

### Erzeugungsmuster (creational patterns)

- Abstraktion der Erzeugungsmechanismen von Objekten
- System wird unabhängig davon wie seine Objekte erzeugt, verküpft und repräsentiert
- Klassennmuster: Vererbung von allen die erzeugten Objekte
- Objektmuster: Umwandlung der Instanzierung an andere Zeigt auf

### Definition

Jedes Pattern (Muster) beschreibt ein Problem, das in unserem Umfeld immer wieder auftritt, und beschreibt dann den Kern einer Lösung für dieses Problem in einer Art und Weise, dass diese Lösung millionenfach wiederverwendet werden kann, ohne dass sie auch nur zweimal identisch ist.

Alexander et al.: A Pattern Language, 1977



### Strukturmuster (structural patterns)

- fassen Klassen und Objekte zu größeren Strukturen zusammen
- Klassennmuster: Zusammenfassen von Schnittstellen und Implementierungen
- Objektmuster: Eindringen von Objekten in eine Struktur um neue Funktionalität zu erhalten

# Design Patterns

Vortrag von Svenja Nußbaum

### Verhaltensmuster (behavioral patterns)

- beschreiben die Interaktion zwischen Objekten und komplexen Kontrollflüssen
- Klassennmuster: Aufteilung von Kontrolle auf verschiedene Klassen durch Vererbung
- Objektmuster: Verwendung von Komposition an Stelle von Vererbung

### Und was haben wir jetzt davon?

- Warum das Bild neu entsteht?
- Häufig gemeinsame Fehler werden verhindert
- Änderungen an einem Teil ändern nicht den anderen
- Effizientere Verwendung des Codes
- Einsparung von Speicherplatz durch Verzweigung
- Minderung der programmierten Komplexität durch geschickte Strukturierung und Konsistenzprüfung

Projektpraktikum Computergrafik

Thema: Game Engine

Wintersemester 2014/2015

Universität Koblenz- Landau



Prezi

# Definition

Jedes Pattern (Muster) beschreibt ein Problem, das in unserem Umfeld immer wieder auftritt, und beschreibt dann den Kern einer Lösung für dieses Problem in einer Art und Weise, dass diese Lösung millionenfach wiederverwendet werden kann, ohne dass sie auch nur zweimal identisch ist.

Alexander et al.: A Pattern Language, 1977



# Klassenmuster

- beschäftigen sich mit den Beziehungen zwischen Klassen und deren Subklassen
- Beziehungen bauen auf Vererbung auf und stehen zur Compile- Zeit fest

# Objektmuster

- beschäftigen sich mit Beziehungen zwischen Objekten
- Beziehungen können zur Laufzeit verändert werden und sind dynamischer



# Dokumentation eines Pattern

- Name des Pattern
- Problembeschreibung
- Beschreibung der Lösung
- Konsequenzen

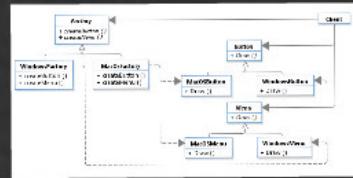
# Erzeugungsmuster (creational patterns)

- Abstraktion der Erzeugungsprozesse von Objekten
- System wird unabhängig davon wie seine Objekte erzeugt, verknüpft und repräsentiert werden
- Klassenmuster: Vererbung variiert die erzeugten Objekte
- Objektmuster: Übertragung der Instanziierung an andere Objekte





Wie sieht die GUI jetzt aus?



#### Problembeschreibungen

- ein System soll unabhängig davon sein, wie seine Produkte (Objekte) erzeugt, zusammengesetzt und repräsentiert werden
- ein System soll mit einer oder mehreren Produktfamilien konfiguriert werden
- eine Gruppe von verwandten Produkten soll erzeugt und gemeinsam genutzt werden
- in einer Klassenbibliothek sollen die Schnittstellen von Produkten ohne deren Implementierungen bereitgestellt werden

# Abstract Factory Pattern

auch: Kit

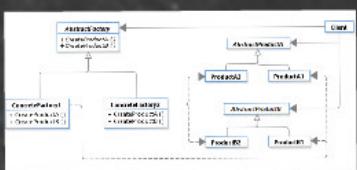
Bereitstellen einer Schnittstelle zur Erzeugung einer Familie von verwandten oder voneinander abhängigen Objekten ohne deren Klassen zu spezifizieren

#### Konsequenzen

- Vorteile**
- Konkrete Klassen werden isoliert
  - Austausch von Produktgruppen wird vereinfacht
  - Konsistenz unter Produkten wird gefördert

- Nachteile**
- Einführung neuer Produktarten ist aufwändig

#### Struktur



#### Zusammenspiel

- eine Instanz einer "ConcreteFactory" wird normalerweise erst zu Laufzeit erzeugt
- "AbstractFactory" verlagert dann die Erzeugung der entsprechenden Produkte in diese konkrete Fabrik

#### Beteiligte Akteure

- AbstractFactory**
- definiert eine Schnittstelle für Operationen, die abstrakte Produkte einer Produktfamilie erzeugen
- AbstractProduct**
- definiert eine Schnittstelle für eine Produktart
- Client**
- verwendet nur die Schnittstellen "AbstractFactory" und "AbstractProduct"
- ConcreteFactory**
- implementiert die Operationen der (AbstractFactory) Schnittstelle zur Erzeugung von konkreten Produkten
- Concrete Product**
- definiert durch Implementierung der (AbstractProduct) Schnittstelle ein konkretes Produkt einer Produktart, das durch die korrespondierende Fabrik erzeugt wird

# Abstract Factory Pattern

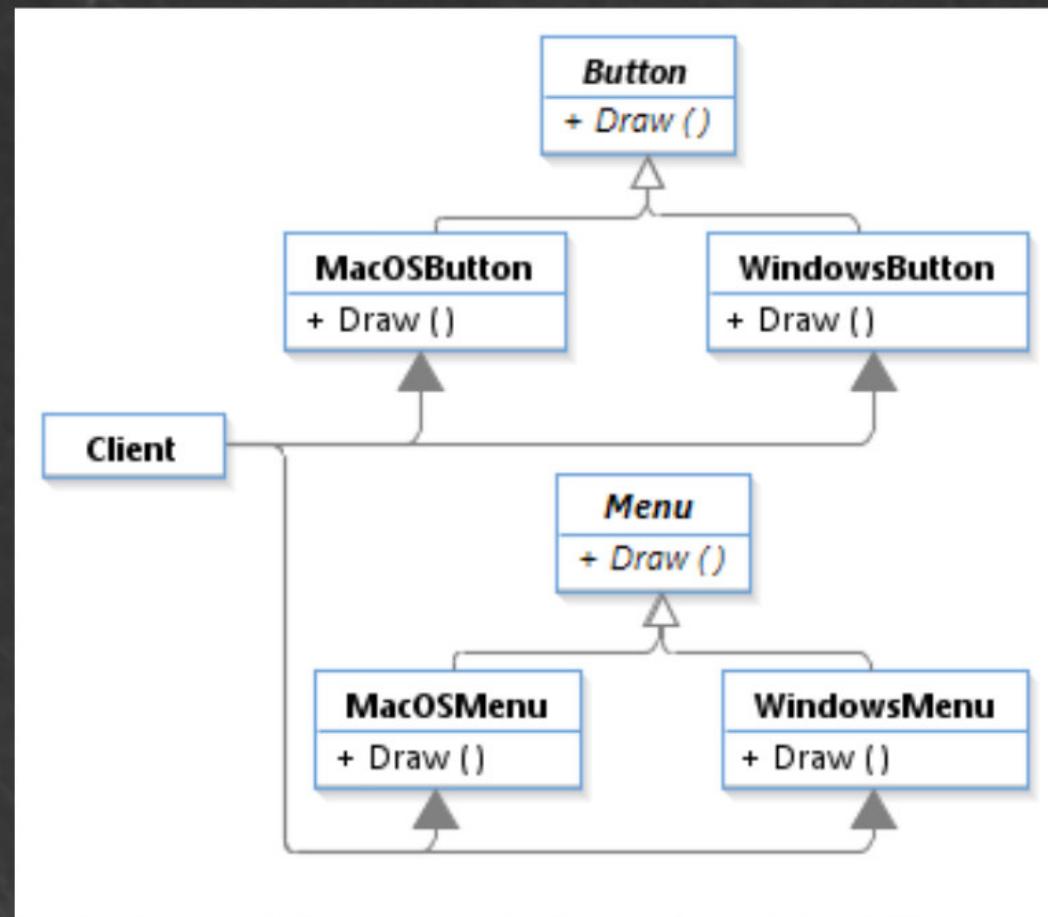
auch: Kit

Bereitstellen einer Schnittstelle zur Erzeugung einer Familie von verwandten oder voneinander abhängigen Objekten ohne deren Klassen zu spezifizieren



# Motivation

## Erstellen einer GUI

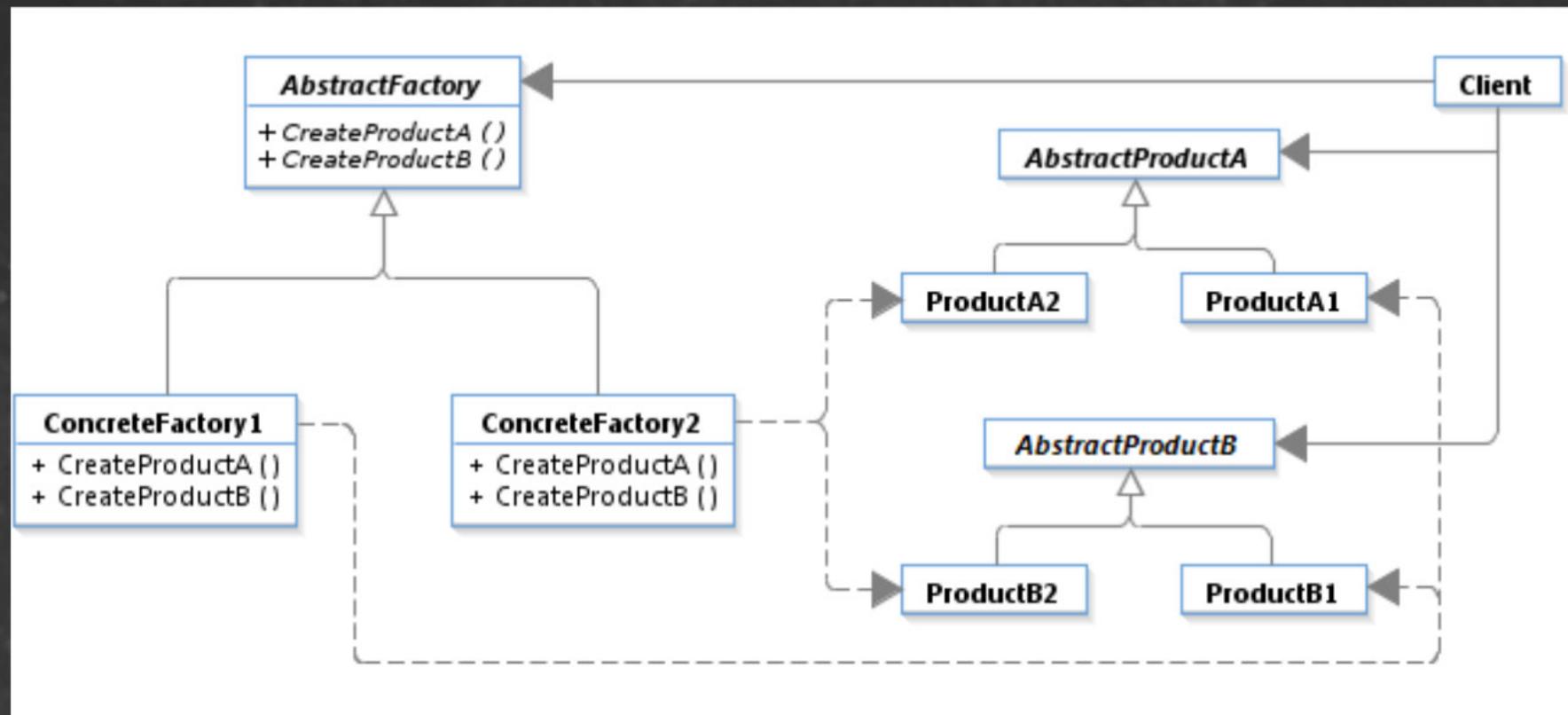


# Problembeschreibungen

- ein System soll unabhängig davon sein, wie seine Produkte (Objekte) erzeugt, zusammengesetzt und repräsentiert werden
- ein System soll mit einer oder mehreren Produktfamilien konfiguriert werden
- eine Gruppe von verwandten Produkten soll erzeugt und gemeinsam genutzt werden
- in einer Klassenbibliothek sollen die Schnittstellen von Produkten ohne deren Implementierungen bereitgestellt werden



# Struktur



# Beteiligte Akteure

## AbstractFactory

- definiert eine Schnittstelle für Operationen, die abstrakte Produkte einer Produktfamilie erzeugen

## AbstractProduct

- definiert eine Schnittstelle für eine Produktart

## Client

- verwendet nur die Schnittstellen "AbstractFactory" und "AbstractProduct"

## ConcreteFactory

- implementiert die Operationen der (AbstractFactory) Schnittstelle zur Erzeugung von konkreten Produkten

## Concrete Product

- definiert durch Implementierung der (AbstractProduct) Schnittstelle ein konkretes Produkt einer Produktart, das durch die korrespondierende Fabrik erzeugt wird



# Zusammenspiel

- eine Instanz einer "ConcreteFactory" wird normalerweise erst zu Laufzeit erzeugt
- "AbstractFactory" verlagert dann die Erzeugung der entsprechenden Produkte in diese konkrete Fabrik



# Konsequenzen

## Vorteile

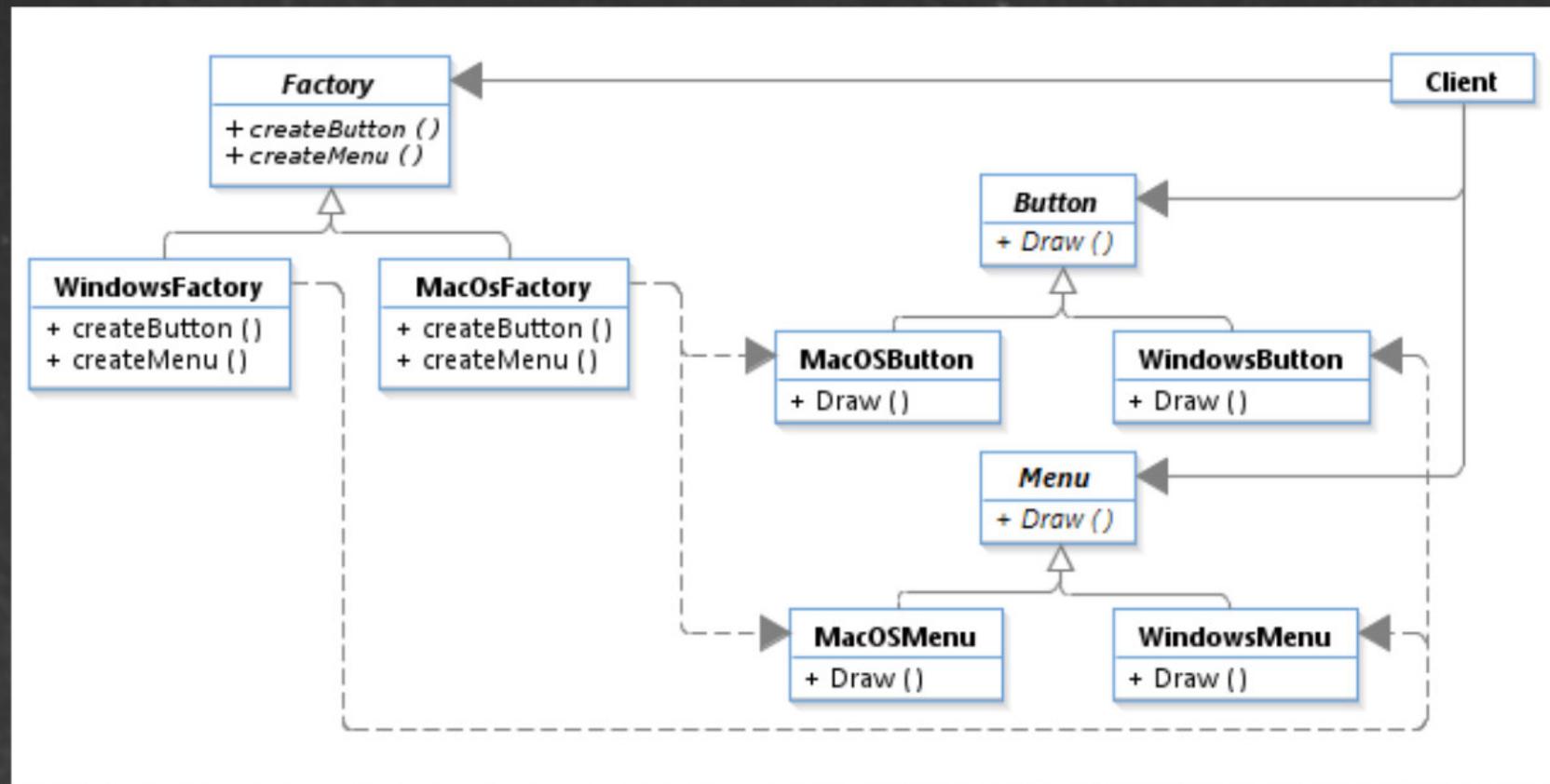
- Konkrete Klassen werden isoliert
- Austausch von Produktgruppen wird vereinfacht
- Konsistenz unter Produkten wird gefördert

## Nachteile

- Einführung neuer Produktarten ist aufwändig



# Wie sieht die GUI jetzt aus?



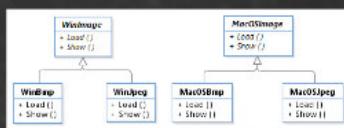
# Strukturmuster (structural patterns)

- fassen Klassen und Objekte zu größeren Strukturen zusammen
- Klassenmuster: Zusammenfassen von Schnittstellen und Implementierungen
- Objektmuster: Einordnung von Objekten in eine Struktur um neue Funktionalität zu erhalten



### Motivation

Darstellen von Bildern



### Hierarchien mit Bridge



### Problembeschreibungen

- Die permanente Bindung zwischen Implementierung und Abstraktion soll verhindert werden.
- Sowohl Abstraktion als auch Implementierung sollen erweiterbar bleiben.
- Die Veränderung der Implementierung einer Abstraktion soll keine Auswirkungen auf den Nutzer haben.
- Die Implementierung einer Abstraktion soll dem Nutzer vollständig verborgen sein.

# Bridge Pattern

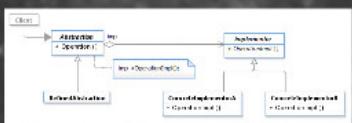
auch: Handle/ Body

Trennung der Implementierung von ihrer Abstraktion (Schnittstelle), sodass diese unabhängig voneinander verändert werden können

### Konsequenzen

- Trennung von Schnittstelle und Implementierungen
  - Implementierung kann zur Laufzeit gewählt werden
  - bessere Struktur
  - Verbesserte Erweiterbarkeit
  - Details der Implementierungen werden vor Klienten verborgen

### Struktur



### Beteiligte Akteure

- Abstraction**
- definiert die Schnittstelle für Abstraktionen
  - erhält eine Referenz auf eine Objekt des Typs "Implementor"
- Implementor**
- definiert die Schnittstelle für Implementierungsklassen
- ConcreteAbstraction**
- erweitert die Schnittstelle "Abstraction"
- ConcreteImplementor**
- Implementiert die "Implementor"- Schnittstelle und definiert deren konkrete Implementierungen

### Zusammenspiel

- "Abstraction" leitet die Anfragen des Klienten an sein "Implementor"- Objekt weiter

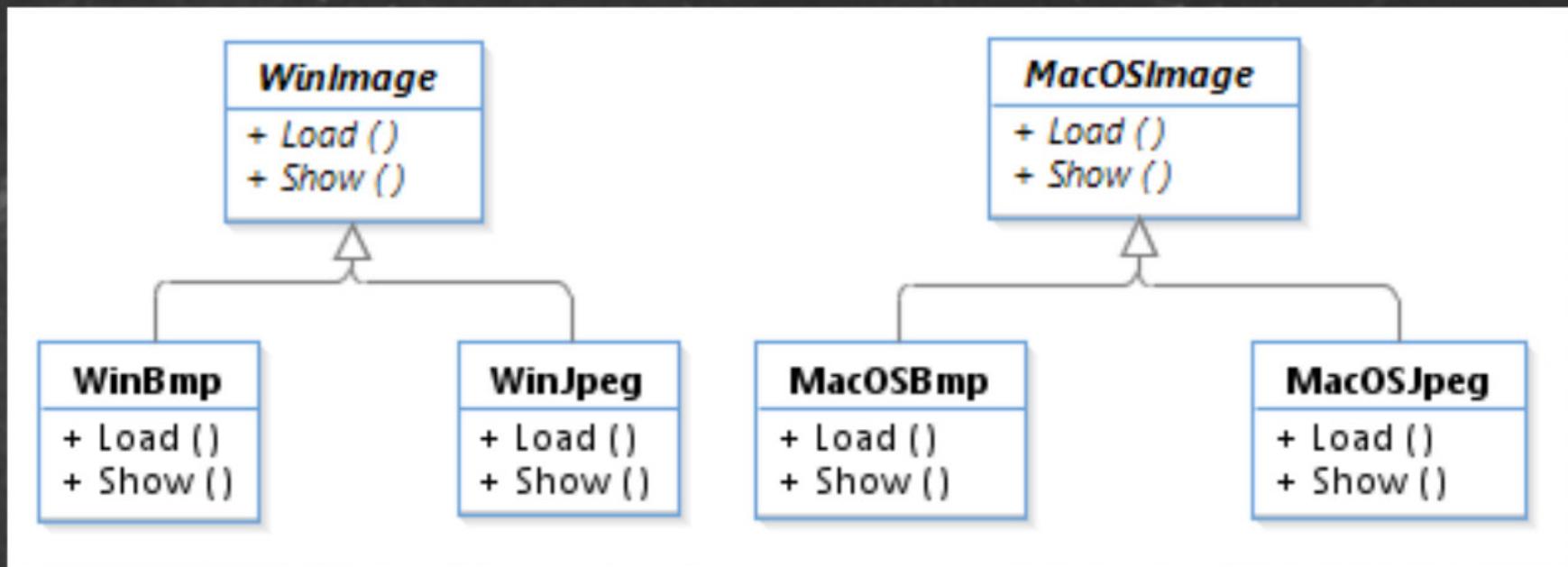
# Bridge Pattern

auch: Handle/ Body

Trennung der Implementierung von ihrer Abstraktion (Schnittstelle), sodass diese unabhängig voneinander verändert werden können

# Motivation

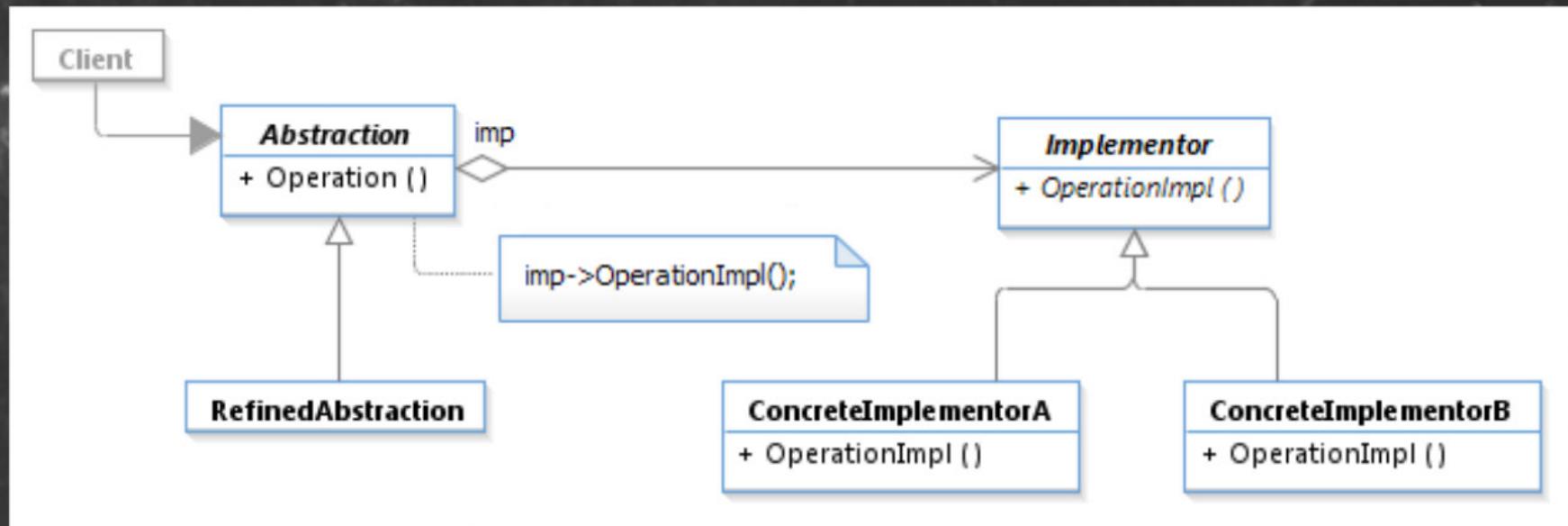
## Darstellen von Bildern



# Problembeschreibungen

- Die permanente Bindung zwischen Implementierung und Abstraktion soll verhindert werden.
- Sowohl Abstraktion als auch Implementierung sollen erweiterbar bleiben.
- Die Veränderung der Implementierung einer Abstraktion soll keine Auswirkungen auf den Nutzer haben.
- Die Implementierung einer Abstraktion soll dem Nutzer vollständig verborgen sein.

# Struktur



# Beteiligte Akteure

## Abstraction

- definiert die Schnittstelle für Abstraktionen
- erhält eine Referenz auf eine Objekt des Typs "Implementor"

## Implementor

- definiert die Schnittstelle für Implementierungsklassen

## RefinedAbstraction

- erweitert die Schnittstelle "Abstraction"

## ConcreteImplementor

- implementiert die "Implementor"- Schnittstelle und definiert deren konkrete Implementierungen



# Zusammenspiel

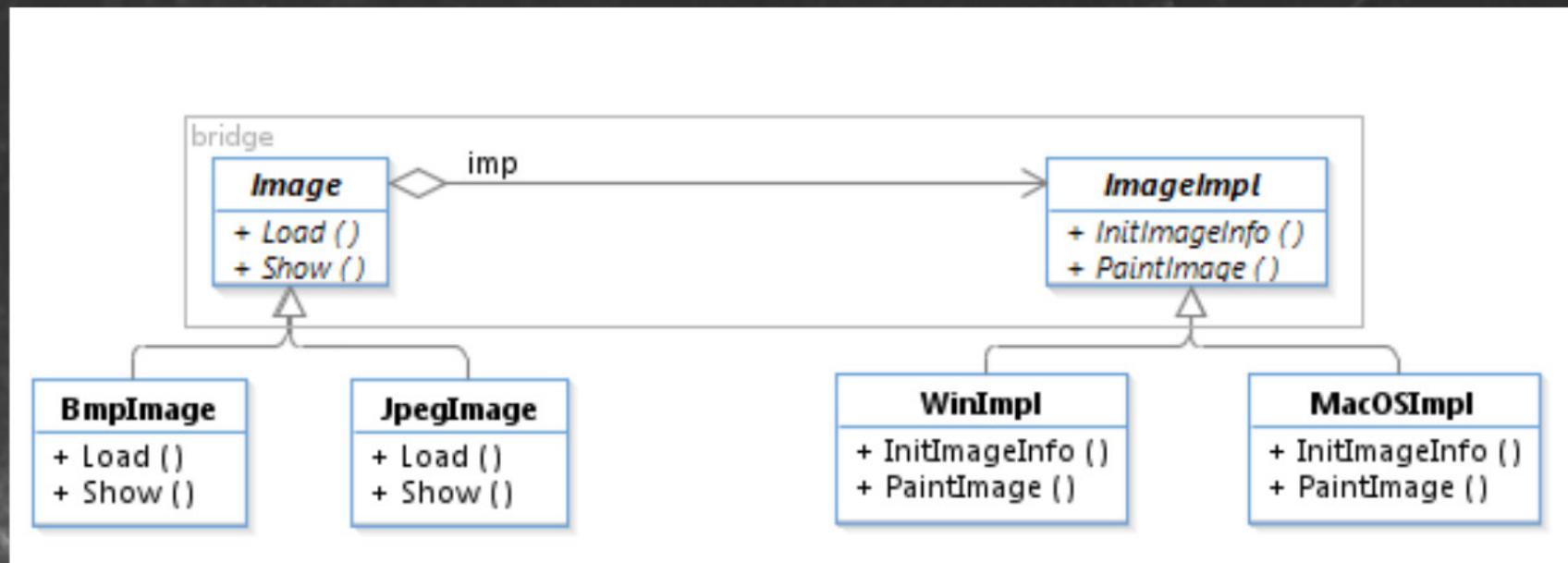
- "Abstraction" leitet die Anfragen des Klienten an sein "Implementor"- Objekt weiter

# Konsequenzen

- Trennung von Schnittstelle und Implementierungen
  - Implementierung kann zur Laufzeit gewählt werden
  - bessere Struktur
- Verbesserte Erweiterbarkeit
- Details der Implementierungen werden vor Klienten verborgen



# Hierarchien mit Bridge



# Verhaltensmuster (behavioral patterns)

- beschreiben die Interaktion zwischen Objekten und komplexen Kontrollflüssen
- Klassenmuster: Aufteilung von Kontrolle auf verschiedene Klassen durch Vererbung
- Objektmuster: Verwendung von Komposition an Stelle von Vererbung



## Motivation

### Erstellung eines kleinen Spiels

- Wir haben:
  - Edelstein (jewel)
  - Wachmänner (Guards)
  - Sicherheitstüren (Door)
- Edelstein liegt auf druckempfindlicher Platte
- Spieler ist ein Dieb, der den Edelstein stehlen will
- Es soll Alarm geschlagen werden, wenn der Edelstein entfernt wird (Wachmänner informieren, Türen verriegeln)

## DEMO

Der Beispielcode steht hier zum Download bereit:  
<http://www.dreamincode.net/forums/topic/197421-the-listener-pattern/>

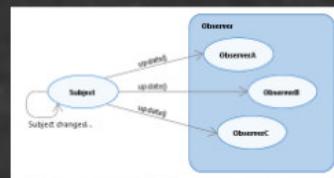
## Problembeschreibungen

- Das Modell beinhaltet zwei voneinander abhängige Aspekte, die jedoch unabhängig voneinander verändert und verwendet werden sollen.
- Die Veränderung eines Objektes zieht Veränderung unbekannt vieler weiterer Objekte nach sich.
- Ein Objekt muss andere Objekte benachrichtigen ohne deren Typ zu kennen.

# Observer Pattern

auch: Publish- Subscribe, Listener

Definition einer 1 : n - Beziehung zwischen Objekten, so dass alle abhängigen Objekte benachrichtigt und aktualisiert werden, wenn sich der Zustand des Objekts, von dem sie abhängen, ändert



## Struktur



## Beteiligte Akteure

**Subject**  
- kennt seine Beobachter (beliebig viele)  
- stellt eine Schnittstelle zum Anhängen und Lösen von Beobachterobjekten zur Verfügung

**Observer**  
- definiert eine Schnittstelle für Objekte, die über die Veränderungen des Subject benachrichtigt werden sollen

**ConcreteSubject**  
- spezifiziert den für die konkreten Beobachter interessanten Zustand  
- sendet eine Benachrichtigung an seine Beobachter, wenn dieser Zustand sich ändert

**ConcreteObserver**  
- erhält eine Referenz auf ein "ConcreteSubject" Objekt  
- spezifiziert den Zustand, der mit dem Zustand des Subjekts konsistent sein soll  
- implementiert die "Observer"- Schnittstelle um die Aktualisierung des eigenen Zustands zu ermöglichen

## Konsequenzen

### Vorteile

- Subjekt und Beobachter sind abstrakt gekoppelt
- Broadcast - Kommunikation wird ermöglicht

### Nachteile

- fehlerhafte Aktualisierungen
- Aktualisierungskaskaden

## Zusammenspiel

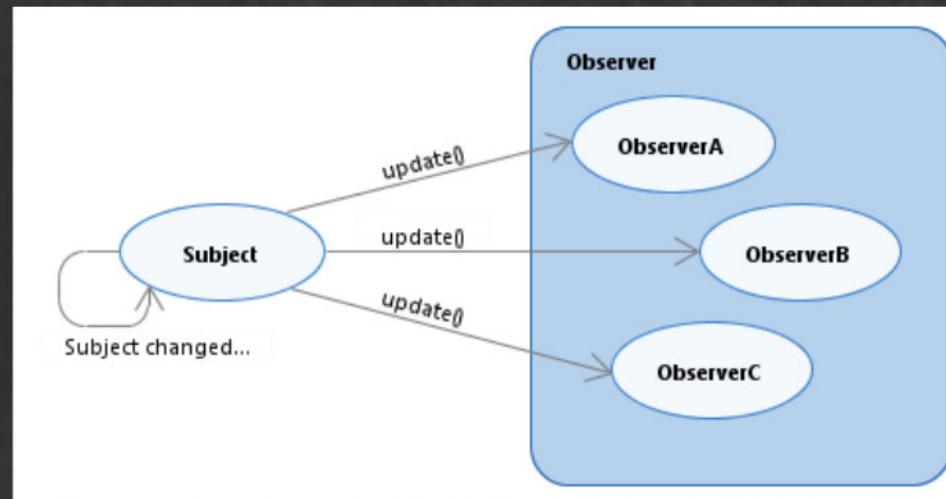
- "ConcreteSubject" benachrichtigt seine Beobachter, wenn eine für diese relevante Veränderung auftritt
- Nach Erhalt der Benachrichtigung können die Beobachter Informationen vom Subjekt abfragen um ihren eigenen Zustand entsprechend zu aktualisieren



# Observer Pattern

auch: Publish-Subscribe, Listener

Definition einer 1 : n - Beziehung zwischen Objekten, so dass alle abhängigen Objekte benachrichtigt und aktualisiert werden, wenn sich der Zustand des Objekts, von dem sie abhängen, ändert



# Motivation

## Erstellung eines kleinen Spiels

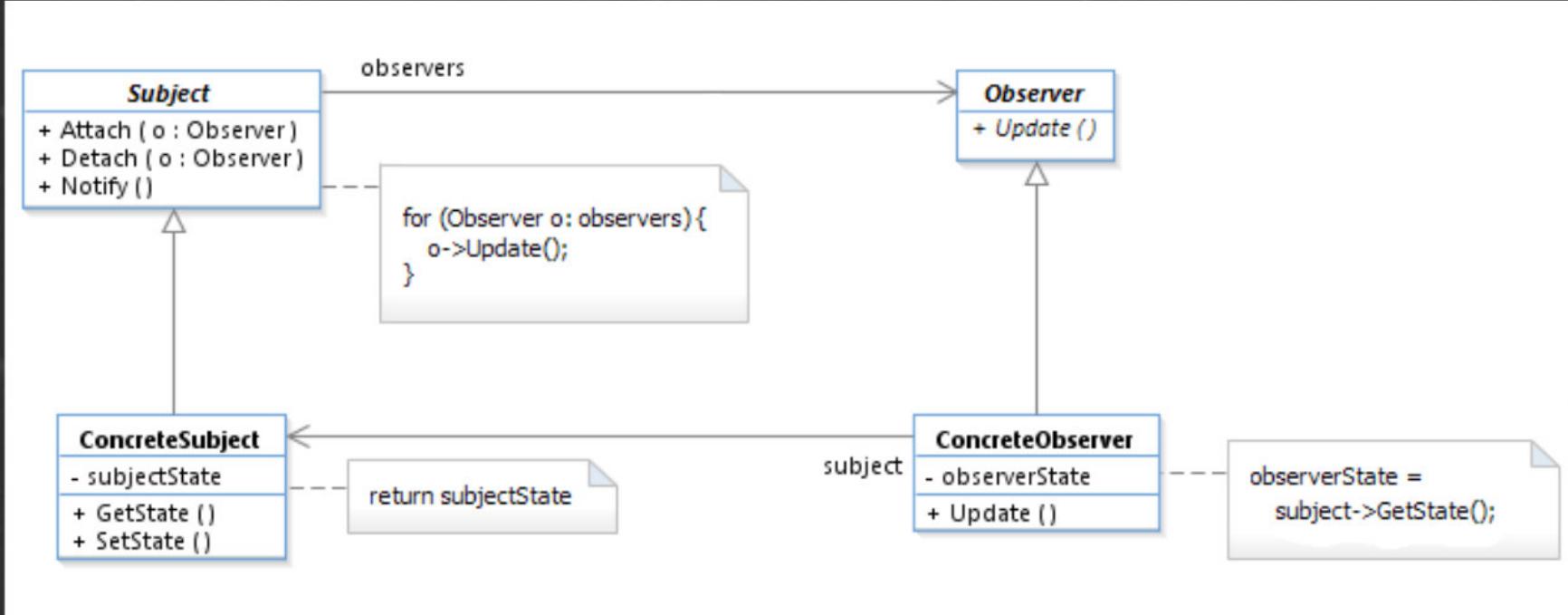
- Wir haben:
  - Edelstein (Jewel)
  - Wachmänner (Guards)
  - Sicherheitstüren (Door)
- Edelstein liegt auf druckempfindlicher Platte
- Spieler ist ein Dieb, der den Edelstein stehlen will
- Es soll Alarm geschlagen werden, wenn der Edelstein entfernt wird (Wachmänner informieren, Türen verriegeln)

# Problembeschreibungen

- Das Model beinhaltet zwei voneinander abhängige Aspekte, die jedoch unabhängig voneinander verändert und verwendet werden sollen.
- Die Veränderung eines Objektes zieht Veränderung unbekannt vieler weiterer Objekte nach sich.
- Ein Objekt muss andere Objekte benachrichtigen ohne deren Typ zu kennen.



# Struktur



# Beteiligte Akteure

## Subject

- kennt seine Beobachter (beliebig viele)
- stellt eine Schnittstelle zum Anhängen und Loslösen von Beobachterobjekten zur Verfügung

## Observer

- definiert eine Schnittstelle für Objekte, die über die Veränderungen des Subject benachrichtigt werden sollen

## ConcreteSubject

- speichert den für die konkreten Beobachter interessanten Zustand
- sendet eine Benachrichtigung an seine Beobachter, wenn dieser Zustand sich ändert

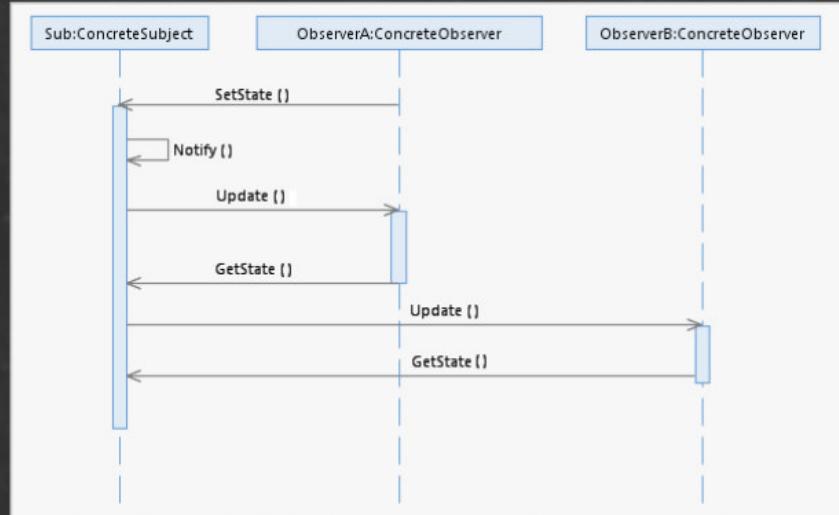
## ConcreteObserver

- erhält eine Referenz auf ein "ConcreteSubject" Objekt
- speichert den Zustand, der mit dem Zustand des Subjektes konsistent sein soll
- implementiert die "Observer"- Schnittstelle um die Aktualisierung des eigenen Zustands zu ermöglichen



# Zusammenspiel

- "ConcreteSubject" benachrichtigt seine Beobachter, wenn eine für diese relevante Veränderung auftritt
- Nach Erhalt der Benachrichtigung können die Beobachter Informationen vom Subjekt abfragen um ihren eigenen Zustand entsprechend zu aktualisieren



# Konsequenzen

## Vorteile

- Subjekt und Beobachter sind abstrakt verknüpft
- Broadcast- Kommunikation wird ermöglicht

## Nachteile

- fehlerhafte Aktualisierungen
- Aktualisierungskaskaden

# DEMO

Der Beispielcode steht hier zum Download bereit:  
<http://www.dreamincode.net/forums/topic/197421-the-listener-pattern/>



# Und was haben wir jetzt davon?

- Warum das Rad neu erfinden?
- häufig gemachte Fehler werden vermieden
- deutlich bessere Strukturierung des Codes
- effizientere Verwendung des Codes
- Einsparung von Speicherplatz und Rechenleistung
- ABER: Beschränkung der eigenen Kreativität durch schon strukturierte und klar vorgegebene Pfade

## Quellen

- Lehr-Kompendium vom Fach Informatik: "Was ist Programmierung?".  
Autoren: Christian Hirsch, Michael Klemm, Stephan Lippert  
Verlag: Carl Hanser Verlag, München, Wien  
ISBN: 978-3-446-41600-0
- <http://www.cs.york.ac.uk/~mccann/101/lectures/03.html>
- <http://www.cs.york.ac.uk/~mccann/101/lectures/04.html>
- <http://www.cs.york.ac.uk/~mccann/101/lectures/05.html>



Prezi

# Quellen

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: **Design Patterns. Elements of Reusable Object- Oriented Software**, Addison-Wesley, 1995
- Siri Sjöqvist, Erik Balgård: **3D game engines and design patterns**,  
[http://www.idt.mdh.se/kurser/cd5130/msl/2006lp4/reports/drafts/3d\\_game\\_engines\\_and\\_design\\_patterns.pdf](http://www.idt.mdh.se/kurser/cd5130/msl/2006lp4/reports/drafts/3d_game_engines_and_design_patterns.pdf)
- [http://sourcemaking.com/design\\_patterns/abstract\\_factory/cpp/1](http://sourcemaking.com/design_patterns/abstract_factory/cpp/1)
- <http://www.codeproject.com/Articles/890/Bridge-Pattern-Bridging-the-gap-between-Interface>
- <http://www.dreamincode.net/forums/topic/197421-the-listener-pattern/>