Ansätze wie einfacher Durchschnittsbildung, ein Ergebniszähler oder komplexere statistische und numerische Verfahren zur Verfügung.
Belässt man nun sowohl die Implementierungen der Slaves als auch die einzelnen Eingabeparameter für die Teilberechnungen gleich, sollten alle Slaves auch das gleiche Ergebnis zurückliefern. Dieser auf den ersten Blick sinnlose Ansatz hat aber den Vorteil von (mehrfacher) Redundanz, was vorallem für kritische Berechnungen oder in relativ instabilen Kommunikationsumgebungen ein gewünschtes Ziel sein kann[31].
In diesem Fall liegt die einzige Aufgabe des Masters zur Berechnung des Gesamtergebnis liegt darin, auf das erste, fehlerfreie Ergebnis eines Slaves zu warten, um dieses dann zum Client weiterzuleiten. Das Ergebnis ist also immer möglich, solange nur ein Slave arbeitet, der mit dem Master kommunizieren kann. Neben der Redundanz lässt sich so mitunter auch Zeit gewinnen, da nur die Antwort des schnellsten, korrekt anwortenden Slaves relevant ist.

2.3.3 Vor- und Nachteile

Die größten Nachteile des Master-Slave Musters sind die üblichen Probleme beim Multithreading[32]. Da im ursprünglichen Muster keine Kommunikation zwischen den Slaves vorgesehen ist[33], ist alleine der Master für Synchronization und Locking von Ressourcen zuständig. Außerdem ist er alleine für jegliche Kommunikation verantwortlich, was ihn in jedem der oben genannten Anwendungsfälle zum Flaschenhals macht. Ein Fehlverhalten oder gar ein Ausfall des Masters ist also nicht ohne Weiteres korrigierbar. Dies gilt dann auch in dem Fall, in dem die Slaves redundant angeordnet sind.
Durch die Verwendung mehrere Threads (oder Prozesse) entsteht natürlich auch ein gewisser Overhead, der nicht immer durch die parallele Berechnung der Teilaufgaben kompensiert werden kann. Dies ist natürlich stark davon abhängig ob und wieviel Threads die Hardware oder das Betriebssystem unterstützen. Parallele Berechnung bringt logischerweise nur dann einen Vorteil, wenn auch tatsächlich parallel gerechnet werden kann.
Dies macht Programme mit dem Master-Slave Muster auch schwer auf andere Systeme portierbar, da Thread -und Prozessverwaltung sowie die gültige maximale Zahl parallel laufender Threads (Prozesse) sich von System zu System stark unterscheiden können[34].
Das Master-Slave Muster bringt bei richtiger Anwendung dennoch große Vorteile. Je nach Art der Verwendung sind dies entweder Geschwindigkeitsgewinn, Redundanz oder Genauigkeitserhöhung. Möglich sind natürlich auch Kombinationen, diese sind allerdings nicht Teil des ursprünglichen Musters.

[31] [POSA96], S.255
[32] [POSA96], S. 260
[33] [POSA96], S.257
[34] Ebd.

2.3.4 Anwendungsbeispiele

Trotz einiger Nachteile, der hohen Implementierungskomplexität und den relativ speziellen Anwendungsfällen findet das Master-Slave Muster dennoch Verwendung. Wie bereits in der Eingangsproblemstellung beschrieben sind hier zum Beispiel graphische Berechnungen zu erwähnen, wie sie Programme wie ParaGL[35] oder K-3D[36] ermöglichen. Beide wählen den Ansatz der parallelen Berechnung, um die Last auf mehrere CPUs zu verteilen. ParaGL ist dabei auf Cluster optimiert und erlaubt so die Erzeugung von 3D Bildern in Echtzeit, währed K-3D eher auf verteilte Systeme im Netzwerk spezialisiert ist.
Die Verwendung von parallelen Berechnungen findet mit den heute üblichen Prozessoren aber auch in anderen Bereichen immer mehr Anhang. Zu nennen sind hier komplexe Matrizenberechnungen, wie sie in der Physik oder Chemie üblich sind oder bei NP-harten Problemen, die durch etwa Devide-and-Conquer, Brute-Force oder dynamische Ansätze gelöst werden können.
Im Gegensatz dazu finden sich eher weniger Einsatzgebiete zur Genauigkeitserhöhung oder zur Ausfallprävention. Hier sind vorallem Großrechnersysteme und andere kritische Programme, wie sie in Banken oder Versicherungen Anwendung finden, zu erwähnen. Aber auch im Design von Microprozessoren erreicht man mittels des Master-Slave Muster eine gewisse Redundanz[37].

3 Abgrenzung der Muster

Das Master-Slave Muster, dass in [GOF94] noch nicht erwähnt wurde, weist aufgrund seines Aufbaus und seinem bevorzugten Anwendungsfällen verhältnismäßig wenig Gemeinsamkeiten mit den anderen beiden hier vorgestellten Mustern auf und fällt eher in den Bereich der Concurrency Pattern. Dort verrichten z.B. das Active Object und das Monitor Object ähnliche Aufgaben[38].
Im Gegensatz dazu sind das Decorator und das Proxy Muster nahezu identisch, wie sich schon an ihren jeweiligen Klassendiagrammen klar erkennen lässt. Der einzige relevante Unterschied liegt hier tatsächlich in den anvisierten Verwendungen. Während das Decorator Muster die Funktionalität einer Klasse *erweitert* und sich dazu bildlich um das Interface der Klasse legt ("Wrapper"-Ansatz), ist der Proxy für Funktionen zuständig, die außerhalb des Aufgabenbereichs der verwalteten Klasse(n) liegen. Bildlich gesehen ist dabei der Proxy einfach als Vermittler zwischengeschaltet[39]. Im Gegensatz zum Decorator müssen sich dabei Anfragen des Clients nicht unbedingt in Funktionsaufrufen in der Component niederschlagen, wenn es der Proxy nicht für nötig hält. Auch eine Verschachtelung von mehreren Proxys ist im ursprünglichen Muster nicht vorgesehen, was eine "Kette von Proxys" und das dynamische Hinzufügen mehrerer Proxys zu einem Objetk quasi ausschließt.
Diese Eigenart des Decorators findet sich aber zum Beispiel auch im Kompositum

[35] `http://www.csrl.cs.vt.edu/paragl.html`(18.07.09)
[36] `http://www.k-3d.org/wiki/Main_Page`(18.07.09)
[37] [Feiler], S.8
[38] [POSA07], S.318
[39] vgl. [GOF94], S.219f.

Muster, dass allerdings seine Priorität auf die Objektaggregation legt, während diese beim Decorator auf den Funktionen liegen. Das führt dazu, dass beide Muster häufig gleichzeitig verwendet werden um so einen "Baukasten" an Objekten zu erzeugen[40].
Abgesehen von diesen eher marginalen Unterschieden lassen sich verwandte Muster auch durchaus untereinander austauschen, wie etwa in [GOETZ] anhand von Java und dem Decorator und Proxy Muster beschrieben wird. Dabei ist natürlich vorher abzuwägen inwiefern solch ein Ersatz, auch wenn er zunächste bequemer erscheint, die eigentliche Problemstellung erfüllt. Dieser Ansatz ist auch mit anderen, verwandten Mustern vorstellbar, allen voran der Adapter, der im Gegensatz zu Decorator und Proxy das nach außen sichtbare Interface eines Objektes verändert (worin auch seine hauptsächliche Aufgabe besteht). Eine ähnliche Aufgabe könnte hier ein Schutzproxy erfüllen, der bestimmte Interfacezugriffe sperrt oder umlenkt oder ein erweitertes Interface anbietet.
Da Kombinationen von Mustern im realen Programmiereralltag nichts ungewöhnliches sind, sind aber strenge Abgrenzungen der Muster untereinander wie hier ohnehin eher theoretischer Natur.

4 Zusammenfassung

Die Austauschbarkeit und die Kombinationsmöglichkeiten der Muster sind es auch dann, die es nahezu unmöglich machen ein konkretes Muster als veraltet oder überflüssig zu bezeichnen. Obwohl jedes Muster auch Nachteile aufweist, sind die Vorteile zumindest in den derzeitig am häufigsten verwendeten Sprachen (C++, Java) durch keine anderen Sprachfeatures zu kompensieren.
Aus diesem Grund bieten die genannten Sprachen auch oft bereits Standardimplementierungen einiger Muster (z.B. C++: STL), was die Komplexität und die Nachteile der Verwendung zum Teil weiter senkt.
Darüber lassen sich die hier genannten Nachteile der Muster aber nicht komplett verbergen, so dass der Nutzen immer stark von der Anwendung abhängt und damit von einem wohlüberlegten Programmdesign. Insbesondere der Speicher und -Kommunikationsoverhead einiger Muster lohnt erst ab einer bestimmten Programmgröße -und komplexität.
Eine Programm, dass "pattern-driven" entwickelt wurde, ist deshalb nicht zwangsmäßig auch das effizienteste. Die 3 hier vorgestellten Muster haben aber auch nach nun fast 15 Jahren nicht an Gültigkeit verloren.

[40] Ebd.

5 Quellen

Literatur

[POSA96] Buschmann, F.; Meunier R. et al.
Pattern-Oriented Software Architecture - A System of Patterns
John Wiley & Sons, 1996

[POSA07] Buschmann, F.; Henney, K.; Schmidt, D.C.
Pattern-Oriented Software Architecture
A Pattern Language for Distributed Computing
John Wiley & Sons, 2007

[GOF94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.
Design Patterns
Elements of Reusable Object-Oriented Software
Addison-Wesley, 1994

[Feiler] Feiler, Peter H.;Gluch, David P.; Hudak, John J.;Lewis, Bruce A.
Pattern-Based Analysis of an Embedded Real-time System Architecture

[GEARY] David Geary
http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-designpatterns.html? (09.06.2009)

[GOETZ] Brian Goetz
http://www.ibm.com/developerworks/java/library/j-jtp08305.html
(13.07.2009)