

Design Patterns

Dr. Matthias Hözl

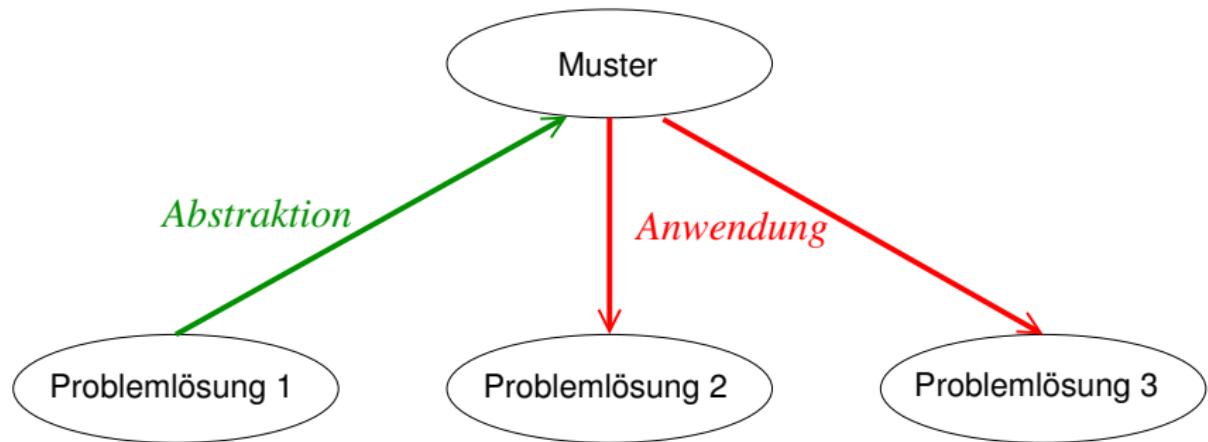
16. Dezember 2020

Entwurfsmuster

Teil 1

Grundlagen

Dasselbe (bewährte) Lösungsmuster kann für Probleme, die einander ähnlich sind, wiederverwendet werden.



Vorteile

- ▶ Wiederverwendung von erprobten Lösungsprinzipien
(Qualität, Kostenersparnis)
- ▶ abstrakte Dokumentation von Entwürfen
- ▶ gemeinsames Vokabular zur (schnellen) Verständigung unter Entwicklern

Nachteile

- ▶ Oftmals erhöhte Komplexität
- ▶ Oftmals weniger statische Information
- ▶ Oftmals geringere Performance

Fensterplatz

Jeder liebt Fensterplätze, Erker und große Fenster mit niedrigen Fensterbänken und bequemen Stühlen, die an sie herangezogen werden.

In einem Raum, der nicht über einen solchen Platz verfügt, kann man sich nur selten gemütlich zu fühlen oder sich vollkommen wohl zu fühlen...

Architekturpattern (2, C. Alexander)

Wenn der Raum kein Fenster enthält, welches ein „Platz“ ist, wird eine Person in dem Raum zwischen zwei Einflüssen hin- und hergerissen sein: 1. Sie will sich hinsetzen und es sich bequem machen. 2. Sie fühlt sich zum Licht hingezogen. Offensichtlich, wenn die bequemen Plätze—die Plätze im Raum, an denen man sich am liebsten aufhält, von den Fenstern entfernt sind, gibt es keine Möglichkeit, diesen Konflikt zu überwinden.

Deshalb: In jedem Raum, in dem Sie sich tagsüber längere Zeit aufhalten, machen Sie mindestens ein Fenster zu einem „Fensterplatz“.

Was ist ein Entwurfsmuster? (C. Alexander)

Jedes Entwurfsmuster ist eine dreiteilige Regel, die eine Beziehung ausdrückt zwischen einem bestimmten **Kontext**, einem **Problem** und einer **Lösung**.

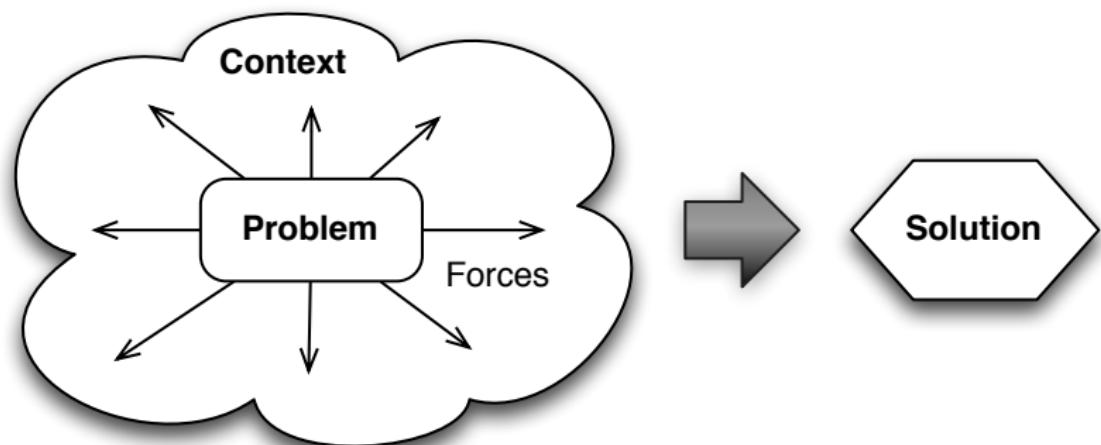
Als ein Element **in der Welt**, ist jedes Muster eine Beziehung zwischen einem bestimmten **Kontext**, einem bestimmten System von **Einflüssen** (oder **Kräften**), die in diesem Kontext wiederholt auftreten, und einer bestimmten räumlichen Konfiguration, die es diesen Kräften erlaubt, sich aufzulösen.

Was ist ein Entwurfsmuster? (C. Alexander)

Als ein Element der **Sprache** ist ein Muster eine Anweisung, die zeigt, wie diese räumliche Konfiguration immer wieder verwendet werden kann, um das gegebene Kräftesystem aufzulösen, wo immer der Kontext es relevant macht.

Das Muster ist, kurz gesagt, gleichzeitig ein Ding, das in der Welt geschieht, und die Textregel, die uns sagt, wie und wann wir es erschaffen müssen. Es ist **beides** ein Prozess und ein Ding; sowohl eine Beschreibung eines Dings, das lebendig ist, als auch eine Beschreibung des Prozesses, der dieses Ding erzeugt.

Komponenten eines Patterns



- ▶ 1977 Alexander: Architekturmuster für Gebäude und Städtebau
- ▶ 1980 Smalltalks MVC-Prinzip (Model View Controller)
- ▶ Seit 1990 Objektorientierte Muster im Software-Engineering
- ▶ 1995 Design Pattern Katalog von Gamma, Helm, Johnson, Vlissides („Gang of Four“, GoF)
- ▶ Martin Fowler (1996). Analysis Patterns
- ▶ Frank Buschmann et al. (1996-2007(?)). Pattern-Oriented Software Architecture, Volumes 1–5
- ▶ Martin Fowler (2002). Patterns of Enterprise Application Architecture
- ▶ Gregor Hohpe, Bobby Woolf (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions

Arten von Pattern

Pattern können in verschiedenen Abstraktionsgraden definiert werden:

- ▶ Architektur-Pattern: Beschreiben die komplette Architektur eines Systems
Beispiele: Geschichtete Architektur, „Pipes und Filter“ Architektur
- ▶ Design-Pattern: Beschreiben Lösungen für Design-Probleme auf einer niedrigeren Abstraktionsebene als Architektur-Pattern
Beispiele: Singleton, Abstract Factory, Composite
- ▶ Idiome: Beschreiben Sprachspezifisch Lösungen.
Beispiele: Smart Pointers (C/C++), Resource Allocation is Initialization (RAII, C++)

Beispiel: GUI in Smalltalk 80

System Transcript

Snapshot at: (31 May 1983 10:37:52 am)

System Browser

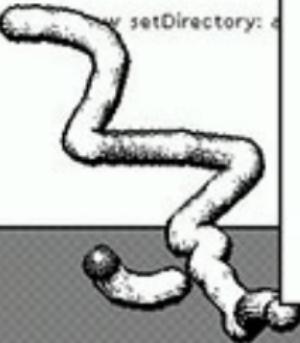
System-Changes	AltoFile
System-Compiler	AltoFileAddress
System-Releasing	AltoFileDirector
Files-Streams	AltoFilePage
Files-Abstract	
Files-Xerox Alto	

instance

directory: aFileDialog direct

"Answer an instance of me aString."

setDirectory: aFileDialog



System Workspace

The Smalltalk-80tm System Version 2

Copyright (c) 1983 Xerox Corp.

File List All rights reserved.

* A

DiskDescriptor
Smalltalk-80.changes
Smalltalk-80.sources
SysDir
toothpaste.st

'From Smalltalk-80, Version 2.2 of July 4, 1987 on 9 July 1987 at 11:27:15 am'!

!Form class methodsFor: 'examples'!

toothpaste

"Form toothpaste"

"Draw spheres ala Ken Knowlton, Computer Graphics, v15 n4 p352.
Draws while red is held, terminated by yellow."

| facade outliner filter point queue cursorPoint |
facade + Form
extent: 20@20
fromArray: #(65471 61440 63191 61440 64699
61440 67453 61440 55307)

Smalltalk 80 war eine der ersten graphischen Programmierumgebungen. Um die Entwicklung graphischer Oberflächen zu vereinfachen wurde in den 1980er Jahren in Smalltalk 80 die Model-View-Controller (MVC)-Architektur eingeführt.

Es wurden viele Varianten des ursprünglichen MVC Patterns vorgeschlagen (e.g., Model-View-Presenter, Model-Document-Abstraction, Model-View-Viewmodel). Leider werden viele dieser Varianten oft ebenfalls als MVC-Pattern bezeichnet.

Name: Model-View-Controller

Kontext:

Sie entwickeln eine Applikation mit graphischer Benutzeroberfläche und wollen Benutzeroberfläche und Geschäftslogik unabhängig voneinander halten.

Problem:

Wenn man den Code, der die Benutzeroberfläche verwaltet direkt in die Geschäftslogik einer Anwendung einbettet erhält man Anwendungen, deren eigentliche Funktionalität eng mit der Präsentation der Daten verwoben ist. Das erschwert die Unterstützung verschiedener Plattformen und verkompliziert die Geschäftslogik.

Lösung:

Spalte die Anwendung in drei Arten von Objekten/Komponenten auf:

- ▶ Das *Modell* ist verantwortlich dafür, die Geschäftsprozesse und Domänenobjekte abzubilden, und es verarbeitet die Daten entsprechend den geschäftsspezifischen Anforderungen. Das Modell hängt nicht von spezifischen Ein-Ausgabeverhalten oder Benutzeroberflächen ab.

Lösung:

Spalte die Anwendung in drei Arten von Objekten/Komponenten auf:

- ▶ View-Komponenten sind für die graphische Darstellung der Daten verantwortlich. Sie erhalten die anzuzeigenden Daten vom Modell. Mehrere Views können die gleichen Daten des Modells darstellen.

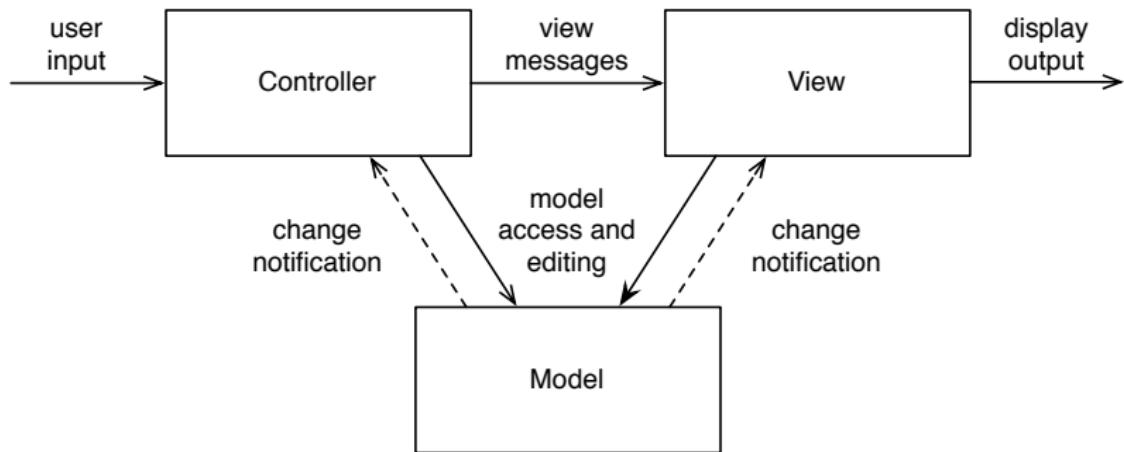
Lösung:

Spalte die Anwendung in drei Arten von Objekten/Komponenten auf:

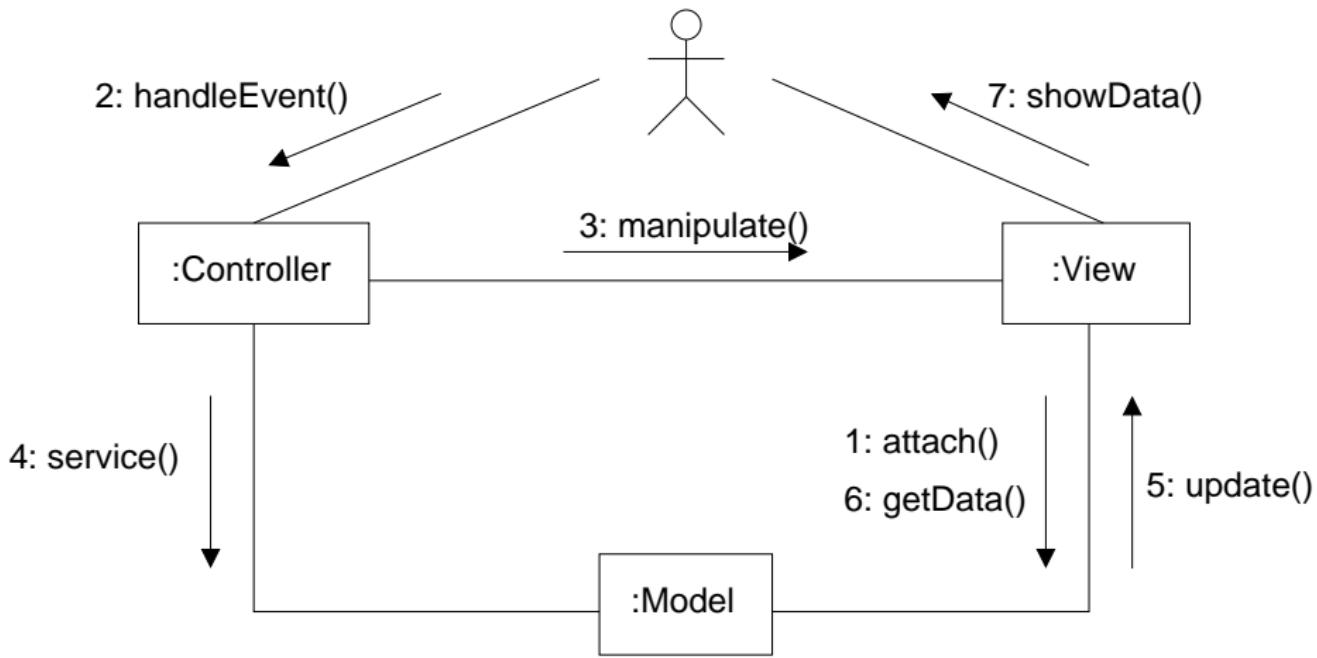
- ▶ Jede View hat einen oder mehrere *Controller*, die Eingaben vom Benutzer bekommen. Diese Eingaben werden in Events für die View oder das Modell umgewandelt. Jegliche Benutzerinteraktion wird durch Controller geregelt.

MVC Architektur

Struktur: (Kein UML Diagramm!)



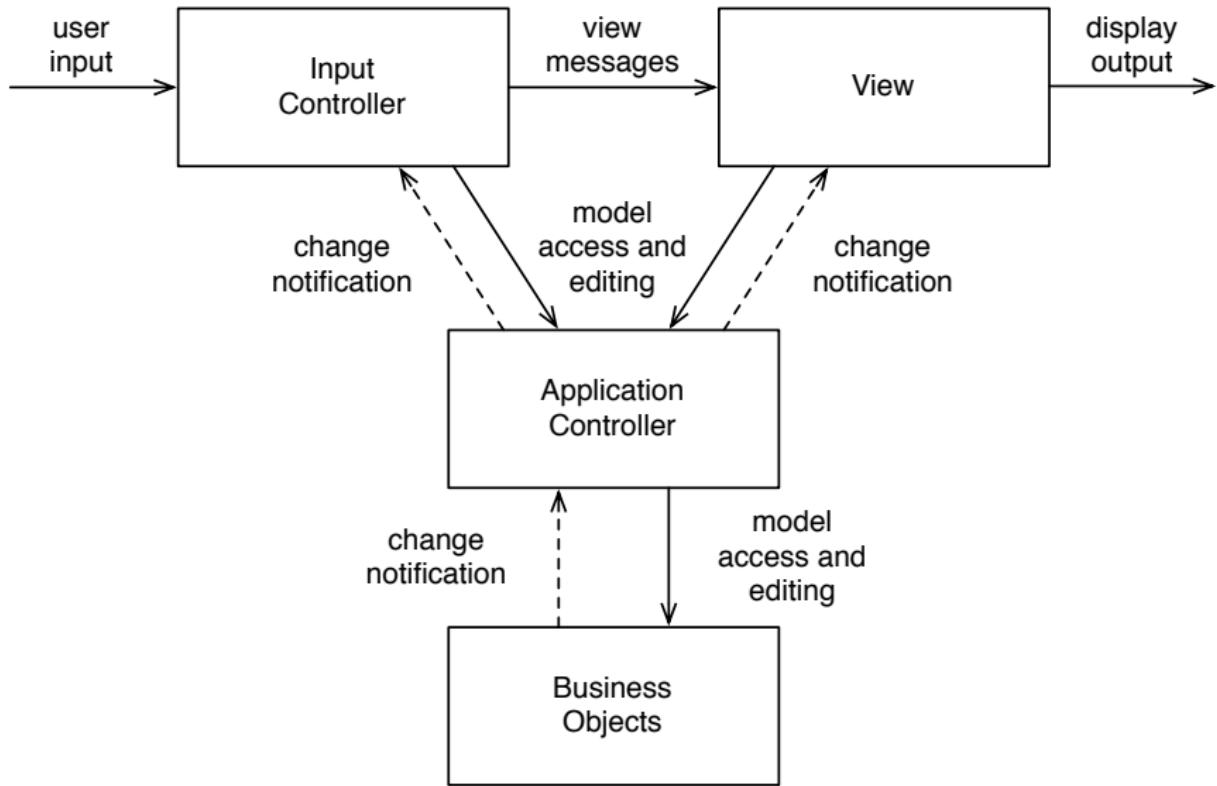
Interaktion (Kommunikationsdiagramm):



Konsequenzen:

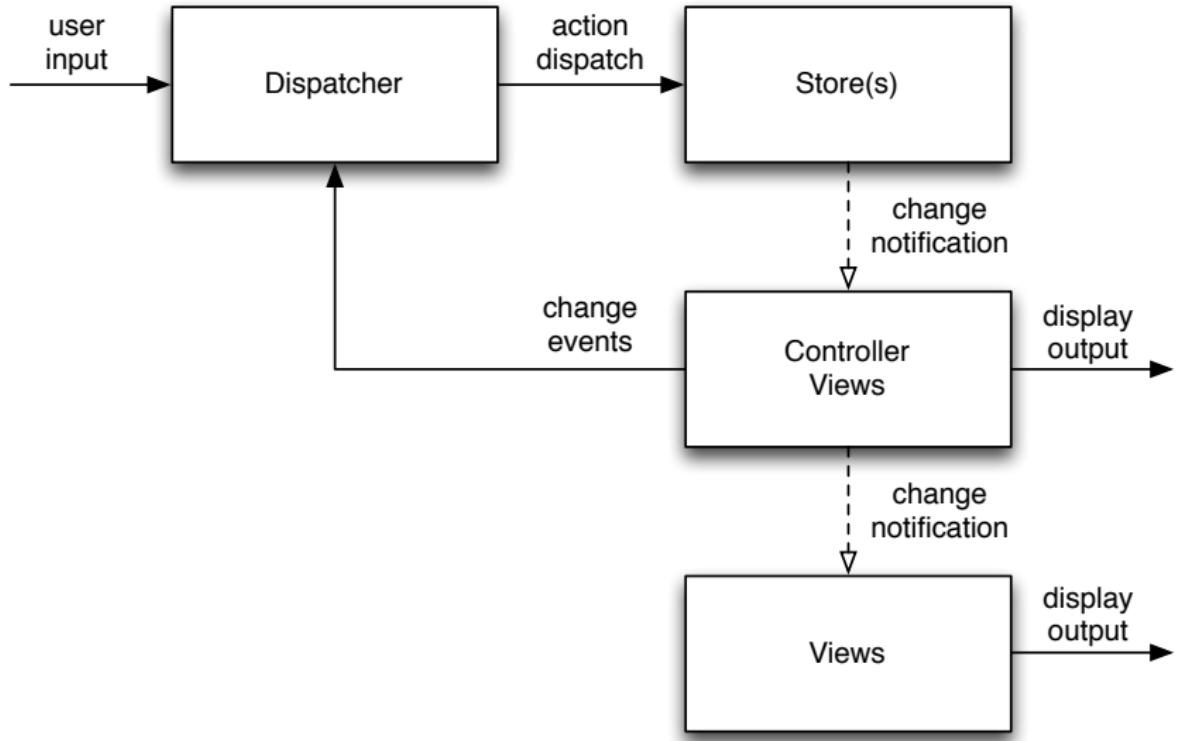
- ▶ Modell und Benutzeroberfläche sind vollständig entkoppelt
- ▶ Es können leicht verschiedene Benutzeroberflächen für die gleiche Geschäftslogik erstellt werden
- ▶ Das Modell muss einen generischen Notifikationsmechanismus bereitstellen um Views und Controller über Änderungen informieren zu können

Web-MVC Architektur

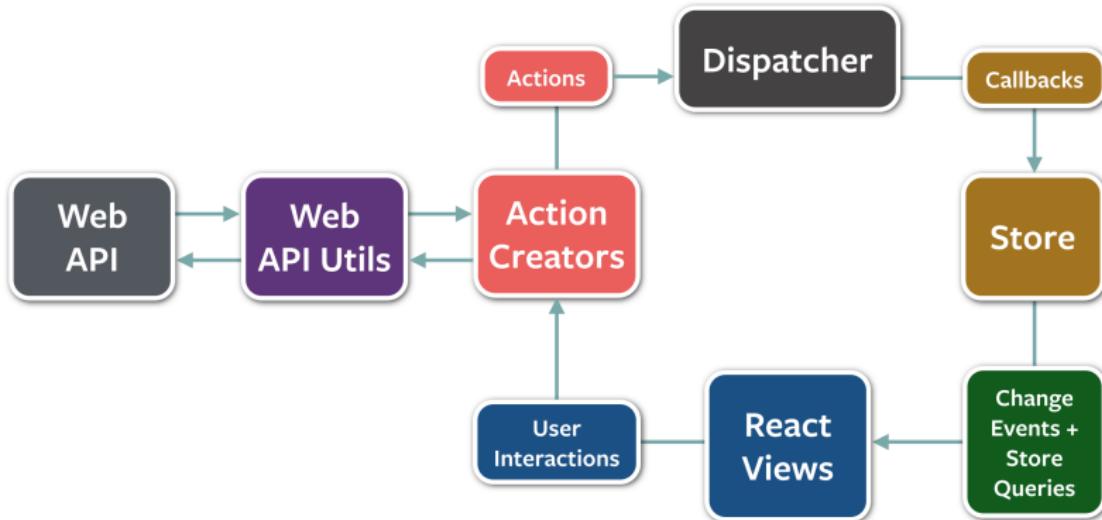


Viele andere Architekturen wurden in der Literatur ebenfalls als MVC-Architektur bezeichnet. Z.B. wird in Web-Applikationen oft der „Application Controller“ im Diagramm auf der vorhergehenden Folie als „Controller“ bezeichnet. In einer traditionellen MVC-Architektur ist der „Application Controller“ Teil des Modells und kein Controller.

Alternative zu Web-MVC: Flux



Alternative zu Web-MVC: Flux



Andere Anwendung eines MVC-Artigen Patterns

You Win!

Collection Level



Pattern-Sprachen und -Kataloge

- ▶ Pattern werden zu verschiedenen Zwecken und auf verschiedenen Abstraktionsebenen verwendet
- ▶ Daher gibt es verschiedene „Pattern-Sprachen,“ die sich in den Elementen unterscheiden, die für jedes Pattern aufgeschrieben werden
- ▶ Typischerweise werden nicht einzelne Pattern aufgeschrieben sondern Sammlungen miteinander verwandter Patterns, die alle in der gleichen Pattern-Sprache notiert sind
- ▶ Eine solche Sammlung von Patterns nennt sich Pattern-Katalog
- ▶ Pattern-Kataloge existieren für verschiedenartige Themengebiete
 - ▶ Analyse, Software-Design, ...
 - ▶ Enterprise-Architektur, verteilte Systeme, ...
 - ▶ Programmiersprachen (JavaScript, Smalltalk...)

Wesentliche Elemente eines Entwurfmusters

- ▶ Name des Musters
- ▶ Beschreibung der Problemklasse, bei der das Muster anwendbar ist
- ▶ Beschreibung eines Anwendungsbeispiels
- ▶ Beschreibung der Lösung (Struktur, Verantwortlichkeiten, ...)
- ▶ Beschreibung der Konsequenzen (Nutzen/Kosten-Analyse)

Beispiel: Design-Pattern im GoF-Katalog

Beschreibungsform für Design Pattern

► **Pattern-Name und Klassifizierung**

Der Name des Musters vermittelt die Essenz des Musters kurz und bündig. Ein guter Name ist wichtig, denn er wird Teil Ihres Design-Vokabulars.

► **Ziel**

Eine kurze Aussage, die die folgende Frage beantwortet: Was macht das Entwurfsmuster? Was ist sein Grund und Ziel?
Welche bestimmte Designfrage oder -probleme adressiert es?

► **Also bekannt als (aka)**

Andere bekannte Namen für das Muster, falls vorhanden.

Beispiel: Design-Pattern im GoF-Katalog

► Motivation

Ein Szenario, das ein Entwurfsproblem illustriert und zeigt, wie die Klassen- und Objektstrukturen in dem Muster das Problem lösen. Das Szenario wird Ihnen helfen, die abstrakteren Beschreibungen des Patterns, die darauf folgen zu verstehen.

► Anwendbarkeit

In welchen Situationen kann das Entwurfsmuster angewendet werden? Was sind Beispiele für schlechte Designs, die das Muster adressieren? Wie können Sie diese Situationen erkennen?

Beispiel: Design-Pattern im GoF-Katalog

- ▶ **Teilnehmer**

Die am Entwurfsmuster beteiligten Klassen und/oder Objekte und ihre Verantwortlichkeiten.

- ▶ **Kollaborationen**

Wie die Teilnehmer zusammenarbeiten, um ihre Verantwortlichkeiten.

Beispiel: Design-Pattern im GoF-Katalog

► Struktur

Eine grafische Darstellung der Klassen des Musters unter Verwendung einer Notation, die auf der Object Modelling Technique (OMT) basiert. Wir verwenden auch Interaktionsdiagramme zur Veranschaulichung von Abfolgen von Anforderungen und Kollaborationen zwischen Objekten.

► Konsequenzen

Wie unterstützt das Muster seine Ziele? Was sind die Zielkonflikte und Ergebnisse der Verwendung des Musters? Welche Aspekte der Systemstruktur können Sie damit unabhängig variieren?

► Implementierung

Auf welche Fallstricke, Hinweise oder Techniken sollten Sie achten, wenn Sie der Implementierung des Musters beachten? Gibt es sprachspezifische Probleme?

Beispiel: Design-Pattern im GoF-Katalog

► **Beispielcode**

Codefragmente, die veranschaulichen, wie Sie das Muster implementieren könnten (in C++ oder Smalltalk).

► **Bekannte Anwendungen**

Beispiele für das Pattern, die in realen Systemen zu finden sind. Wir führen mindestens zwei Beispiele aus verschiedenen Domänen an.

► **Verwandte Patterns**

Welche Entwurfsmuster sind eng mit diesem verwandt? Was sind die wichtigen Unterschiede? Mit welchen anderen Mustern sollte dieses verwendet werden?

Klassifikation der GoF Design Pattern

- ▶ **Creational Patterns** (befassen sich mit der Erzeugung von Objekten)
- ▶ **Structural Patterns** (befassen sich mit der strukturellen Komposition von Klassen oder Objekten)
- ▶ **Behavioral Patterns** (befassen sich mit der Interaktion von Objekten und der Verteilung von Verantwortlichkeiten)

- ▶ **Abstract Factory** Bietet eine Schnittstelle zum Erstellen von Familien von verwandten oder abhängigen Objekten ohne Angabe ihrer konkreten Klassen.
- ▶ **Builder** Trennt die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.
- ▶ **Factory Method** Definiert eine Schnittstelle zur Erzeugung eines Objekts, aber lässt Unterklassen entscheiden, welche Klasse sie instanziieren. Mit der Factory-Methode kann eine Klasse die Instanzierung an Unterklassen aufschieben.

- ▶ **Prototype** Gibt die Arten von Objekten an, die erstellt werden sollen; Objekte werden aus einer prototypischen Instanz erzeugt, neue Objekte werden durch Kopieren dieses Prototyps erstellt.
- ▶ **Singleton** Stellt sicher, dass eine Klasse nur eine Instanz hat, und bieten einen globalen Zugriffspunkt darauf.

Structural Patterns

- ▶ **Adapter** Wandelt die Schnittstelle einer Klasse in eine andere Schnittstelle, die Clients erwarten. Adapter lassen Klassen zusammenarbeiten, die sonst aufgrund inkompatibler Schnittstellen nicht zusammenarbeiten könnten.
- ▶ **Bridge** Entkoppelt eine Abstraktion von ihrer Implementierung, so dass die beiden unabhängig voneinander variieren können.
- ▶ **Composite** Setzt Objekte in Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien darzustellen. Mit Composite können Clients einzelne Objekte und Zusammensetzungen von Objekten einheitlich behandeln.

- ▶ **Decorator** Fügt dynamisch zusätzliche Verantwortlichkeiten zu einem Objekt hinzu. Dekoratoren bieten eine flexible Alternative zu Unterklassen zur Erweiterung der Funktionalität.
- ▶ **Facade** Bietet ein einheitliches Interface zu einer Menge von Schnittstellen in einem Subsystem. Facade definiert eine übergeordnete Schnittstelle die die Verwendung des Subsystems erleichtert.
- ▶ **Flyweight** Verwendet Sharing, um eine große Anzahl von feinkörnigen Objekten effizient zu unterstützen.
- ▶ **Proxy** Verwendet ein Surrogat oder einen Platzhalter für ein anderes Objekt, um den Zugriff darauf zu steuern.

- ▶ **Chain of Responsibility** Vermeidet die Kopplung des Senders einer einer Anfrage an ihren Empfänger, indem mehr als ein Objekt die Chance bekommt die Anforderung zu bearbeiten. Verkettet die empfangenden Objekte und gibt die Anforderung entlang der Kette weiter, bis ein Objekt sie bearbeitet.
- ▶ **Command** Kapselt eine Anforderung als Objekt, wodurch Clients mit verschiedenen Anforderungen parametrisiert werden, Anforderungen in eine Warteschlange gestellt oder Anfragen protokolliert und rückgängig gemacht werden können.
- ▶ **Interpreter** Definiert eine Sprache und eine Repräsentation ihrer Grammatik zusammen mit einem Interpreter, der die Repräsentation verwendet, um Sätze in der Sprache zu interpretieren.

- ▶ **Iterator** Stellt eine Möglichkeit bereit, auf die Elemente eines Aggregatobjekts zugreifen, ohne seine zugrunde liegende Darstellung zu kennen.
- ▶ **Mediator** Definiert ein Objekt, das kapselt, wie ein Satz von Objekten interagiert. Ein Mediator fördert die lose Kopplung, indem er Objekte davon abhält, sich explizit aufeinander zu beziehen, und er lässt Sie ihre Interaktion unabhängig voneinander variieren.
- ▶ **Memento** Externalisiert den internen Zustand eines Objekts ohne die Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.

Behavioral Patterns

- ▶ **Observer** Definiert eine Eins-zu-viele-Abhangigkeit zwischen Objekten, so dass bei einer Zustandsnderung eines Objekts alle abhangigen Objekte benachrichtigt und automatisch aktualisiert werden.
- ▶ **State** Erlaubt einem Objekt, sein Verhalten zu ndern, wenn sich sein interner Zustand ndert. Das Objekt ndert scheinbar seine Klasse.
- ▶ **Strategy** Definiert eine Familie von Algorithmen, kapselt jeden einzelnen und macht sie austauschbar. Strategy lsst den Algorithmus unabhangig von den Clients, die ihn verwenden, variieren.

- ▶ **Template Method** Definiert das Skelett eines Algorithmus in einer Operation, wobei einige Schritte auf Unterklassen verschoben werden. Die Template Method lässt Unterklassen bestimmte Schritte eines Algorithmus umdefinieren ohne die Struktur des Algorithmus zu ändern.
- ▶ **Visitor** Stellt eine Operation dar, die an den Elementen einer Objektstruktur durchgeführt wird. Mit Visitor können Sie eine neue Operation definieren, ohne die Klassen des Elements zu ändern, auf dem sie operiert.

Anwendungsbeispiel: Computerspiel

Game Over!



Anwendungsbeispiel: Computerspiel

- ▶ Spieler kann verschiedenartige Bonus-Gegenstände aufsammeln
 - ▶ Die Gegenstände haben unterschiedliche Effekte
 - ▶ Die Interaktion der Gegenstände mit dem Spieler folgt immer dem gleichen Muster
- ▶ Die Energie des Spielers nimmt mit der Zeit ab
- ▶ Durch Aufsammeln geeigneter Gegenstände kann der Spieler seine Energie auffüllen
- ▶ Es werden laufend neue Bonus-Gegenstände erzeugt und in der Welt platziert



Designproblem: Eine Instanz des Spiels

- ▶ Das Spiel benötigt mehrere Ressourcen die von verschiedenen Subsystemen aus ansprechbar sein sollen
 - ▶ System-Ressourcen (Bildschirm/Renderer)
 - ▶ Spielelemente (Figuren, Level)
- ▶ Es soll sichergestellt werden, dass jede Ressource nur einmal erzeugt wird und alle Subsysteme auf die gleiche Ressource zugreifen
- ▶ Globale Variablen?
- ▶ Statische Attribute?
- ▶ Flexibler: Eine Klasse, die ihre einzige Instanz selber verwaltet.

Singleton (Creational Pattern)

Ziel

Stellt sicher, dass eine Klasse nur eine Instanz hat; stellt einen globalen Punkt für den Zugang zu ihr dar.

Motivation

Einige Klassen sollten genau eine Instanz haben, z. B. Klassen, die eindeutigen physischen Ressourcen entsprechen. Diese Instanz sollte leicht zugänglich sein. Singleton macht die Klasse selbst verantwortlich den Überblick über ihre einzelne Instanz zu behalten.

Singleton (Creational Pattern)

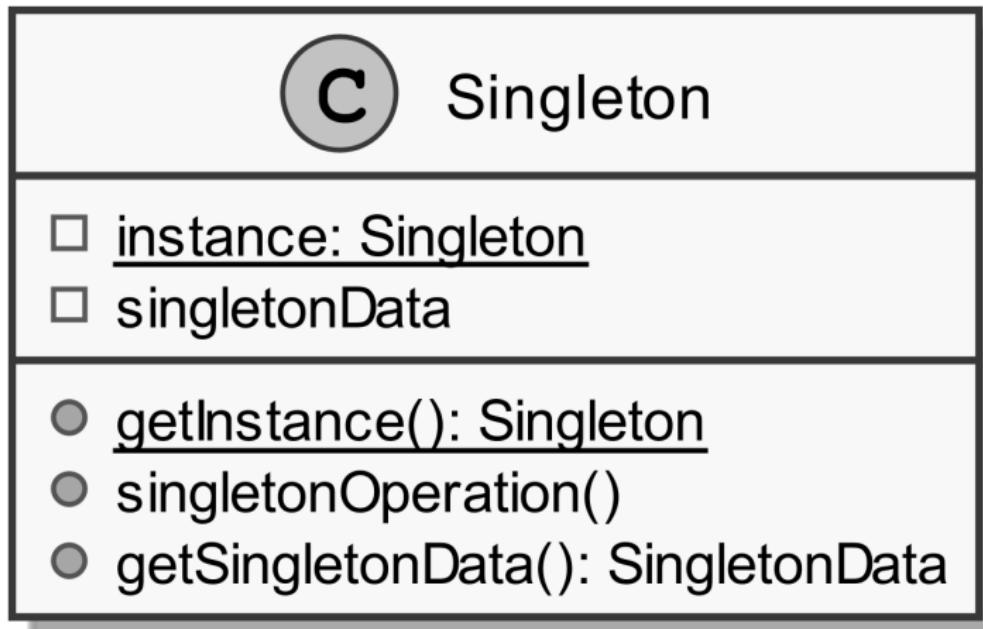
Anwendbarkeit

Verwenden Sie das Singleton-Muster, wenn

- ▶ es genau eine Instanz einer Klasse geben muss, und diese für Clients über einen bekannten Zugangspunkt zugänglich sein soll,
- ▶ die einzige Instanz durch Vererbung erweiterbar sein soll, und Clients in der Lage sein sollten, eine erweiterte Instanz zu verwenden, ohne ihren Code zu modifizieren.

Singleton (Creational Pattern)

Struktur



Singleton (Creational Pattern)

Konsequenzen

Das *Singleton* hat mehrere Vorteile:

- ▶ Kontrollierter Zugriff auf die einzige Instanz.
- ▶ Reduzierter Namensraum.
- ▶ Erlaubt eine Verfeinerung der Operationen und der Darstellung.
- ▶ Erlaubt eine variable Anzahl von Instanzen.
- ▶ Flexibler als Klassenoperationen.

Bekannte Anwendungen `java.lang.Runtime;`
`org.eclipse.core.runtime.Plugin.`

Beispiel ...

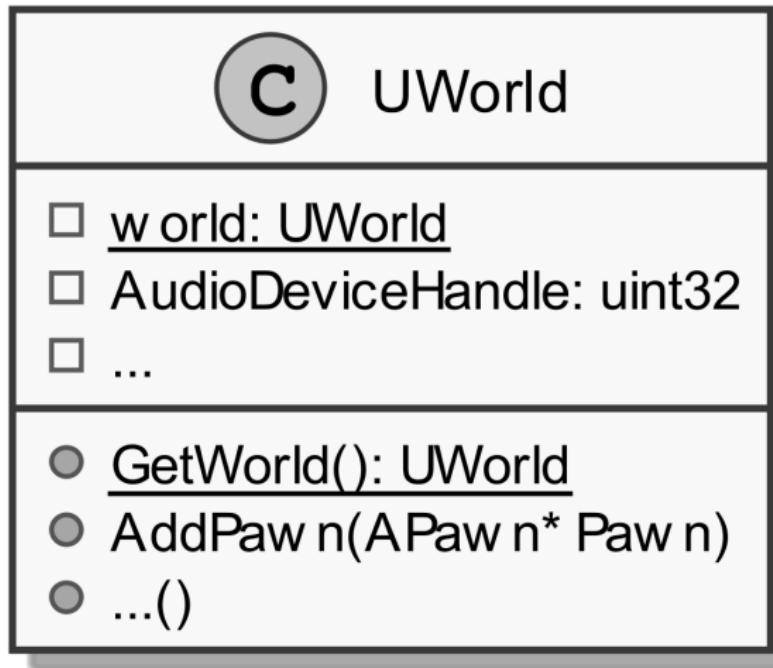
Aufgabe Singleton

Erstellen Sie eine Klasse MySingleton, die das Singleton Pattern implementiert, und eine private int-Variable `my_state`, sowie Getter und Setter dafür hat.

Verwenden Sie nach Möglichkeit TDD zum Bearbeiten dieser Aufgabe. Ergeben sich dabei Probleme durch das Singleton Pattern?

Singleton (Creational Pattern)

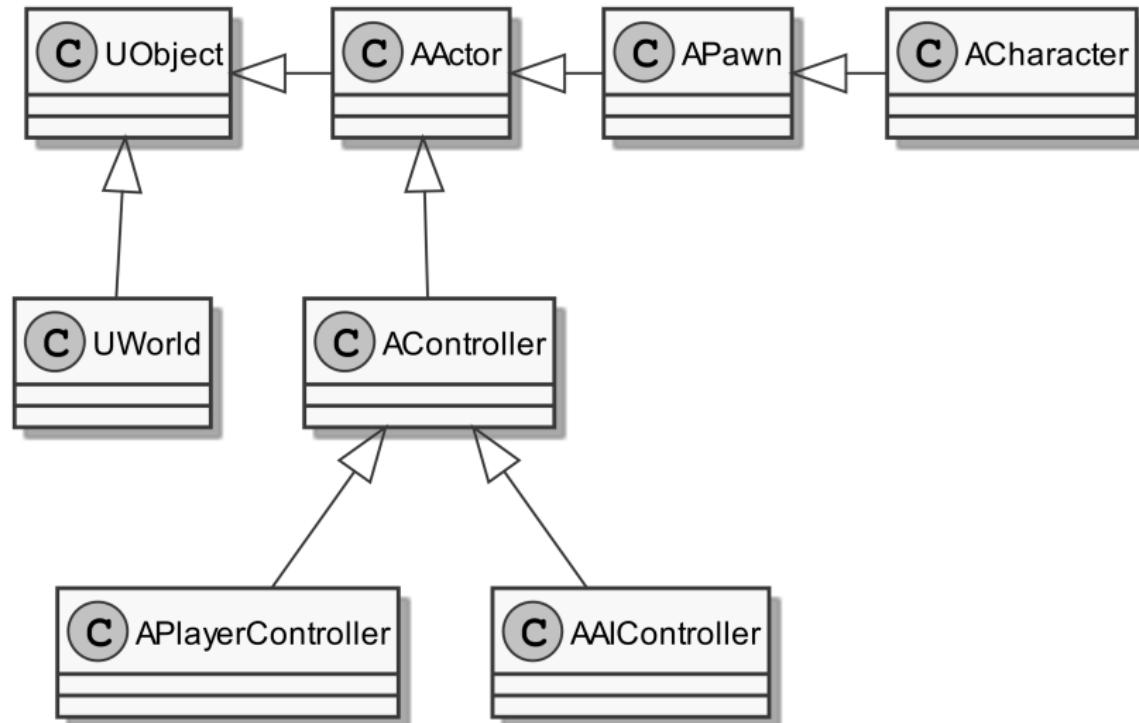
Beispiel



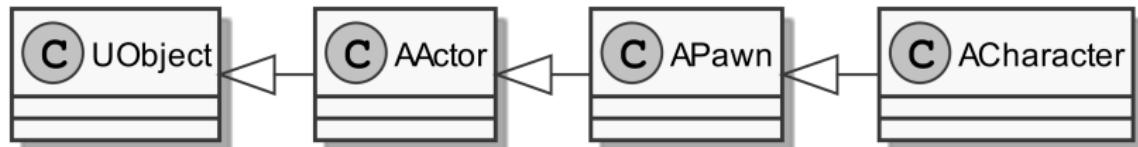
(**Bemerkung:** Die Unreal Engine funktioniert nicht so.)

Kurze Einführung in die Unreal Engine

UE4 Klassenhierarchie



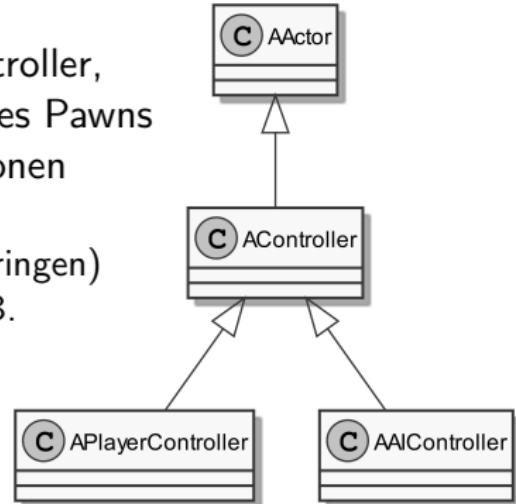
UE4 Klassenhierarchie



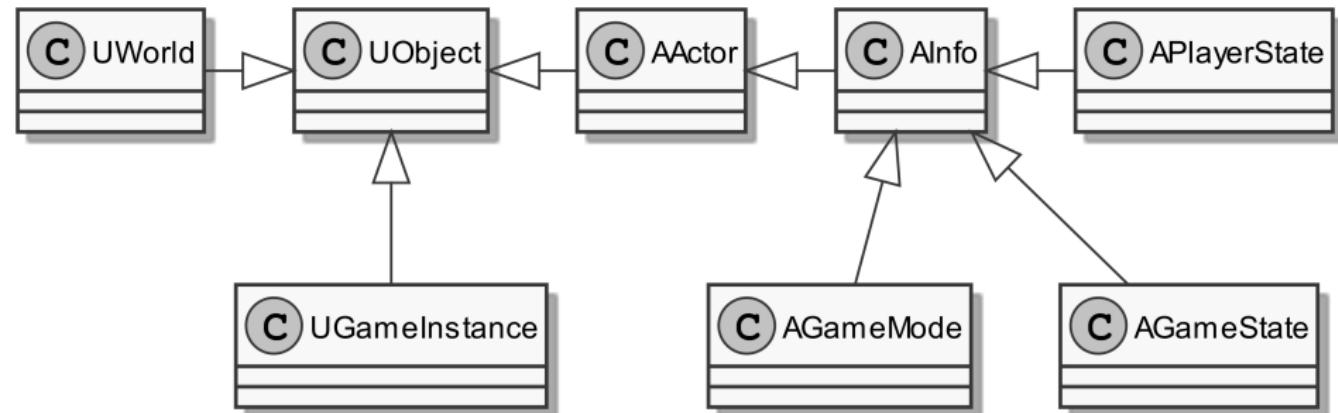
- ▶ **UObject** erweitert C++ um Funktionalität wie Garbage Collection und Reflection
- ▶ **AActor** ist die Oberklasse aller Objekte, die innerhalb eines Levels existieren können (Aktoren). Aktoren können für Netzwerkspiele zwischen Clients und Server repliziert werden.
- ▶ **APawn** ist die Oberklasse aller Aktoren, die durch einen Spieler oder den Computer kontrolliert werden können (Pawns)
- ▶ **ACharacter** sind Pawns, die herumlaufen können

UE4 Klassenhierarchie

- ▶ **AController** ist die Klasse aller Controller, d.h. aller Aktoren, die die Aktion eines Pawns kontrollieren können. Derartige Aktionen sind z.B.
 - ▶ Bewegung (z.B. Gehen/Laufen/Springen)
 - ▶ Interaktion mit der Umgebung (z.B. Aufheben von Gegenständen)
 - ▶ Abfeuern einer Waffe
 - ▶ ...
- ▶ **APlayerController** übersetzt die Eingaben des Benutzers in Aktionen für den kontrollierten Pawn
- ▶ **AAIController** kontrolliert den Pawn mit Hilfe eines Programms (der sog. künstlichen Intelligenz, KI)



UE4 Klassenhierarchie



UE4 Klassenhierarchie

Für ein Spiel können benutzerdefinierte Unterklassen erzeugt werden für

- ▶ AGameMode
- ▶ AGameState
- ▶ APlayerState
- ▶ UGameInstance
- ▶ APawn (für Spieler-Figuren)
- ▶ APlayerController (für Spieler-Figuren)

Die Klasse der UWorld-Instanz kann nicht verändert werden.

- ▶ **UGameInstance** repräsentiert die aktuelle Spielinstanz. Eine (indirekte) Instanz von UGameInstance wird automatisch erzeugt sobald das Spiel gestartet wird und existiert bis das Spiel beendet wird.
- ▶ **UWorld** repräsentiert die Umgebung, in der alle Aktoren und Komponenten existieren und erlaubt den Zugriff auf die aktuelle Game Instance, den aktuellen Game Mode, den Game State, etc.

- ▶ **AIInfo** ist eine Oberklasse für Aktoren, die keine physikalische Repräsentation im Level haben aber Informationen über die Welt enthalten
- ▶ **AGameMode** definiert die Regeln des aktuellen Spiels, die Aktoren, die im Spiel existieren, etc. (Existiert nur auf dem Server.)
- ▶ **AGameState** stellt den relevanten Zustand des Spiels dar, z.B. wo sich die Figuren auf einem Schachbrett befinden, wie viele Punkte die einzelnen Spieler im Moment haben, usw. (Wird zwischen Server und Clients repliziert.)
- ▶ **APlayerState** Instanzen existieren für alle Spieler, die an einem Spiel beteiligt sind und werden zwischen Clients und Server repliziert

Wieder zurück zu Patterns

Designproblem: Bonusitems

- ▶ Wir wollen mehrere Arten von Bonusitems (Pickups) haben, die unterschiedliche Effekte auf den Spieler haben
- ▶ Teile des Verhaltens sollen aber immer gleich bleiben

Template Method (Behavioral Pattern)

Ziel

Definieren Sie das Skelett eines Algorithmus in einer Operation und verschieben Sie einzelne Schritte in Unterklassen.

Mit Template Method können Unterklassen bestimmte Schritte eines Algorithmus umdefinieren, ohne die Struktur des Algorithmus zu verändern.

Motivation

- ▶ Ein Anwendungsframework, das sich mehreren Dokumentenklassen umgehen muss. Es gibt einige Ähnlichkeiten, aber auch Unterschiede im Umgang mit den verschiedenen Dokumenttypen.
- ▶ Das Framework definiert konkrete Operationen, die abstrakte Operationen als Teil ihrer Ausführung aufrufen.
- ▶ Instanzen des Frameworks implementieren die virtuellen Methoden.

Template Method (Behavioral Pattern)

Motivation (Beispiel)

- ▶ Pickups sollen verschiedene Effekte haben können.
- ▶ Verschiedene Player/AI-Controller sollen Benutzerdefiniertes Verhalten haben, aber auch invariantes Verhalten, das für die Engine notwendig ist.

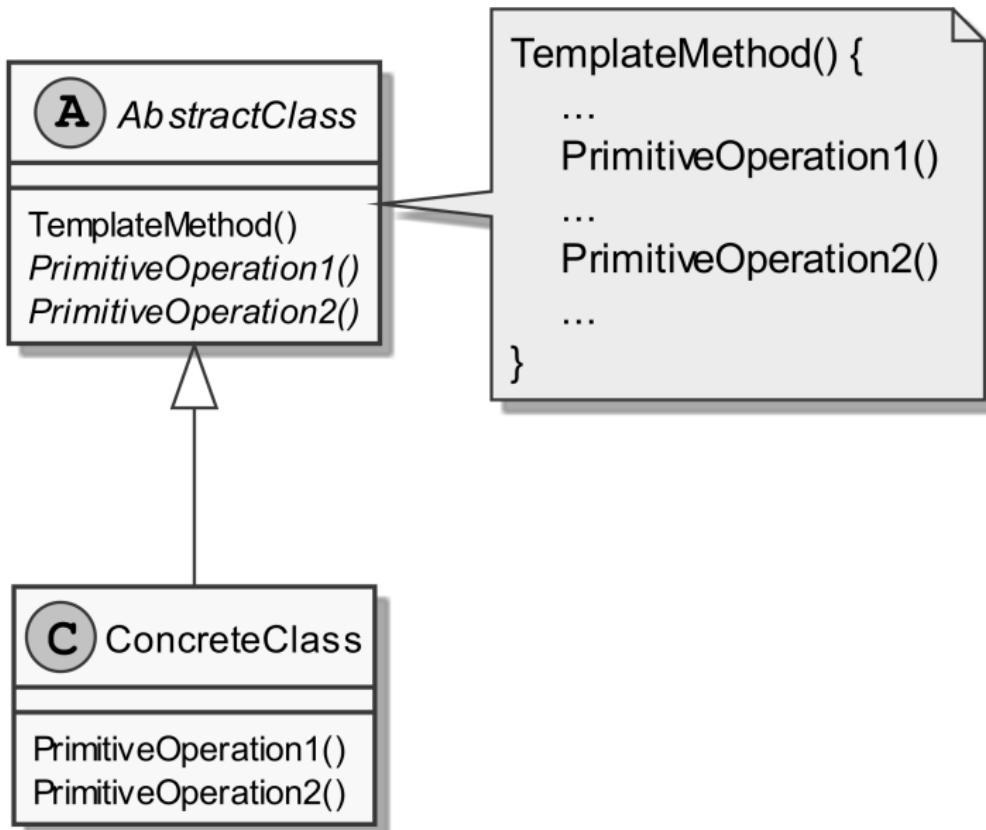
Template Method (Behavioral Pattern)

Anwendbarkeit

- ▶ um die unveränderlichen Teile eines Algorithmus einmal zu implementieren und es den Unterklassen zu überlassen, das Verhalten zu implementieren, das variieren kann
- ▶ wenn gemeinsames Verhalten von Unterklassen in einer gemeinsamen Klasse faktorisiert und lokalisiert werden soll, um Code-Duplizierung zu vermeiden
- ▶ zur Steuerung von Erweiterungen von Unterklassen

Template Method (Behavioral Pattern)

Struktur



Template Method (Behavioral Pattern)

Participants

- ▶ **AbstractClass:**
 - ▶ definiert abstrakte *primitive Operationen*, die konkrete Unterklassen implementieren (Hooks)
 - ▶ implementiert eine Template-Methode, die das Skelett eines Algorithmus darstellt
- ▶ **KonkreteKlasse:** implementiert die primitiven Operationen

Template Method (Behavioral Pattern)

Collaborations

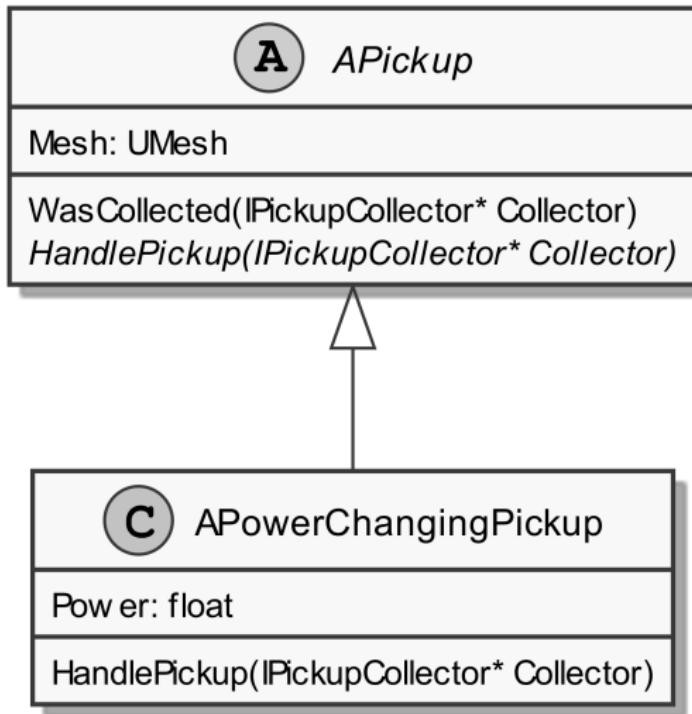
ConcreteClass stützt sich auf AbstractClass, um die invarianten Schritte des Algorithmus auszuführen

Konsequenzen

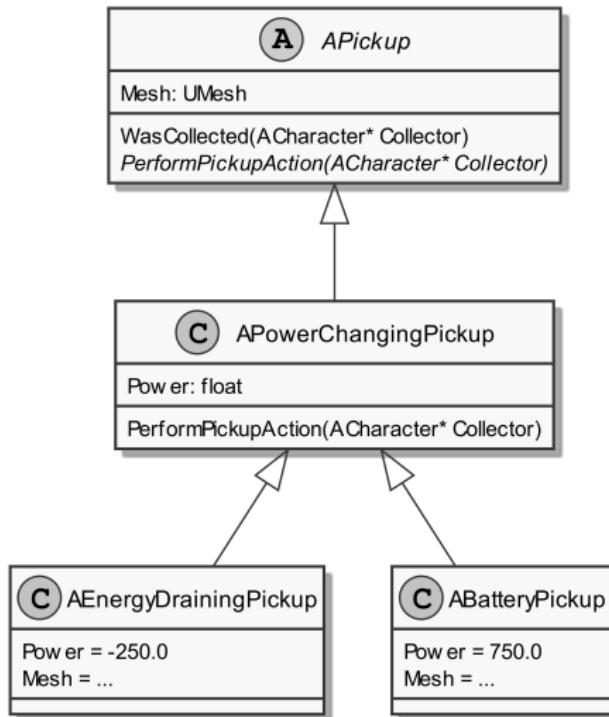
Template-Methoden sind eine grundlegende Technik der Code-Wiederverwendung. Sie führen zu einer invertierten Kontrollstruktur, die manchmal als das “Hollywood-Prinzip” bezeichnet wird.

Es ist wichtig, dass Template-Methoden spezifizieren, welche Operationen überschrieben werden *dürfen* und welche überschrieben werden *müssen*.

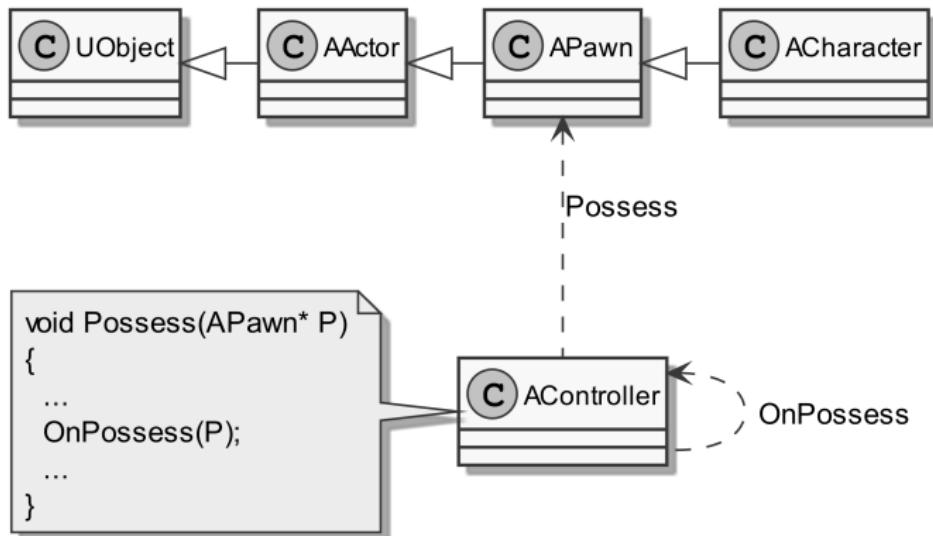
Template Method (Behavioral Pattern)



Template Method (Behavioral Pattern)



Template Method (Behavioral Pattern)



Template Method (Beispiel)

Implementieren Sie eine Klasse Aggregator, die eine Methode

```
void print_sum_and_product_up_to(int n)
```

bereitstellt, die das Produkt und die Summe aller Zahlen von 1 bis n ausgibt.

Gestalten Sie `print_sum_and_product_up_to` als Template-Methode, die zwei abstrakte Methoden

```
int sum(int n)  
int prod(int n)
```

verwendet um die Ausgabe zu erzeugen.

Erzeugen Sie zwei Unterklassen

- ▶ LoopAggregator, die die abstrakten Methoden mittels for-Schleifen berechnet und
- ▶ FormulaAggregator, die die Formeln $n(n + 1)/2$ für die Summe und entweder eine rekursive Implementierung oder

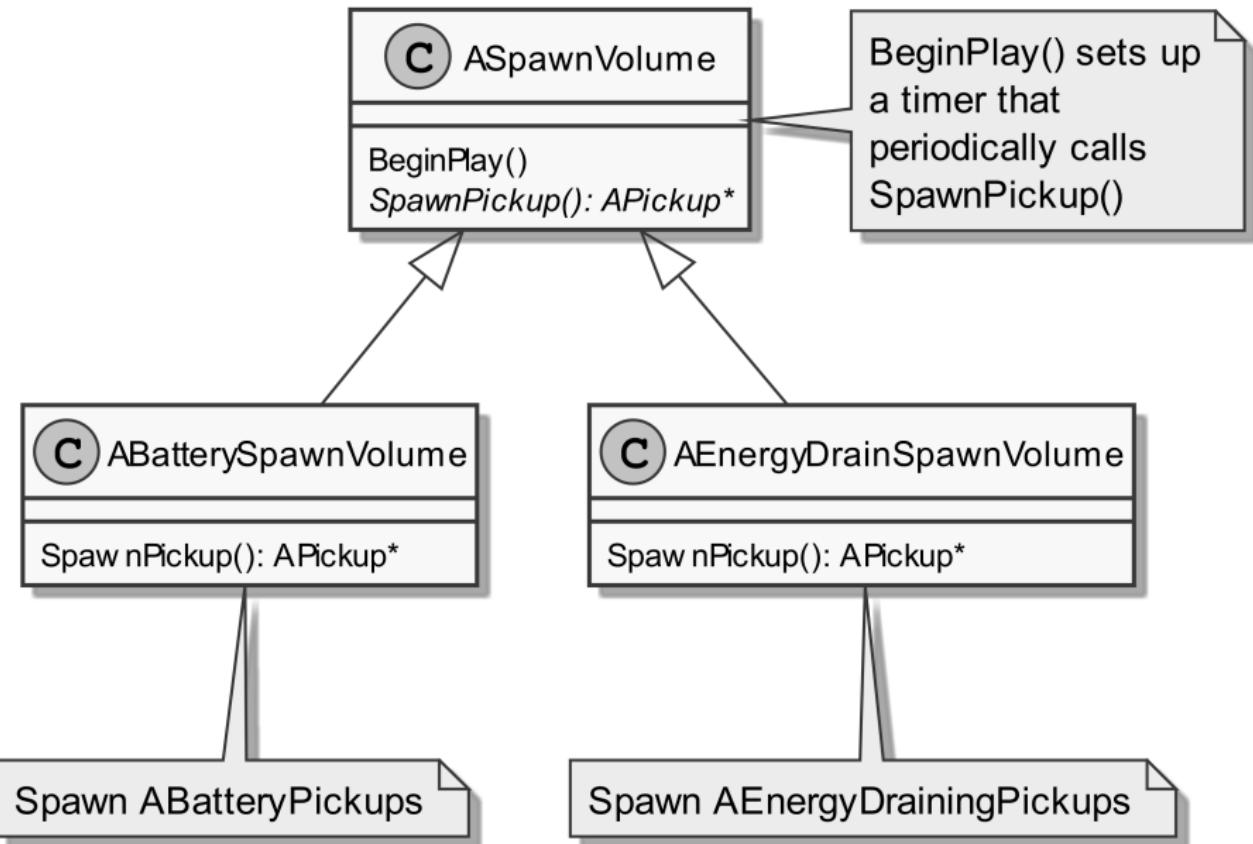
Designproblem: Erzeugen von Bonusitems

Während des Spiels werden Bonusitems erzeugt, die der Spieler aufsammeln kann.

- ▶ Verschiedenartige Bonusitems sollen verschiedenen Auswirkungen auf den Spieler haben
- ▶ An verschiedenen Orten sollen unterschiedliche Bonusitems erzeugt werden

Die Orte, an denen Bonusitems erzeugt („gespawned“) werden sollen vom Level-Designer durch sogenannte Spawn-Volumes festgelegt werden.

Erster Entwurf: Verschiedene Spawn-Volumes



Factory Method (Creational Pattern)

Ziel

Definiert eine Schnittstelle zur Erzeugung eines bestimmten Objekttyps; lässt Unterklassen entscheiden, welche konkrete Klasse instanziert werden soll.

Motivation

Betrachten Sie eine Office-Suite wie Microsoft Office oder Libre Office. Wenn die Programme in der Suite ein gemeinsames Framework verwenden, um ihre Dokumente zu verwalten, muss dieses Framework Funktionalitäten wie z. B. öffnen, schließen oder speichern für jeden Dokumenttyp bereitstellen. Jede Anwendung hat einen bestimmten Dokumententyp, mit dem sie arbeitet, aber viele der Funktionen, die das Framework bereitstellt, sind unabhängig vom dem spezifischen Dokumenttyp.

Factory Method (Creational Pattern)

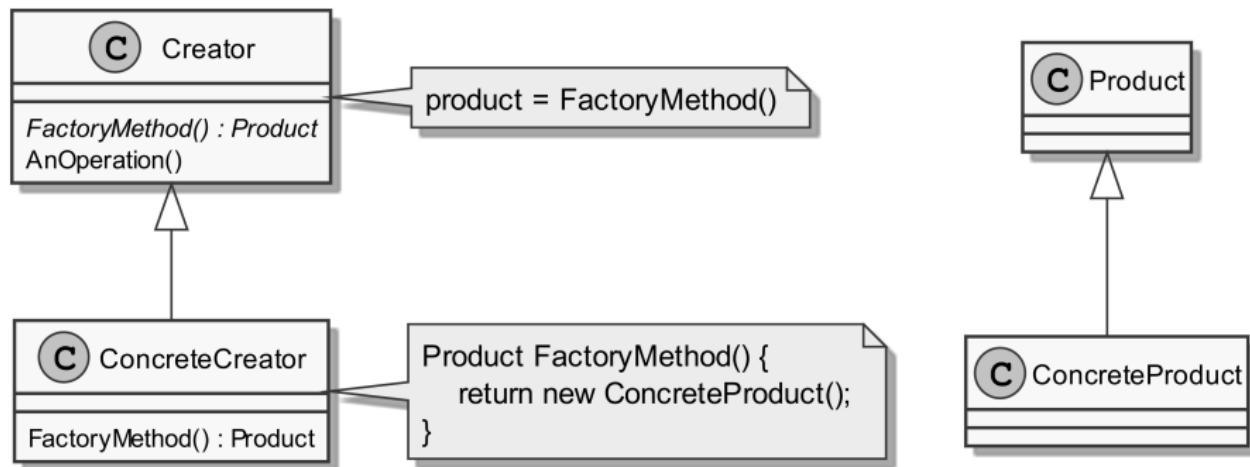
Anwendbarkeit

Verwende Factory Method wenn

- ▶ Eine Klasse eine unbeschränkte Anzahl an verschiedenen Objekttypen erzeugen muss.
- ▶ Unterklassen für die konkreten Typen, die erzeugt werden, verantwortlich sein sollen.

Factory Method (Creational Pattern)

Struktur



Factory Method (Creational Pattern)

Teilnehmer

- ▶ Produkt: Der Typ des Objekts, das erstellt werden soll (oft eine reine Schnittstelle)
- ▶ ConcreteProduct: Implementiert die Produkt Schnittstelle
- ▶ Creator: Deklariert die Factory-Methode, die eine Instanz von Produkt zurückgibt. Kann eine Standard Implementierung definieren.
- ▶ ConcreteCreator: Überschreibt die Factory-Methode.

Factory Method (Creational Pattern)

Kollaborationen

Creator verlässt sich darauf, dass seine Unterklassen die erforderliche Factory Method bereitzustellen.

Konsequenzen

- ▶ Keine Notwendigkeit für anwendungsspezifische Klassen im Framework-Code.
- ▶ Framework-Code kann mit jeder benutzerdefinierten konkreten Klasse arbeiten.
- ▶ Clients müssen die Klasse Creator erben, wodurch Teile des Clients eng an das Framework koppeln.
- ▶ Factory Method bietet Hooks für Unterklassen.

Beispiel ...

Designproblem: Erzeugen von Bonusitems

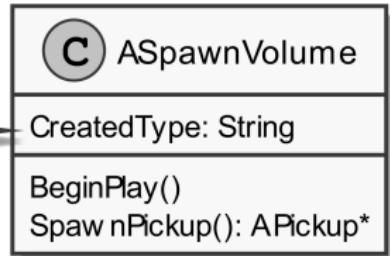
Zur Erinnerung: Die Orte, an denen Bonusitems erzeugt werden sollen vom Level-Designer durch sogenannte Spawn-Volumes festgelegt werden.

Problem mit dem angegebenen Design

Es ist für den Designer nicht möglich, den von einem Spawn-Volume erzeugten Objekttyp zu verändern.

Zweiter Entwurf: Konfigurierbares Spawn-Volume

```
APickup* ASpawnVolume::SpawnPickup() {
    APickup* P;
    if (CreatedType == "Battery")
    {
        P = new ABatteryPickup();
    }
    else
    {
        P = new AEnergyDrainingPickup();
    }
    // Place P into the level and return it
}
```

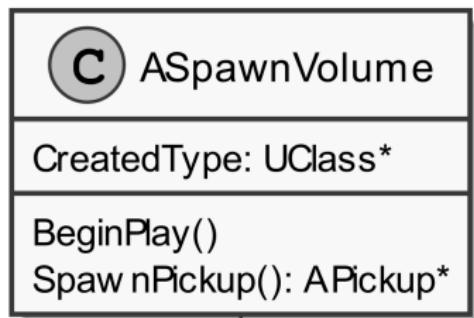


Zweiter Entwurf: Konfigurierbares Spawn-Volume

Dieser Entwurf erlaubt die Konfiguration des erzeugten Objekts, hat aber einige Nachteile:

- ▶ Die Verwendung von Strings zur Angabe des Typs ist fehleranfällig
Besser: Verwendung einer Enumeration, aber dann muss für jeden neuen Pickup-Typ die Enumeration angepasst werden
- ▶ Die SpawnPickup()-Methode muss alle Pickup-Typen kennen und per Fallunterscheidung den richtigen Typ auswählen

Dritter Entwurf: Spawn-Volume mit Reflection



```
APickup* ASpawnVolume::SpawnPickup() {  
    APickup* P { NewObject<APickup>(this, CreatedType) };  
    // Place P into the level and return it  
}
```

Dritter Entwurf: Spawn-Volume mit Reflection

C

ASpawnVolume

CreatedType: TSubclassOf<APickup>

BeginPlay()

SpawnPickup(): APickup*

```
APickup* ASpawnVolume::SpawnPickup() {
```

```
...
```

```
    APickup* P { SpawnActor<APickup>(CreatedType, ...);  
    return P;
```

```
}
```

Reflection (Architectural Pattern)

From: Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael. *Pattern-Oriented Software Architecture, A System of Patterns: Volume 1*, p. 193. Wiley. (See also PLoPD2, p. 271)

Abstract

Das Reflection-Architekturmuster bietet einen Mechanismus zur Struktur und Verhalten von Softwaresystemen dynamisch zu verändern. Es unterstützt die Modifikation grundlegender Aspekte, wie Typ-Strukturen und Funktionsaufrufmechanismen. Bei diesem Muster wird eine Anwendung in zwei Teile aufgeteilt. Eine Metaebene liefert Informationen über ausgewählte Systemeigenschaften und macht die Software selbst-reflexiv. Eine Basisebene umfasst die Anwendungslogik. Ihre Implementierung baut auf der Metaebene auf. Änderungen an den Informationen in der Metaebene wirken sich auf das nachfolgende Verhalten der Basisebene aus.

Reflection (Architectural Pattern)

Auch Bekannt als

Open Implementation, Meta-Level Architecture

Beispiel

[Persistenz Layer für C++ der für beliebige C++-Programme funktionier.]

Kontext

Systeme, die sich selber dynamisch inspizieren oder verändern können. . .

Designproblem: Mehrere Bonusitems pro Region

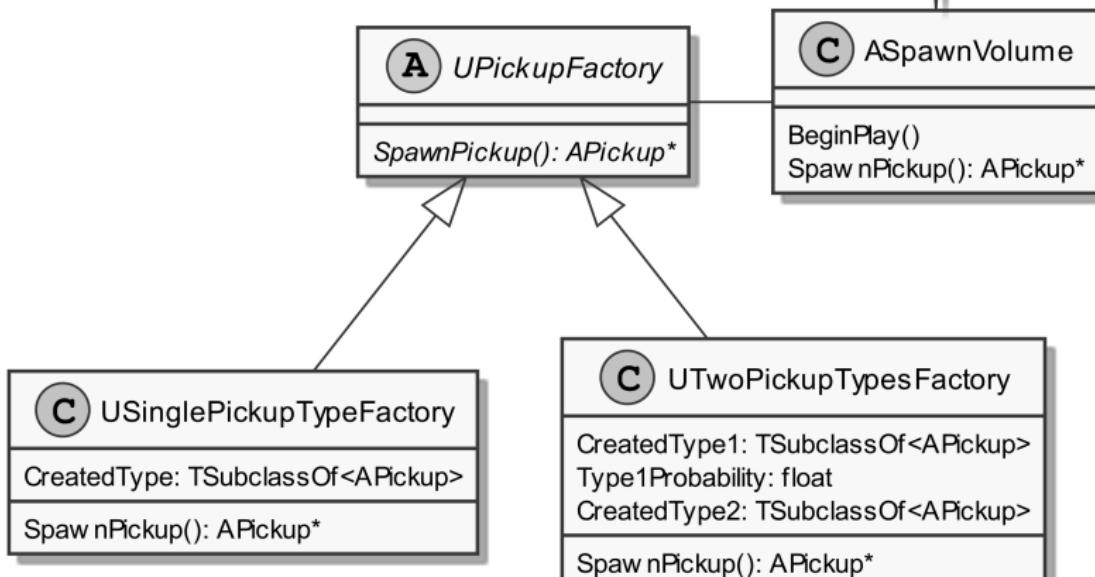
Unser jetziges Design hat einige positive Eigenschaften:

- ▶ Der von einem Spawn-Volume erzeugte Pickup-Typ kann leicht geändert werden
- ▶ Das Hinzufügen von neuen Pickup-Typen geht ohne jede Änderung an der Spawn-Volume Implementierung

Aber: Es ist nicht leicht möglich in einem Spawn-Volume mehrere Bonusitem-Typen zu erzeugen.

Vierter Entwurf: Spawn-Volume mit Abstract Factory

```
Pickup* ASpawnVolume::SpawnPickup() {  
    ...  
    return { PickupFactory->SpawnPickup();  
}
```



Abstract Factory (Creational Pattern)

Ziel

Bietet eine Schnittstelle zum Erstellen von Familien verwandter oder abhängiger Objekten ohne Angabe ihrer konkreten Klassen.

Motivation

UI-Toolkit, das mehrere Looks und Feels unterstützt. Das Widget-Look und feel sollten nicht hart kodiert sein. Definiert eine abstrakte WidgetFactory-Klasse, die eine einheitliche Schnittstelle für Widgets für verschiedene Looks und Feels bietet, instanziert eine konkrete Unterklasse für jedes Look and Feel.

Abstract Factory (Creational Pattern)

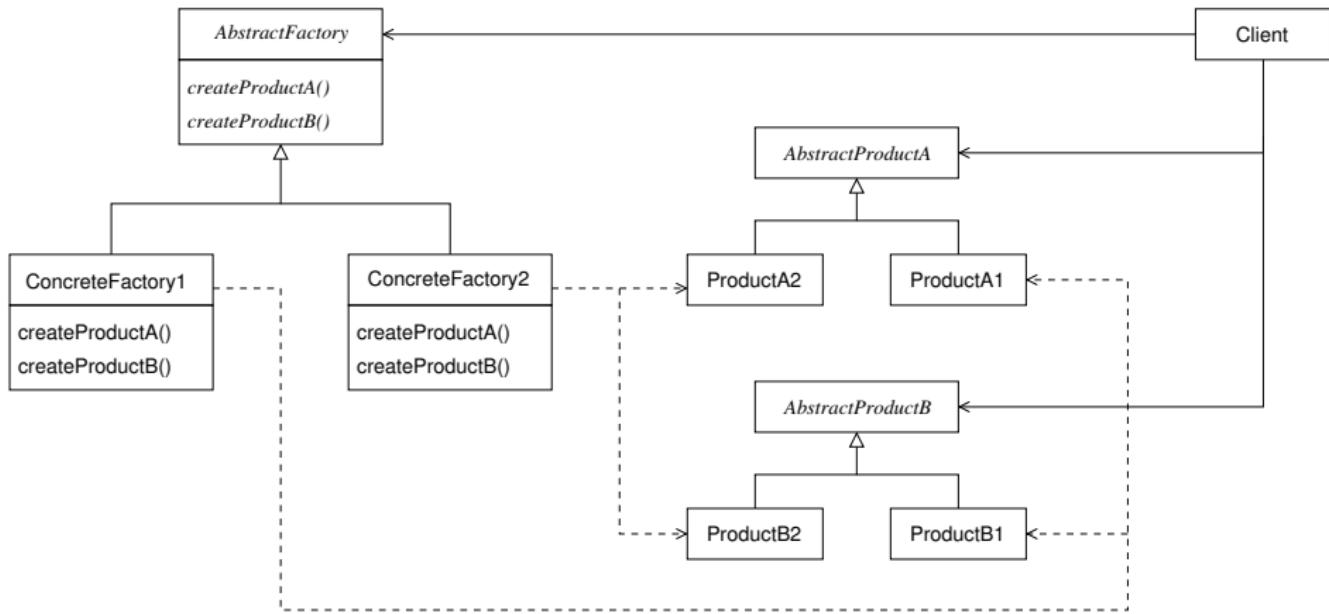
Anwendbarkeit

Verwenden Sie das abstrakte Fabrikmuster, wenn

- ▶ ein System unabhängig davon sein soll, wie seine Produkte erstellt, komponiert und dargestellt werden
- ▶ ein System mit einer von mehreren Produktfamilien konfiguriert werden soll
- ▶ ein System eine Familie von verwandten Produktobjekten ist, die nur gemeinsam verwendet werden sollen und diese Einschränkung erzwungen werden soll
- ▶ Sie eine Klassenbibliothek von Produkten bereitstellen wollen und nur ihre Schnittstellen, nicht ihre Implementierungen offenlegen wollen

Abstract Factory (Creational Pattern)

Struktur



Abstract Factory (Creational Pattern)

Konsequenzen

Das Muster *Abstract Factory*

- ▶ isoliert konkrete Klassen
- ▶ macht den Austausch von Produktfamilien einfach
- ▶ fördert die Konsistenz zwischen Produkten
- ▶ erschwert die Unterstützung neuer Arten von Produkten
- ▶ erhöht die Code-Komplexität (erheblich)

Bekannte Anwendungen Toolkit in AWT.

Verwandte Patterns

- ▶ Factory Method
- ▶ Prototype
- ▶ Singleton (the factory is often a Singleton)

Abstract Factory (Creational Pattern)

Implementieren Sie eine Klassenhierarchie für Ebooks mit abstrakten Klassen Book und Chapter. Ein Buch soll den Namen des Autors, einen Titel und einen Vektor von Kapiteln beinhalten.

Implementieren Sie zwei Konkrete Implementierungen der Klassen: HtmlBook und HtmlChapter, sowie PdfBook und PdfChapter.

Schreiben Sie eine Abstract Factory Klasse, die ein Buch erzeugen kann, wenn Autor und Zahl der Kapitel angegeben werden.

Schreiben Sie konkrete Implementierungen, die PDF und HTML-Bücher erzeugen.