Entwurfsmuster - von Kopf bis Fuß

Zusammenfassung

Inhaltsverzeichnis

1 Entwurfsprinzipien

Essenz jedes Musters kurz zusammengefasst.

Entwurfsprinzipien

- Identifizieren Sie die Aspekte Ihrer Anwendung, die sich aendern koennen und trennen Sie sie von denen, die konstant bleiben.
- Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.
- Ziehen Sie die Komposition der Vererbung vor.
- Streben Sie bei Entwürfen mit interargierenden Objekten nach lockerer Kopplung (einfachere Erweiterung moeglich).
- Klassen sollten fuer Erweiterung offen, aber fuer Veraenderung geschlossen sein. (Mit Vorsicht zu betrachten, dies fuehrt zu hoeherer Komplexität und sollte nur in sinnvollen Faellen angewendet werden.)
- Stützen Sie sich auf Abstraktionen. Stützen Sie sich nicht auf konkrete Klassen.
- Sprechen Sie nur mit Freunden.
- Versuchen Sie nicht, was anzurufen, wir rufen Sie an.
- Eine Klasse sollte nur einen Grund haben, sich zu ändern.

2 Strategy-Muster

2.1 Problemstellung

- Es soll eine Simulation für ein Spiel mit diversen Entenarten geschrieben werden.
- Entenarten besitzen verschiedene Eigenschaften.
- In manchen Fällen nicht nur verschiedene Ausprägungen, sondern sogar nicht existent.
- Verschiedene Arten sollen aber alle von der selben Klasse erben.

2.2 Lösung

Problem hierbei (Lsg.): Wenn dies direkt in die Klasse geschrieben wird ist diese sehr schlecht wartbar, denn die Klassen selbst sind sehr schlecht wartbar. Die Lösung hierbei, das Strategy-Muster. Man versucht ähnliche Algorithmen wie z.B. das Flug- oder Quakverhalten in eine Gruppe/Familie von Algorithmen zusammenzufassen (zu kapseln). Dies geschieht mithilfe verschiedener Interfaces z.B. Quakverhalten / Flugverhalten. Die Mutterklasse besitzt entsprechende Felder von den Typen Quak-/Flugverhalten. Diese werden entsprechend gesetzt und in den jeweiligen Methoden aufgerufen. Die aufrufenden erbenden Klassen implementieren das entsprechende Interface und können so das gewünschte Verhalten implementieren. Hierbei gestaltet sich die Wartung des Codes deutlich einfacher.

2.3 Erklärung des Musters

Definition Das Strategy-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients die ihn einsetzen, variieren zu lassen.

Vorteile Bei diesem Entwurf können andere Typen von Objekten unsere Flug- und Quakverhalten wieder verwenden (Bezug zu Kap. 1), weil diese Verhalten nicht mehr in unseren Ente-Klassen verborgen sind. Wir können neue Verhalten hinzufügen, ohne irgendeine unserer bestehenden Verhaltensklassen zu ändern oder Hand an eine der Enten-Klassen zu legen, die Flugverhalten nutzen.

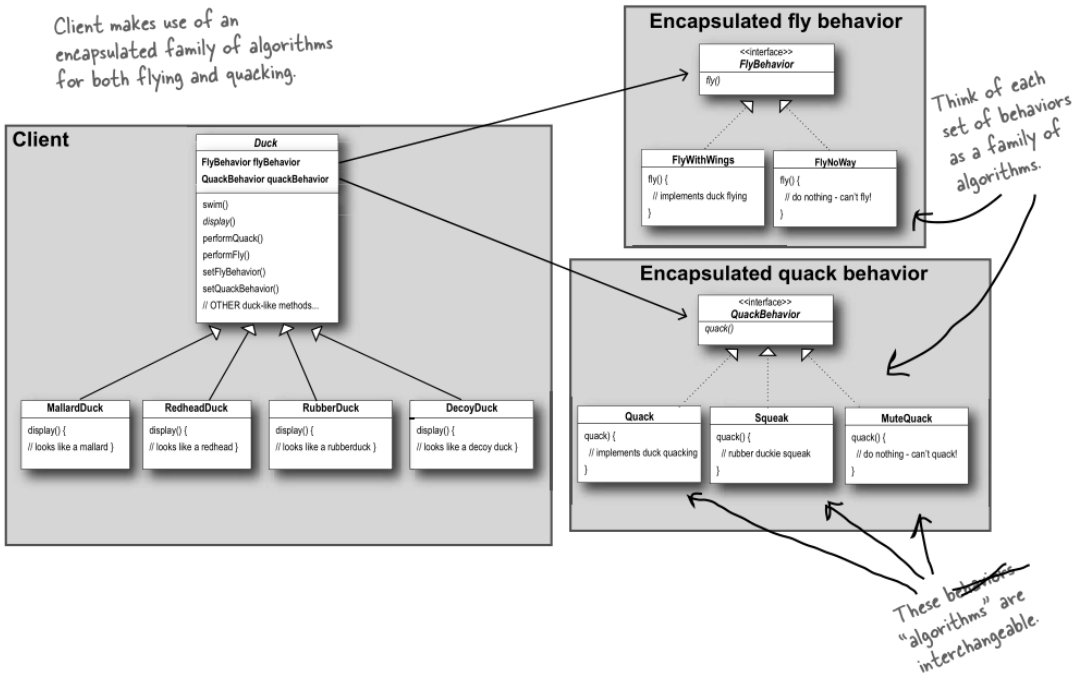

Abbildung 1: UML-Darstellung des Strategy-Musters

3 Decorator-Muster

3.1 Problemstellung

- Ein Bäcker möchte verschiedene Kaffeesorten verkaufen.
- Es soll möglich sein verschiedene Bonuszutaten (z.B. Sahne, Zucker, etc.) hinzuzufügen.
- Hierbei kann die Menge der variieren. Es ist also möglich sowohl eine Extra-Portion, als auch zwei extra Portionen Sahne dazu zu bestellen.
- Ausserdem ist es möglich verschiedene Zutaten zu kombinieren.
- Problem: Wenn man sich auf einfache Vererbung beschränkt wird die Stuktur sehr schnell unübersichtlich.

3.2 Lösung

Es wird zu allererst eine abstrakte Klasse Getränk gebildet, von der sowohl die Zutaten als auch die Kaffeesorten erben. Die Kaffesorten tun dies direkt. Sie besitzen zwei Felder für die Beschreibung und den Preis, über entsprechende Getter sind sie abrufbar. Nun bildet man eine abstrakte Wrapper-Klasse von der sämtliche Zutaten erben sollen. Jede Zutat erbt von der Wrapper Klasse und muss die Getter für die Beschreibung / Preis neu implementieren, denn jede Zutat hält eine Referenz auf ein Getränk welches im Konstruktor gesetzt wird. Bei den Gettern wird die Beschreibung / Preis an die Werte des Getränks angehängt bzw. drauf addiert. Da alle Klassen von der selben Superklasse ableiten kann die Referenz nicht nur einen Kaffee ohne Zutaten sondern auch einen mit diesen beinhalten. Die Wrapper-Klasse ist letztendlich dazu da eine Schnittstelle herzustellen, damit es sowohl geordnet ist und die Referenz wie erwähnt bereits modifizierte Objekte halten kann.

Beim Bestellen wird eine zugrunde liegende Kaffesorte gewählt (z.B. Espresso). Diese wird mit den einzelnen Zutaten "dekoriert"(Bsp. siehe main Klasse im Code).

3.3 Erklärung des Musters

Definition Decorator - Fügt einem Objekt dynamisch zusätzliche Verantwortlichkeit hinzu. Dekorierer bieten eine flexible Alterntive zur Ableitung von Unterklassen zum Zweck der Erweiterung der Funktionalität.

Punkt für Punkt (S.105)

- Vererbung ist eine Form von Erweiterung, aber nicht notwendigerweise der beste Weg, um Ihren Entwürfen Flexiblität zu verleihen.
- Unsere Entwürfe sollen die Erweiterung von Verhalten ermoeglichen, ohna dass dazu bestehnder Code geändert werden müsste.

- Oft keonnen Komposition und Delegierung verwendet werden, um zur Laufzeit neue Verhalten hinzuzufügen.
- Für die Erweiterung von Verhalten bietet das Decorator-Muster eine Alternative zur Ableitung von Unterklassen.
- Das Decorator-Muster schliesst einen Satz von Dekorierer-Klassen ein, die verwendet werden, um konkrete Komponenten einzupacken.
- Dekorierer-Klassen spiegeln den Typ der Komponente wider, die sie dekorieren. (Sie haben sogar tatsächlich den gleichen Typ wie die Komponente, die sie dekorieren, entweder durch Vererbung oder durch die Implementerung eines Interface.)
- Dekorierer ändern das Verhalten der Komponenten, indem sie vor und / oder nach (oder auch an Stellen von) Methodenaufrufen auf der Komponente neue Funktionalitäten hinzufügen.
- Sie koennen eine Komponente mit einer bliebigen Zahl vn Dekorierern einpacken.
- Dekorierer sind für die Clients der Komponente üblichweise transparent, ausser wenn sich der Client auf den konkreten Typ der Komponente stützt.
- Dekorierer koennen in Ihren Entwürfen zu vielen kleinen Objekten fürhen, und eine übermässige Verwendung kann den unübersichtlich machen.

Wann das Decorator-Muster ungeeignet ist

- Wenn Sie mit Code arbeiten, der auf den Typ einer konkreten Komponente angewiesen ist, zerbrechen Dekorierer diesen Code. Solange Sie nur Code auf Basis des abstrakten Komponententyps schreiben, bleibt die Verwendung von Dekorierern für ihren Code transparent. (Muster z.B. für Rabatsystem beim Kaffeehaus aus diesem Kapitel ungeeignet.)
- Dekorierer sollen den Objekten, die sie einpacken Verhalten hinzufügen. Wenn Sie begninnen, auf mehreren Schichten in der Dekoriererkette zu blicken, dann strecken Sie Decorator über seinen eigentlichen Zweck.

Gut zu wissen

- Die java.io-Klassen verwenden ebenfalls das Decorator-Muster. Es gibt diverse Streams mit jeweils unterschiedlichen Aufgaben, und es gibt die Klasse FilterInputStream die als Superklasse für verschiedene Dekorierer arbeitet.

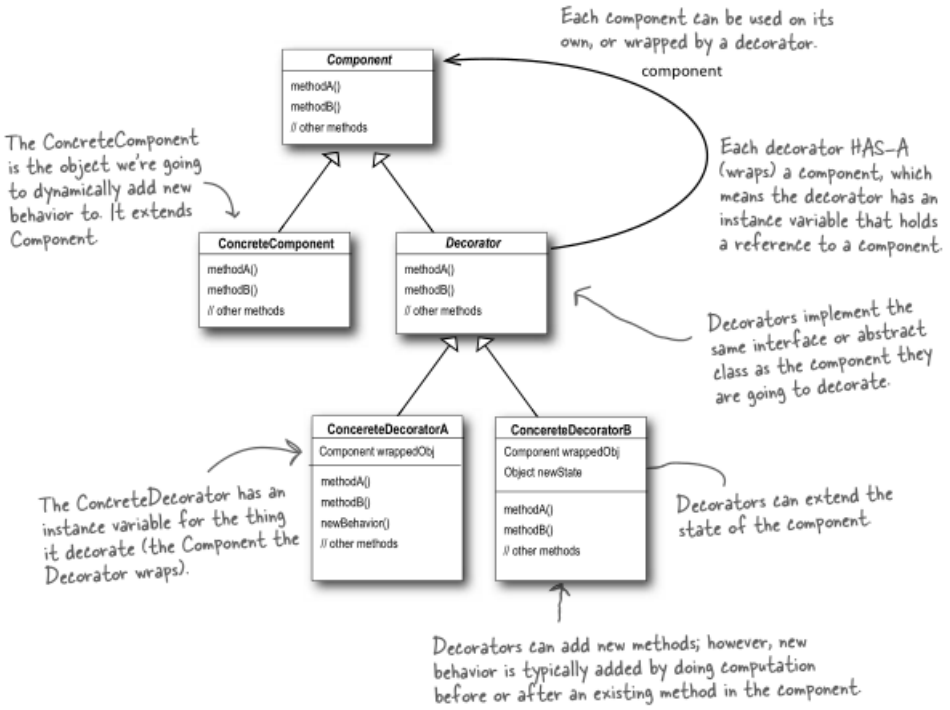

Abbildung 2: UML-Darstellung des Decorator-Musters

4 Observer-Muster

4.1 Problemstellung

- Wetterstation liefert Daten (Luftdruck, Temperatur, etc.).
- Gefragt ist ein WetterDaten-Objekt.
- Diverse Anzeigegeräte sollen in der Lage sein, sich die Daten vom WetterDaten-Objekt zu ziehen und entsprechend darzustellen. Problematisch wird es, wenn verschiedene Anzeigegeräte nur bestimmte Daten anzeigen sollen.
- Besonderes Augenmerk bei dieser Aufgabe, die Daten sollen mit einem einzigen Aufruf aktualisiert werden.
- Wichtig: Es soll möglich sein neue Anzeigeräte einzubinden. Hierbei soll der zus. Aufwand so gering wie möglich gehalten werden.

4.2 Lösung

Muster (Grundsätzliches Prinzip): Man definiert ein Interface z.B. »Subjekt« das die Schnittstelle für das WetterDaten-Objekt angibt. Hierzu gehören die Methoden um einen Beobachter zu registrieren, zu entfernen oder diesen über die Aktualisierung der Daten zu informieren (bzw. ihm diese zukommen zu lassen). Die Idee beim Observer-Muster ist nämlich, dass ein Datenobjekt mit entsprechenden Feldern ein Attribut mit einer Liste von Beobachtern hält die Zugriff auf die Felder besitzen. Dies kann mithilfe einer Übergabe des Objektes geschehen oder aber mit einer tatsächlichen Übergabe von Parametern. Mit den Methoden zum Registrieren und Entfernen werden die Beobachter der Liste hinzugefügt oder entfernt. Das WetterDaten-Objekt aus dem Bsp. implementiert jetzt dieses Interface. Anschliessend wird ein zweites Interface für die Beobachter (im Bsp. die verschiedenen Anzeige-Klassen) geschrieben. Die Beobachter informieren dieses jeweils und haben damit Zugriff auf eine Methode zum Aktualisieren der Werte. Hier können direkt die Parameter übergeben werden, oder aber das Datenobjekt selbst (um mit Gettern die Daten zu übermitteln).

Muster aus java.util.Observable / Observer Es gibt bereits ein vorgefertigtes Observer-Muster in der Standardbibliothek *util*. Hier erbt das jeweilige Datenobjekt von der Superklasse Observable. Diese Klasse bringt bereits die Liste aus Beobachtern und weitere Methoden wie die setChanged() und notfyObservers() Methoden. Wichtig hierbei ist zu beachten, dass man erst die Beobachter benachrichtigen kann wenn setChanged aktiviert wurde. NotifyObservers() ist ausserdem überladen, denn es ist ist möglich neben dem eigentlichen Datenobjekt (hier WetterDaten-Objekt) noch ein anderes zu übergeben,

welches z.B. weitere Werte enthalten kann. Zum genauen Aufbau siehe Code.

Nachteile der Standardbibliothek

- Reihenfolge der Auswertung ändert sich -> Ungeeignet für Anwendungen wo dies von Bedeutung sein sollte.
- Observable ist eine Klasse und kein Interface -> Klasse kann daher keine andere Klasse erweitern und schlecht wartbar sowie wiederverwendbar.
- Observable schützt entscheidende Methoden -> z.B. setChanged(), kann daher auch nur von erbenden Klassen aufgerufen werden.

4.3 Erklärung des Musters

Definition Das Observer-Muster definiert eine Eins-zu-viele-Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts ändert.

Gutes Alltagsbeispiel Zeitungsabonnementsdienst

- Man möchte Abonnent werden -> man wird auf die Liste der Abonennten gesetzt.
- Neue Ausgabe wird ausgegeben (aktualisiert) -> Alle Abonnementen der Liste werden benachrichtigt.
- Bei der Standardbibliothek *utils* ist bereits ein solches Muster vorgegeben mit dem es für die Abonennten sogar möglich ist sich jederzeit per Getter-Methoden die Daten zu ziehen ohne das eine entsprechende Methode im Subjekt aufgerufen werden muss.
- Auf Wunsch ist es ebenfalls möglich wieder aus der Liste der Abonennten auszutreten.

Punkt für Punkt Zusammenfassung (S. 74)

- Das Observer-Musster definiert ein Eins-zu-viele-Verhältnis zwischen Objekten.
- Subjekte oder, wie wir sie auch kennen, Observables aktualisieren Beobachter über eine Schnittstelle.
- Die Beobachter sind insofern locker angebunden, als das Observable über sie nichts anderes weiß, als dass sie das Interface Observer implementiern.
- Sie können Daten aus dem Observable herausgeben oder herausziehen, wenn Sie das Muster verwenden (wobei das Herausziehen als die »richtigere« Methode betrachtet wird).

- Verlassen Sie sich nicht auf eine bestimmte Reihenfolge der Benachrichtigung Ihrer Beobachter.
- Java besitzt eine Reihe von Implementierungen des Observer-Musters, einschließlich des allgemeinen java.util.Observable.
- Nehmen Sie sich vor den Haken der Implementierung von java.util.Observable in Acht. (Siehe Erklärung Kap. 1).
- Haben Sie keine Hemmungen, Ihre eigene Observable-Implementierung zu schreiben, wenn dies erforderlich ist.
- Swing macht wie andere GUI-Frameworks extensiven Gebrauch vom Observer-Muster.
- Sie finden das Muster auch an vielen anderen Orteien einschliesslich JavaBeans und RMI.

4.4 Aufgabe zu den Entwurfsprinzipien

Aufgabe zu den Entwurfsprinzipien Beschreiben Sie für jedes Entwurfsprinzip, wie das Observer-Muster das Prinzip umsetzt.

Prinzipien

1. Identifizieren Sie die Aspekte Ihrer Anwendung, die sich ändern können und trennen Sie sie von denen, die konstant bleiben.
2. Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.
3. Ziehen Sie die Komposition der Vererbung vor.

Eigene Erklärung

1. Beobachter können sich ändern, werden daher zusammengefasst und ausgelagert.
2. Da die Beobachter lediglich ein Interface implementieren und keine Klasse erweitern, besteht hier eine lose Bindung d.h. man Programmiert auf eine Schnittstelle.
3. Die beiden Parteien (Beobachter, Subjekt) implementieren jeweils nur ein Interface und erweitern keine Superklasse.

Erklärungen aus dem Buch (S. 77)

1. Das, was beim Observer-Muster variiert, ist der Zustand des Objekts und die Anzahl sowie die Typen der Beobachter. Mit diesem Muster können Sie die Objekte variieren, die vom Zustand des Objekts abhängig sind, ohne das Subjekt verändern zu müssen. Das nennt man verausschauend handeln!

2. Subjekt und Beobachter nutzen beide Interfaces. Das Subjekt hält Objekte nach, die das Interface Observer implementieren, während die Beobachter sich registrieren und vom Subjekt-Interface benachrichtigt werden. Wie wir gesehen haben, hält das die Dinge ordentlich und locker gebunden.

3. Das Observer-Muster nutzt Komposition um eine beliebige Anzahl von Beobachtern mit ihren Subjekten zu verbinden. Diese Beziehungen werden nicht durch irgendeine Art von Vererbungshierarchie implementiert. Nein sie werden zur Laufzeit durch Komposition eingerichtet!

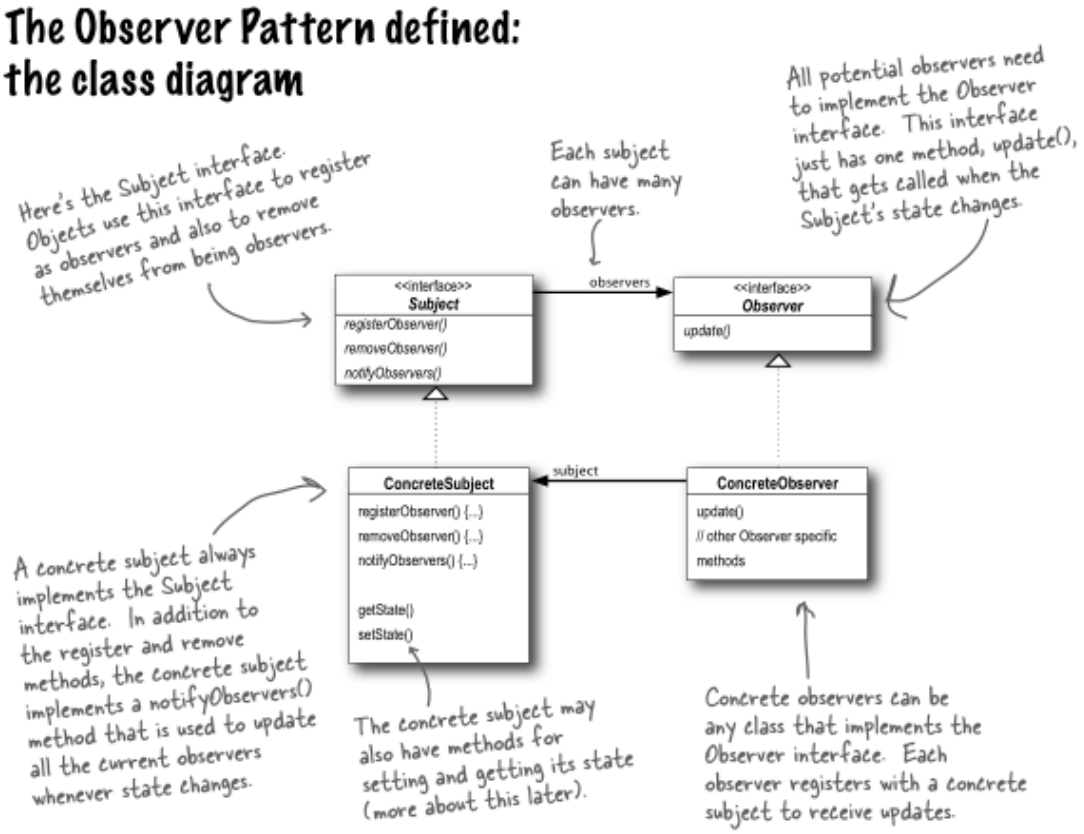

Abbildung 3: UML-Darstellung des Observer-Musters

5 Singleton-Muster

5.1 Definition

Das Singleton-Muster sicher, dass es nur eine Instanz einer Klasse gibt, und bietet einen globalen Zugriffspunkt fuer diese Instanz.

5.2 Erklaerung

- Wir nehmen eine Klasse und lassen sie eine einzige Instanz von sich selbst verwalten. Wir verhindern auch, dass irgendeine andere Klasse eigenstaendig eine neue Instanz erstellt. Um eine Instanz zu erhalten, muss man ueber die Klasse selbst gehen.
- Wir bieten ausserdem einen globalen Zugriffspunkt fuer die Instanz: Jedes Mal wenn Sie eine Instanz benoetigen, fragen Sie einfach be der Klasse nach, und diese reicht Ihnen, die eine Instanz. Wie Sie gesehen haben, koennen wir das so implementieren, dass das Singleton verzoegert erstellt werden kann, was bei ressourcenintensiven Objekten besonders wichtig ist.

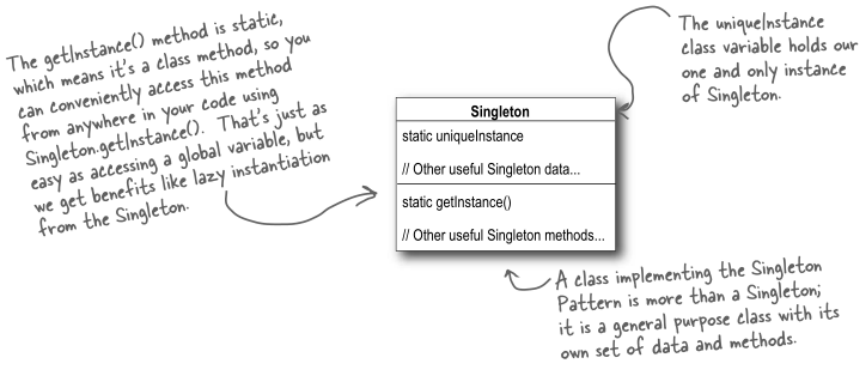

Abbildung 4: UML-Darstellung des *einfachen* Singleton-Musters

6 Factory-Muster

6.1 Problemstellung

Einfach

- Man moechte eine Pizzeria mit mehreren Zweigstellen erstellen.
- Bei der Zubereitung der Pizza gibt diverse Arbeitsablaeufe die von den Zweigstellen allesamt unterschiedliche abgearbeitet werden koennen (Z.B. Schneiden: Vierteln vs. in acht Stuecke schneiden, etc.).
- ‘ Problem: Um Code-Dopplung zu vermeiden sollen die Pizza in einer einzigen Klasse implementiert sein -> Es muss ein Pizza-Objekt entsprechend den Anforderungen (welche Zweigstelle?) erstellt werden.

6.2 Loesung

Es wird eine abstrakte Superklasse *Pizzeria* erstellt. Verschiedene Zweigstellen beerben diese. Jede dieser Zweigstellen implementiert eine Methode *erstellePizza()*. Diese dient als so genannte Factory, denn sie erstellt ein Objekt des gewünschten Typs. Bei der Erstellung wird ein entsprechender Konstruktor einer speziellen Pizza-Klasse aufgerufen. Diese Klasse implementiert die Methoden der Arbeitsschritte (backen, schneiden, etc.) entsprechend. In der Superklasse wird eine Methode deklariert ("bestellePizza()") die mit diesem Objekt arbeitet. Durch den Polymorphismus / Abstraktion ist es irrelevant mit welchem Pizza-Objekt man arbeitet, da alle erbenden Klassen des Pizza-Typs die dort aufgerufenen Methoden implementiert / geerbt haben. Die Methode "bestellePizza()"wird auch fabrikMethode genannt.

6.3 Problemstellung: Erweitert

- Einzelne Zweigstellen benutzen minderwertige Zutaten.
- Wie kann man man Konsistenz bei den Zutaten sichern?

6.4 Loesung

noch unbearbeitet

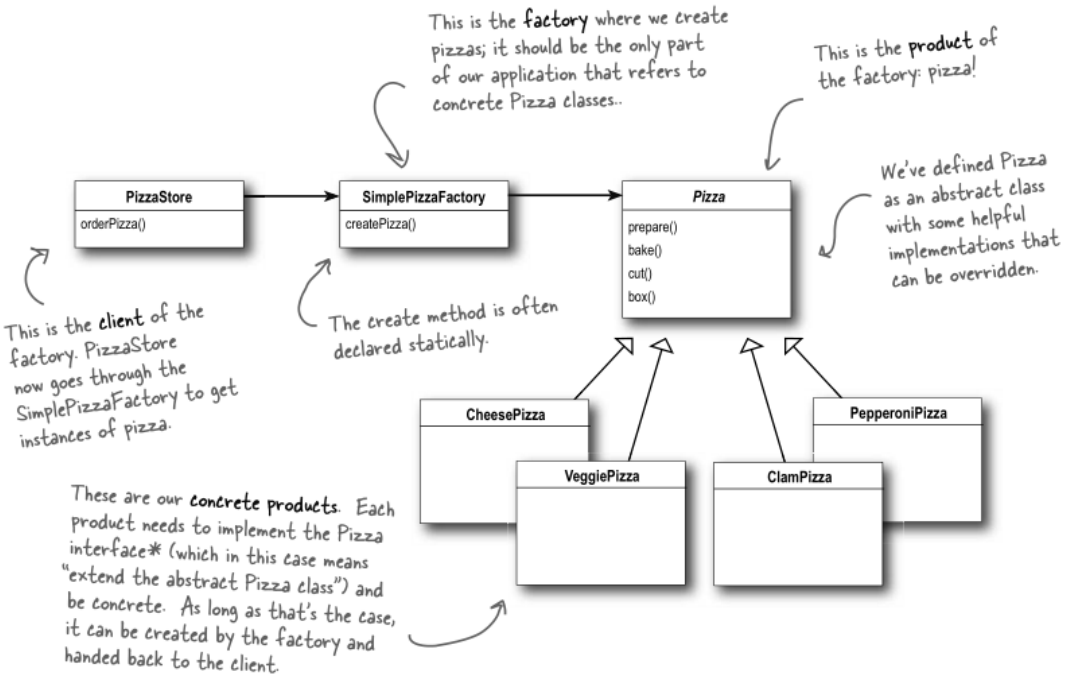

Abbildung 5: UML-Darstellung des *einfachen* Factory-Musters

7 Command-Muster

7.1 Problemstellung

Es soll eine Universalfernbedienung entworfen werden. Hierbei können verschiedene Haushaltsgeräte angeschlossen werden. Die Fernbedienung soll für jedes angeschlossene Gerät einen An- und Ausschalter zur Verfügung stellen. Problem hierbei. Diverse Gerätschaften besitzen haben unterschiedliche Schnittstellen. Bei manchen heißen diese schlicht und ergreifend anders als gefordert, bei anderen gibt es eine derart schlichte Kontrollfunktion nicht. Ein Beispiel wäre hier zum Beispiel ein Garagentor welches auf und zu gemacht werden kann aber auch eine Stereoanlage soll nicht nur angeschaltet werden, es soll auch ein Lied mit einer bestimmten Lautstärke gespielt werden.

Außerdem soll es möglich sein eine gerade angefragte Aktion rückgängig zu machen.

Lösung Anfragen werden per Schnittstelle gekapselt. Jede Art von Gerät API welche an die Fernbedienung angeschlossen werden soll muss von einer Klasse welche das Command-Interface für eine Art von Gerät aufgenommen werden können. Diese Command-Klasse birgt allerdings nur die Methode(n) zum tatsächlichen Ausführen der Kommandos. Diese Methode kann anschließend von der eigentlichen Fernbedienung aufgerufen werden. Eine Fernbedienung besitzt nun irgendeine Art von Collection um diversen Slots die gerade beschriebenen Kommando-Objekte zuweisen zu können.

Um Slots zu sperren welche keine Belegung besitzen kann man entweder eine Prüfung auf die Nullreferenz einbauen oder ein Default-Kommando implementieren was nichts tut und somit auch keine Nullpointerexception auslöst.

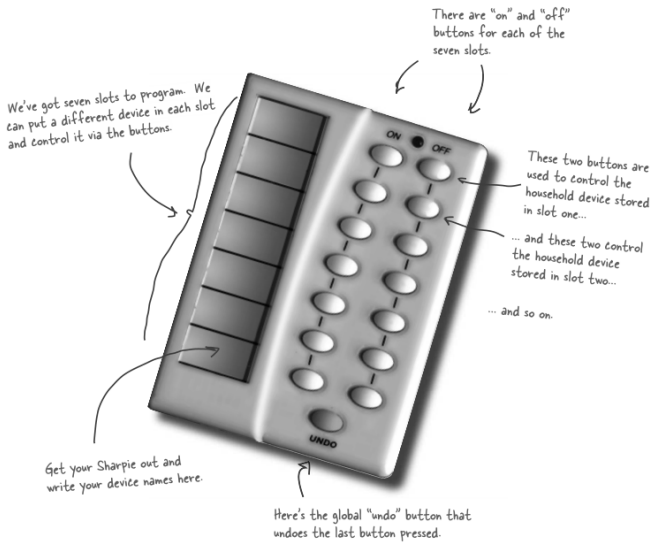

Abbildung 6: UML-Darstellung der zu implementierenden Fernsteuerung

Undo-Fkt. Die einfachste Art dies umzusetzen ist jeweils zwei getrennt Command-Objekt für das An-/Ausschalten von Geräten zu implementieren. Wenn man nun in der Fernbedienung eine Instanzvariable mit dem jeweils zuletzt ausgeführten Befehl setzt. Damit ein Aufruf von undo funktionieren kann muss die Commandschnittstelle jeweils diese Funktion implementieren. Um mehrere Undo's hintereinander auszuführen reicht ein rekursiver Aufruf, da die *gemerkte* Instanzvariable selbst ein Command-Objekt ist und somit ebenfalls einen Verlauf besitzt.

7.2 Erklärung des Musters

Definition Das Command-Muster kapselt eine Anfrage als ein Objekt, dies ermöglicht polymorphes Verhalten mit unterschiedlichen Anfragen, Warteschlangen oder Log-Anfragen. Mit diesem Muster wird außerdem eine Undo-Option ermöglicht.

Beschreibung Ein Command-Objekt kapselt einzelne Anfragen indem ein Interface nach spezifischen Gruppen implementiert wird. Dieses Interface bietet eine wie bereits erwähnt eine Methode *execute*. Diese Methode arbeitet mit einer im Konstruktor übergebenen Instanzvariable. Da es eine klare Trennung zwischen dem Erstellen der Command-Objekte und dem Ausführen gibt, gibt es hierbei viele Freiheiten. Es ist bspw. möglich erst mehrere Befehle zu sammeln bevor sie nacheinander (oder gar gleichzeitig) ausgeführt werden (siehe Meta Command Muster).

Macro-Commands Um mehrere Command-Objekte über einen Slot in der Fernbedienung auszuführe, wird kann man einfach eine Instanzvariable mit einer Collection von Commando-Objekten welche ausgeführt werden sollen implementieren. In der *execute*-Methode werden diese dann alle nacheinander ausgeführt.

7.3 Vorteile

- Die Bearbeitung der Aufgaben ist komplett gekapselt
- Ideal für parallele Verarbeitung
- Einfach Erweiterbar, z.B. Queue welche von mehreren Threads abgearbeitet wird.
- Möglich einfach logging und load Methode in die Schnittstelle aufzunehmen. Dies ermöglicht es im Fehlerfall einen Datenverlust zu vehindern.

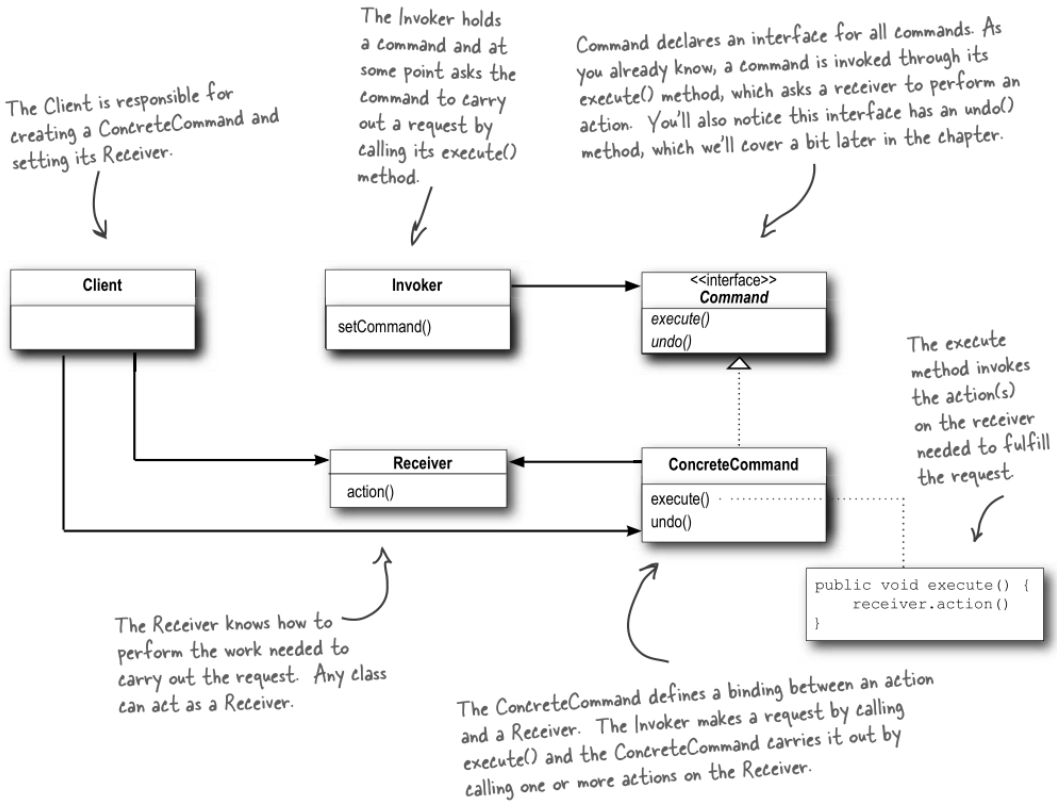

Abbildung 7: UML-Darstellung der zu implementierenden Fernsteuerung

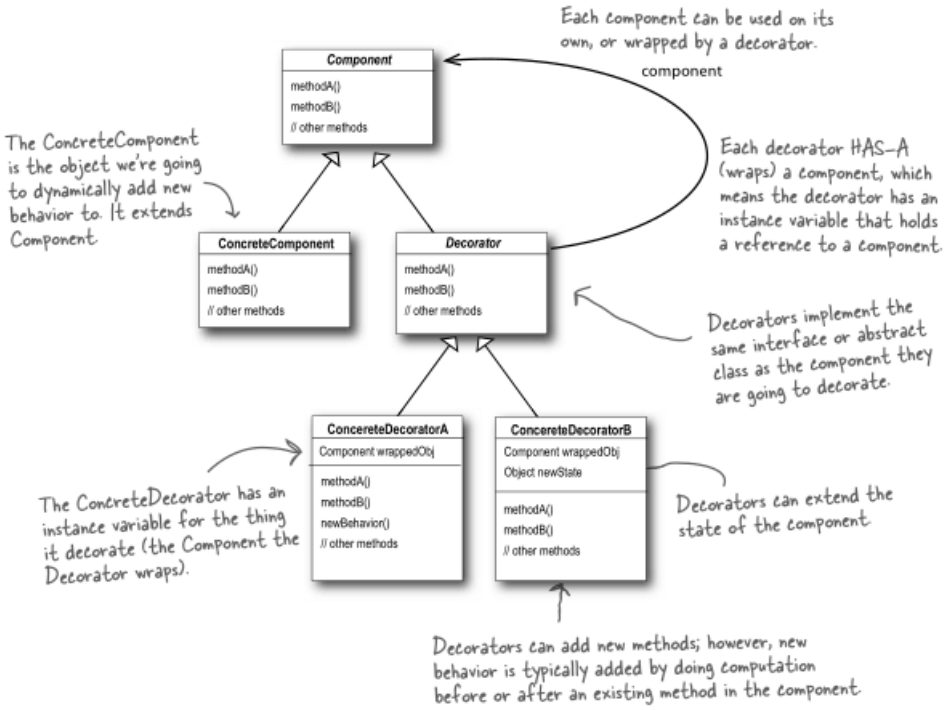

Abbildung 8: UML-Darstellung der zu implementierenden Fernsteuerung

8 Adapter-Muster

8.1 Problemstellung

Man möchte ein ähnliches Objekt in gleicher Weise verarbeiten. Wie man es mit einer geg. Objektinstanz tun würde.

Beispiel aus dem Buch Man möchte eine Truthahn-Instanz in gleicher Weise wie eine Enten-Instanz verwalten. Beide sind in der Lage Laute von sich zu geben und (zumindest kurze Strecken) zu fliegen. Ein Adapter-Interface ermöglicht dies. Der Klient arbeitet nun mit diesem Adapter welcher von dem Enten-Interface erbt. Das Interface tut nun nichts anderes als entsprechenden Methoden einer Instanzvariablen aufzurufen, er dient in gewisser Weise als Übersetzer.

8.2 Erklärung des Musters

Das Adapter Muster konvertiert ein geg. Interface einer Klasse in eins welches vom Klienten erwartet wird. Adapter lassen verschiedene Klassen zusammenarbeiten welche dies sonst nicht ohne weiteres könnten.

8.3 Sonderfall: Facade-Muster

Problemstellung Klient besitzt neues Heimkino. Alle Projektoren, Hifi und sonstige Anlagen jedes Mal *per Hand* zu starten / beenden ist nervig. Das Facade-Muster versteckt diese low-level Zugriffe hinter einem Interface.

Vorteile: Teilweises Entkoppeln des Klients von Low-level-Komponenten (falls gewünscht besitzt der Klient weiterhin Zugriff auf die Komponenten). Vereinfacht die Nut-

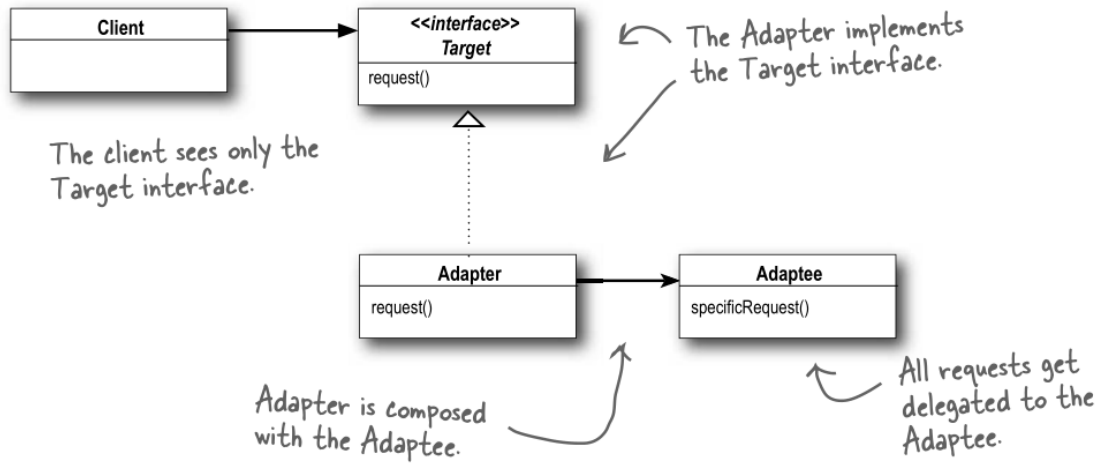

Abbildung 9: UML-Darstellung des Adapter-Musters

zung der Komponenten stark und es ist sogar möglich mehrere verschiedene Interfaces für verschiedene Anwendungszwecke zu bauen (hohe Flexibilität).

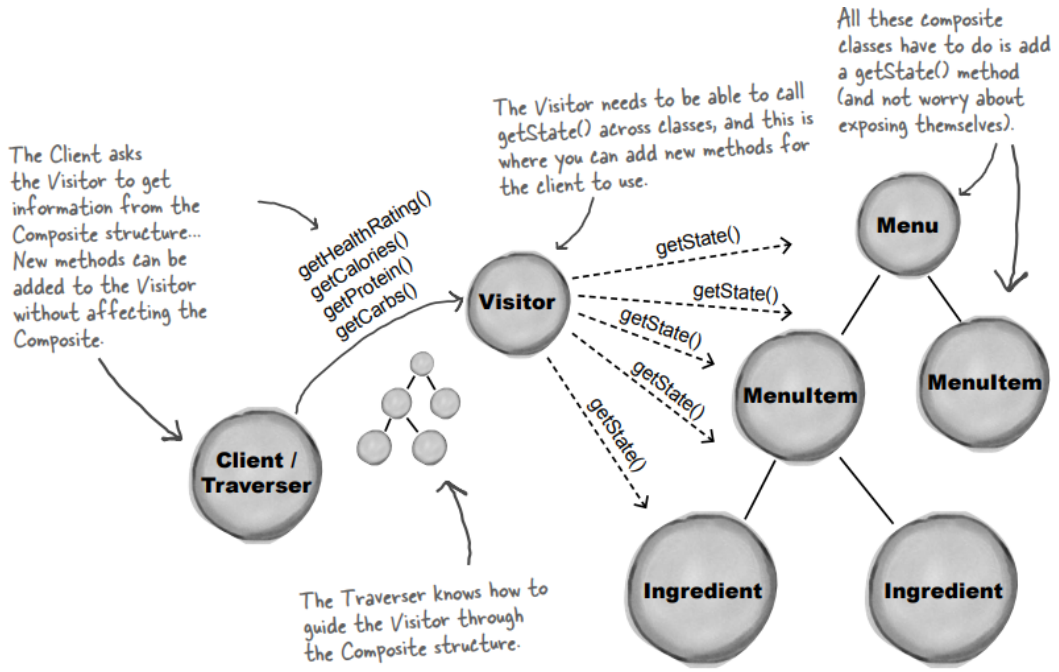

Abbildung 10: UML-Darstellung des Visitor-Musters

9 Visitor-Muster

Verwenden Sie das Visiotr-Muster, wenn Sie ein Objekt-Kompositum um neue Fähigkeiten erweitern möchten und neue Kapselung unwichtig ist.

9.1 Problemstellung

Die Gäste, die das Restaurant und das Pfannkuchenhaus besuchen, sind in letzter Zeit gesundheitsbewusster geworden. Bevor sie ihr Essen bestellen, fragen sie nach Nahrungsmittelinformationen. Da beide Restaurantsbereit sind Sonderbestellungen entgegenzunehmen, fragen manche Gäste sogar nach Nahrungsmittelinformationen zu einzelnen Zutaten.

9.2 Erklärung des Musters

Der Visitor muss jedes Element des Kompositums besuchen; diese Funktionalität steckt in einem Durchquerer-Objekt. Der Visitor wird von Durchquerer angeleitet, den Zustand von allen Objekten im Kompositum zu erfassen. Wenn der Zustand einmal erfasst ist, kann der Client den Visitor verschiedene Operationen auf dem Zustand ausführen lassen. Ist neue Funktionalät erforderlich, muss lediglich der Visitor erweiter werden.

9.3 Vorteile

1. Ermöglicht Operationen zu einer Kompositumsstruktur hunzuzufügen, ohne die Struktur selbst zu ändern.
2. Das Hinzufügen neuer Operationen ist relativ einfach
3. Der Code für Operationen, die der Visitor durchführt, ist zentralisiert.

9.4 Nachteile

1. Bei der Verwendung des Visitors wird die Kapselung der Kompositumsklassen zerstört.
2. Da die Durchquererfunktion mit in die Sache verwickelt ist, sind die Veränderungen an der Kompositumsstruktur schwieriger.

10 Iterator-Muster

10.1 Problemstellung

Beispiel aus dem Buch: Zwei verschiedene Speisekarten sollen beide von einer Kellner-Klasse verwendet werden. Die Kellnerin soll in der Lage sein, alle Gerichte gemeinsam auszugeben, sowie die einzelne Gerichte zu filtern (Frühstückstisch, etc.). Problem hierbei, die vorgegebenen Klassen nutzen jeweils eine andere Listenimplementierung. Wie kann man nun dennoch geschlossen über die Listen iterieren ohne den kompletten vorgegebenen Code umzuschreiben.

10.2 Erklärung des Musters

Das Iterator-Muster bietet eine Möglichkeit, auf die Elemente in einem Aggregat-Objekt sequenziell zuzugreifen, ohne die zu Grunde liegende Implementierung zu offenbaren. Das Muster gibt die Aufgaben der Durchquerung außerdem dem Iterator-Objekt statt dem Aggregat. Das vereinfacht die Schnittstelle und die Implementierung des Aggregats und schiebt die Verantwortung für die Iteration dahin, wo sie hingehört.

Um das beschriebene Problem zu lösen reicht es die Iterator-Schnittstelle zu implementieren. Diese kapselt die Iteration und die Kellnerin kann nun polymorph über alle Elemente iterieren. Eine übergeordnete Speisekarten-Schnittstelle lässt nun folgendes UML-Diagramm entstehen:

10.3 Punkt für Punkt - S. 380

- Ein Iterator ermöglicht Zugriff auf die Elemente eines Aggregats, ohne seine interne Struktur zu offenbaren.
- Ein Iterator übernimmt die Aufgabe, über ein Aggregat zu iterieren, und kapselt es in einem anderen Objekt.
- Wenn wir einen Iterator verwenden, erlösen wir das Aggregat von der Verantwortung, Operationen für die Durchquerung seiner Daten zur Verfügung zu stellen.
- Ein Iterator bietet eine allgemeine Schnittstelle für die Durchquerung der Elemente eines Aggregats und ermöglicht es Ihnen, beim Schreiben von Code, der die Elemente eines Aggregats verwendet, Polymorphismus zu verwenden.

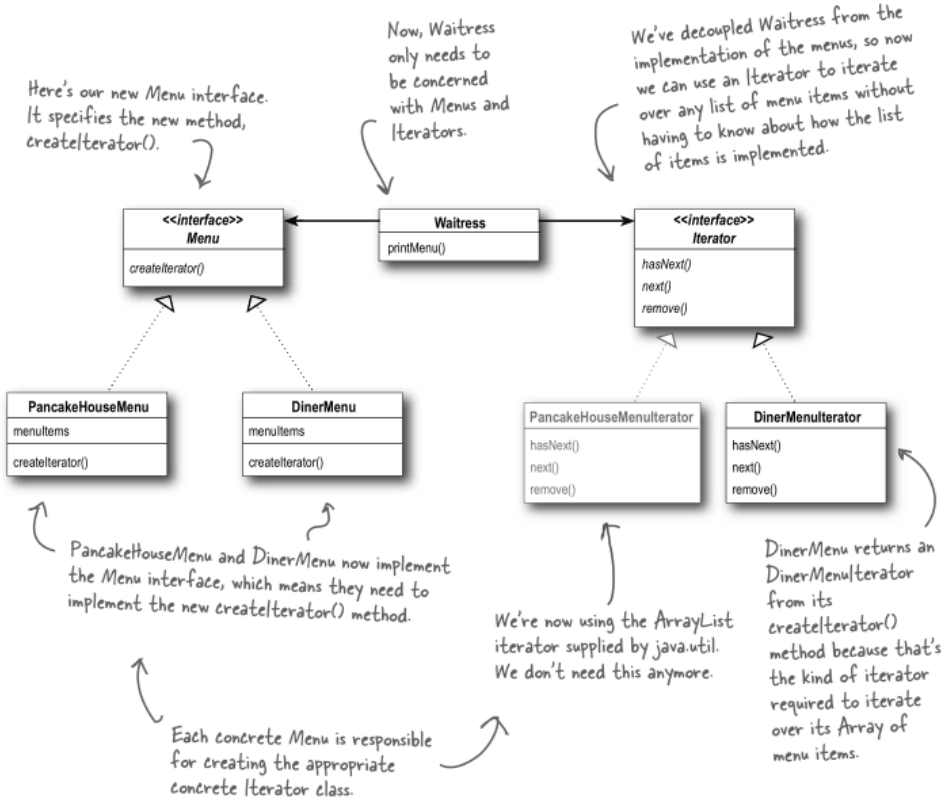

Abbildung 11: UML-Darstellung Composite-Musters

11 Composite-Muster

11.1 Problemstellung

Das Beispiel aus dem vorherigen Kapitel (Iteratoren) soll erweitert werden. *Jetzt möchten sie eine Dessertkarte hinzufügen. Gut. Und was jetzt? Jetzt müssen wir nicht nur mehrere Speisekarten, sondern auch noch Speisekarten in Speisekarten unterstützen. Es wäre schön, wenn wir die Dessertkarte einfach zu einem Element der Collection RestaurantSpeisekarte machen könnten, aber so wie das jetzt implementiert ist, funktioniert es nicht.*

11.2 Erklärung des Musters

Definition: Das Composite-Muster ermöglicht es Ihnen, Objekte zu einer Baumstruktur zusammenzusetzen, um Teil/Ganzes-Hierarchien auszudrücken. Das Composite-Muster erlaubt den Clients, individuelle Objekte und Zusammensetzungen von Objekten auf gleiche Weise zu behandeln.

Wesentlicher Nachteil: Alle Komponenten müssen die Schnittstelle Speisekarten-Komponente implementieren. Aber weil die Blätter und die Knoten unterschiedliche Rollen haben, können wir nicht für alle Methoden eine sinnvolle Default-Implementierung definieren. Manchmal ist es noch dasd Beste, einfach eine Runtime-Exception auzulösen.

Das Iterieren innerhalb des Kompositum-Baums lässt sich mithilfe eines neuen Iterators bewerkstelligen. Hierbei speichert jedes Element des Kompositums eine Liste von Iteratoren seiner Kindsknoten / Blätter. Dies wird rekursiv für alle Kinder implementiert. Die Blätter bilden hierbei die Austrittsbedingung und geben einen Iterator zurück welcher lediglich über den Wert des eigenen Blattes iteriert. In den jeweils höheren Elementen wird zur Laufzeit dieser Stack von Element-Iteratoren nach und nach abgearbeitet.

Punkt für Punkt - S. 380

- Das Composite-Muster bietet eine Struktur, die einzelne Objekte und Komposita aufnehmen kann.
- Das Composite-Muster erlaubt Clients, Komposita und einzelne Objekte auf gleiche Weise zu behandeln.
- Jede Objekt in einer zusammengesetzten Struktur ist eine Komponente. Komponenten können andere Komponenten oder Blattknoten sein.
- Bei der Implementierung von Composite gibt es viele Designkompromisse. Sie müssen einen Ausgleich zwischen Transparenz und Sicherheit und Ihren Anforderungen finden.

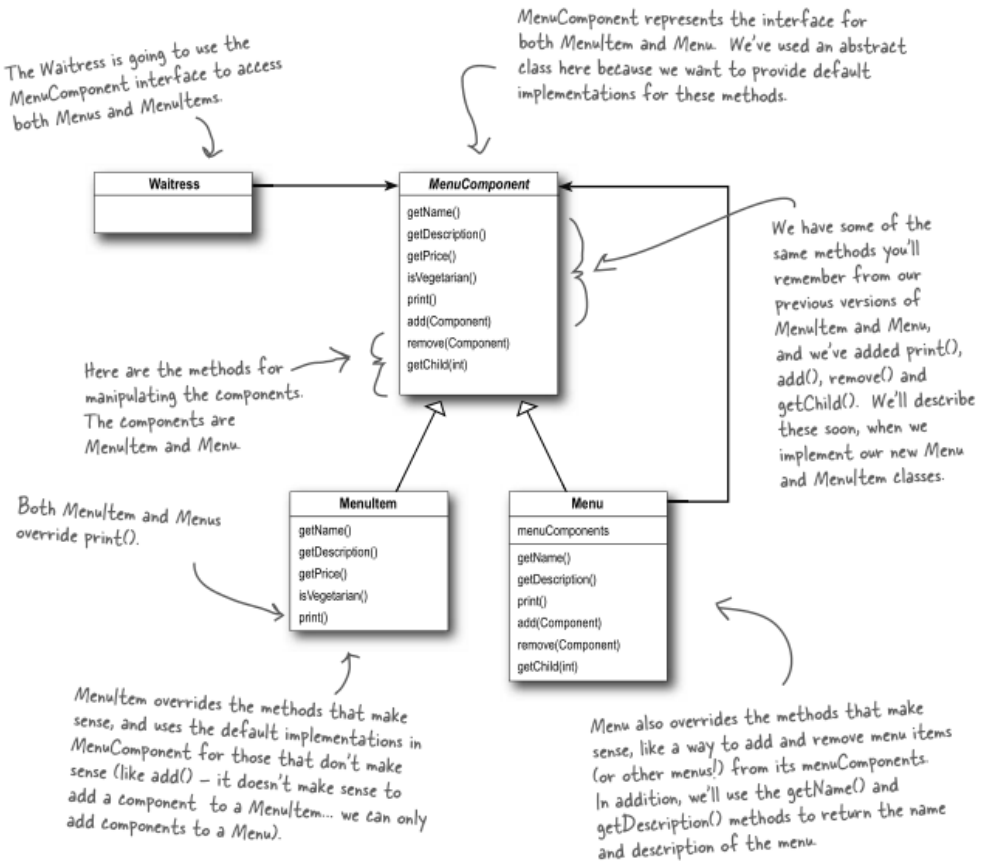

Abbildung 12: UML-Darstellung des Composite-Musters

12 Builder-Muster

12.1 Problemstellung

Sie haben soeben den Auftrag erhalten, einen Urlaubsplaner für einen neuen Themenpark zur erstellen. Die Gäste des Parks können ein Hotel und verschiedene Sorten von Eintrittskarten auswählen, Tische in Restaurants servieren lassen und sogar Special Events buchen. Um den Urlaubsplaner zu erstellen, müssen Sie strukturen wie die folgenden erzeugen können.

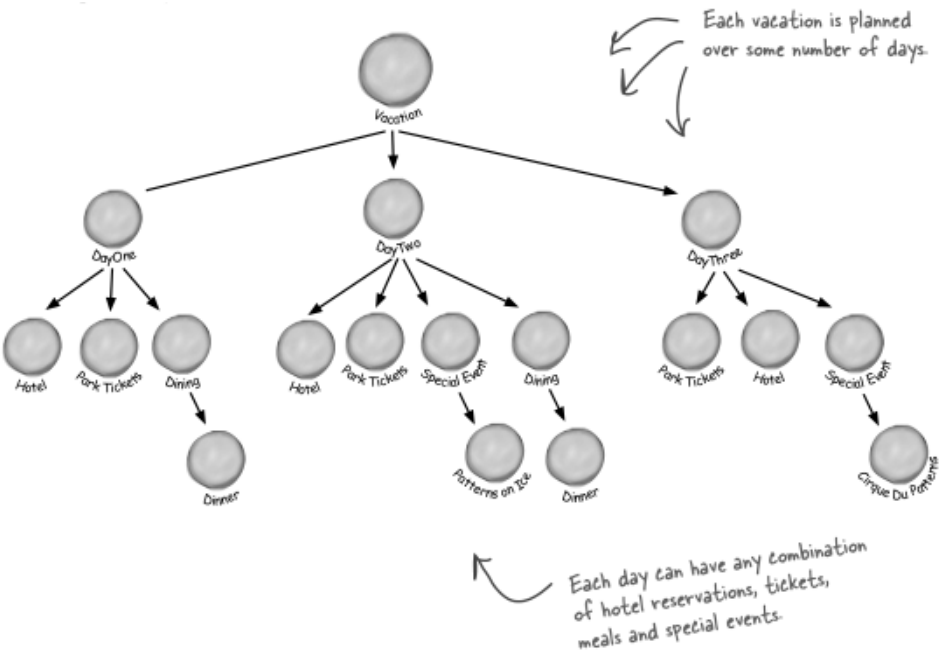

Abbildung 13: Darstellung der Datenstruktur

Es soll besonders Wert auf eine flexible Datenstruktur gelegt werden, da verschiedene Besucher auch sehr verschiedene Wünsche haben. Zudem benötigen Sie eine Abfolge von möglicherweise komplexen Schritten für die Erzeugung des Planers. Wie können Sie die komplexe Struktur erzeugen, ohne sie mit den Schritten für die Erzeugung zu vermischen?

12.2 Erklärung des Musters

Definition: Verwenden Sie das Builder-Muster, um die Konstruktion eines Produkts zu kapseln und seinen Aufbau in mehreren Schritten zu ermöglichen.

Ähnlichkeiten zum Iterator-Muster. Dort wurde die Iteration in einem separaten Objekt gekapselt und die interne Repräsentation der Collection vor dem Client verborgen. Hier steckt der gleiche Gedanke dahinter: Wir kapseln die Erzeugung des Reiseplaners in einem Objekt (dem Builder oder Erbauer, hier nennen wir es Ersteller), und unser

Client bittet dann den Ersteller, die Struktur des Reiseplaners für ihn zu erstellen. Ein Blick in das folgende Diagramm sollte hierbei Klarheit verschaffen.

12.3 Vorteile des Builder-Musters

- Kapselt die Art und Weise, wie ein komplexes Objekt hergestellt wird.
- Ermoglicht die Herstellung von Objekten in einem Prozess, der aus mehreren Schritten besteht und variiert (im Gegensatz zu Ein-Schritt-Fabriken).
- Verbirgt die interne Reprasentation des Produkts vor dem Client.
- Produkt-Implementierungen können ausgetauscht werden, weil der Client nur eine abstrakte Schnittstelle sieht.

12.4 Verwendung und Nachteile

1. Häufig für den Aufbau von Composite-Strukturen verwendet.
2. Die Herstellung von Objekten erfordert mehr Wissen üer den Arbeitsbereich des Clients., als wenn man eine Fabrik verwendet.

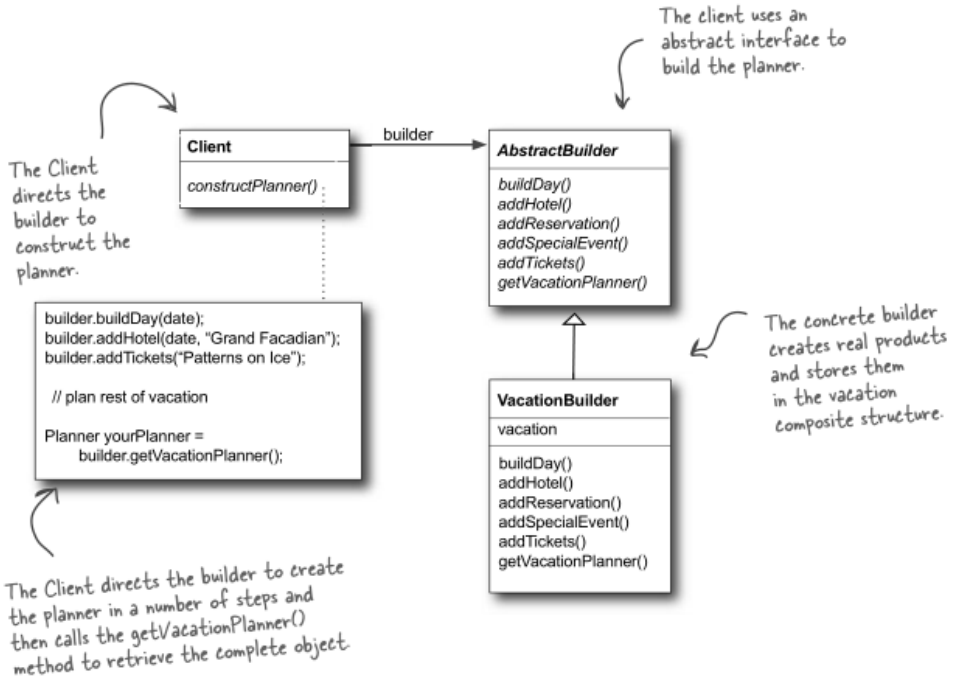

Abbildung 14: UML-Darstellung des Builder-Musters

13 Memento-Muster

13.1 Problemstellung

Sie entwickeln ein Computerspiel und sollen eine Speichern-Funktionalität hinzufügen, die die Spielfigur zu einem späteren Zeitpunkt an den gespeicherten Platz zurück befördert falls dieser im späteren Spielverlauf etwas zustoßen sollte.

13.2 Erklärung des Musters

Definition: Verwenden Sie das Memento-Muster, wenn es erforderlich ist, dass Sie ein Objekt in einen seiner früheren Zustände zurücksetzen können, z.B. wenn Ihr Benutzer die Anweisung *rückgängig* eingibt.

Ziele des Musters: Das Memento/Muster hat zwei Ziele

- Den wichtigen Zustand des Schlüssel-Objekts von einem System speichern.
- Die Kapselung des Schlüssel-Objekts bewharen.

Vor dem Hintergrund des Prinzips *nur eine Zuständigkeit pro Klasse* ist es auch eine gute Idee, den zu speichernden Zustand von dem Schlüssel-Objekt zu trennen. Dieses separate Objekt, das den Zustand hält, wird Memento-Objekt genannt.

13.3 Vor/Nachteile

Vorteile des Memento-Musters:

- Wenn der gespeicherte Zustand außerhalb des Schlüsselobjekts gehalten wird, bleibt die Kohäsion *(innerer Zusammenhalt)* erhalten.
- Bewahrt die Kapselung der Daten des Schlüssel-Objekts.
- Bietet eine leicht zu implementierende Wiederherstellungsmöglichkeit.

Verwendung und Nachteile:

- Das Memento-Muster wird zur Speicherung eines Zustands verwendet.
- Speicherung und Wiederherstellung können zeitaufwendig sein.
- In Java-Szstemen sollten Sie in betracht ziehen, Systemzustände stattdessen über Serialisierung zu speichern.

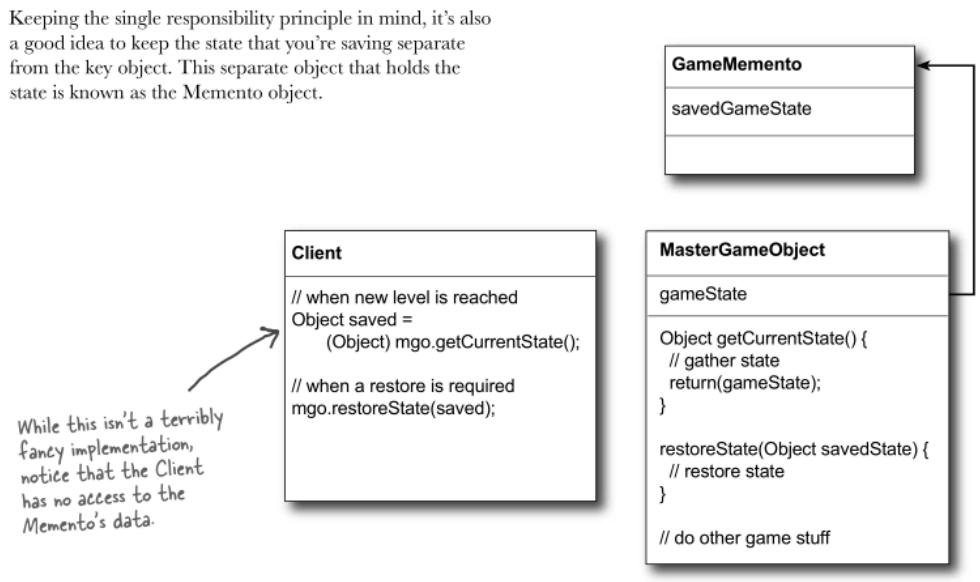

Abbildung 15: UML-Darstellung des Memento-Musters