Zusammenfassung

Objekt-orientiertes Design

von Simon Flüeli

Inhaltsverzeichnis

Entwurfsmuster - Design Patterns 4
Was sind Entwurfsmuster 4
Beschreibung durch 4
Klassifikation 4
Klassenbibliothek 5
Topologien 5
Framework 6
Muster vs. Framework 6
Fabrikmethode-Muster (Factory Pattern) 7
Motivation 7
Problem 7
Lösung 7
Anwendbarkeit: Verwendung des Musters, wenn 7
Allgemeine Struktur 7
Interaktion 7
Konsequenzen 7
Singleton-Muster (Singleton Pattern) 8
Motivation 8
Anwendbarkeit: Verwendung des Musters, wenn: 8
Allgemeine Struktur 8
Interaktionen 8
Konsequenzen 8
Kompositum Muster (Composite Pattern) 9
Motivation 9
Anwendbarkeit: Verwenden, wenn 9
Allgemeine Struktur 9
Interaktion 9
Konsequenzen 9
Proxy Muster (Proxy Pattern) 10
Motivation 10
Anwendbarkeit: Muster anwenden wenn 10
Allgemeine Struktur 10
Interaktionen 11
Konsequenzen 11
Fassaden-Muster (Facade-Pattern) 12
Motivation 12
Anwendbarkeit: Muster verwenden, wenn 12

Allgemeine Struktur 12
Interaktionen 12
Konsequenzen 12
Beobachter Muster (Observer Pattern) 13
Motivation 13
Anwendbarkeit: Muster anwenden, wenn gilt: 13
Allgemeine Struktur 13
Interaktion 14
Konsequenzen 14
Schablonen Muster (Template-Method Pattern) 15
Motivation 15
Anwendbarkeit: Muster verwenden 15
Allgemeine Struktur 15
Interaktionen 16
Konsequenzen 16
Strategie Muster (Strategy Pattern) 17
State Pattern 18
Feinheiten 18
Unterschied State ←→ Strategy 18
Decorator Pattern 19
The Iterator Pattern 19

Entwurfsmuster - Design Patterns

Was sind Entwurfsmuster

- Generische Lösung für ein immer wiederkehrendes Problem

Beschreibung durch

- Name
- Problembeschreibung
  - Wann ist das Muster anwendbar
- Lösungsbeschreibung
  - Kein konkreter Entwurf und keine Implementierung
  - Abstrakte Beschreibung des Problems
  - Beschreibt allgemeine Anordnung der Klassen bzw. Objekte
- Konsequenzen
  - Zeit- und Speichereffizienz
  - Sprach- und Implementierungseigenschaften
  - Flexibilität, Erweiterbarkeit und Portabilität

Klassifikation

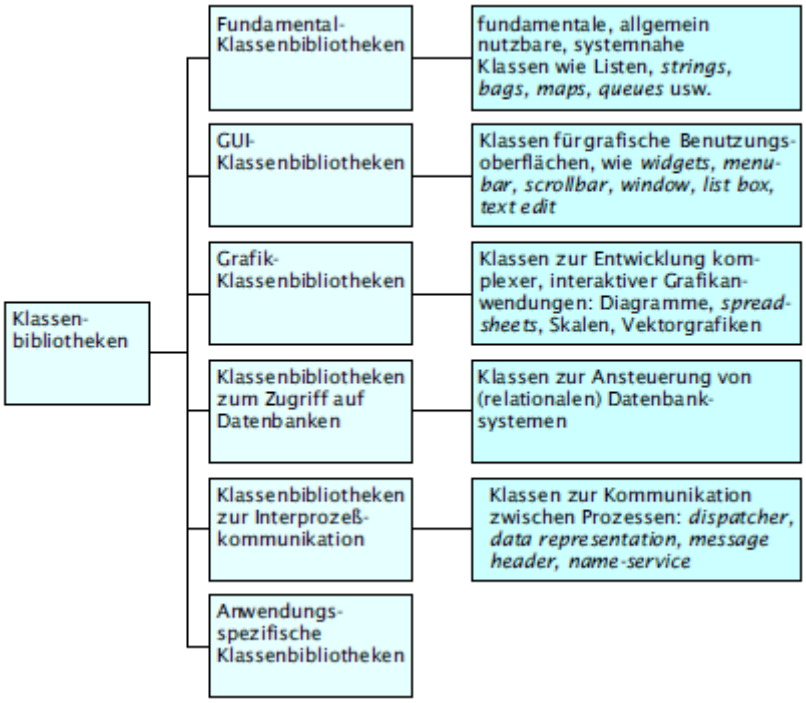
- Erzeugungsmuster (creational patterns)
  - Helfen, ein System unabhängig davon zu machen, wie seine Objekte erzeugt, zusammengesetzt und repräsentiert werden
- Strukturmuster (structural patterns)
  - Zusammensetzung von Klassen und Objekten zu grösseren Strukturen
- Verhaltensmuster (behavioural patterns)
  - Interaktion zwischen Objekten und Klassen
  - Beschreiben komplexe Kontrollflüsse, die zur Laufzeit schwer nachvollziehbar sind

- Klassenbasierte Muster
  - Beziehungen zwischen Klassen
  - Ausgedrückt durch Generalisierungsstrukturen
  - Festgelegt zur Übersetzungszeit
- Objektbasierte Muster
  - Beschreiben Beziehungen zwischen Objekten, die zur Laufzeit geändert werden können

Benutzen auch bis zu einem gewissen Grad die Generalisierung

Klassenbibliothek

- Organisierte Softwaresammlung, aus der ein Entwickler nach Bedarf Einheiten verwendet



Topologien

- Baum Topologie
  - Gemeinsame Wurzelklasse
- Wald-Topologie
  - Bibliothek besteht aus mehreren Baumhierarchieren
  - Vorteil: Flache Generalisierungshierarchie im Vergleich zu Baum-Topologie
- Baustein-Topologie
  - Unabhängige Klassen
  - Verwendung des Konzepts der generischen Klasse zur spezifischen Anpassung

Framework

- Menge von zusammenarbeitenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren
- Besteht aus konkreten und - insbesondere - aus abstrakten Klassen, die Schnittstellen definieren
- Definition von Unterklassen zur Verwendung und Anpassung des Frameworks
  - Selbstdefinierte Unterklassen empfangen Botschaften von vordefinierten Framework-Klassen
  - Hollywood-Prinzip "Don't call us, we'll call you"
- Ist immer spezifisch auf einen Anwendungsbereich ausgelegt
  - Erstellung grafischer Editoren
  - Erstellung von Finanzsoftware
- Spezialisierung für eine konkrete Anwendung durch Ableiten von Unterklassen aus den abstrakten Framework-Klassen
- Realisierung der Frameworks mittels Programmiersprachen
  - Frameworks können also ausgeführt und direkt wiederverwendet werden
- Ermöglicht hohe Wiederverwendung
- Bestimmt Architektur der Anwendung
- Definiert Struktur der Klassen und Objekte und deren Verantwortlichkeiten
- Legt fest, wie Klassen und Objekte zusammenarbeiten
- Legt fest, wie der Kontrollfluss aussieht
- Konzentrierung auf Details der Anwendung

Muster vs. Framework

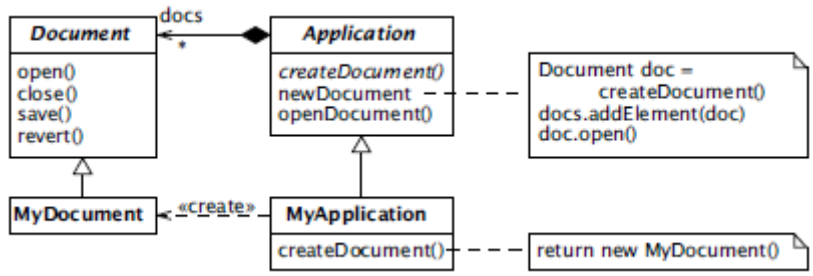
- Entwurfsmuster sind abstrakter als Frameworks
  - Werden nur beispielhaft durch Code repräsentiert
  - Anwendung von Mustern mit neuer Implementierung verbunden
- Muster sind kleiner als Frameworks
  - Framework enthält mehrere Muster
- Muster sind weniger spezialisiert als Frameworks
  - Keine Beschränkung auf bestimmten Anwendungsbereich

Fabrikmethode-Muster (Factory Pattern)

- Klassenbasiert
- Bietet Schnittstelle zum Erzeugen eines Objekts an
- Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekte ist
- Bezeichnung auch: Virtueller Konstruktor (virtual constructor)

Motivation

- Verwendung eines Frameworks für Anwendung, die mehrere Dokumente gleichzeitig anzeigen kann
- Verwendung der abstrakten Klasse Application und Document und Modellierung einer Assoziation zwischen ihren Objekten



- Application: für Erzeugung neuer Dokumente zuständig

Problem

- Framework muss Objekte (MyDocument) erzeugen, kennt aber nur die abstrakte Oberklasse, von der es keine Objekte erzeugen darf

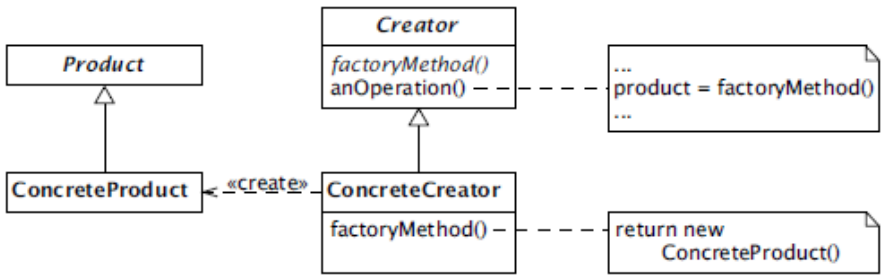
Lösung

- Unterklasse MyApplication muss die abstrakte Methode createDocument() überschreiben
  → Exemplar von MyDocument wird zurückgegeben
  - Nach der Erzeugung eines Objektes von MyApplication, kann dieses spezifische Dokumente erzeugen, ohne deren exakte Klasse zu kennen
  - Operation createDokument() heisst **Fabrikmethode**
    - Für die Fabrikation eines Objektes verantwortlich

Anwendbarkeit: Verwendung des Musters, wenn

- Eine Klasse die von ihr zu erzeugenden Objekte nicht im Voraus kennen kann
- Die Unterklassen festlegen sollen, welche Objekte sie erzeugen

Allgemeine Struktur



Interaktion

- Creator verlässt sich darauf, dass Unterklassen die Fabrikmethode korrekt implementieren

Konsequenzen

- Fabrikmethoden verhindern, dass sie anwendungsspezifische Klassen in den Code des Frameworks einbinden müssen!

Singleton-Muster (Singleton Pattern)

- Objektbasiert
- Stellt sicher, dass eine Klasse genau ein Objekt besitzt
- Ermöglicht globalen Zugriff auf dieses Objekt

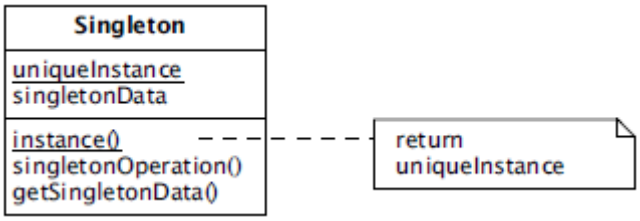
Motivation

- Bei manchen Klassen soll genau ein Objekt existieren
- Einfacher Zugriff auf dieses Objekt von mehreren anderen Objekten

Anwendbarkeit: Verwendung des Musters, wenn:

- Es genau ein Objekt von einer Klasse geben und ein einfacher Zugriff darauf bestehen soll
- Das einzige Exemplar durch Spezialisierung mittels Unterklassen erweitert wird und Klienten das erweiterte Exemplar verwenden können, ohne ihren Code zu ändern

Allgemeine Struktur




```
public final class Singleton {
   private static Singleton uniqueInstance = null;
   private Singleton() {}
   public static Singleton getInstance() {
       if(uniqueInstance == null)  {
              uniqueInstance = new Singleton();
       }
       return uniqueInstance;
   }
}
```

Interaktionen

- Klienten holen über getInstance() eine Referenz auf das einzige Objekt

Konsequenzen

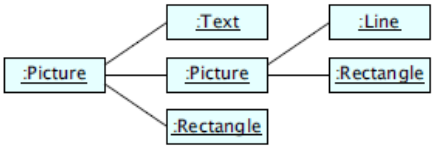
- Verbesserung gegenüber globalen Variablen
- Singleton-Klasse kann durch Unterklasse spezialisiert werden
- Änderungen auf mehrere Exemplare leicht durchführbar

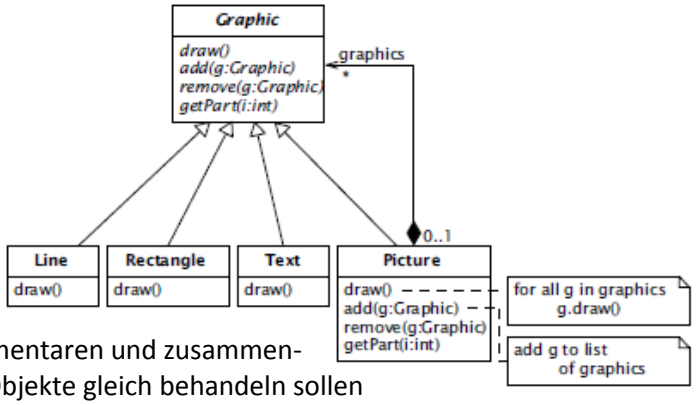
Kompositum Muster (Composite Pattern)

- Objektbasiert
- Setzt Objekte zu Baumstrukturen zusammen, um whole-part-Hierarchien darzustellen
- Ermöglichet es, sowohl einzelne Objekte als auch einen Baum von Objekten einheitlich zu behandeln

Motivation

- Zusammensetzen von Grafiken zu komplexen Grafiken
- Komplexe und einfache Grafiken sollen aus Sicht des Klienten gleich behandelt werden

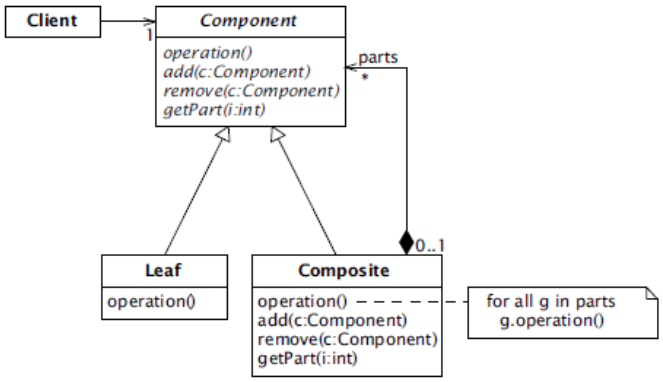




Anwendbarkeit: Verwenden, wenn

- whole-part-Hierarchien von Objekten dargestellt werden sollen
- Klienten keinen Unterschied zwischen elementaren und zusammengesetzten Objekten wahrnehmen und alle Objekte gleich behandeln sollen

Allgemeine Struktur



- Component
  - Deklariert Schnittstelle für alle Objekte, implementiert default-Verhalten und deklariert Schnittstelle zum Zugriff und Verwalten von Teilobjekten
- Leaf
  - Repräsentiert elementare Objekte
- Composite
  - Aggregatklasse, definiert das Verhalten von zusammengesetzten Objekten, speichert Teilobjekte und implementiert Operationen, die sich auf Teilobjekte beziehen
- Client
  - Repräsentiert die Klienten

Interaktion

- Klienten verwenden nur die Schnittstelle von Component
- Wenn Empfänger elementares Objekt, wird die Botschaft direkt bearbeitet, sonst leitet es die Botschaft weiter

Konsequenzen

- Klient wird einfacher
- Es ist einfach, neue Arten von Komponenten einzufügen

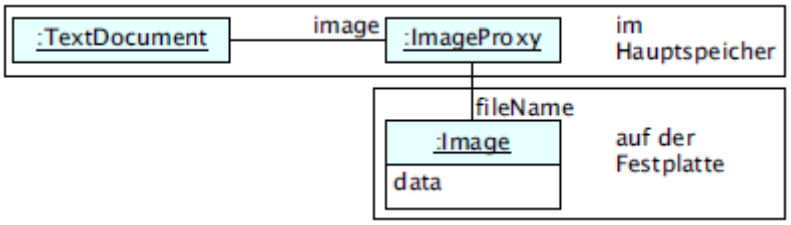
Proxy Muster (Proxy Pattern)

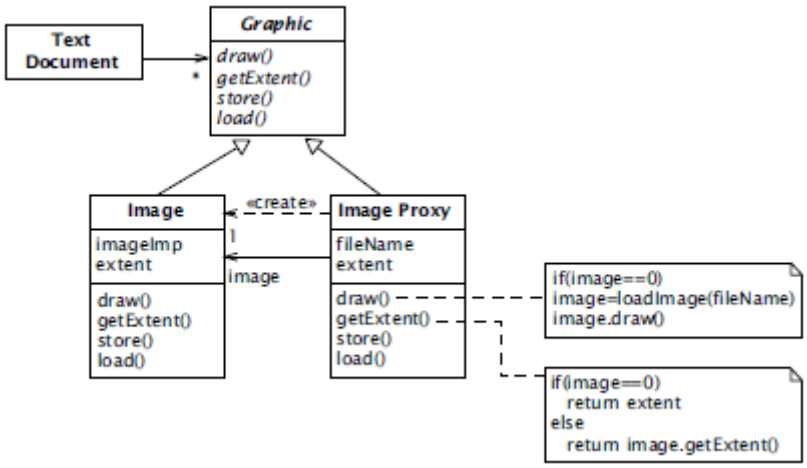
- Objektbasiert
- Kontrolliert den Zugriff auf ein Objekt mithilfe eines vorgelagerten Stellvertreter-Objekts
- Auch: Surrogat

Motivation

- Integration grafischer Objekte in einen Text
- Darstellung grosser Bilder benötigt viel Computerleistung

→ Verwendung eines **Platzhalters** (Proxy) anstelle des echten Bildes
→ Bild-Proxy erzeugt tatsächliches Bild erst, wenn es benötigt wird
→ Proxy kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreter-Objekts

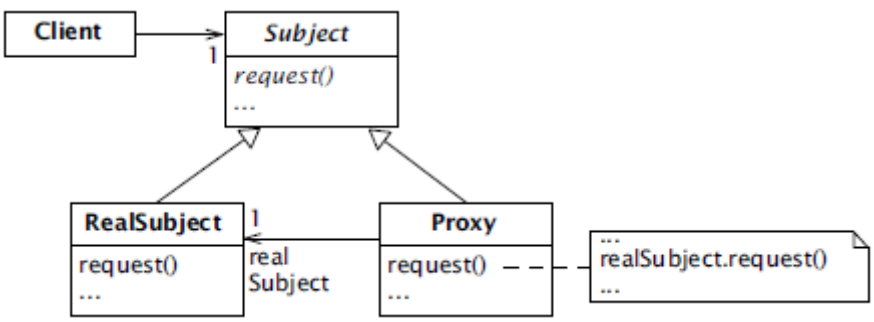




Anwendbarkeit: Muster anwenden wenn

- Remote-Proxy als lokaler Vertreter für ein Objekt auf einem anderen Computer
- Virtuelles Proxy erzeugt "teure" Objekte auf Verlangen
- Schutz-Proxy kontrolliert Zugriff auf Original
- Smart reference als Ersatz für einen einfachen Zeiger, der zusätzlich folgende Funktionen anbietet
  - Zählen der Referenzen auf eigentliches Objekt
  - Automatische Freigabe, wenn es keine Referenzen mehr besitzt
  - Laden eines persistenten Objekts, wenn es erstmalig referenziert wird
  - Testen eines Objekts auf locking, bevor darauf zugegriffen wird

Allgemeine Struktur



- Proxy
    - Kontrolliert Zugriff auf eigentliches Objekt und ist dafür zuständig, es zu erzeugen und zu löschen
        - Bietet eine Schnittstelle, die mit der von Subject identisch ist, so dass ein Proxy-Objekt für Subject-Objekt eingesetzt werden kann
        - Remove-Proxies kodieren eine Botschaft und senden sie an das RealSubject in einem anderen Adressraum
        - Viertuelle Proxies können zusätzliche Informationen über das RealSubject speichern

-Subject
    - Definiert die gemeinsame Schnittstelle des echten Objekts und des Proxy-Objekts
- RealSubject: Definiert das echte Objekt

Interaktionen

- Proxy leitet Befehle an das echte Objekt weiter

Konsequenzen

- Remote-Proxy verbirgt die Tatsache, dass sich ein Objekt in einem anderen Adressraum befindet
- Virtuelles Proxy dient der Optimierung
- Schutz-Proxies und smart references ermöglichen die Durchführung zusätzlicher Verwaltungsaufgaben beim Zugriff auf das Objekt

Fassaden-Muster (Facade-Pattern)

- Objektbasiert
- Bietet einfache Schnittstelle zu einer Menge von Klassen (Paket) an
- Definiert eine abstrakte Schnittstelle, um die Benutzung des Pakets zu vereinfachen

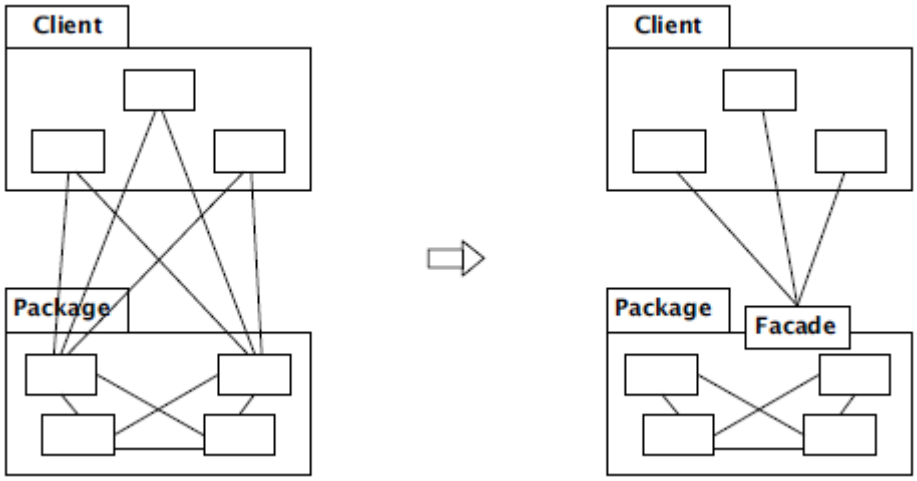
Motivation

- Lose Kopplung von Paketen
  - Erleichtert den Austausch der Pakete und deren unabhängige Implementierung
- Fassadenklasse zeigt eine vereinfachte Schnittstelle für die - umfassendere - Funktionalität des Pakets
  - Vereinfachte Sicht genügt vielen Klienten
  - Klienten, denen diese Schnittstelle nicht reicht, müssen hinter die Fassade schauen

Anwendbarkeit: Muster verwenden, wenn

- Einfache Schnittstellen zu einem komplexen Paket angeboten werden sollen
- Es zahlreiche Abhängigkeiten zwischen Klienten und einem Paket gibt
- Pakete in Schichten organisiert werden sollen

Allgemeine Struktur



- Fassade weiss, welche Klasse des Pakets für die Bearbeitung einer Botschaft zuständig sind und delegiert Botschaften vom Klienten an die zuständige Klasse
  - Definiert keine neue Funktionalität
  - Oft wird nur ein Objekt der Fassadenklasse benötigt
- Package-Klassen
  - Führen die von der Fassade zugewiesenen Aufgaben durch und wissen nichts von der Fassade

Interaktionen

- Klienten kommunizieren mit dem Paket, indem sie Botschaften an die Fassade schicken, welche diese dann an das entsprechende Objekt innerhalb des Pakets weiterleitet

Konsequenzen

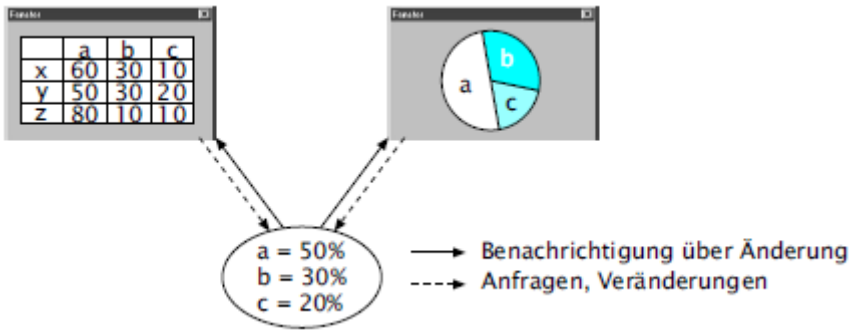
- Vereinfachung der Benutzung des Systems durch Reduzierung der Klassen, die den Klienten bekannt sein müssen
- Lose Kopplung erleichtert Austausch von Paketen und deren unabhängige Implementierung
- Klienten können die Fassade umgehen und direkt auf die Klassen des Pakets zugreifen

Beobachter Muster (Observer Pattern)

- Objektbasiert
- Sorgt dafür, dass bei Änderungen eines Objekts alle davon abhängigen Objekte benachrichtigt und automatisch aktualisiert werden

Motivation

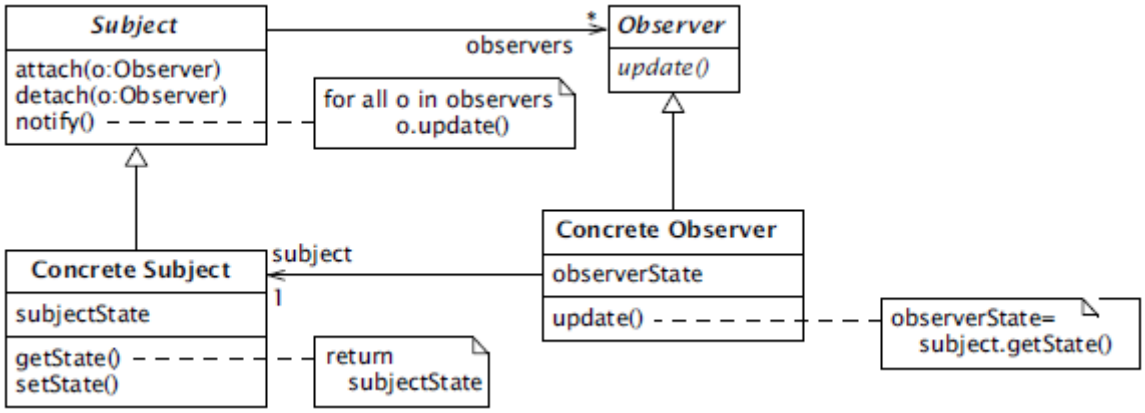
- Objekt enthält Anwendungsdaten
- Darstellung dieser Daten auf verschiedene Arten
- Kreisdiagramm soll sich ändern, wenn die Daten in der Tabelle verändert werden und umgekehrt



Anwendbarkeit: Muster anwenden, wenn gilt:

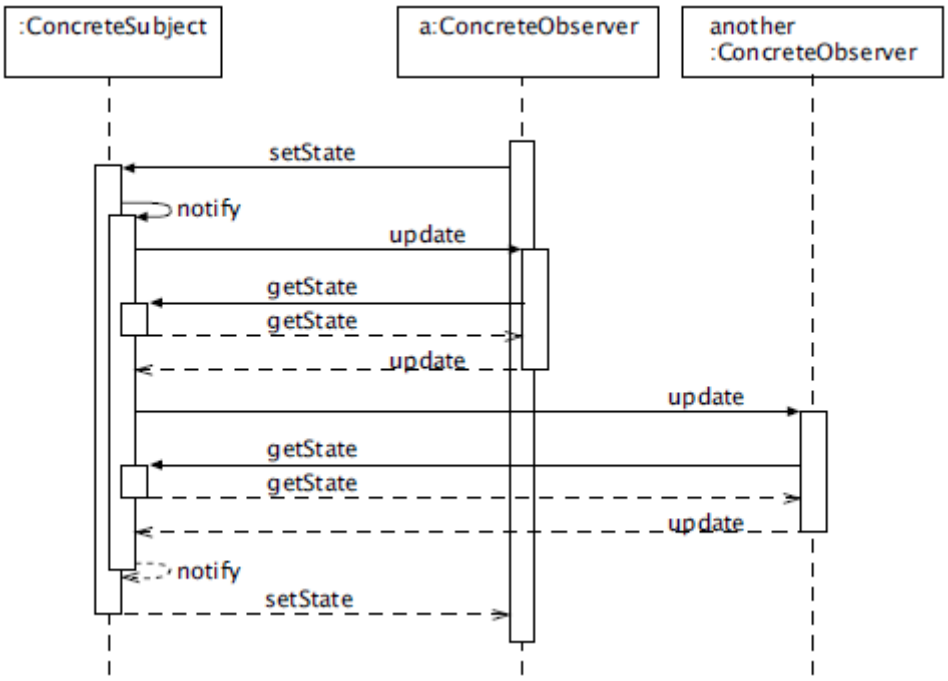
- Eine Abstraktion besitzt zwei Aspekte, die wechselseitig voneinander abhängen
- Änderung eines Objekts impliziert die Änderung anderer Objekte unbekannter Anzahl
- Ein Objekt soll andere Objekte benachrichtigen und diese Objekte sind nur lose gekoppelt

Allgemeine Struktur



- Subject
  - Kennt eine beliebige Anzahl von Beobachtern
- Observer
  - Definiert die Schnittstelle für alle konkreten Objekte, die über Änderungen eines subjects informiert werden müssen
- ConcreteSubject
  - Objekte speichern die Daten, die für die konkreten Beobachter relevant sind
- ConcreteObserver
  - Objekte kennen das konkrete Subjekt und speichern den Zustand, der mit dem des Subjekts konsistent sein soll
    - Implementiert die Operationen der Observer-Klasse

Interaktion



Konsequenzen

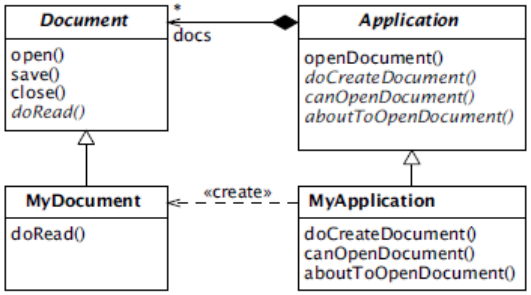
- Subjekte und Beobachter können unabhängig voneinander modifiziert und einzeln wiederverwendet werden
- Neue Beobachter können ohne Änderung des Subjekts hinzugefügt werden

Schablonen Muster (Template-Method Pattern)

- Objektbasiert
- Definiert den Rahmen eines Algorithmus in einer Operation und delegiert Teilschritte an Unterklassen

Motivation

- Framework, das die Klassen Document und Application bereitstellt
- Schablonenmethode openDocument()



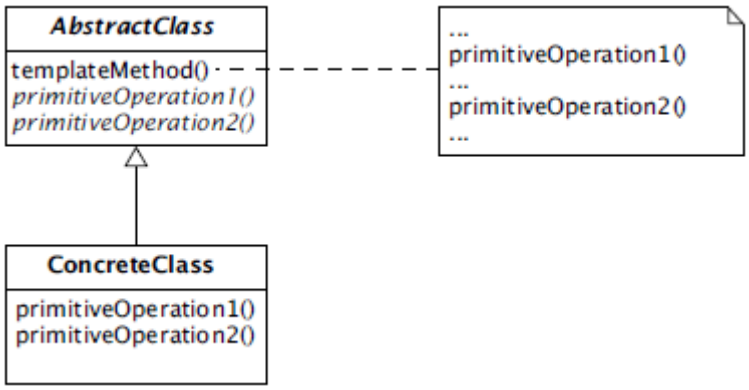

```
abstract class Application {
   public boolean canOpenDocument(String Name) { return true; )
   public void addDocument(Document doc) {};
   public abstract Document doCreateDocument();
   public abstract void aboutToOpenDocument(Document doc);

   public void openDocument(String name) {
      if(!canOpenDocument(name)) {
         return true;
      }
      Document doc = doCreateDocument();
      if(doc != null) {
         this.addDocument(doc);
         this.aboutToOpenDocument(doc);
         doc.open();
         doc.doRead();
      }
   }
}
```

Anwendbarkeit: Muster verwenden

- Um die invarianten Teile eines Algorithmus genau einmal festzulegen; konkrete Ausführung der variierenden Teile wird den Unterklassen überlassen
- Wenn gemeinsames Verhalten von Unterklassen in einer Oberklasse realisiert werden soll; Vermeidung der Duplikation von Code

Allgemeine Struktur

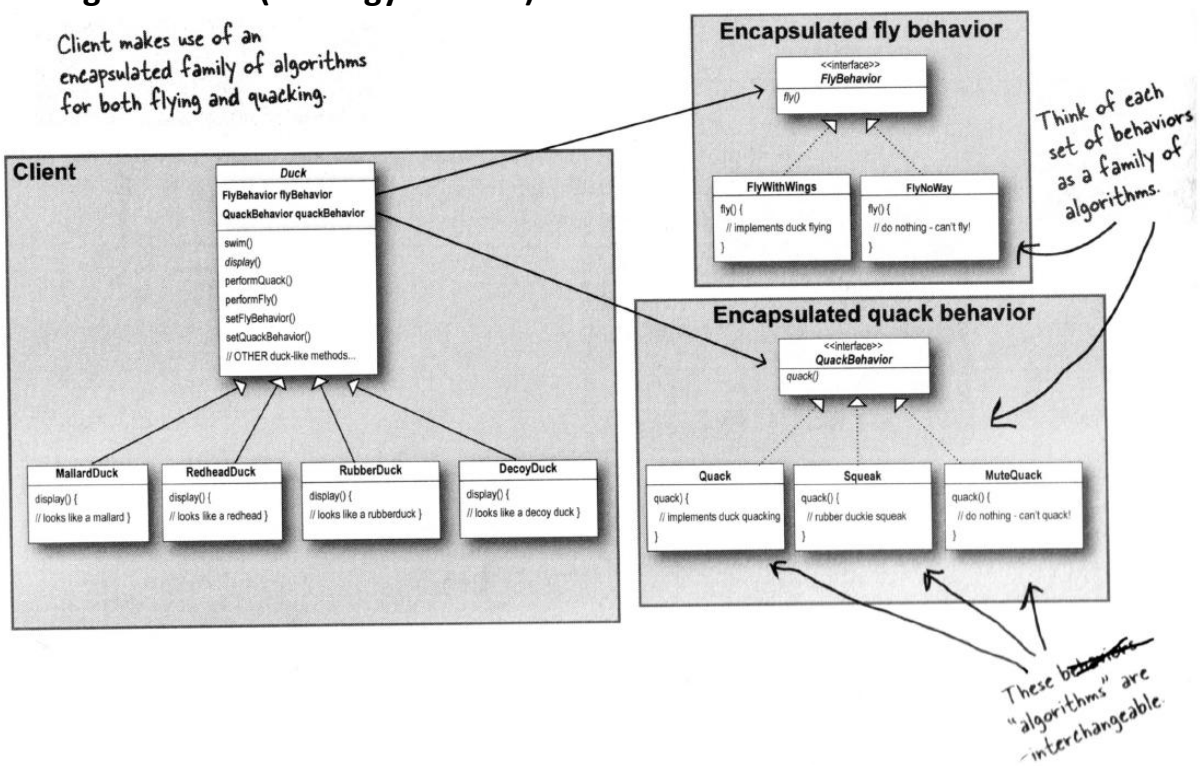


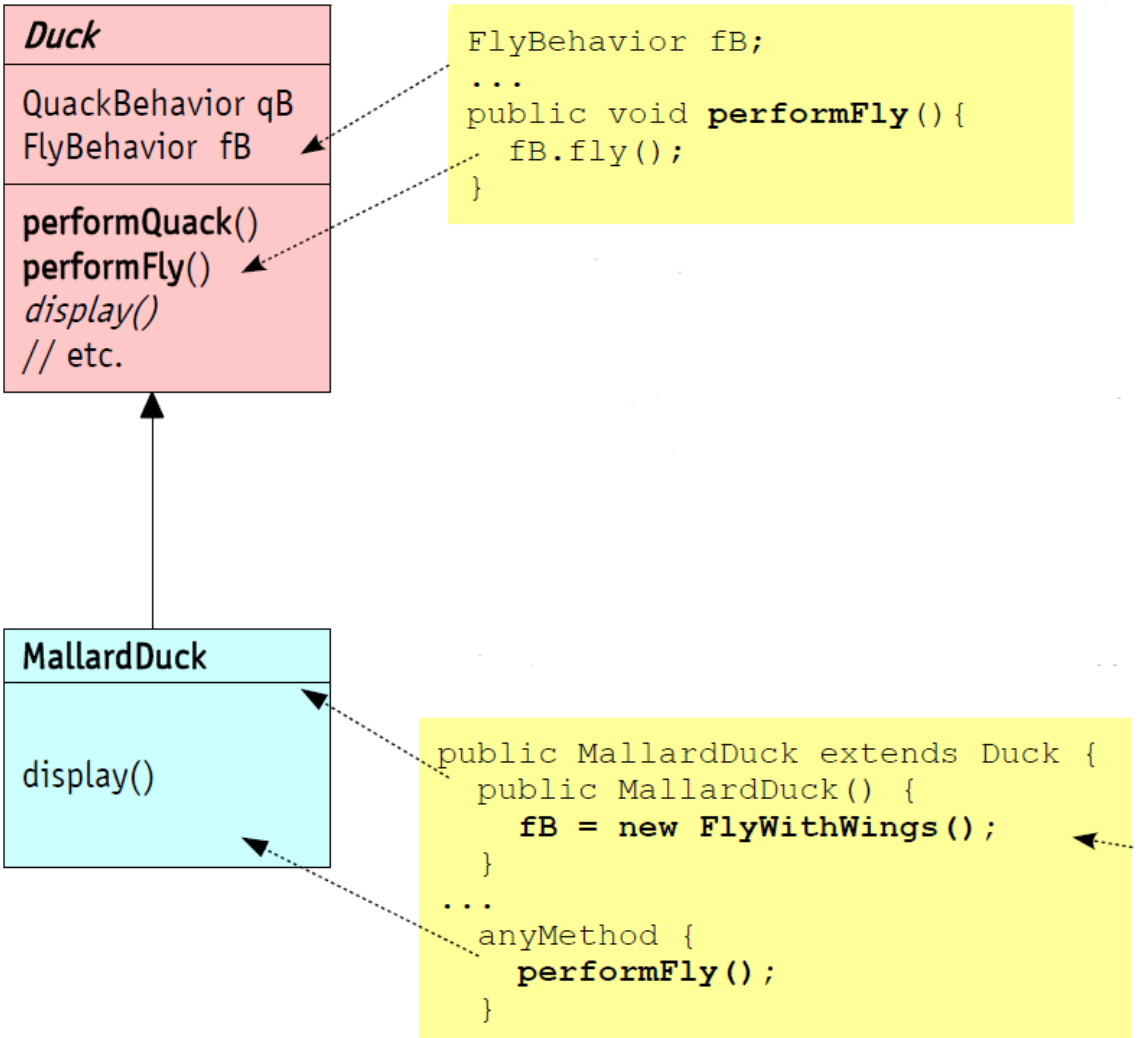
- AbstractClass
  - Definiert abstrakte primitive Operationen und implementiert die Schablonenmethode
- ConcreteClass
  - Implementiert die primitiven Operationen der abstrakten Oberklasse

Interaktionen
- ConcreteClass setzt voraus, dass die AbstractClass die invarianten Teile implementiert

Konsequenzen
- Grundlegende Technik zur Wiederverwendung von Code
- Für Klassenbibliotheken, um das gemeinsame Verhalten in Bibliotheksklassen darzustellen
- Realisieren das Hollywood-Prinzip "Don't call us, we'll call you"

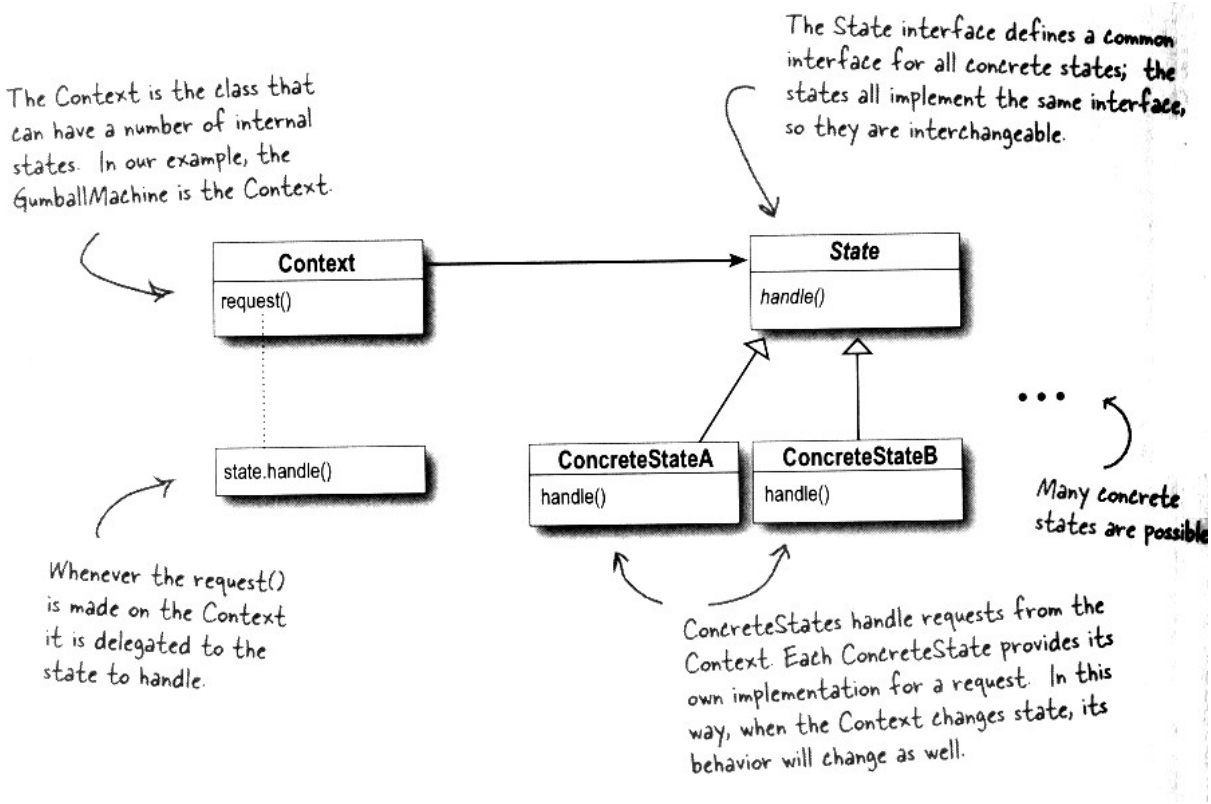
Strategie Muster (Strategy Pattern)

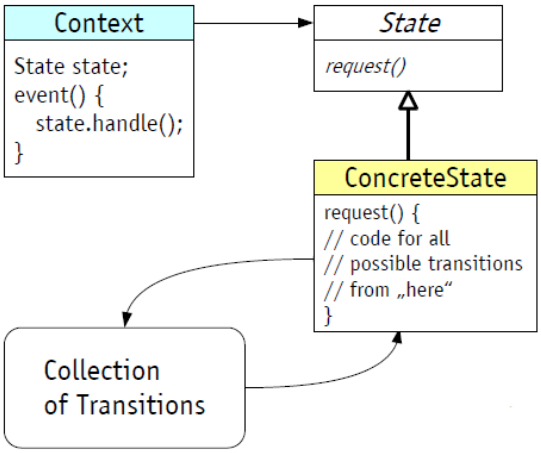




State Pattern

- Erlaubt es einem Objekt sein Verhalten gemäss seines Status zu ändern
- Erlaubt einem Objekt "seine Klasse zu ändern"





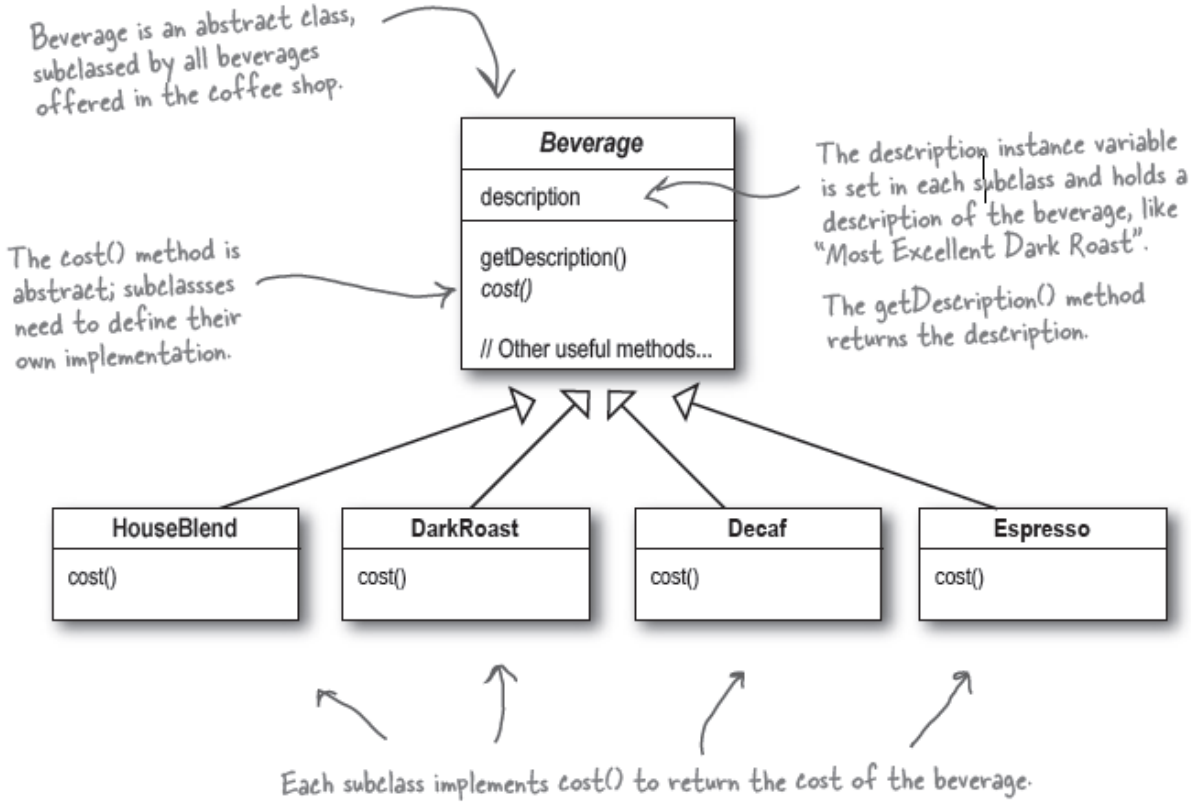
Feinheiten

- Tradeoff zwischen Context und State
- "State" kann zum Interface reduziert werden, wenn der Context den gesamten "Zustand" hält
- Der Context bzw. State kann den "Zustand" auch über Methoden anderer Klassen in Erfahrung bringen (Vorsicht: "externe" Abhängigkeiten)
- Eine Klasse pro Zustand: "sehr viele" Klassen?

Unterschied State ←→ Strategy

Strategy:	Zweck ist spezifisches Verhalten
State:	Zweck ist dynamische Variation des Verhaltens
Strategy:	Alternative zum exzessiven Gebrauch von "subclassing"
State:	Alternative zum Spaghetti-Code

Decorator Pattern



The Iterator Pattern

- Zugriff auf die Elemente einer Struktur (Aggregat), ohne sich um die Details der Realisierung des Aggregats zu kümmern
- Weg um z.B. eine Liste / Baum zu traversieren