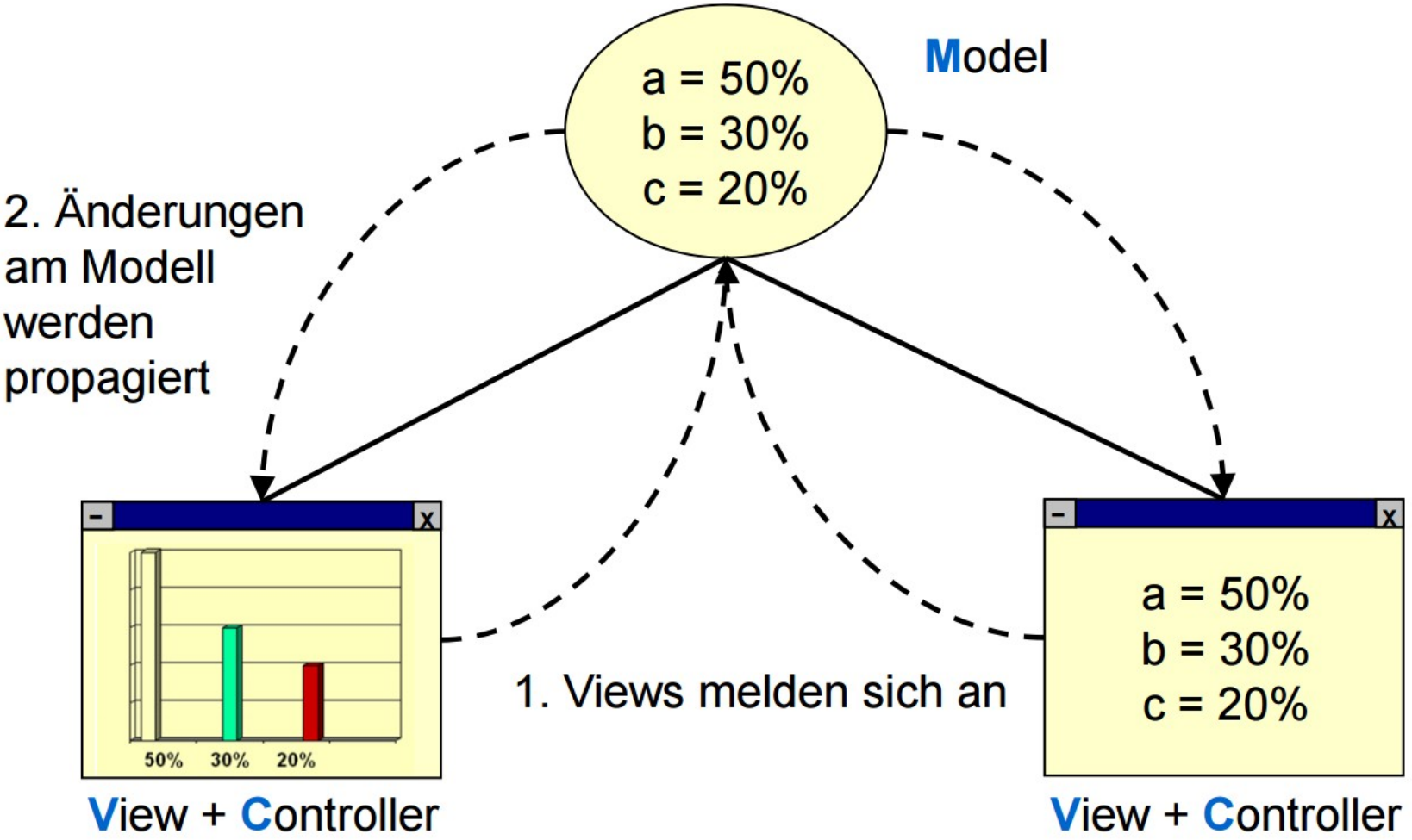
Entwurfsmuster I

Ziele

- Dokumentation von Lösungen wiederkehrender Probleme, um Programmierer bei der Softwareentwicklung zu unterstützen.
- Schaffung einer gemeinsamen Sprache, um über Probleme und ihre Lösungen zu sprechen.
- Bereitstellung eines standardisierten Katalogisierungsschemas um erfolgreiche Lösungen aufzuzeichnen

Das MVC


a = 50%
b = 30%
c = 20%
Model
2. Änderungen am Modell werden propagiert
50%
30%
20%
1. Views melden sich an
a = 50%
b = 30%
c = 20%
View + Controller
View + Controller

Das MVC

- Propagierung von Änderungen: Observer Pattern
  - Kommt z.B. auch bei Client/Server-Programmierung zur Benachrichtigung der Clients zum Einsatz
- Geschachtelte Views: Composite Pattern
  - View enthält weitere Views, wird aber wie ein einziger View behandelt.
- Reaktion auf Events im Controller: Strategy Pattern
  - Eingabedaten können validiert werden
  - Controller können zur Laufzeit gewechselt werden

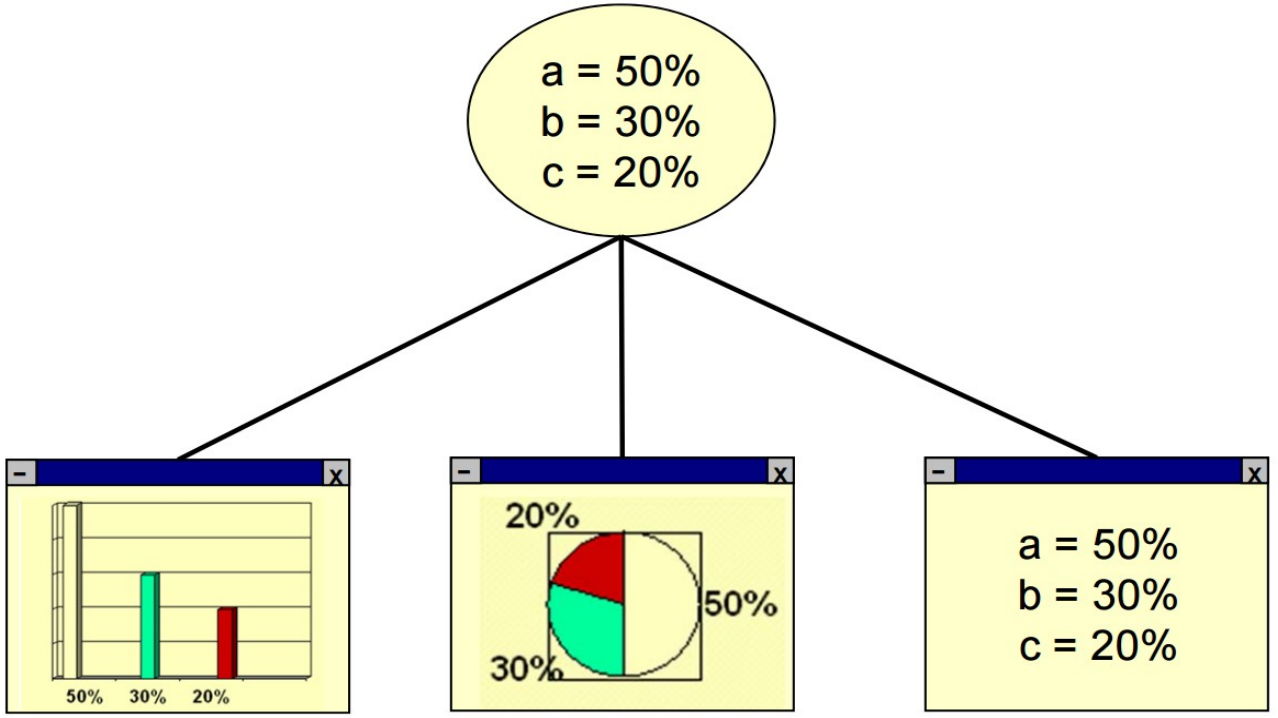
Observer

- Stellt eine 1:N Beziehung zwischen Objekten her
- Wenn das eine Objekt seinen Zustand ändert, werden die davon abhängigen Objekte benachrichtigt und entsprechend aktualisiert
- Verschiedene Objekte sollen zueinander konsistent gehalten werden
- Andererseits sollen sie dennoch nicht eng miteinander gekoppelt sein

Observer

Wenn in einer Sicht Änderungen vorgenommen werden, werden alle anderen Sichten aktualisiert

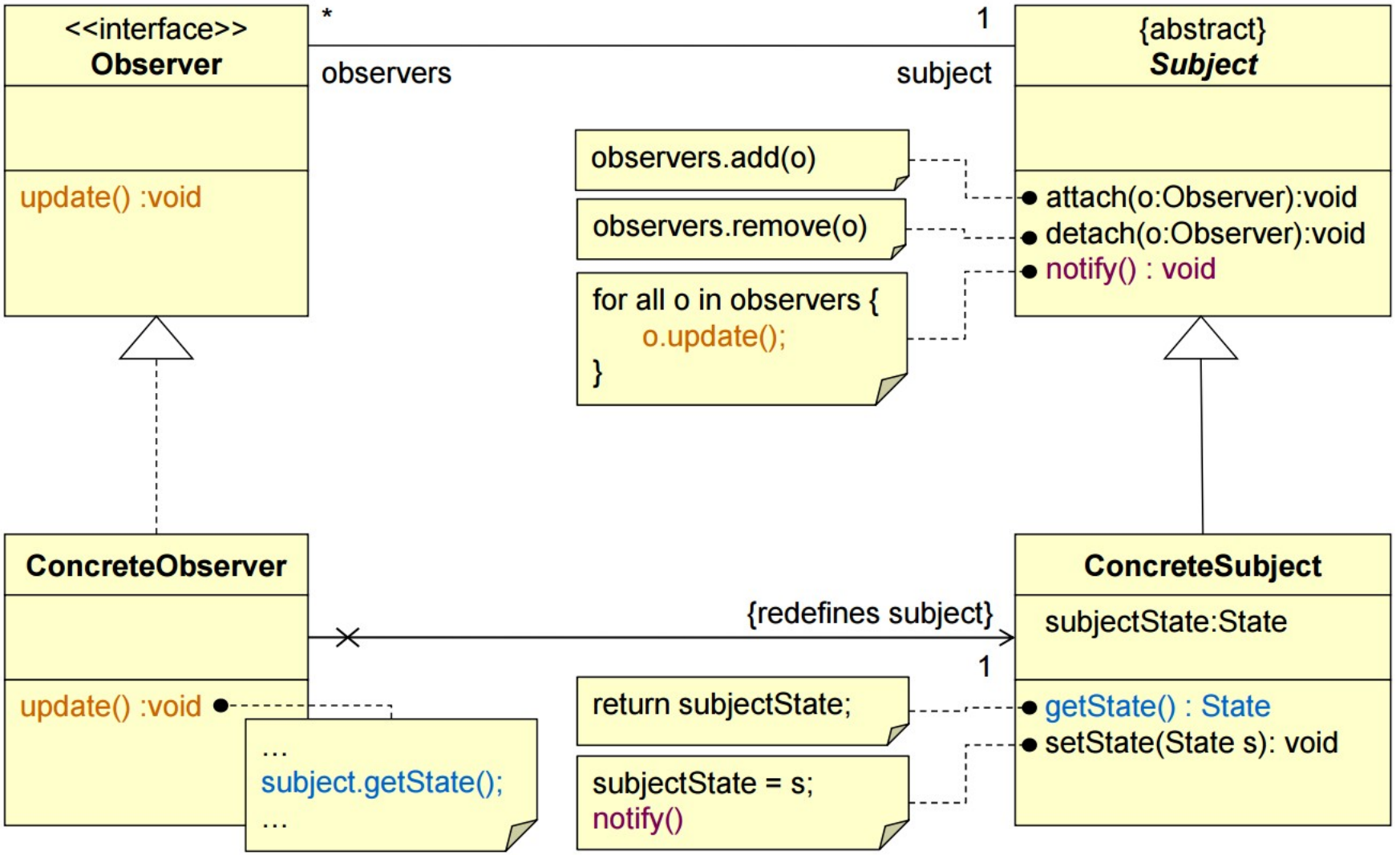
- Sichten sind aber unabhängig voneinander!



Kontext

- Abhängigkeiten
  - Ein Aspekt einer Abstraktion ist abhängig von einem anderen Aspekt
- Folgeänderungen
  - Änderungen an einem Objekt erfordert Änderungen an anderen Objekten
  - Es ist nicht bekannt, wie viele Objekte geändert werden müssen
- Lose Kopplung
  - Objekte sollen andere Objekte benachrichtigen können, ohne Annahmen über die Beschaffenheit dieser Objekte machen zu müssen

Observer - Pull



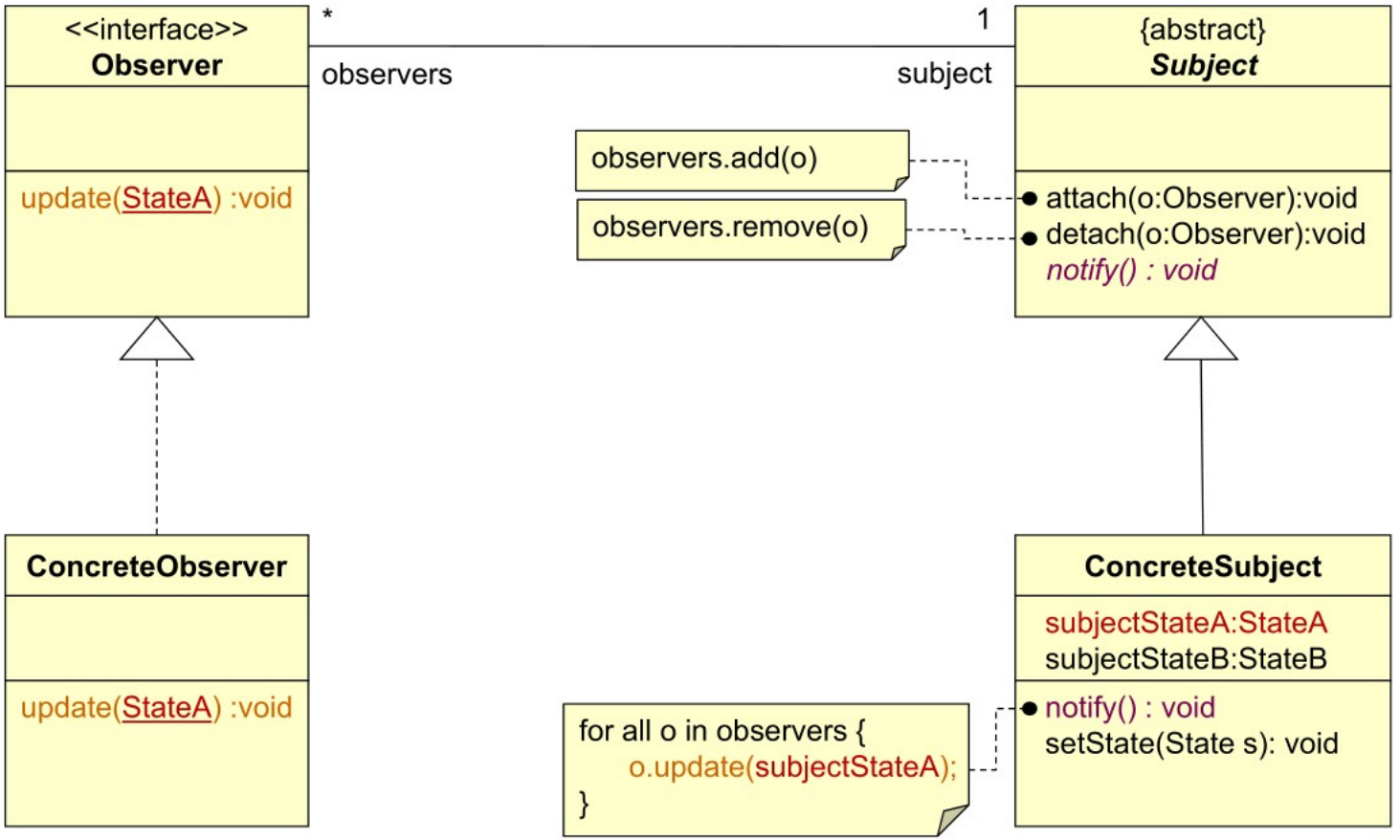
Modell

- Observer
  - update() -- auch: handleEvent
- Subject
  - attach(Observer o)
  - detach(Observer o)
  - notify()
  - setState(…)
  - getState()

Implementierung

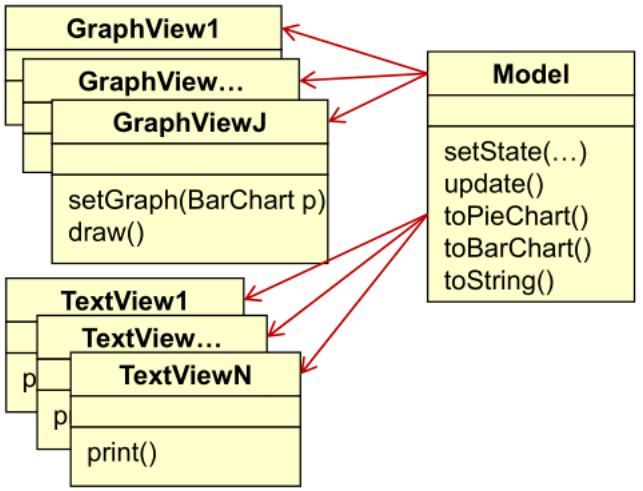
- „push“ versus „pull“
- Pull: Subjekt übergibt in „update()“ keinerlei Informationen, aber die Beobachter müssen sich die Informationen vom Subjekt holen
  - Berechnungen werden häufiger durchgeführt
- Push: Subjekt übergibt in Parametern von „update()“ detaillierte Informationen über Änderungen
  - Beobachter sind weniger wiederverwendbar (Abhängig von den Parametertypen)

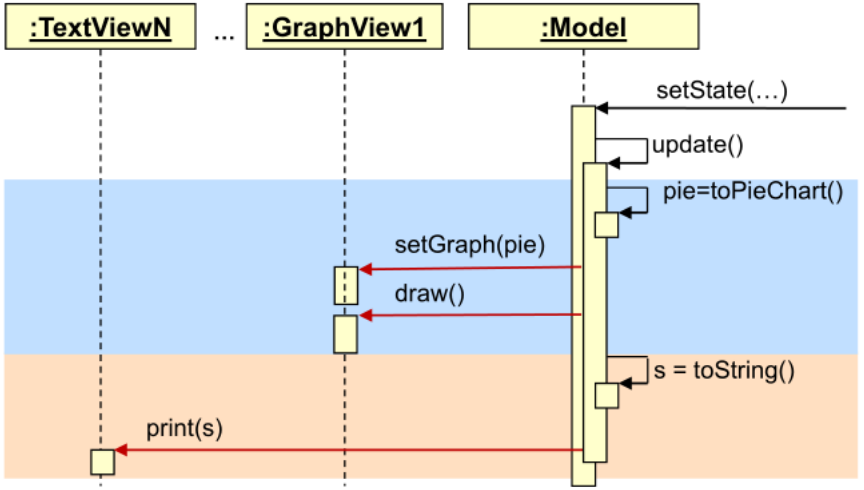
Oberver - Push



Abhängigkeiten ohne Observer

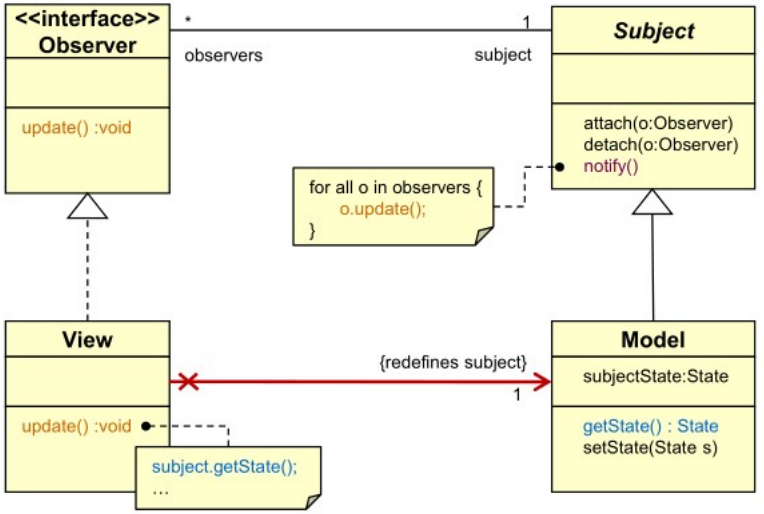
- myGraphView1.setGraph(this.toPieChart());
  myGraphView1.draw();
- myGraphView2.setGraph(this.toBarChart());
  myGraphView2.draw();
- ...

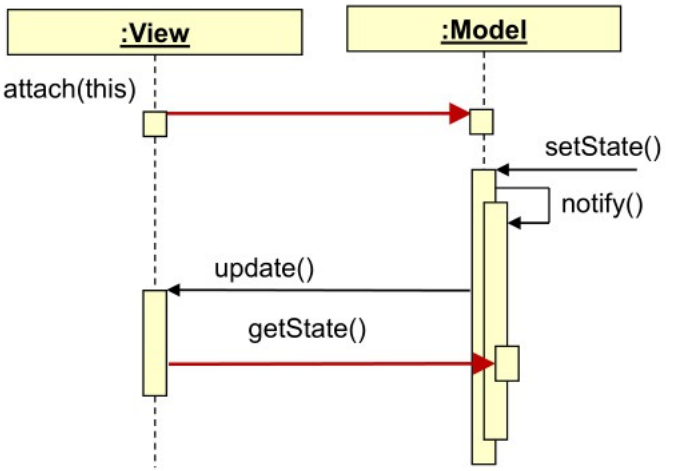




Abhängigkeiten mit Observer

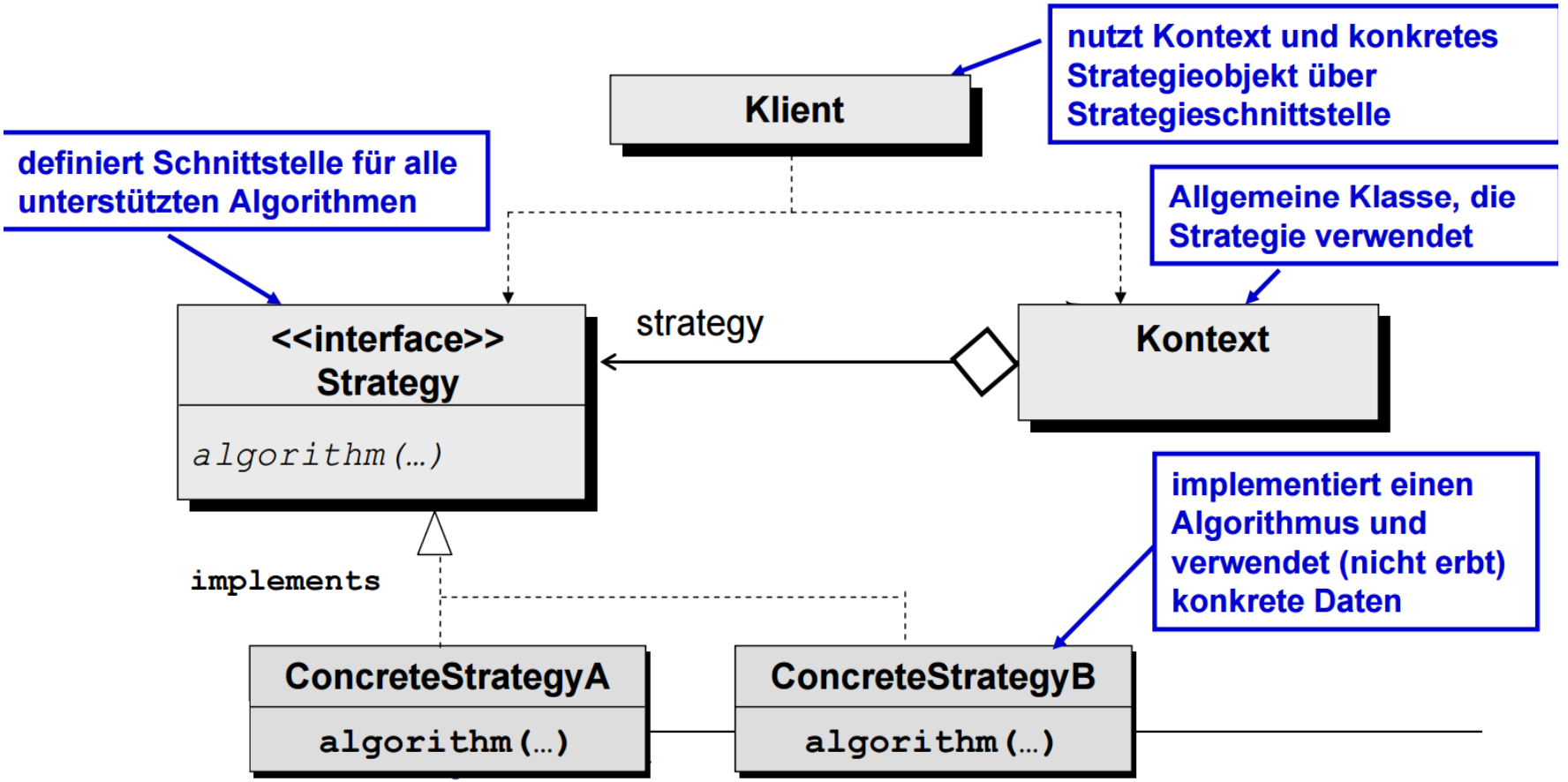
- model.getState1();
- model.getState2();





Strategy

- Das Strategiemuster entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen



Vorteile

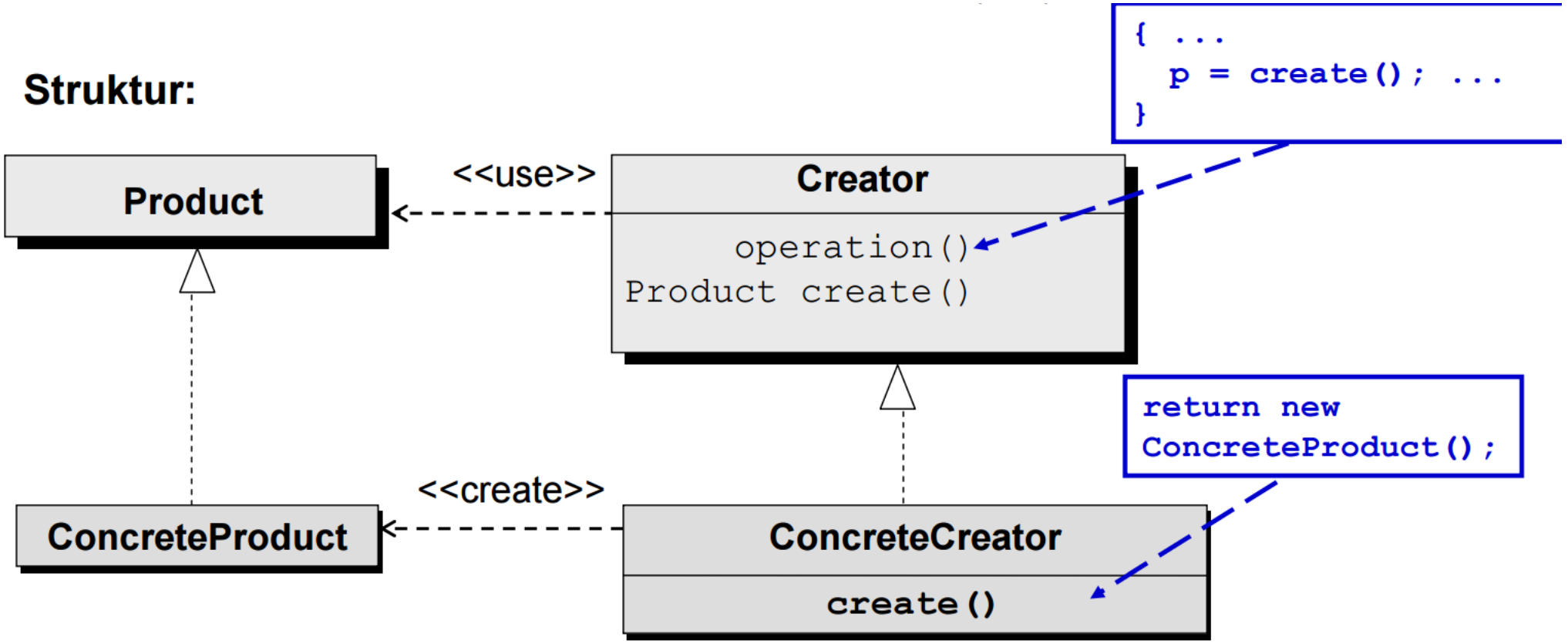
- Es wird eine Familie von Algorithmen definiert
- Strategien bieten eine Alternative zur Unterklassenbildung, helfen Mehrfachverzweigungen zu vermeiden und verbessern dadurch die Wiederverwendung
- Strategien ermöglichen die Auswahl aus verschiedenen AlgorithmenImplementationen (“Algorithmen-Polymorphie”) und erhöhen dadurch die Flexibilität

Nachteile

- Klienten müssen die unterschiedlichen Strategien kennen, um zwischen ihnen auswählen zu können
- Gegenüber der direkten Implementation der Algorithmen im Klienten erzeugen Strategien zusätzlichen Kommunikationsaufwand zwischen Strategie und Klient
- Die Anzahl der Objekte wird erhöht

Factory Method

Struktur:



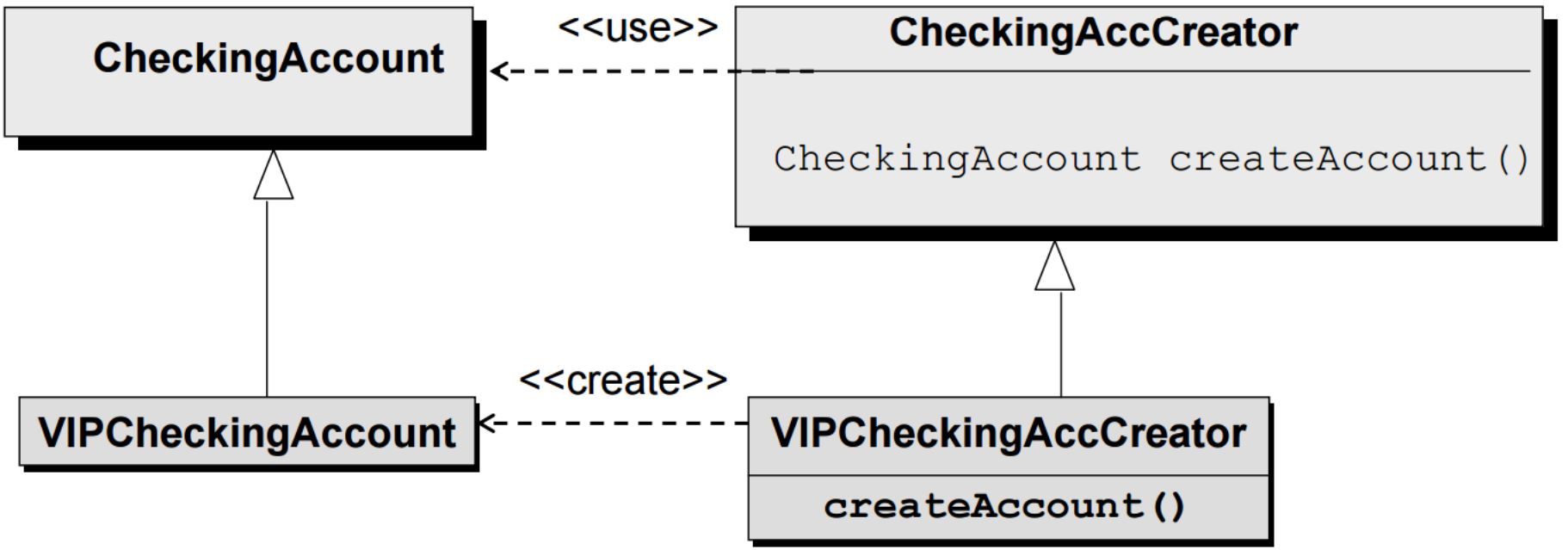
Factory Method

- Fabrikmethodenmuster definiert eine Schnittstelle zur Erzeugung eines Objektes, wobei es den Unterklassen überlassen bleibt, von welcher Klasse das zu erzeugende Objekt ist
- Beteiligte Klassen
  - Product
  - Creator
  - ConcreteProduct
  - ConcreteCreator

Factory Method

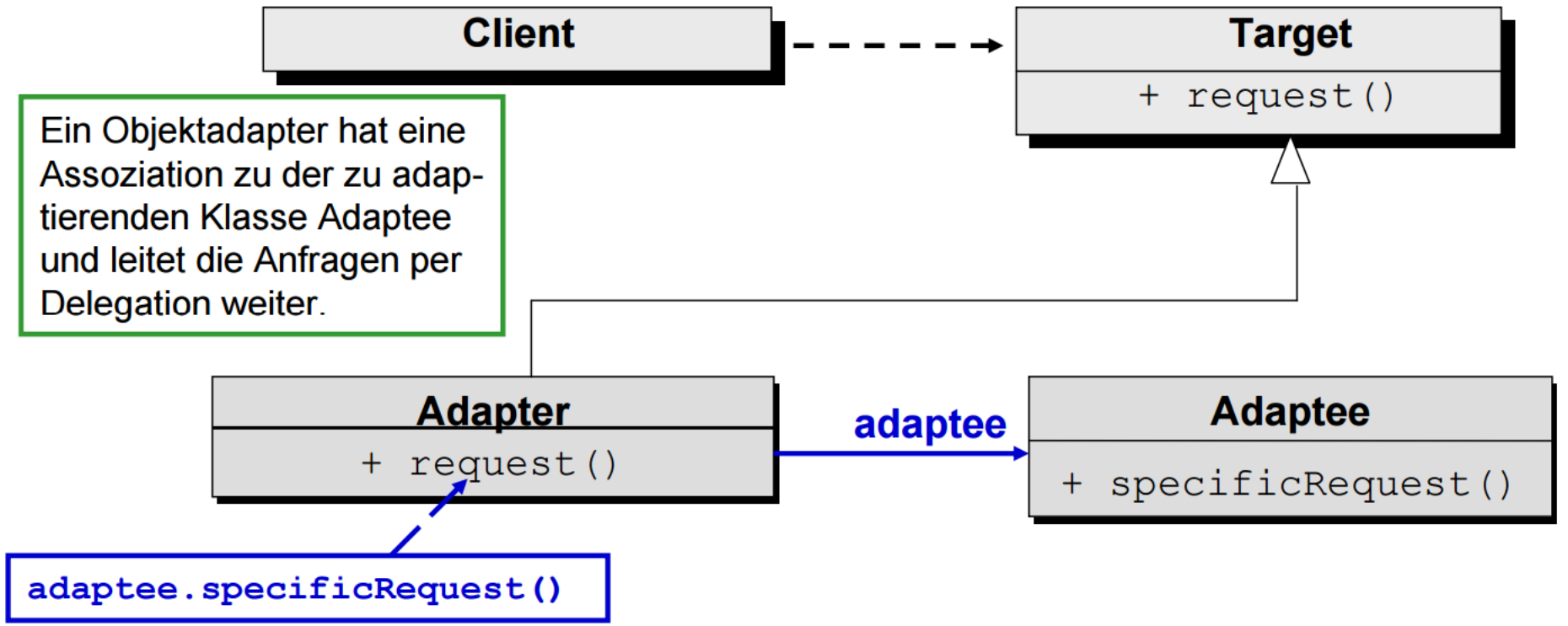
- Die abstrakte Methode create() sehr spezifisch: Sie erzeugt ein Objekt
- Das „Wie“ der Objekterzeugung ist hinter create() versteckt
- Fabrikmethoden entkoppeln ihre Aufrufer von Implementierungen konkreter Produktklassen
- Die Verwendung dieses Erzeugungsmusters läuft auf Unterklassenbildung hinaus

Fabrikmethode für Konten



Adapter

- übersetzt eine Schnittstelle in eine andere



Adapter

```
interface ITarget
{
  List<string> GetProducts();
}


public class VendorAdaptee
{
   public List<string> GetListOfProducts()
   {
      List<string> products = new List<string>();
      products.Add("Gaming Consoles");
      products.Add("Television");
      products.Add("Books");
      products.Add("Musical Instruments");
      return products;
   }
}


class VendorAdapter:ITarget
{
   public List<string> GetProducts()
   {
      VendorAdaptee adaptee = new VendorAdaptee();
      return adaptee.GetListOfProducts();
   }
}


class ShoppingPortalClient
{
   static void Main(string[] args)
   {
      ITarget adapter = new  VendorAdapter();
      foreach (string product in adapter.GetProducts())
      {
        Console.WriteLine(product);
      }
      Console.ReadLine();
   }
}
```

Vorteile

- Ein Klassenadapter passt genau eine Dienstklasse (Adaptee) an
- Ein Klassenadapter kann dadurch das Verhalten des Adaptees überschreiben
- Ein Klassenadapter wird eingesetzt, wenn ein Teil einer ganz konkreten Schnittstelle variiert werden soll

Zusammenfassung

- Entwurfsmuster sind Regeln oder Richtlinien, um häufig auftretende Probleme bei der Erstellung eines Programms zu lösen
- Das Fabrikmethodenmuster (Factory Method) definiert eine Schnittstelle zur Erzeugung eines Objektes, wobei es den Unterklassen überlassen bleibt, von welcher Klasse das zu erzeugende Objekt ist

Zusammenfassung

- Das Adaptermuster (Adapter, Wrapper) übersetzt eine Schnittstelle in eine andere. Dadurch können Klassen miteinander kommunizieren, die zueinander inkompatible Schnittstellen zur Verfügung stellen
- Das Strategiemuster (Strategy) entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen
- Das Beobachtermuster (Observer) ermöglicht die Weitergabe von Änderungen eines Objekts an abhängige Objekte

Entwurfsmuster II (GRASP)

Objektorientierter Entwurf

Beim Entwurf objektorientierter Programme hat man viele Entscheidungsmöglichkeiten

- Aufteilung in Packages
- Klassen und Vererbung
- Kapselung von Daten und Schnittstellen
- Datenstrukturen

Ziele

- Korrektheit
- Wiederverwendbarkeit
- Verständlichkeit
- Effizienz

Erfahrungswerte

- Design Patterns
- Anti-Patterns
- Design Smells
- Code Smells

Entwurfsmuster

Beschreibung

- eines Problems
- dessen Lösung
- Hinweise, wann diese Lösung anzuwenden ist und wie sie unter veränderten Rahmenbedingungen anzuwenden ist

Anwendung von GRASP

- „It is possible to communicate the detailed principles and reasoning required to grasp basic object design, and to learn to apply these in a methodical approach that removes the magic and vagueness“
- Anders als die GoF Entwurfsmuster
  - Was soll man beachten, um bessere Designergebnisse zu erhalten
  - Entwurfsmuster, um Klassen und Objekten Zuständigkeiten (responsibilities) zuzuweisen

Gang of Four

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

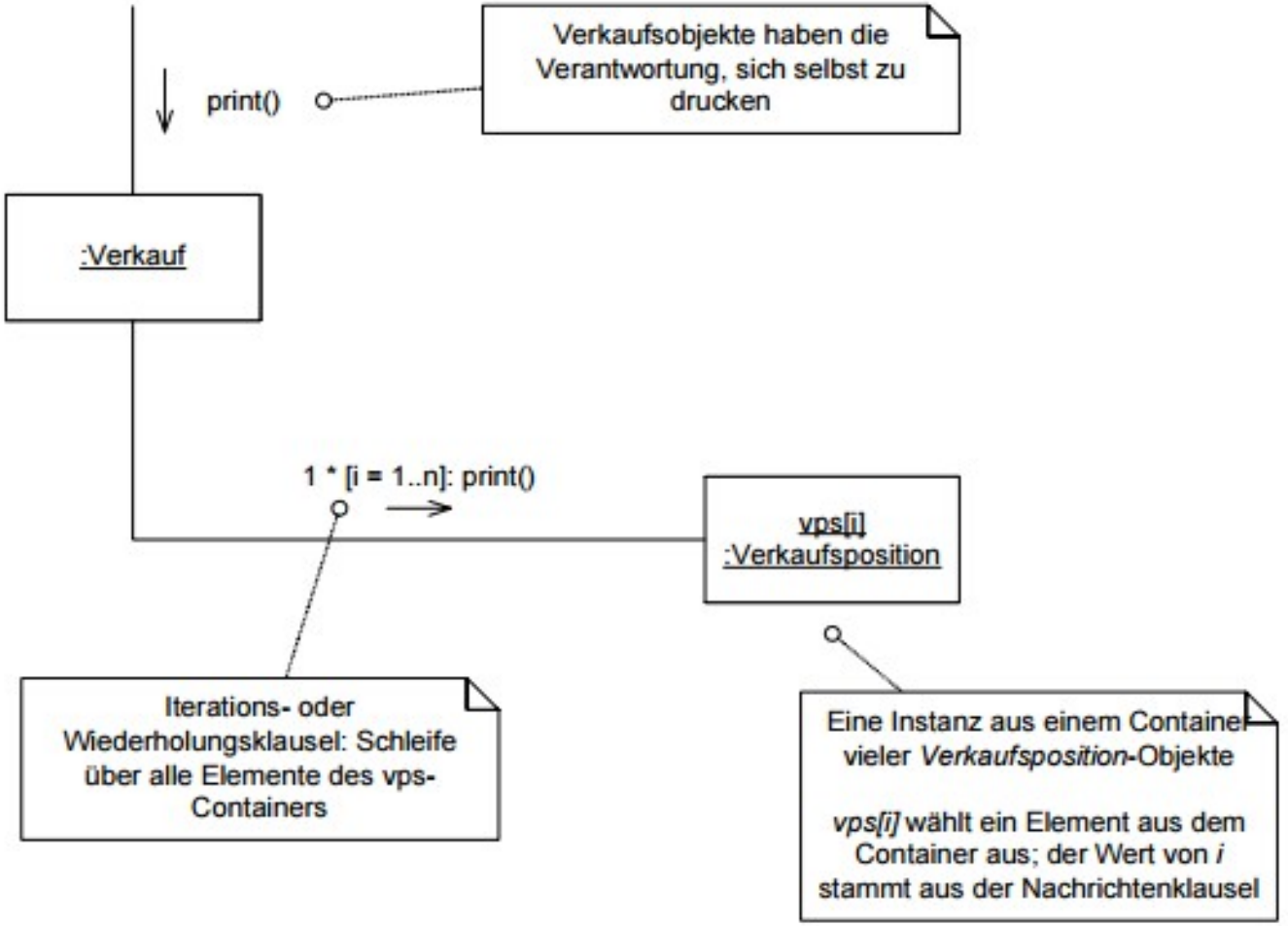
Gang of Four

- Beschreibt die Element, aus denen das Design besteht
- Beschreibt Beziehungen und Zusammenarbeit zwischen den Elementen
- Gibt kein konkretes Design oder eine Implementierung

Anwendung von GRASP - Verantwortung

- Verkauf-Objekte haben die Verantwortung zugeteilt bekommen, sich selbst zu drucken
  - Dies wird durch eine print Message ausgelöst und von der entsprechenden print Methode durchgeführt.
- Damit diese Verantwortlichkeit durchgeführt werden kann, bedarf es der Zusammenarbeit mit den Verkaufsposition-Objekten, die ihrerseits aufgefordert werden, sich selbst zu drucken.

Anwendung von GRASP - Verantwortung



Expert Pattern

- ein Expert Pattern bezeichnen

Pattern Name	**Expert**
Absicht	Gib derjenigen Klasse eine Zuständigkeit, die die *Information* hat, diese zu erfüllen
Problemlösung	Grundlegendes Prinzip der Zuordnung von Zuständigkeiten an Objekte.

- Warum Entwurfsmuster?
  - Entwurfsmuster zu definieren und mit Namen zu versehen hat viele Vorteile
  - Man versteht einander viel besser bei der Diskussion verschiedener Entwurfsalternativen

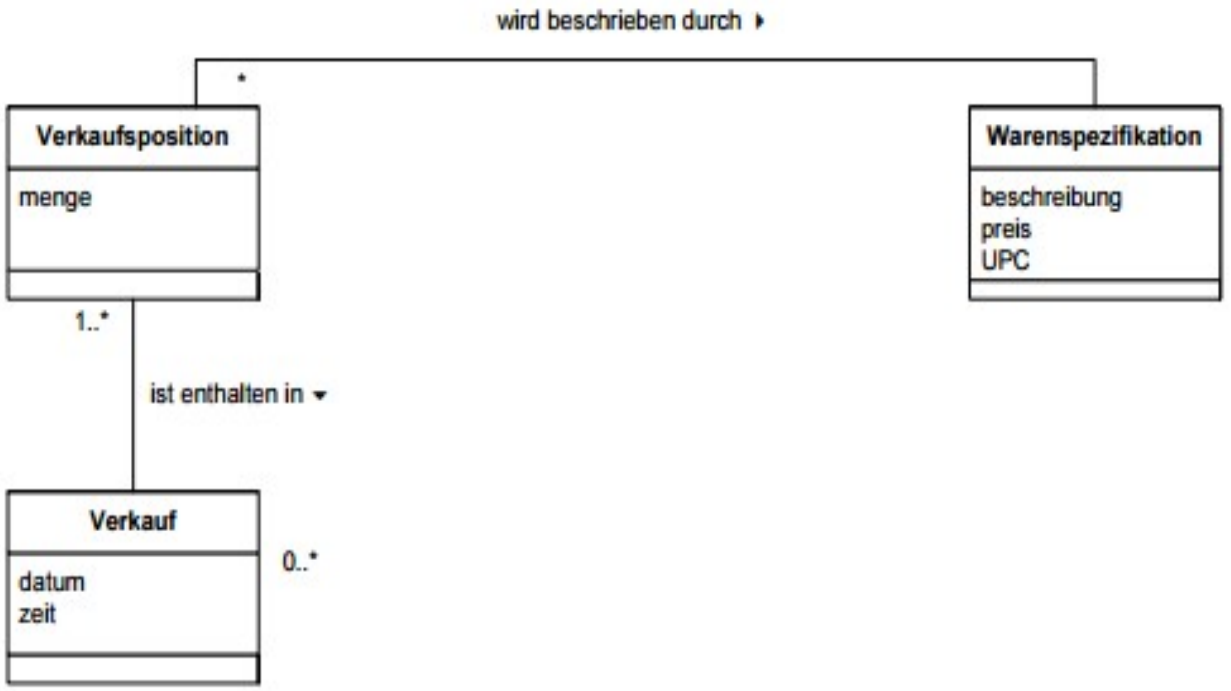
GRASP

GRASP-Patterns (General Responsibility Assignment Software Patterns):

- Expert
- Creator
- Low Coupling
- High Cohesion
- Polymorphismus
- Pure Fabrication
- Indirection

Information Expert

- Wer sollte zuständig sein, die Gesamtsumme des Verkaufs kennen?
- Gemäß dem Expert Pattern sollten wir nach der Klasse suchen, die alle Informationen besitzt, um die Gesamtsumme zu bestimmen



Information Expert

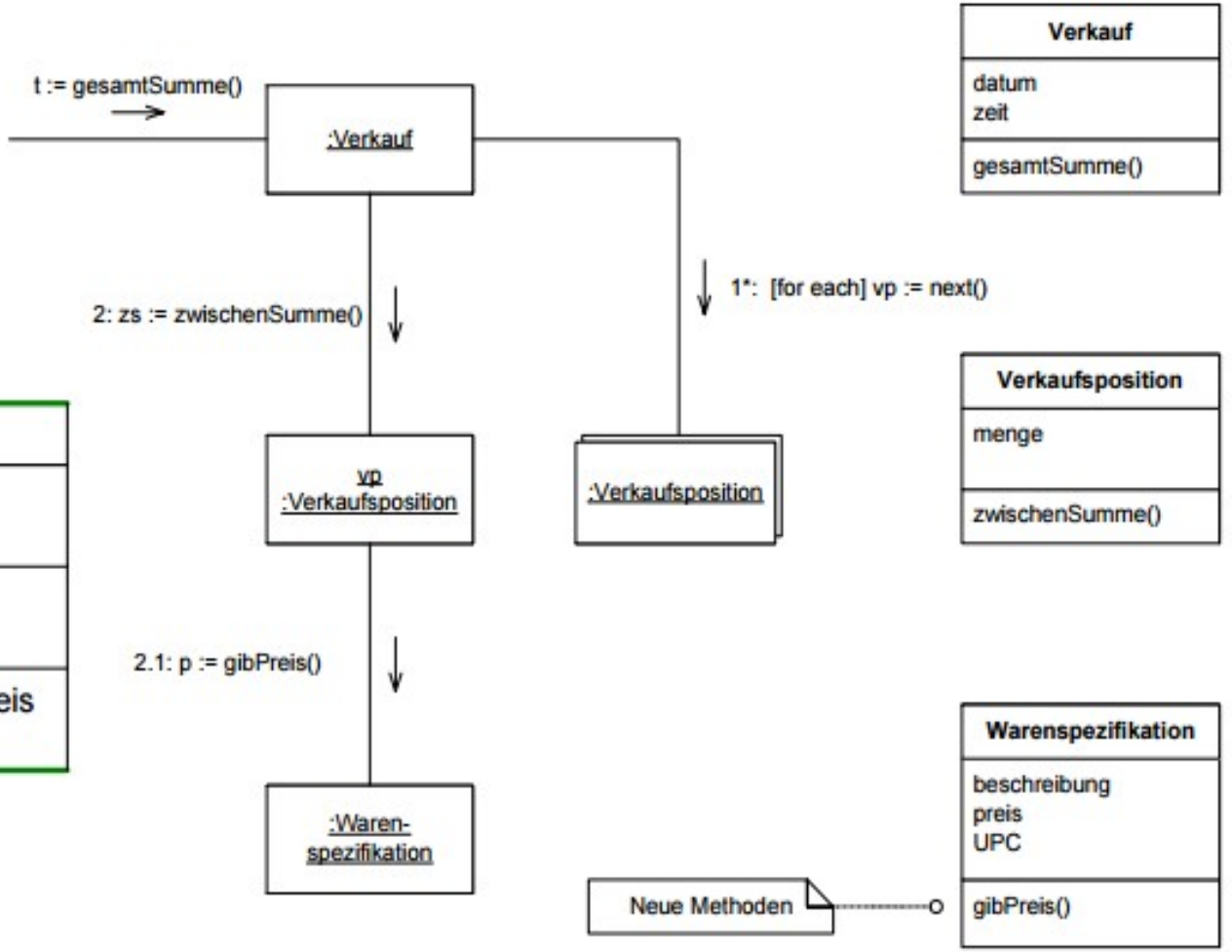
- Um die Gesamtsumme bestimmen zu können, benötigt man die Verkaufspositionen (menge) und die zugehörigen Warenspezifikation Objekte (preis)
- Alle Informationen laufen bei Verkauf zusammen
- Also ist Verkauf unser Informationsexperte

Information Expert

Methoden

3 Zuständigkeiten:

Klasse	Zuständigkeit
Verkauf	kennt die Gesamtsumme
Verkaufs-position	kennt die Zwischensumme
Warenspezi-fikation	kennt den Warenpreis



Information Expert

- Gib die Zuständigkeit dem Objekt, das alle notwendigen Informationen besitzt
- Entspricht dem grundlegenden Verständnis der objektorientierten Denkweise:
  - Objekte sollen Dinge tun, die mit den Informationen, die sie enthalten, zusammenhängen
- Oft nur Teilexperten, die zusammenarbeiten müssen, um das Gesamtbild zu erhalten
  - Die Informationen sind tatsächlich auf die beteiligten Klassen verteilt

Information Expert - Vorteile

- Datenkapselung ist sichergestellt, da Objekte ihre eigenen Informationen zur Erfüllung ihrer Aufgaben verwenden
- Dies unterstützt lose Kopplung:
  - Robustere und besser zu pflegende Systeme
- Verhalten wird über mehrere Klassen verteilt, die die Informationen besitzen:
  - leichtgewichtige Klassen, die leichter zu verstehen und zu pflegen sind
  - hoher Zusammenhalt

Creator

B ist Creator von A Objekten

- Weise der Klasse B die Zuständigkeit zu, Instanzen der Klasse A zu erzeugen, wenn eine der folgenden Aussagen zutrifft:
  - B enthält A Objekte (Komposition)
  - B nutzt intensiv A Objekte
  - B besitzt die Initialisierungsdaten für A Objekte, die bei deren Erzeugung gebraucht werden

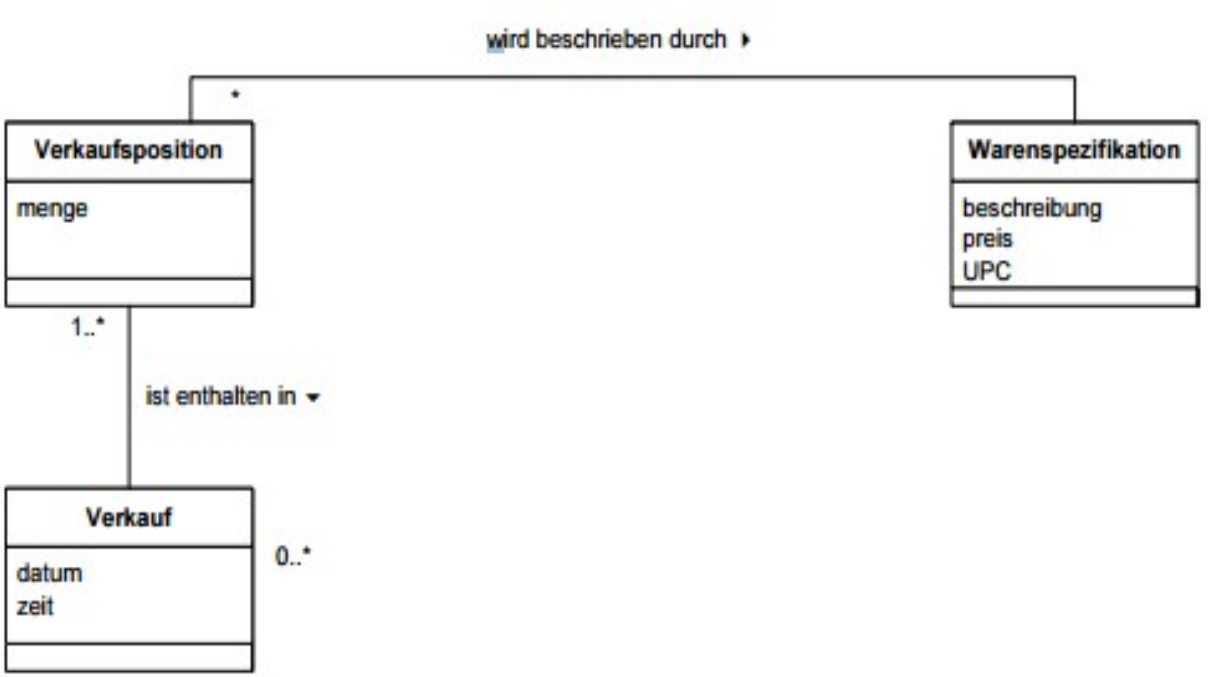
Creator

Problem

- Wer sollte für die Erzeugung neuer Instanzen einer bestimmten Klasse zuständig sein?
  - Erzeugung neuer Instanzen ist in objektorientierten Anwendungen häufig
  - Daher sollte man ein allgemeines Prinzip haben, die Zuständigkeit für Objekterzeugung zu vergeben

Creator

- Verkaufspositionen creator?
- Nach dem Creator Pattern sollten wir nach einer Klasse suchen, die Verkaufspositionen aggregiert, enthält usw.
  - natürlich: Verkauf



Creator

- Durch dieses Pattern wird ein Creator gesucht, der ohnehin mit dem erzeugten Objekt verbunden sein muss
  - Dadurch wird wiederum lose Kopplung unterstützt.
- Ein anderer Hinweis dafür, einer Klasse die Zuständigkeit als Creator zuzuweisen, ist wenn sie die Initialisierungsdaten für das zu erzeugende Objekt besitzt
  - Expert Pattern?

Creator

- Wer sollte für einen Verkauf die Zahlung-Instanz erzeugen?
  - Diese muss mit der Gesamtsumme initialisiert werden
  - Da Verkauf diese kennt, ist Verkauf ein erster Kandidat, Zahlung-Objekte zu erzeugen
- Lose Kopplung:
  - Durch die Erzeugung wird die Kopplung nicht verstärkt

Low Coupling

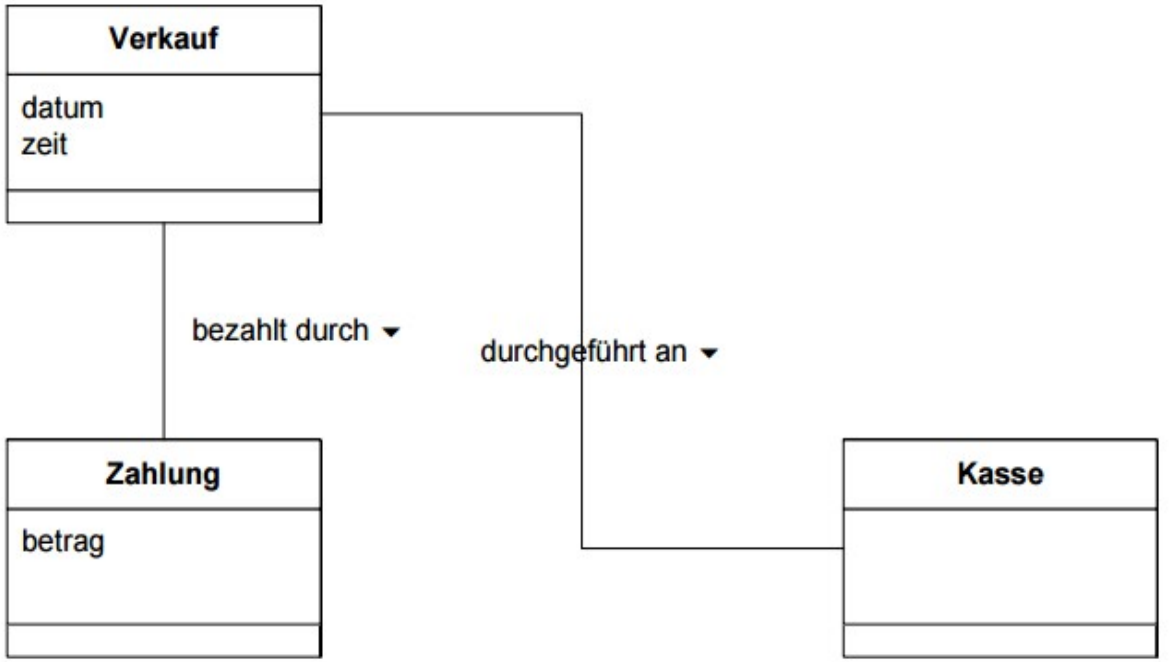
- Weise eine Zuständigkeit derart zu, dass die Kopplung niedrig (lose) bleibt
- Wie unterstützt man geringe Abhängigkeit und vermehrte Wiederverwendbarkeit?
- Kopplung ist ein Maß, wie stark (intensiv) eine Klasse mit einer anderen verbunden ist
- Eine Klasse mit geringer Kopplung ist nicht von zu vielen anderen Klassen abhängig

Low Coupling

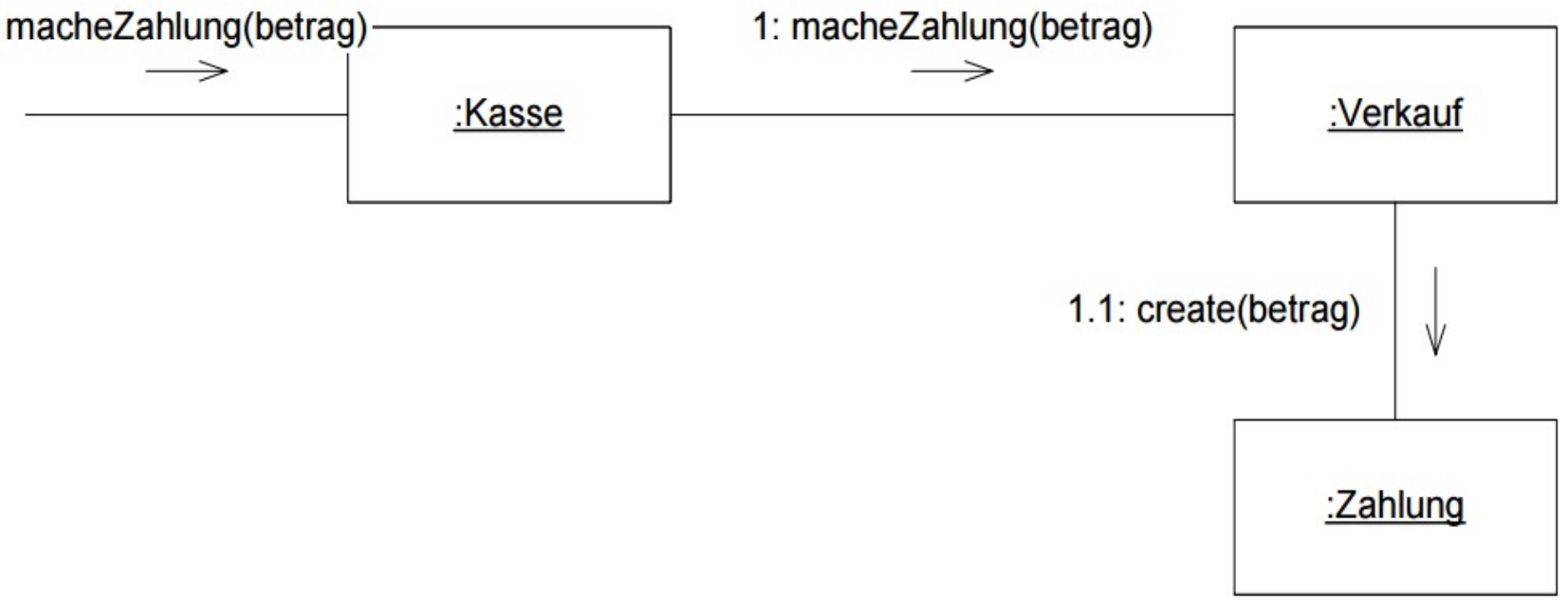
- Eine Klasse mit hoher (starker) Kopplung hängt von vielen anderen Klassen ab
- Dies ist nicht wünschenswert
- Solche Klassen können unter den folgenden Problemen leiden
  - Sind isoliert schwer zu verstehen
  - Sind schwerer wiederzuverwenden
  - Änderungen in verbundenen Klassen rufen lokale Änderungen hervor

Low Coupling

- wir haben eine Instanz von Zahlung zu kreieren und diese mit Verkauf zu verknüpfen
- Welche Klasse sollte dafür zuständig sein?



Low Coupling



Low Coupling

- Das Prinzip der geringen Kopplung muss bei allen Entwurfsentscheidungen berücksichtigt werden
- Formen der Kopplung zwischen ClassX und ClassY
  - ClassX hat ein Attribut, das vom Typ ClassY ist oder sich auf eine ClassY Instanz bezieht
  - ClassX hat eine Methode, die, wie auch immer, ClassY oder eine Referenz auf ClassY verwendet
  - ClassX ist direkt oder indirekt eine Subklasse von ClassY

Low Coupling

- Low Coupling unterstützt den Entwurf von Klassen die unabhängiger voneinander sind
  - der Einfluss auf andere Klassen durch Änderungen wird reduziert
  - die Klassen werden dadurch besser wiederverwendbar
- Low Coupling muss immer in Verbindung mit anderen Designkriterien gesehen werden
- Die Reduktion der Kopplung ist nicht so wichtig wenn...?

Low Coupling

- Natürlich entspricht eine extrem geringe Kopplung nicht dem objektorientierten Paradigma
  - Objekte müssen miteinander kommunizieren, um eine „lebendige“ Anwendung auszumachen
- Ein moderates Maß an Kopplung ist normal, ja sogar notwendig, um objektorientierte Anwendung zu erstellen

High Cohesion

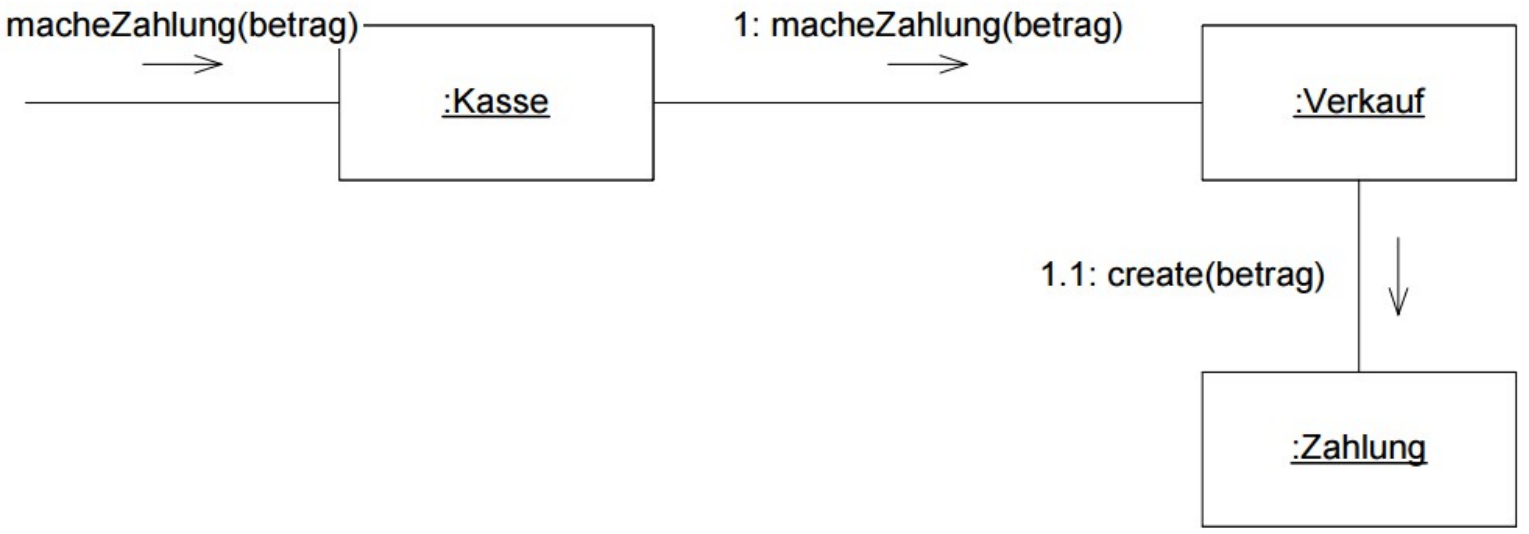
- Weise eine Zuständigkeit derart zu, dass der Zusammenhalt hoch bleibt
- Wie kann auch zunehmende Komplexität gemeistert werden?
- Mit (funktionalem) Zusammenhalt ist hier gemeint, wie stark die Verantwortlichkeiten einer Klasse einen inneren Zusammenhang haben
  - Oft realisieren einzelne Klassen mit hohem Zusammenhalt nicht unmäßig umfangreiche Arbeit

High Cohesion

- Eine Klasse mit niedrigem Zusammenhalt realisiert viele unzusammenhängende Aktivitäten oder tut überhaupt zuviel (alleine)
- Solche Klassen sind nicht wünschenswert
  - Schwer zu verstehen
  - Schwer wiederzuverwenden
- Klassen mit geringem Zusammenhalt haben manchmal Aufgaben selbst übernommen, die besser an andere Objekte delegiert würden

High Cohesion

- High functional cohesion is, if the elements of a component (such as a class) „all work together to provide some well-bounded behavior“



High Cohesion

- Sehr geringer Zusammenhalt
  - Eine Klasse alleine für viele Dinge aus verschiedenen funktionalen Bereichen zuständig.
  - Besser: Zwei verschiedene Familien von Klassen
- Geringer Zusammenhalt
  - Eine Klasse alleine zuständig für komplexe Aufgaben in einem funktionalen Bereich
  - Besser: Aufteilen auf mehrere leichtgewichtigere Klassen, die sich die Arbeit aufteilen

High Cohesion

- Hoher Zusammenhalt
  - Eine Klasse hat mittlere Zuständigkeiten in einem funktionalen Bereich und arbeitet mit anderen Klassen zur Erreichung ihrer Aufgaben zusammen
- Moderater Zusammenhalt
  - Eine Klasse hat leichtgewichtige und alleinige Zuständigkeiten in einigen verschiedenen funktionalen Gebieten, die zwar logisch mit dem Konzept der Klasse korreliert sind, jedoch voneinander unabhängig sind

High Cohesion

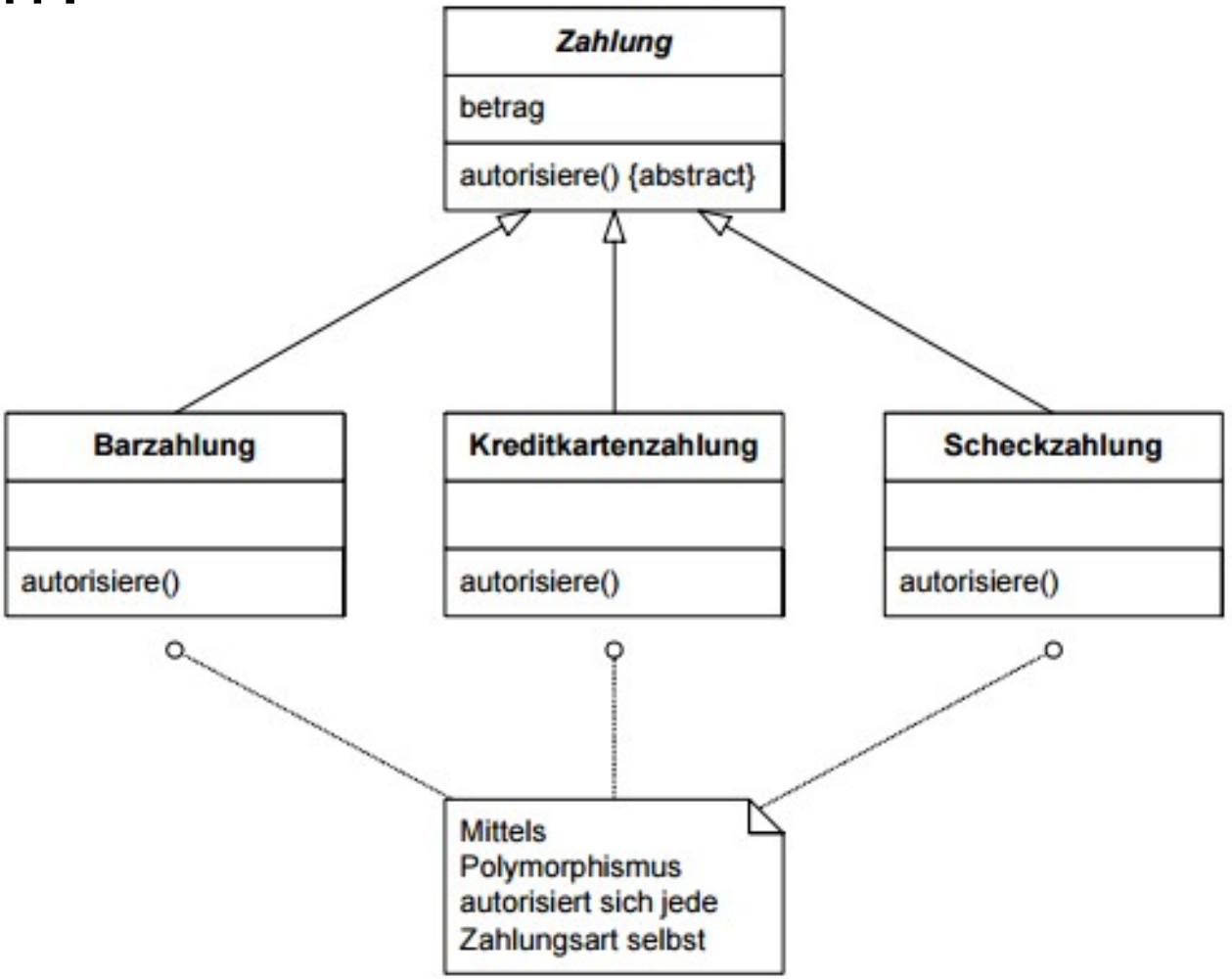
- Eine Klasse mit hohem Zusammenhalt hat relativ wenige Methoden, die aber funktional stark miteinander korreliert sind.
- Die Klasse führt nicht allzu viele Aufgaben selbst durch.
- Sie kollaboriert mit anderen Klassen, um den Aufwand für umfangreichere Aufgaben aufzuteilen

Polymorphismus

- Erweiterungen des Programms bedingen Änderungen an mehreren Stellen
- Gib ähnliche Services in verschiedenen Klassen denselben Namen, aber variiere die Implementierung
- Die verschiedenen Klassen müssen i.a. in einer gemeinsamen Vererbungshierarchie sein
- Zweck??????

Polymorphismus

- Wo sollte die autorisiere Methode implementiert werden?




Pure Fabrication

Pure Fabrication

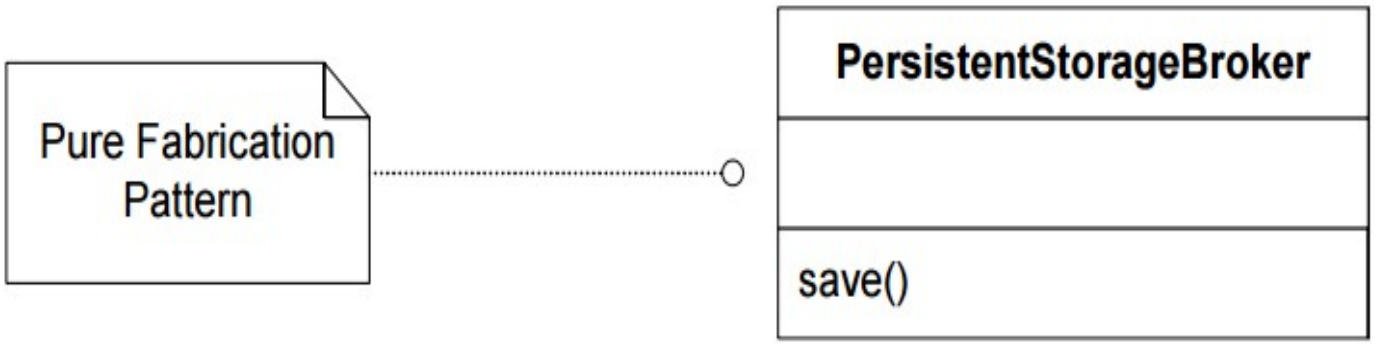
- Weise eine Menge stark zusammenhängender Zuständigkeiten einer künstliche Klasse zu, die nichts Reales in der Anwendungsdomäne repräsentiert
- Etwas, das nur dazu dient, hohe Zusammenhalt, schwache Kopplung und gute Wiederverwendbarkeit zu unterstützen
- Erfindung der Imagination. Design ist üblicherweise sehr sauber

Pure Fabrication

- Angenommen wir benötigen Unterstützung, um eine Verkaufs-Instanz in eine relationale Datenbank zu schreiben
- Zusammenhalt?
- Objekte in einer relationalen Datenbank zu speichern, ist eine häufig benötigte Zuständigkeit
  - Diese sollte nicht jeder Klasse der Anwendungsdomäne gegeben werden, da dadurch wahrscheinlich sehr viel Code dupliziert würde
  - Wiederverwendbarkeit wäre gering

Pure Fabrication

- Vernünftige Lösung?
- Zusätzliche Klasse, deren einzige Zuständigkeit es ist, irgendwelche Objekte auf irgendeinem persistenten Medium zu speichern



Pure Fabrication

- saubere Entwurf
- Die Klasse PersistenStorageBroker ist relativ gut zusammenhängend
  - Einzige Verantwortlichkeit: Speichern von Objekten
- Die Klasse PersistenStorageBroker ist eine sehr generische und wiederverwendbare Klasse

Pure Fabrication

- Pure Fabrication Klassen orientieren sich an zusammenhängenden Funktionalitäten und sind daher eher funktions-zentrierte Klassen
- Viele existierenden Design Patterns sind Pure Fabrication Klassen: Adapter, Visitor, usw
- Der Geist der Objektorientierung kann durch diese funktionsorientierte Argumentation verletzt werden

Indirection

- Weise einem Objekt eine Zuständigkeit zu, so dass es zwischen anderen Komponenten oder Klassen vermittelt, damit diese nicht direkt gekoppelt sind
- Dadurch wird eine Indirection zwischen die anderen Komponenten eingefügt
- Die Klasse PersistentStorageBroker ist auch ein Mittler zwischen einen Objekt und dem Datenbanksystem

Indirection

- Annahme: Unser Kassenterminal muss ein Modem verwenden, um Kreditkartenzahlungen zu autorisieren
  - Das Betriebssystem stellt eine low-level API dafür zur Verfügung
  - Die Klasse KreditAutorisierungsService ist dafür zuständig, mit dem Modem zu kommunizieren
- Würde KreditAutorisierungsService direkt das low-level API aufrufen, wäre diese Klasse eng an diese sehr spezielle API gekoppelt
- Wollte man diese Klasse auf ein anderes Betriebssystem portieren, würde sie massive Modifikationen benötigen

Indirection

- Klasse Modem