Software Design

Julian Huber & Matthias Panny

Design-Patterns

(dt. Entwurfsmuster)

Design-Patterns

- Design-Patterns sind bekannte und bewährte Lösungsmethoden für wiederkehrende Probleme in der Softwareentwicklung
- Sie beschreiben daher ein allgemeines Konzept → unabhängig von einer Programmiersprache
- Bauen auf den Prinzipien der Objektorientierung auf
- Bilden eine Art der Standardisierung in der Softwareentwicklung → verwendete Terminologie stammt dazu aus *"Design Patterns: Elements of Reusable Object-Oriented Software"*

Design-Patterns

- Die meisten gängigen Design-Patterns lassen sich einer der drei folgenden Kategorien zuweisen
- In der Literatur werden aber noch einige weitere definiert

Creational Patterns (dt. Erzeugungsmuster)

- Bilden Methodiken zum Erzeugen von Objekten ab

Structural Patterns (dt. Strukturmuster)

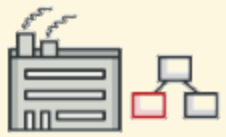
- Bilden Methodiken ab um einzelne Klassen und Objekte zu komplexeren Strukturen zusammenzufassen ohne deren Flexibilität einzuschränken

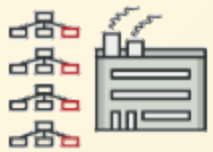
Behavioral Patterns (dt. Verhaltensmuster)

- Bilden Methodiken ab um Verantwortlichkeiten und Aufgaben zwischen Objekten zu verteilen

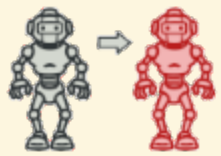
Design-Patterns

Creational patterns


Factory Method

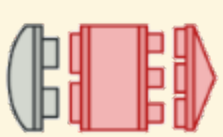

Abstract Factory


Builder


Prototype

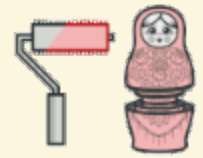

Singleton

Structural patterns

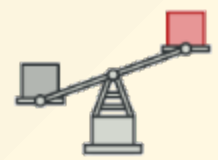

Adapter

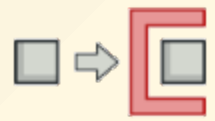

Bridge


Composite


Decorator


Facade


Flyweight

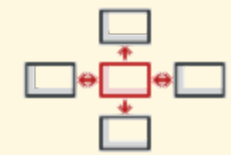

Proxy

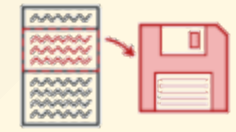
Behavioral patterns


Chain of Responsibility

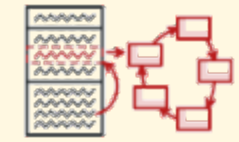

Command

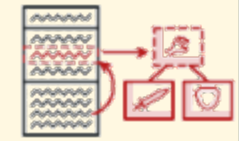

Iterator


Mediator

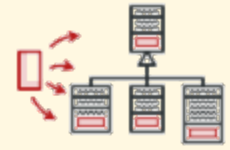

Memento


Observer


State


Strategy


Template Method


Visitor

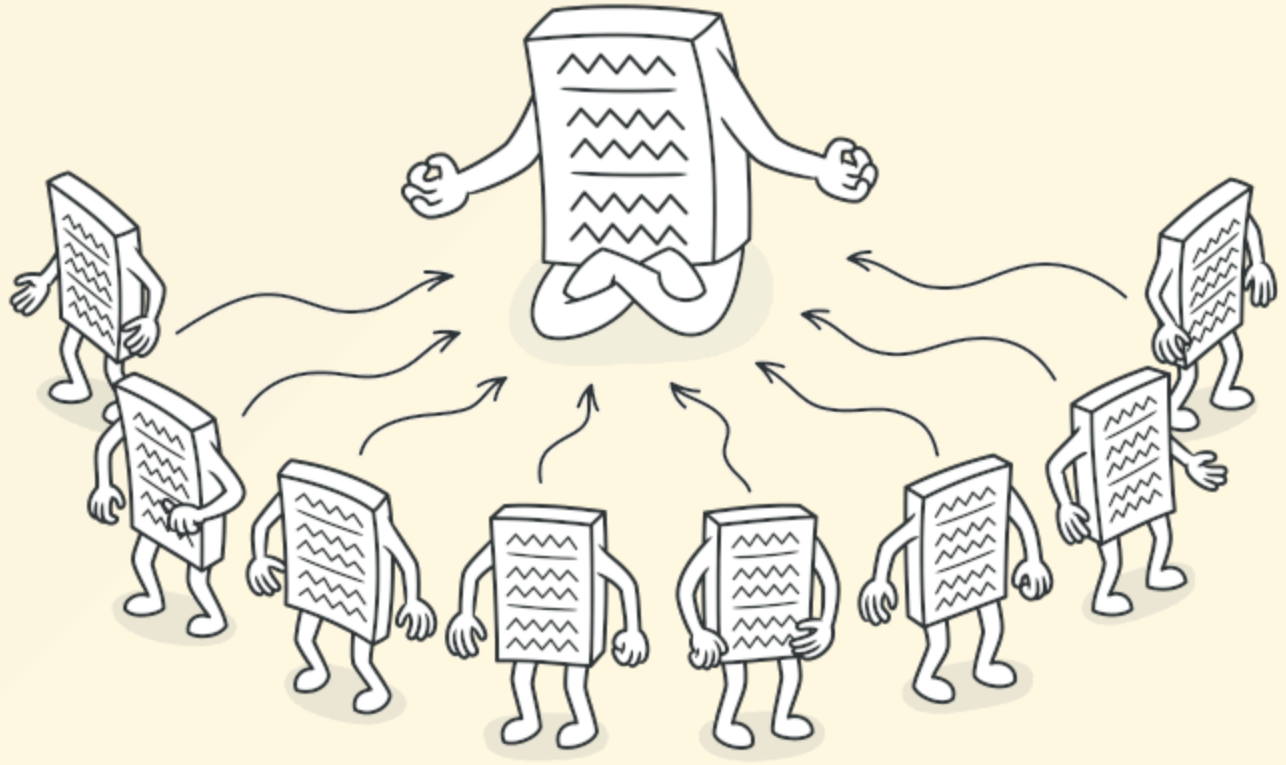
Bildquelle: [Shvets 2019]

Singleton-Pattern

(als Beispiel für ein Creational-Design-Pattern)

Singleton-Pattern

- Creational-Pattern
- Gewährleistet, dass von einer Klasse nur ein einziges Objekt existiert
- Bildet globale Zugriffsmöglichkeit zu diesem einen Objekt ab
- Warum wollen wir das?
  - Zugriff auf eine Ressource, die nur einmal existiert (z.B. Datenbank, Dateien, Configs, besondere Hardware, etc.)



Bildquelle: [Shvets 2019]

Singleton-Pattern

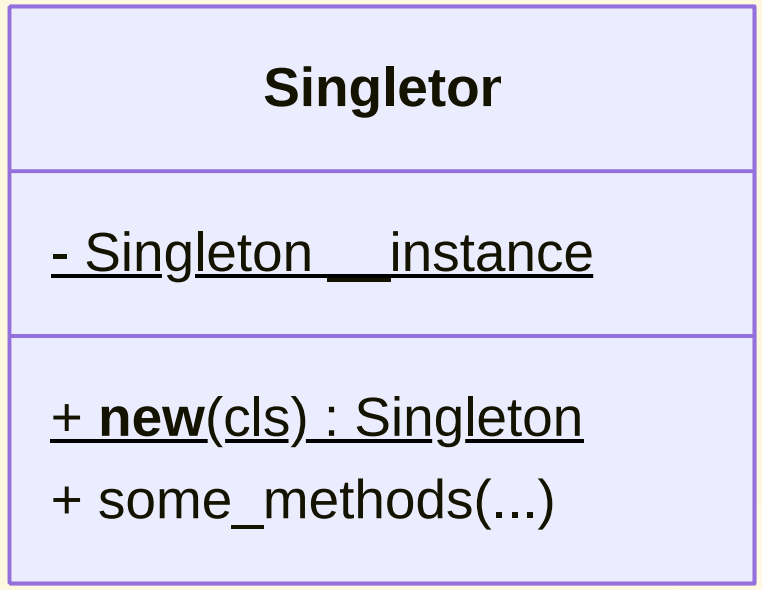
Vorteile

- Globale Zugriffsmöglichkeit auf ein Objekt ohne globale Variablen verwenden zu müssen
- Vermeidung von mehrfacher Instanziierung
- Vermeidung von Ressourcenverschwendung
- Thread-safety möglich → erfordert richtige Implementierung
  - mehrere parallele Zugriffe auf das Singleton-Objekt möglich ohne, dass es zu problematischen Zuständen kommt
    → z.B. in `thread_safety.py` kann nicht sichergestellt werden, dass es keine gleichzeitigen Zugriffe auf die Datei gibt

Beispielimplementierung

Einfache Singelton-Klasse

- Wir halten uns an das UML-Diagramm → was kann daraus abgelesen werden?
- Die Klasse `Singleton` hat eine private, class Variable `__instance`, um die Instanz des Singleton-Objekts zu speichern
- Wir überladen die `__new__()`-Methode und geben dort ein `Singleton`-Objekt zurück



`__new__()`-Methode

- Wird aufgerufen wenn ein neues Objekt erstellt wird und gibt dieses zurück
- Ist dafür zuständig die Klasse selbst zu erzeugen
- Wird normal implizit vor `__init__()` aufgerufen

✍️ Aufgabe

- Wir wollen diese Beispielimplementierung für einen Singleton nun erstellen

Musterlösung - `singleton_class.py`

- `__new__()`-Methode führt Überprüfung durch ob bereits eine Instanz existiert und gibt `__instance` zurück
- Zusätzliche Methode `print()` soll die ID der Instanz ausgeben → hierzu kann die `id()`-Funktion verwendet werden
- Wir wollen unseren Singleton zusätzlich testen

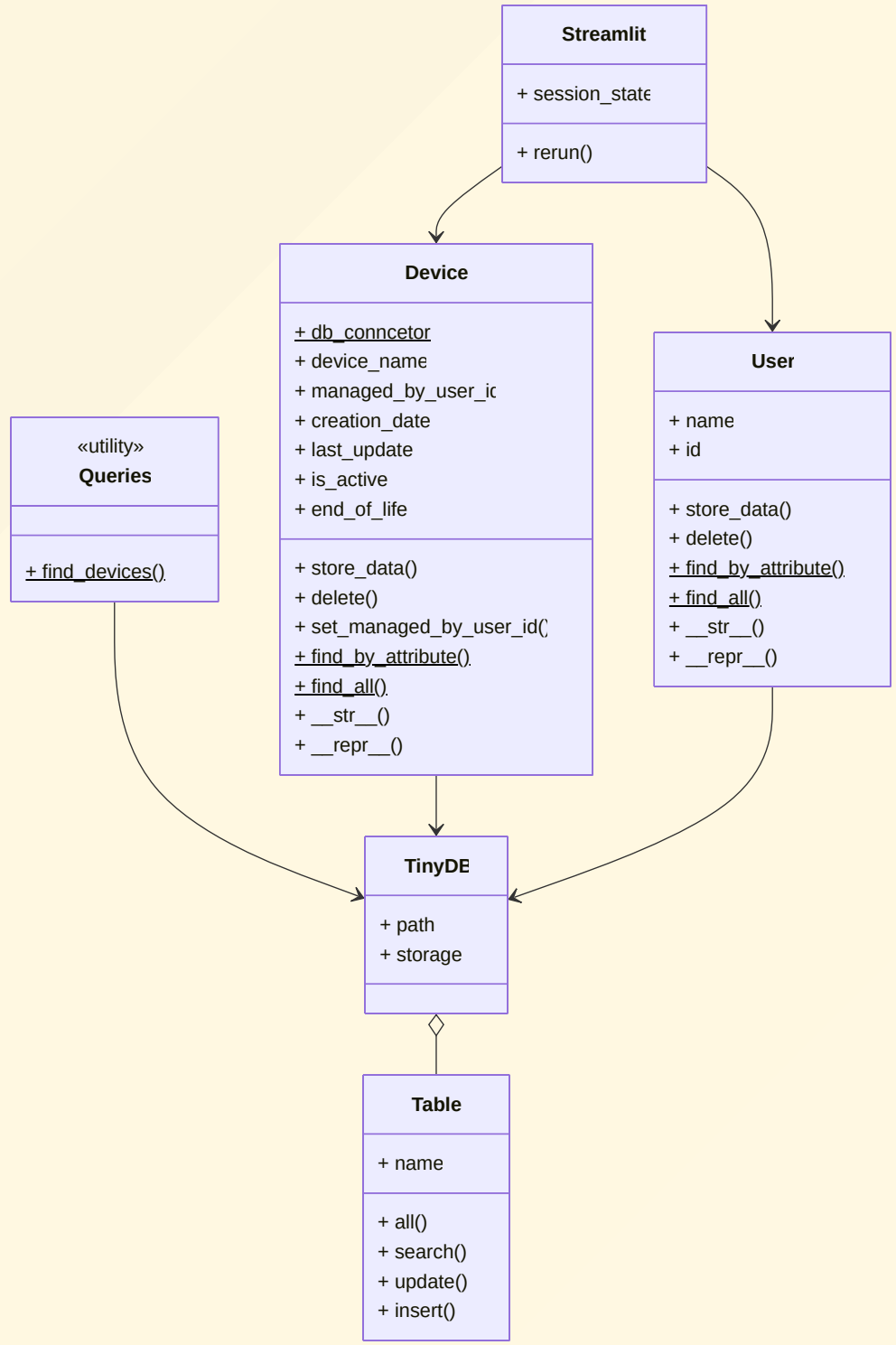
Singleton-Pattern

- Wo haben wir das Singleton-Pattern bereits (unbewusst) im Einsatz gesehen?
  - Das `logging`-Modul stellt einen Logger wie einen Singleton zur Verfügung → die Implementierung unterscheidet sich aber leicht zu unserer. Hier ist ein einfaches Beispiel, wie wir selbst einen Logger schreiben könnten.
- Wo können wir sonst noch intuitiv Einsatzmöglichkeiten finden?
  - Immer wenn wir in unserem Projekt an mehreren Stellen dasselbe Objekt mit den gleich Attributen erstellen → `db_connector`-Objekt in unserer naiven Musterlösung der Case Study I
  - Zugriff auf eine Datei
  - Zugriff auf eine Hardware-Ressource

Singleton-Pattern - Case Study

Beispiel: Case Study I

- Auch in unsere Case Study I ist das Singleton-Pattern sinnvoll
- Unsere Anwendung hat nur eine Datenbank → wir wollen eigentlich nur ein `db_connector`-Objekt haben
- Bis jetzt haben wir aber in jeder Klasse ein neues `db_connector`-Objekt erstellt
- Wenn sich z.B. der Pfad zur Datenbank ändert, müssen wir diesen in jeder Klasse ändern



✍️ Aufgabe

- Wir wollen nun die Klasse `DatabaseConnector` als Singleton implementieren
- Sie soll zusätzlich die Methode `get_table()` bereitstellen, um uns die richtige Tabelle zurückzugeben
- Im selben Modul sollen auch direkt alle dazugehörigen Serializer implementiert werden

Musterlösung - `database_singleton.py`

- `__new__()`-Methode führt Überprüfung durch, ob bereits eine Instanz existiert und gibt `__instance` zurück
- Der Pfad zur Datenbank wird automatisch im Konstruktor gesetzt

Datenbank-Zugang für unsere Case Study I

```python
import os
from tinydb import TinyDB
from tinydb.storages import JSONStorage
from datetime import datetime, date
from tinydb_serialization import Serializer, SerializationMiddleware
from tinydb_serialization.serializers import DateTimeSerializer

class DatabaseConnector:
    """
    Usage: DatabaseConnector().get_devices_table()
    The information about the actual database file path
    and the serializer objects has been abstracted away into this class
    """
    # Turns the class into a naive singleton
    # --> not thread safe and doesn't handle inheritance particularly well
    __instance = None
    def __new__(cls):
        # We need the cls argument to be able to access the class attribute __instance
        if cls.__instance is None:
            cls.__instance = super().__new__(cls)
            cls.__instance.path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'database.json')

        return cls.__instance

    def get_table(self, table_name: str) -> Table:
        return TinyDB(self.__instance.path, storage=serializer).table(table_name)

# Custom Serializer would be added below here...
```

Module als Singleton

- In Python gibt es noch andere Möglichkeiten Singletons zu definieren
- Module sind bereits Singletons → es wird nur eine Instanz pro Modul erzeugt siehe Beispiel `global_object_pattern.py`

```
from my_module import my_instance

my_instance.some_method()  # Output: Value: 0
my_instance.value = 10

from my_module import my_instance
my_instance.some_method()  # Output: Value: 10
```

- Es wird immer die selbe Instanz `my_instance` verwendet, da diese im Modul nur einmal erzeugt wurde
- Dieses Konzept wird in Python auch als Global Object Pattern bezeichnet & meist anstatt des Singleton-Patterns verwendet

Alternative Ansätze

- In Python gibt es noch andere Möglichkeiten identische Eigenschaften, wie jene eines Singletons zu erreichen
- Mehrere Instanzen der Klasse, aber alle mit identen Werten für ihre Attribute → Borg-Pattern

Beispiel `borg.py`

```
class Borg:
  _shared_state = {}
  def __init__(self):
      self.__dict__ = Borg._shared_state

class MyObject(Borg):
  def __init__(self, arg):
    Borg.__init__(self)
    self.val = arg

  def __str__(self):
    return self.val

  obj1 = MyObject("Test1")
  obj2 = MyObject("Test2")
  print(F"{obj1} | {obj2}")
  # Test2 | Test2
  print(obj1 == obj2)
  # False
```

- Die Objekte `obj1` und `obj2` sind weiterhin unterschiedlich

Lazy Initialization vs. Eager Initialization

Lazy Initialization (dt. verzögerte Initialisierung)

- Objekt wird erst bei Bedarf erzeugt → beim Singleton wird die Instanz erst beim ersten Aufruf der `__new__()`-Methode erzeugt
- Vorteil: Ressourcen (v.a. Speicher) werden erst dann verbraucht, wenn sie auch wirklich benötigt werden
- Nachteil:
  - Zur Laufzeit muss der evtl. aufwendige Erzeugungsprozess durchlaufen werden
  - Kann bei Multithreading zu Problemen führen

Eager Initialization (dt. vorzeitige Initialisierung)

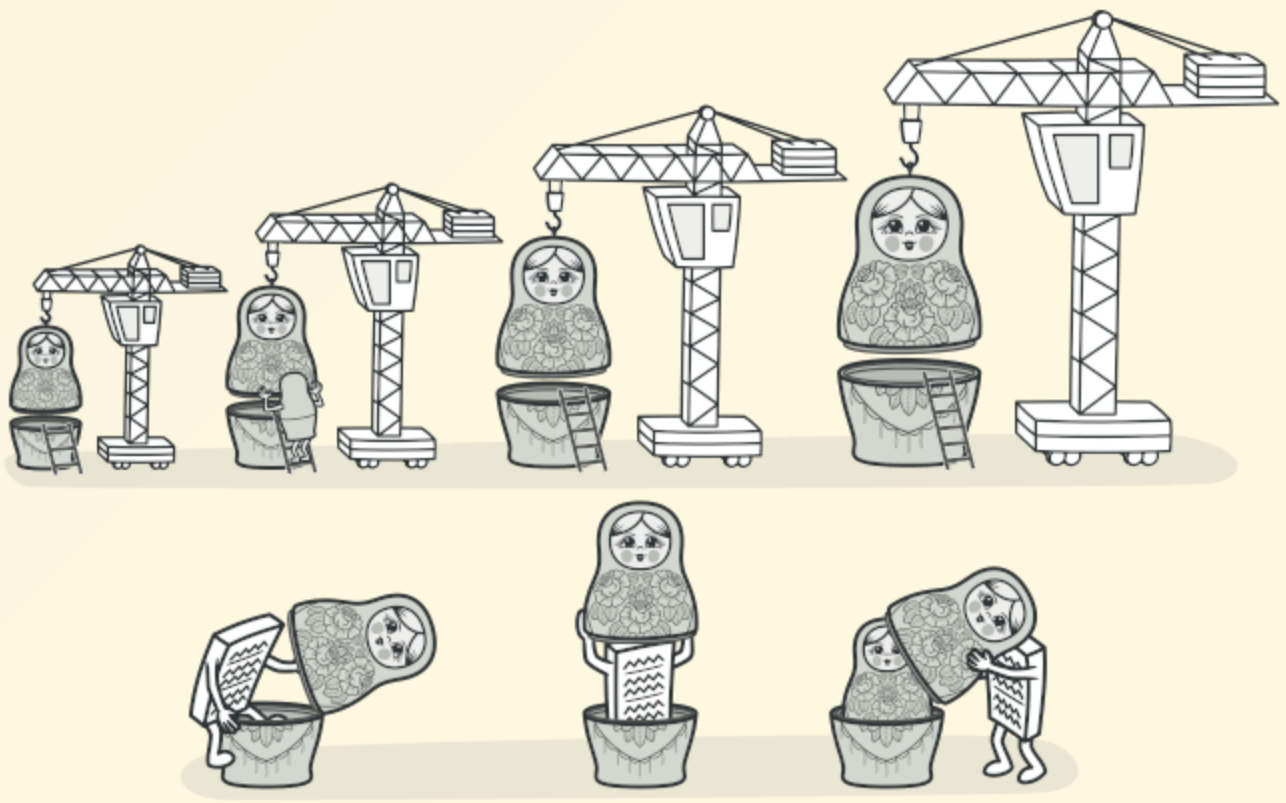
- Objekt wird bereits beim Laden der Klasse erzeugt → beim Singleton nur mit anderer Implementierung umsetzbar
- Vorteil: Garantiert die Verfügbarkeit der Singleton-Instanz während ohne Einbußen in der Laufzeitperformance
- Nachteil: Erstmalige Initialisierung dauert länger

Decorator-Pattern

(als Beispiel für ein Structural-Design-Pattern)

Decorator-Pattern

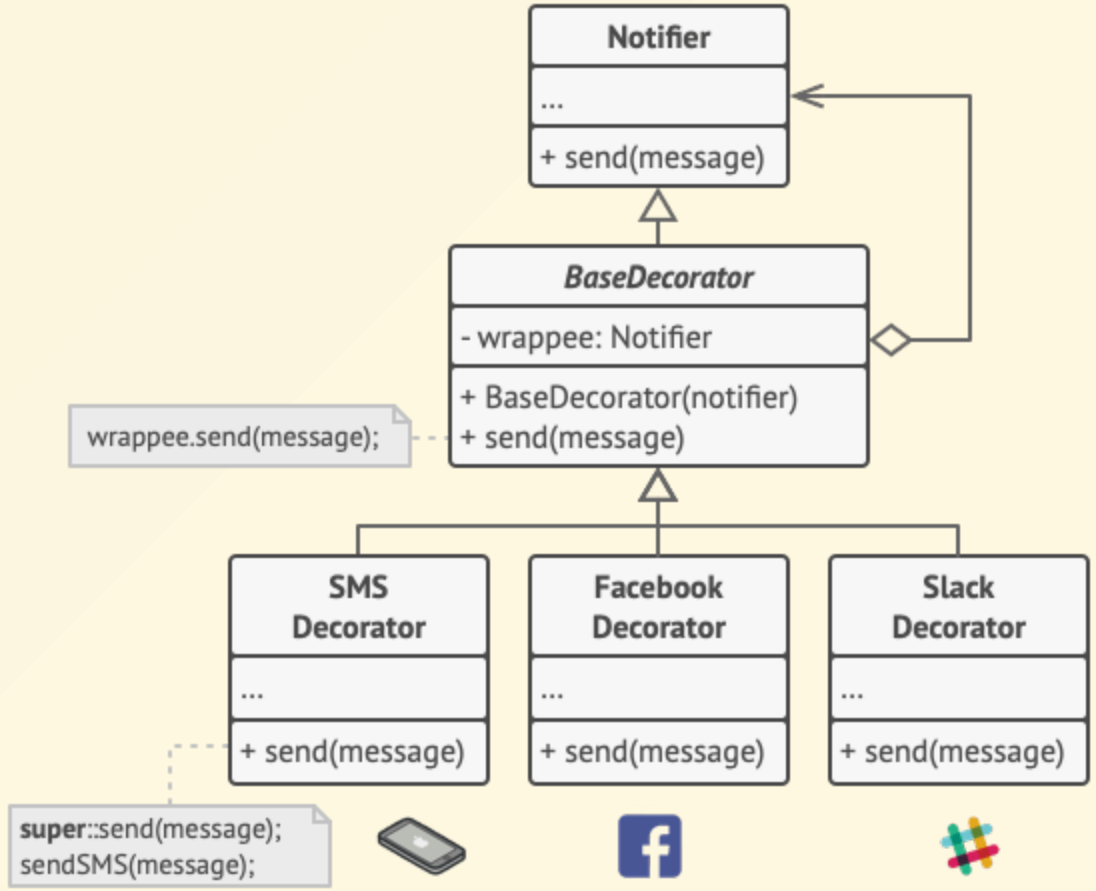
- Structural-Pattern
- Ermöglicht, dass eine Klasse flexibel um Funktionalitäten erweitert werden kann
- Legt schichtweise Funktionalitäten über eine bestehende Klasse
- Warum wollen wir das?
  - Wir wollen eine Klasse nicht verändern, aber zusätzliche Funktionalitäten hinzufügen und dies je nach Bedarf dynamisch entscheiden



Decorator-Pattern

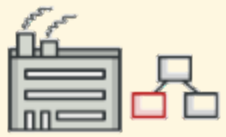
Beispiel `notification.py`

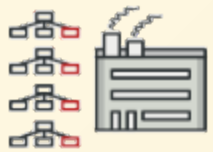
- wir wolle ein Benachrichtigungssystem implementieren, dabei können Nuzter:innen entscheiden, ob sie per Slack oder SMS etc. benachrichtigt werden wollen
- Es soll aber auch möglich sein sich für mehrere Benachrichtigungsarten zu entscheiden



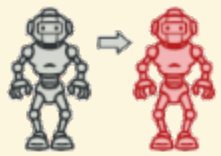
Design-Patterns

Creational patterns


Factory Method

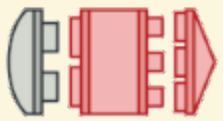

Abstract Factory


Builder


Prototype

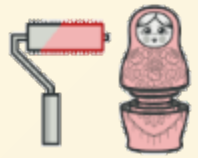

Singleton

Structural patterns

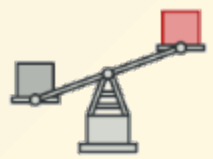

Adapter

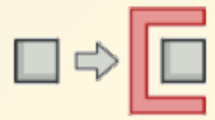

Bridge


Composite


Decorator


Facade


Flyweight

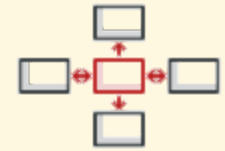

Proxy

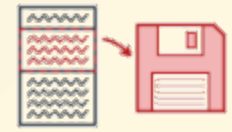
Behavioral patterns


Chain of Responsibility

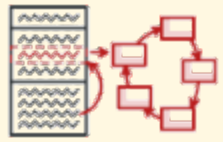

Command

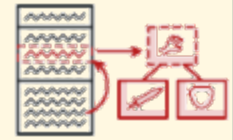

Iterator


Mediator


Memento


Observer


State


Strategy


Template Method


Visitor

Bildquelle: [Shvets 2019]