Software Design Patterns

Ein Vortrag von:
Tim Spuhle, Annemarie Schaumann, Jan Schmalbruch, Alina Schmidt, Moritz Sander

Gruppe CGG

“Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over”

Christopher Alexander

Gliederung

1. Dependency Injection - Creational Pattern
2. Bridge - Structural Pattern
3. Adapter - Structural Pattern
4. State - Behavioural Pattern
5. Thread-Pooling - Concurrency Pattern
6. Abschluss

Dependency Injection

Creational Pattern

Problem:

- Objekte benötigen Instanzen anderer Objekte
  - **ABER:** Klassen sollten entsprechende Objekte nicht erschaffen (Single-Responsibility-Prinzip)

- Genauer Laufzeittyp des benötigten Objekts oft unbekannt oder sogar irrelevant
  - z.B. abstraktes Objekt “Auto” -> Diesel oder Benziner? Welche Reifen?

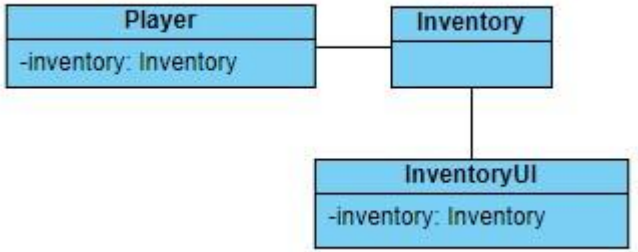
Lösungsansatz: Injizierung von Außen

3 geläufige Möglichkeiten der Implementierung:

- Konstruktor-Injizierung = Abhängigkeit wird direkt im Konstruktor überreicht
- Setter-Injizierung = Abhängigkeit wird über einen Konstruktor gesetzt
- Interface-Injizierung = Definierung einer Schnittschnelle für die Injizierung

Andere Möglichkeiten beinhalten je nach Programmiersprache Reflexion oder auch direkte Verweise auf bestimmte Stellen im Speicher.

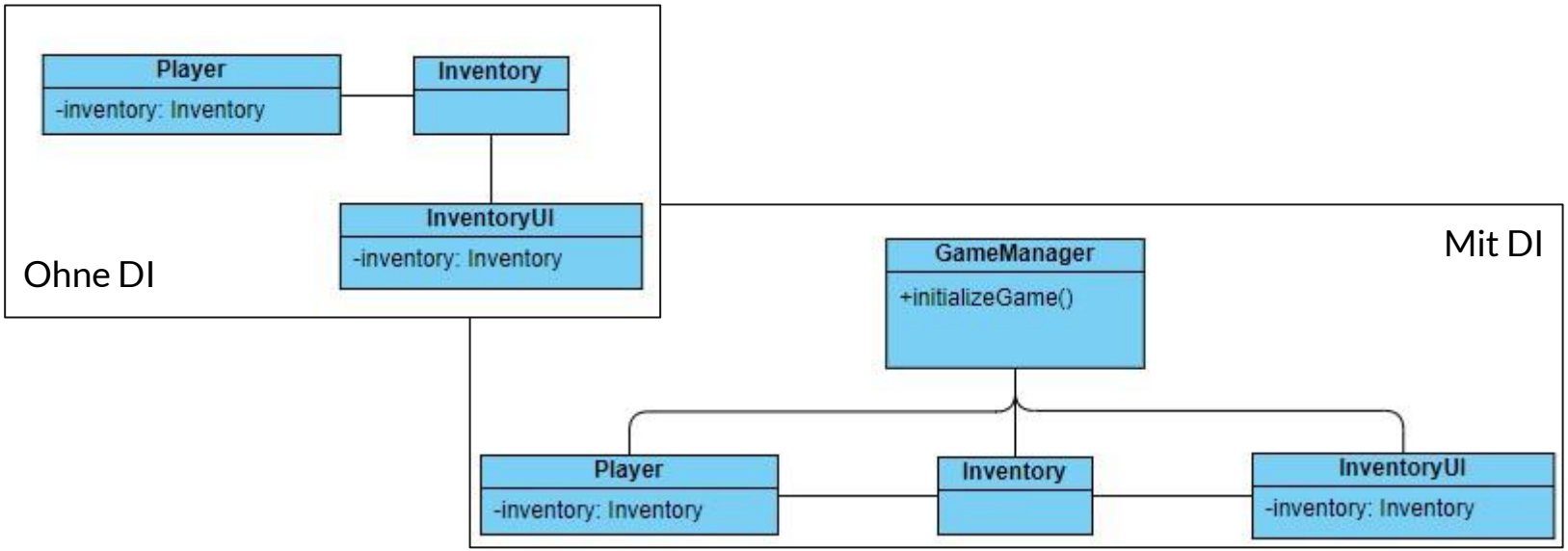
Konkretes Beispiel:



Szenario aus der Spieleentwicklung:

- Klassen “Player”, “Inventory”, “InventoryUI”
- Ohne DI:
  - Variante 1: Player und UI erstellen jeweils ein eigenes Inventory-Object -> nicht funktional
  - Variante 2: Player erstellt Inventory und Inventory erstellt UI -> Vermischung der Schichten
- Mit DI:
  - Variante 1: Player erstellt Inventory und injiziert dieses in die UI -> immernoch Vermischung der Schichten
  - Variante 2: Player und UI bekommen beide das gleiche Inventory-Object von einer injizierenden Klasse

Konkretes Beispiel: Szenario



Konkretes Beispiel: Konstruktor-Injektion

```
class GameManager {
  public void initializeGame(){
    Inventory inventory = new Inventory();
    Player player = new Player(inventory);
    InventoryUI inventoryUI = new InventoryUI(inventory);
  }
}
```

```
class Player {
  private Inventory inventory;

  public Player (Inventory inventory) {
    this.inventory = inventory;
  }
}
```

Konkretes Beispiel: Setter-Injektion

```
class GameManager {
  public void initializeGame(){
    Inventory inventory = new Inventory();
    Player player = new Player();
    InventoryUI inventoryUI = new InventoryUI();

    player.setInventory(inventory);
    inventoryUI.setInventory(inventory);
  }
}
```

```
class Player {
  private Inventory inventory;

  public Player () {}

  public void setInventory(Inventory inventory) {
    this.inventory = inventory;
  }
}
```

Konkretes Beispiel: Interface-Injektion

```
class GameManager {
  public void initializeGame(){
    Inventory inventory = new Inventory();
    Player player = new Player();
    InventoryUI inventoryUI = new InventoryUI();

    player.inject(inventory);
    inventoryUI.inject(inventory);
  }
}
```

```
class Player implements IHasInjectible {
  private Inventory inventory;

  public Player () {}

  public void inject(Injectible injectible) {
    if(injectible instanceof Inventory) {
      inventory = (Inventory)injectible;
    }
  }
}
```

Konkretes Beispiel: Interface-Injektion

```
interface IHasInjectible {
  public void inject(Injectible injectible);
}
```

```
class Inventory implements Injectible {
  // some defining stuff here
}
```

Bonus Pattern:

Marker Interfaces sind leere Interfaces, die man als “Markierung” oder auch “Tags” von Klassen einsetzen kann.

```
interface Injectible {
}
```

Konsequenzen

Pro

- Einhaltung des SRP
- Einzelne Klassen leichter testbar
  - Benötigte Objekte können durch “MockUps” ersetzt werden
- Weniger Berührungspunkte mit den injizierten Klassen = weniger Fehlerpotential bei Änderungen
  - z.B. Ändern eines Konstruktors

Kontra

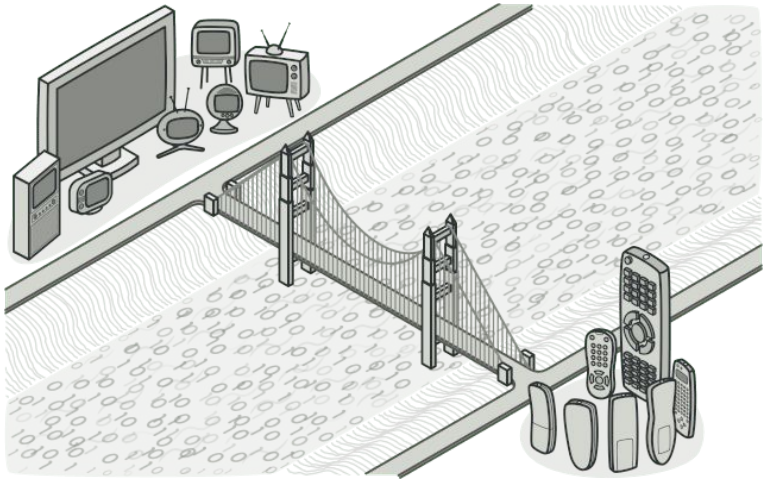
- “Overuse” kann zu unnötigem Aufblähen des Codes führen
- Vor Nutzung des injizierten Objekts muss geprüft werden, ob es injiziert wurde
  - NullPointerException (Java)
  - NullReferenceException (C#)
- Um alle Details eines bestimmten Callstacks zu erfassen, muss evtl zwischen Klassen gesprungen werden

Bridge

Structural Pattern

Was ist ein Bridge?

- Ein Structural Design Pattern
- Grosse Klassen oder eine Gruppe von eng miteinander verwandten Klassen werden in zwei separaten Hierarchien unterteilt

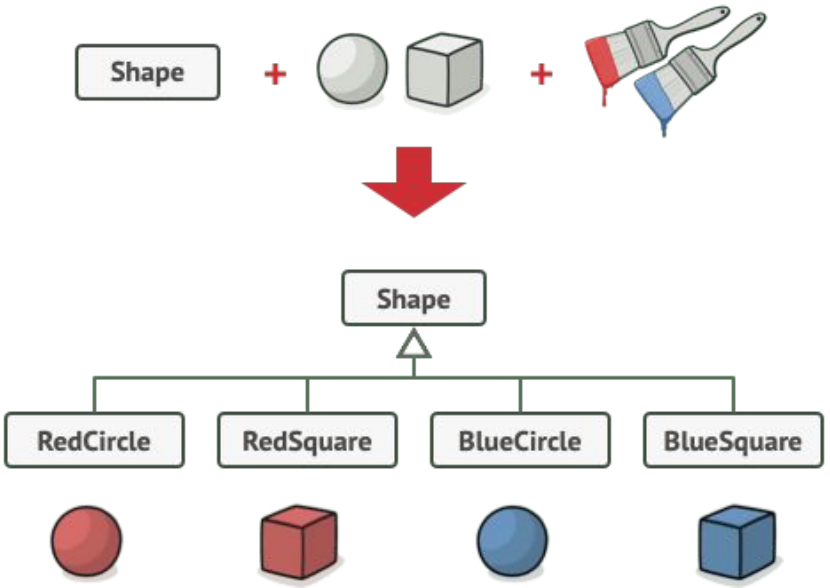


Wann wird es angewendet?

- Jede Änderung in der Implementierung sollte keinen Einfluss auf die Abstraktion haben
- Implementierungen aus verschiedenen Klassen können gleichzeitig verwendet werden
- Eine Implementierung kann von mehreren Objekten gemeinsam genutzt werden
- Eine monolithische Klasse mit mehreren Varianten kann aufgeteilt und organisiert werden
- Eine Implementierung kann für mehrere Objekte freigeben werden

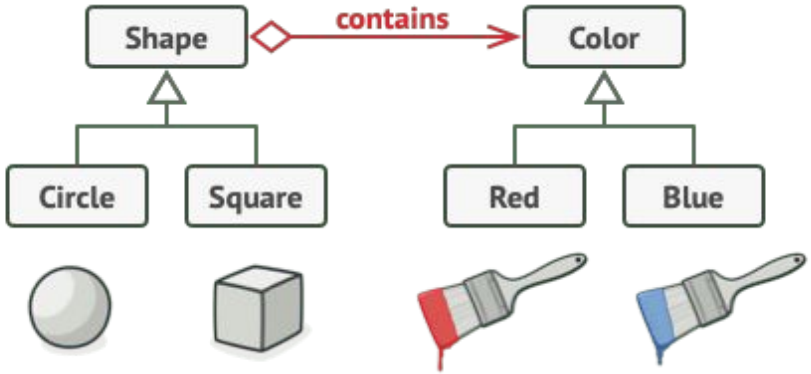
Beispiel

- Wir haben eine geometrische Shape-Klasse mit zwei Unterklassen: Circle und Square.
- Wir wollen diese Klassenhierarchie erweitern, um rote und blaue subclasses zu erstellen.
- Da wir jedoch bereits zwei Unterklassen haben, müssen wir 4 Klassen- kombinationen erstellen.



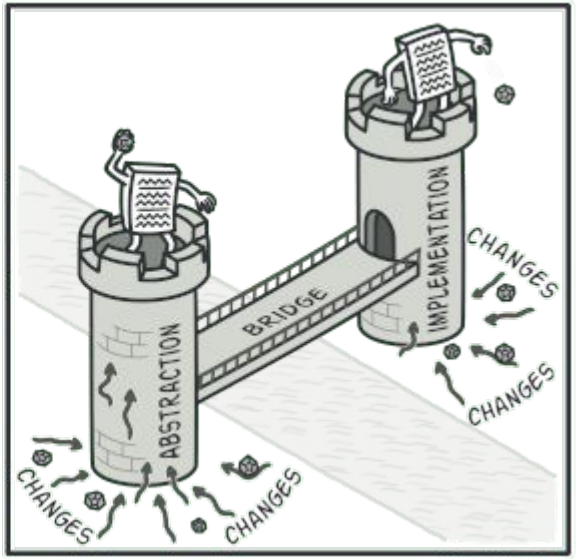
Beispiellösung

- Wir extrahieren eine der Dimensionen in eine separate Klassenhierarchie, sodass die ursprünglichen Klassen auf ein Objekt der neuen Hierarchie verweisen.
- Dadurch können wir die Color und die Shape bezogenen Teile in eine eigene Klassen mit jeweiligen Unterklassen unterteilen.
- Die Referenz fungiert als **Bridge** zwischen den Klassen Shape und Color.



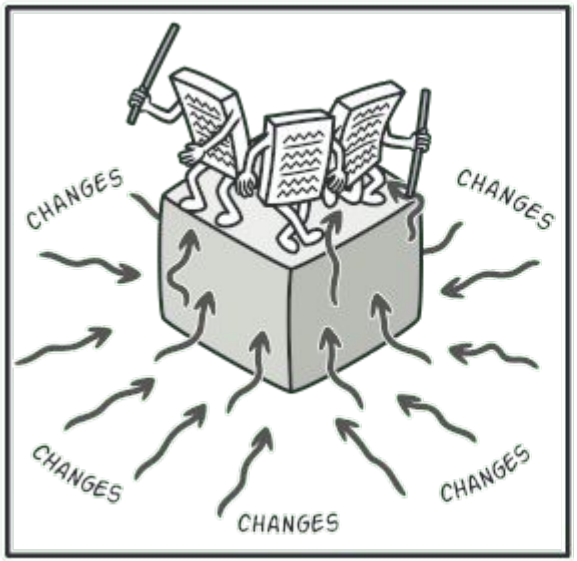
Was sind die Vorteile?

- Es können plattformunabhängige Klassen erstellt werden
- Klassen und Implementierungen vor dem Client ausblenden
- Unabhängig voneinander können neue Abstraktionen und Implementierungen eingeführt werden
- Single Responsibility Principle: Fokus der Abstraktion auf die übergeordnete Logik und Fokus der Implementierung auf die “platform details”
- Eine starke Erhöhung der Klassenzahl kann vermieden werden



Was sind die Probleme?

- Vererbungshierarchien können unübersichtlicher werden
- Wartungen können mehr Zeit beanspruchen

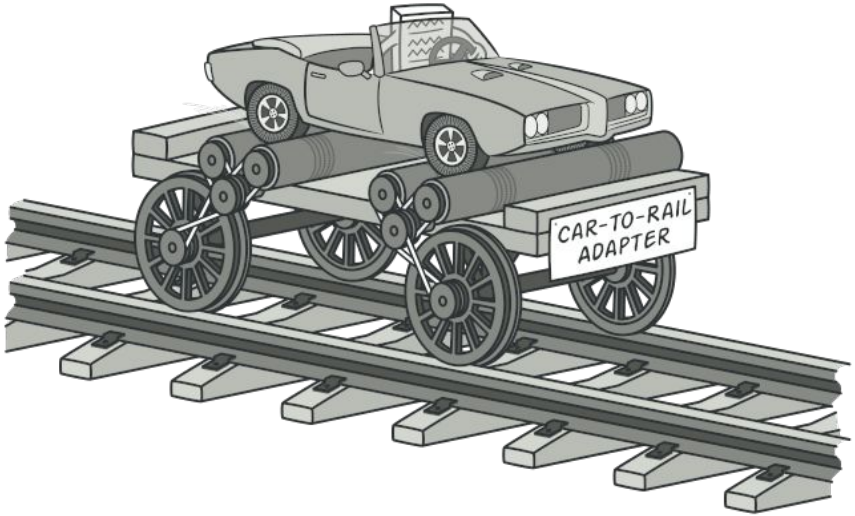


Adapter

Structural Pattern

Problem:

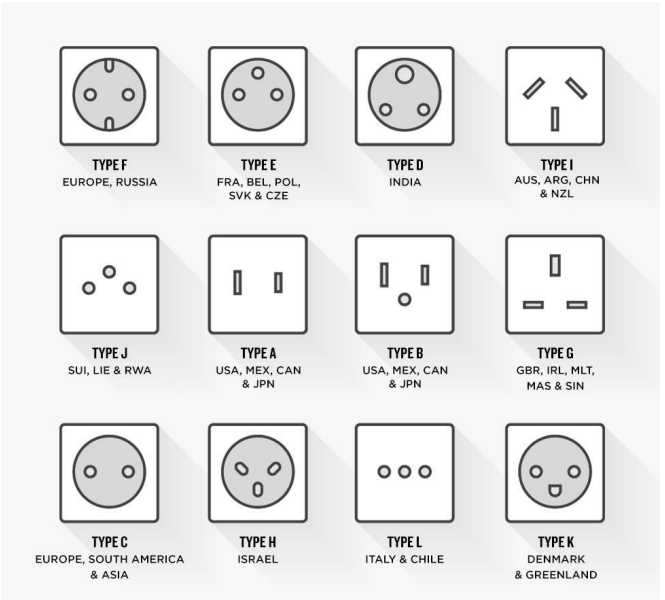
- Zwei Objekte mit inkompatiblen Interfaces
- Können nicht kommunizieren, weil z.B. verschiedene Formate benutzt werden



Analogie - Steckdosen in verschiedenen Ländern

shutterstock.com · 1029011536

?



Lösungsansatz: Einsatz eines Adapters

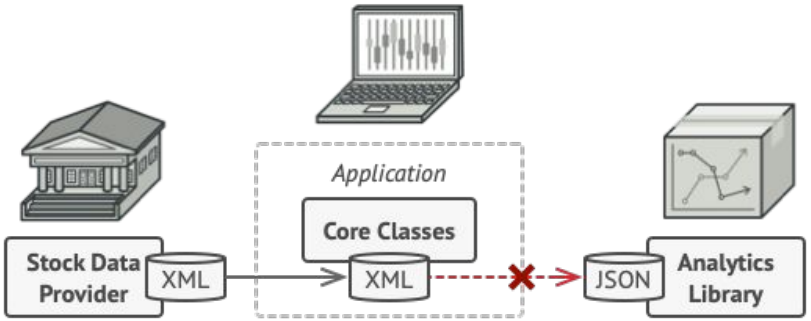
- Konvertiert Interface von Objekt A, so dass Objekt B es verstehen kann
- Adapter umhüllt Objekt A, ohne das Objekt A etwas davon mitbekommt
- Ablauf:

  1. Adapter bekommt Interface, das kompatibel mit Objekt A ist

  2. Mit Interface kann Objekt A Methoden des Adapters aufrufen

  3. Adapter kann Methoden von Objekt B in verständlicher Weise aufrufen

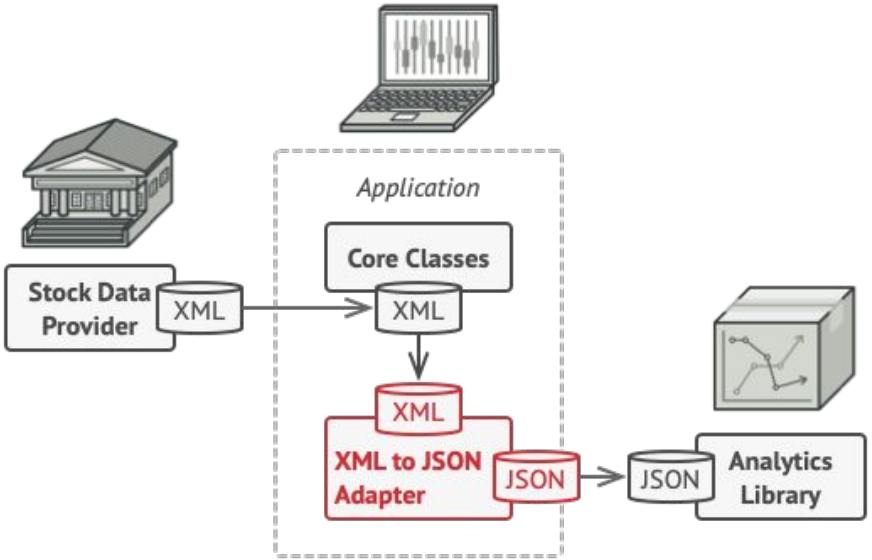
Konkretes Beispiel



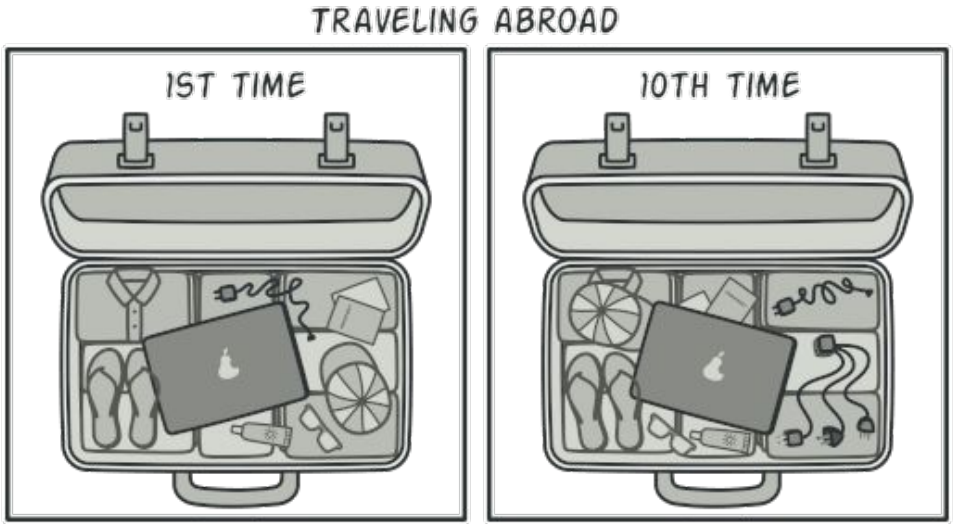
- Applikation zum Verfolgen des Börsenmarkts
- Börsendaten im XML-Format
- Library zum Analysieren im JSON-Format
- Ändern der Library zu XML-Format sehr umständlich und eventuell nicht möglich

Konkretes Beispiel

- XML to JSON Adapter für jede Klasse der Library mit der unsere Applikation interagiert
- Applikation interagiert statt mit der Library mit dem Adapter



Analogie - Steckdosenadapter



Konsequenzen

Pro

- Single Responsibility Principle, Logik der Datenkonvertierung klar von der Fachlogik der Applikation getrennt
- Es können jeder Zeit neue Adapter in das Programm eingefügt werden, ohne den existierenden Code groß zu verändern

Kontra

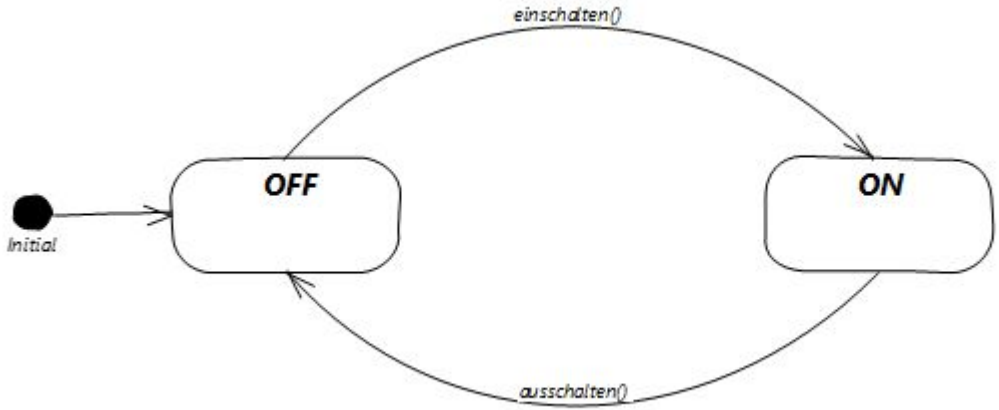
- Komplexität nimmt zu, da neue Interfaces und Klassen eingeführt werden müssen

State

Behavioural Pattern

Problem:

- Unterschiedliche Verhaltensweisen je nach innerem Zustand der Klasse



Lösungsansatz ohne State-Pattern

- Zustand wird als ein Integer-Wert (oder einer Kombination mehrerer Integer-Werte) gespeichert
- in jeder Operation wird dieser Integer-Wert überprüft und dementsprechend wird ein Verhalten ausgeführt

```
class Object{
  private int currentState;
  private static const int STATE1 = 0;
  private static const int STATE2 = 1;

  public void doSomething(){
    if(currentState==STATE1){
      //do something
    }
    else if(currentState==STATE2){
      //do something
    }
  }
}
```

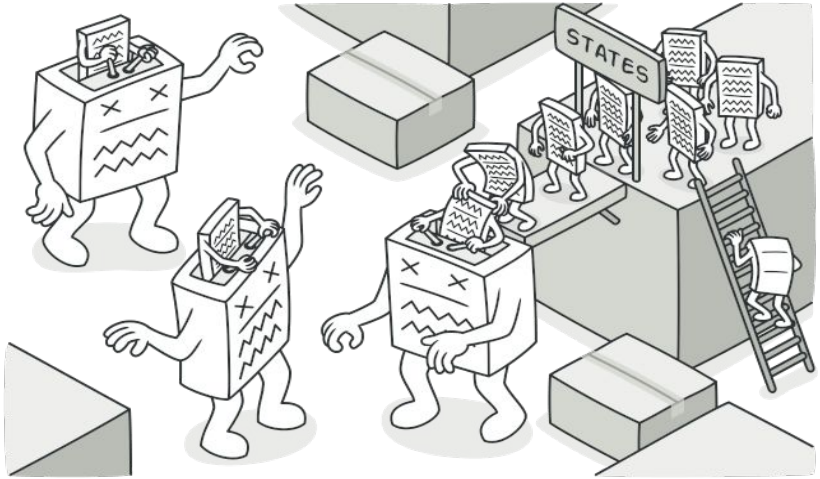
Probleme dieses Lösungsansatzes:

- unübersichtlich
- schlecht erweiterbar
- fehleranfällig

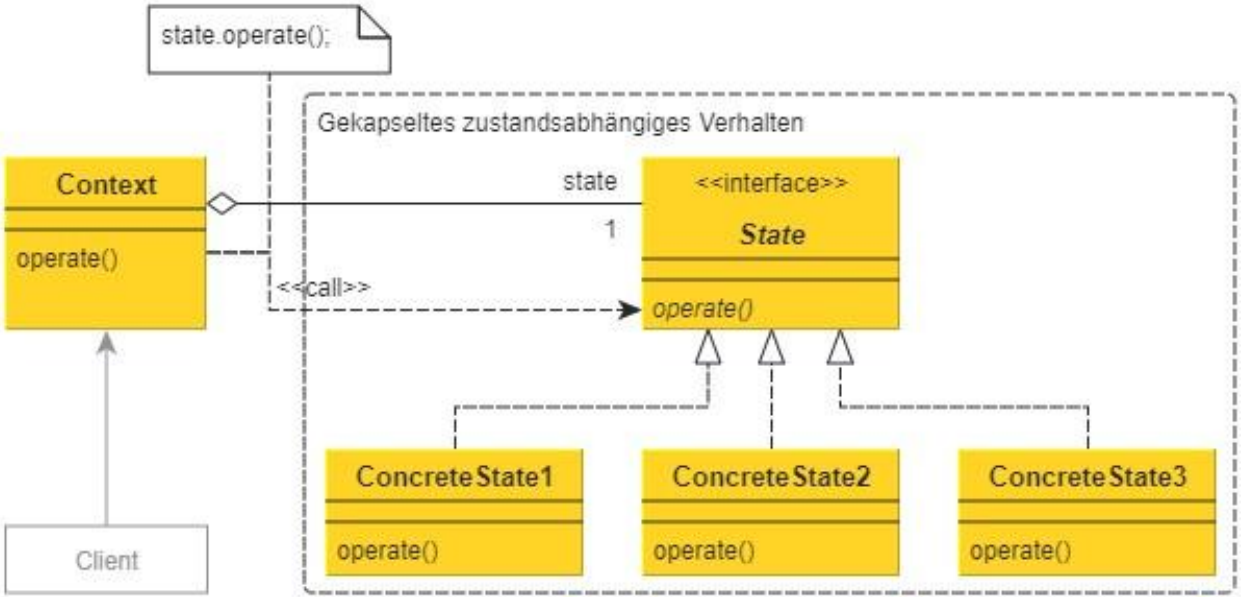
```
class Object{
  private int currentState;
  private static const int STATE1 = 0;
  private static const int STATE2 = 1;
  private static const int STATE3 = 2;
  public void doSomething(){
    if(currentState==STATE1){
      //do something
    }
    else if(currentState==STATE2){
      //do something
    }
    else if(currentState==STATE3){
      //do something
    }
  }
  public void doSomeOtherThing(){
    if(currentState==STATE1){
      //do something
    }
    else if(currentState==STATE2){
      //do something
    }
    else if(currentState==STATE3){
      //do something
    }
  }
}
```

Lösungsansatz mit State-Pattern

- Das Objekt welches sein Verhalten ändern soll wird als “Context” bezeichnet
- States werden als Objekte umgesetzt



Lösungsansatz mit State-Pattern



Konkretes Beispiel

Konsequenzen

Pro

- Single Responsibility Principle
- Open/Closed Principle
- Leicht erweiterbar
- Explizite Zustandsübergänge
- Einfach verständlich

Kontra

- Möglicherweise zu viel Aufwand für Klassen die nur sehr wenige Zustände haben

Thread-pooling

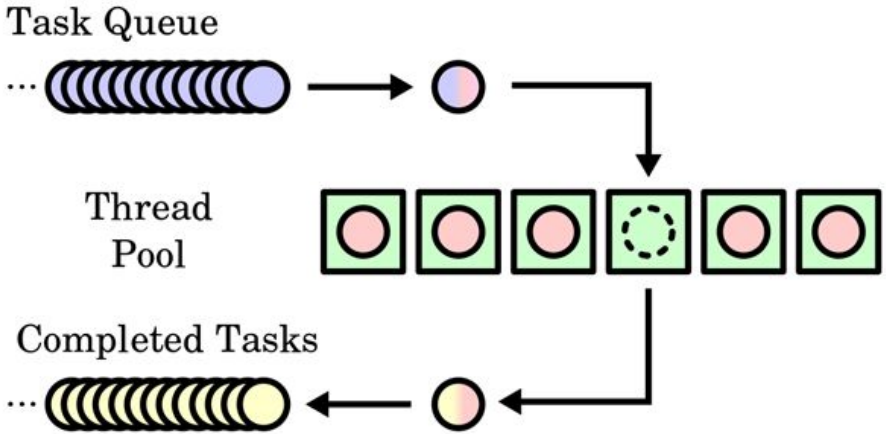
Concurrency Pattern

Was ist ein Thread?

- Aktivitätsträger oder auch Ausführungsstrang bzw. Ausführungsreihenfolge in der Abarbeitung eines Programms
- Somit ein Teil eines Prozesses
- Threads sind mittlerweile ein wichtiges Thema in der Informatik, da sie durch Multicore-CPU's zu einer erheblichen Beschleunigung des Programmablaufs führen können
- Prozessoren haben heutzutage meist zwischen 4 und 8 Kernen
- verwenden zusätzliche Methoden wie Simultaneous Multithreading, um mehrere Threads pro Kern ausführen zu können → logische Prozessoren

Wie funktioniert ein Thread Pool?

- Task Queue beinhaltet die Aufgaben
- Thread Pool bearbeitet die Aufgaben
- Wenn eine Aufgabe vollendet wurde, ist ein Platz im Thread Pool frei, und es kann eine neue Aufgabe von der Task Queue übergeben werden
- Größe des Thread Pools festlegen im Bezug auf Prozessor



Wie wird das Pattern implementiert?

- Es gibt mehrere Wege einen Thread Pool zu implementieren
- Beispielsweise ist es möglich vorgegebene Varianten aus der Java-Library zu nutzen, oder auch die Implementierung selber durchzuführen
- Eigene Implementierung wird in den wenigsten Fällen benötigt z.B. um Funktionalitäten zu ergänzen
- Hier wird das Pattern mit der Java-Library gezeigt
- Man kann beim Erzeugen zwischen Fixed Pool, Cached Pool und Scheduled Pool wählen

Fixed Pool vs. Scheduled Pool vs. Cached Pool

Fixed Pool	Cached Pool	Scheduled Pool
reuses a fixed set of threads operating off a shared unbounded queue *If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.*	*creates new threads as needed, but will reuse previously constructed threads when they are available* *If no existing thread is available, a new thread will be created and added to the pool* *Threads that have not been used for sixty seconds are terminated and removed from the cache*	*can schedule commands to run after a given delay, or to execute periodically.* *threads are kept in the pool, even if they are idle*

Quelle: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html

Beispiel

```java
package tptest;

import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ExecutorService;

public class ThreadPoolTest {

    public static void main(String[] args) {
        int processors = Runtime.getRuntime().availableProcessors();
        System.out.println("Verfügbare Prozessoren:" + processors);
        ExecutorService pool = Executors.newFixedThreadPool(processors);
```

Beispiel

```
for(int i=0; i<10; i++) {
    int taskNo = i;
    pool.execute( () -> {
        long timeStart = System.nanoTime();
        String name = Thread.currentThread().getName();
        String msg = name + ": Task " + taskNo;
        System.out.println(msg);
        long timeEnd = System.nanoTime();
        System.out.println("Runtime for " + name + ": " + (timeEnd - timeStart)/1000 + " µs");
    });
}
```

Beispiel

```
        pool.shutdown();
        try {
            pool.awaitTermination(500, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Beispiel

```
Verfügbare Prozessoren:8
pool-1-thread-1: Task 0
pool-1-thread-2: Task 1
Runtime for pool-1-thread-2: 88 µs
pool-1-thread-3: Task 2
Runtime for pool-1-thread-3: 69 µs
Runtime for pool-1-thread-1: 99 µs
pool-1-thread-4: Task 3
Runtime for pool-1-thread-4: 135 µs
pool-1-thread-6: Task 5
pool-1-thread-5: Task 4
Runtime for pool-1-thread-5: 56 µs
Runtime for pool-1-thread-6: 100 µs
pool-1-thread-7: Task 6
Runtime for pool-1-thread-7: 42 µs
pool-1-thread-8: Task 7
Runtime for pool-1-thread-8: 39 µs
pool-1-thread-2: Task 8
Runtime for pool-1-thread-2: 57 µs
pool-1-thread-4: Task 9
Runtime for pool-1-thread-4: 320 µs
```

```
Verfügbare Prozessoren:8
pool-1-thread-1: Task 0
pool-1-thread-3: Task 2
Runtime for pool-1-thread-1: 167 µs
pool-1-thread-4: Task 3
pool-1-thread-2: Task 1
Runtime for pool-1-thread-3: 185 µs
Runtime for pool-1-thread-2: 434 µs
Runtime for pool-1-thread-4: 43 µs
pool-1-thread-5: Task 4
Runtime for pool-1-thread-5: 87 µs
pool-1-thread-6: Task 5
Runtime for pool-1-thread-6: 100 µs
pool-1-thread-7: Task 6
Runtime for pool-1-thread-7: 201 µs
pool-1-thread-8: Task 7
Runtime for pool-1-thread-8: 37 µs
pool-1-thread-2: Task 8
pool-1-thread-5: Task 9
Runtime for pool-1-thread-5: 42 µs
Runtime for pool-1-thread-2: 35 µs
```

- Unterschiedliche Reihenfolge
- Unterschiedliche Laufzeiten
- Unterschiedliche Wiederverwendung

Abschluss

Letzte Gedanken

- Pattern sind nützliche Werkzeuge zur Vereinfachung des Design Prozesses
- Für fast jedes Problem existiert bereits eine Lösung

Buchempfehlungen:

- Gang of Four:
  Design Patterns: Elements of Reusable Object-Oriented Software
- Besonders einsteigerfreundlich:
  Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software

Quellen und weiterführende Links

- Inversion of Control Containers and the Dependency Injection pattern - Martin Fowler
- What is Dependency Injection? - StackOverflow
- Bridge - RefactoringGuru
- Bridge - Source Making
- Adapter - RefactoringGuru
- State - RefactoringGuru
- Das State Design Pattern - Phillip Hauer
- Zustandsautomat (Bild)

Quellen und weiterführende Links (Thread Pool)

Bilder und statische Informationsquellen:

- https://en.wikipedia.org/wiki/Thread_pool#/media/File:Thread_pool.svg
- https://de.wikipedia.org/wiki/Thread_(Informatik)
- https://stackoverflow.com/questions/18425026/shutdown-and-awaittermination-which-first-call-have-any-difference
- https://de.wikipedia.org/wiki/Simultaneous_Multithreading
- https://en.wikipedia.org/wiki/Thread_pool

Youtube-Tutorial zur Implementierung:

- https://www.youtube.com/watch?v=ZcKt5FYd3bU