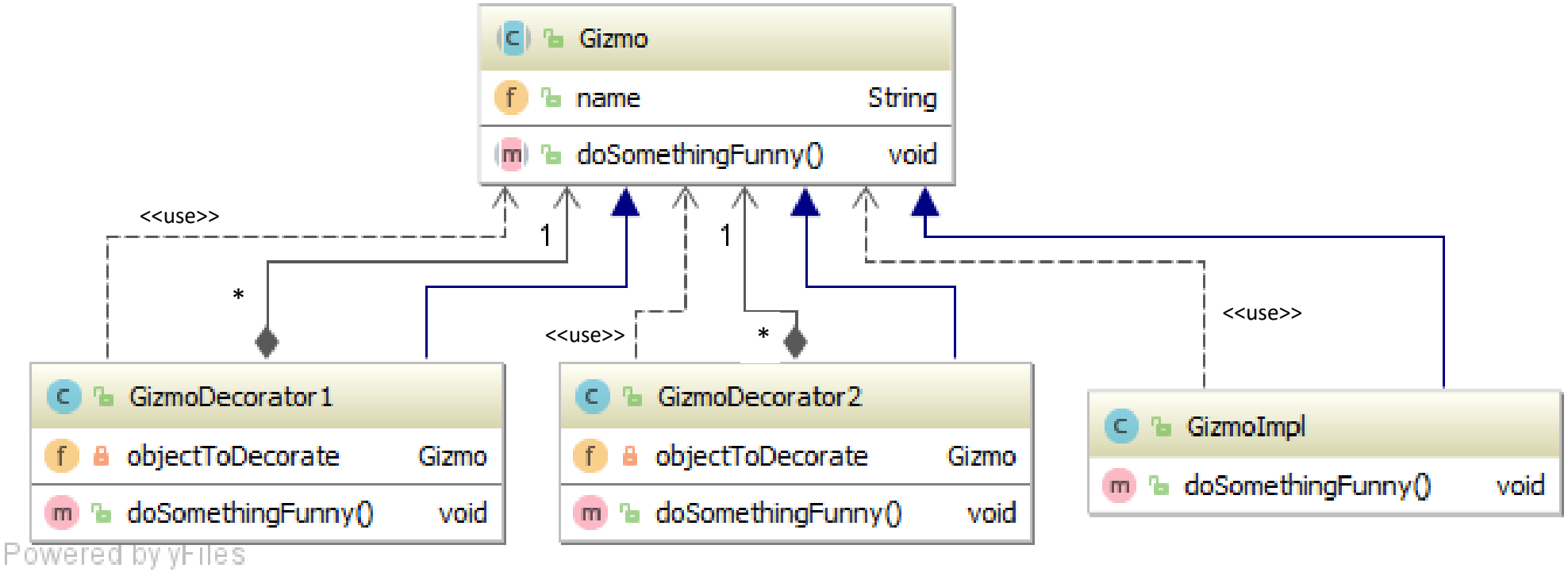
Objektorientierte Programmierung

Hochschule Bochum
WS 19/20
Dr.-Ing. Darius Malysiak

IV. Design Patterns

Decorator Pattern: Strukturmuster

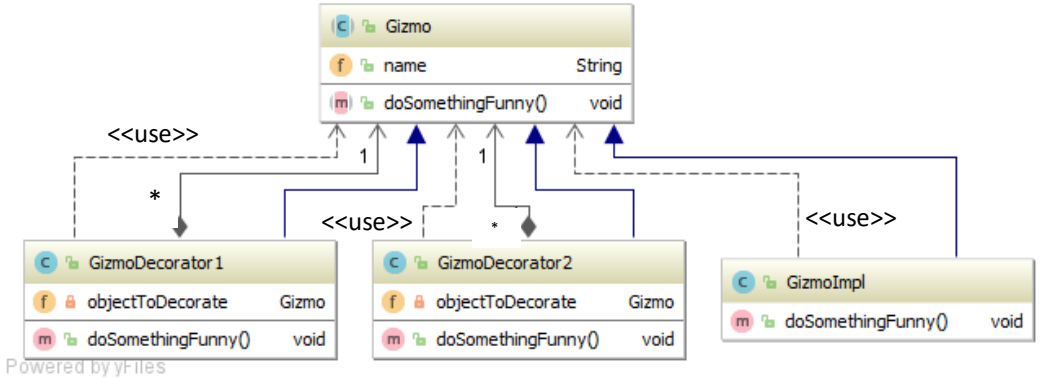
Ziel: Dynamisches augmentieren des Verhaltens eines Objekts.



IV. Design Patterns

Decorator Pattern: Strukturmuster

Ziel: Dynamisches augmentieren des Verhaltens eines Objekts.



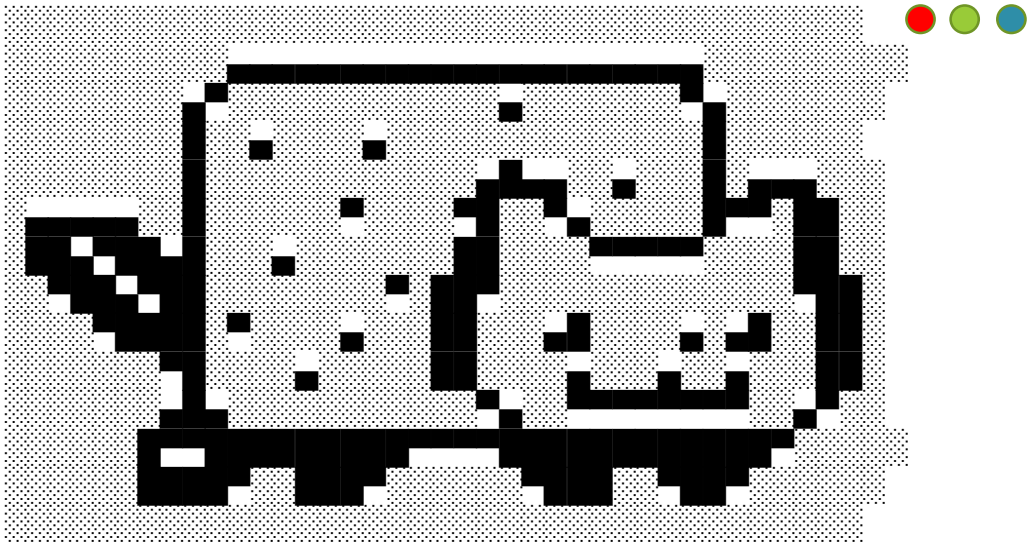

```
public abstract class Gizmo {
    public String name = "";
    public abstract void doSomethingFunny();
}
```

```
public static void main(String[] args) {

    Gizmo g1 = new GizmoImpl();
    g1.doSomethingFunny();

    Gizmo g2 = new GizmoDecorator1(g1);
    g2.doSomethingFunny();

    Gizmo g3 = new GizmoDecorator2(g2);
    g3.doSomethingFunny();
}
```

longcat does not like tacgnol!

longcat does not like nyancat!

Das Verhalten wird erweitert!

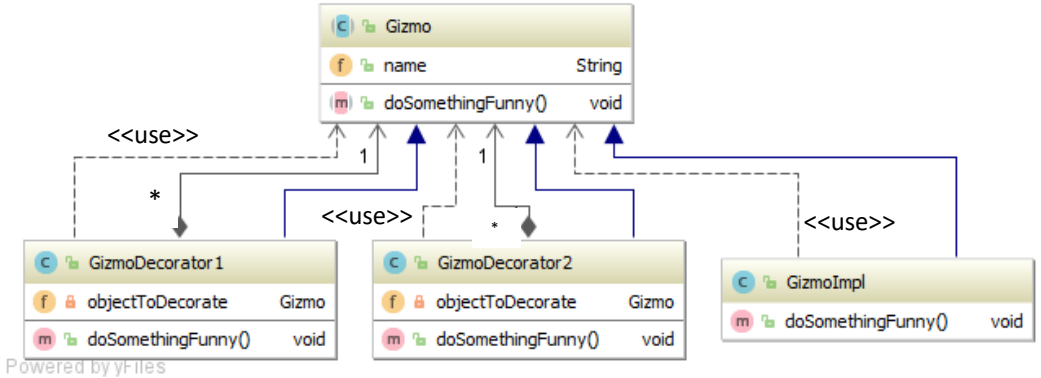


longcat does not like nyancat!

IV. Design Patterns

Decorator Pattern: Strukturmuster

Ziel: Dynamisches augmentieren des Verhaltens eines Objekts.



Vorteile:

- Das Verhalten eines Objekts kann zur Laufzeit erweitert werden ohne polymorphische Implikationen.
- Vererbungshierarchie bleibt flach!

Nachteile:

- Hinzugefügte Verhaltenserweiterungen können nicht entfernt werden.
- Erfordert die Instanziierung eines weiteren Objekts -> Skalierungsfaktor = 2

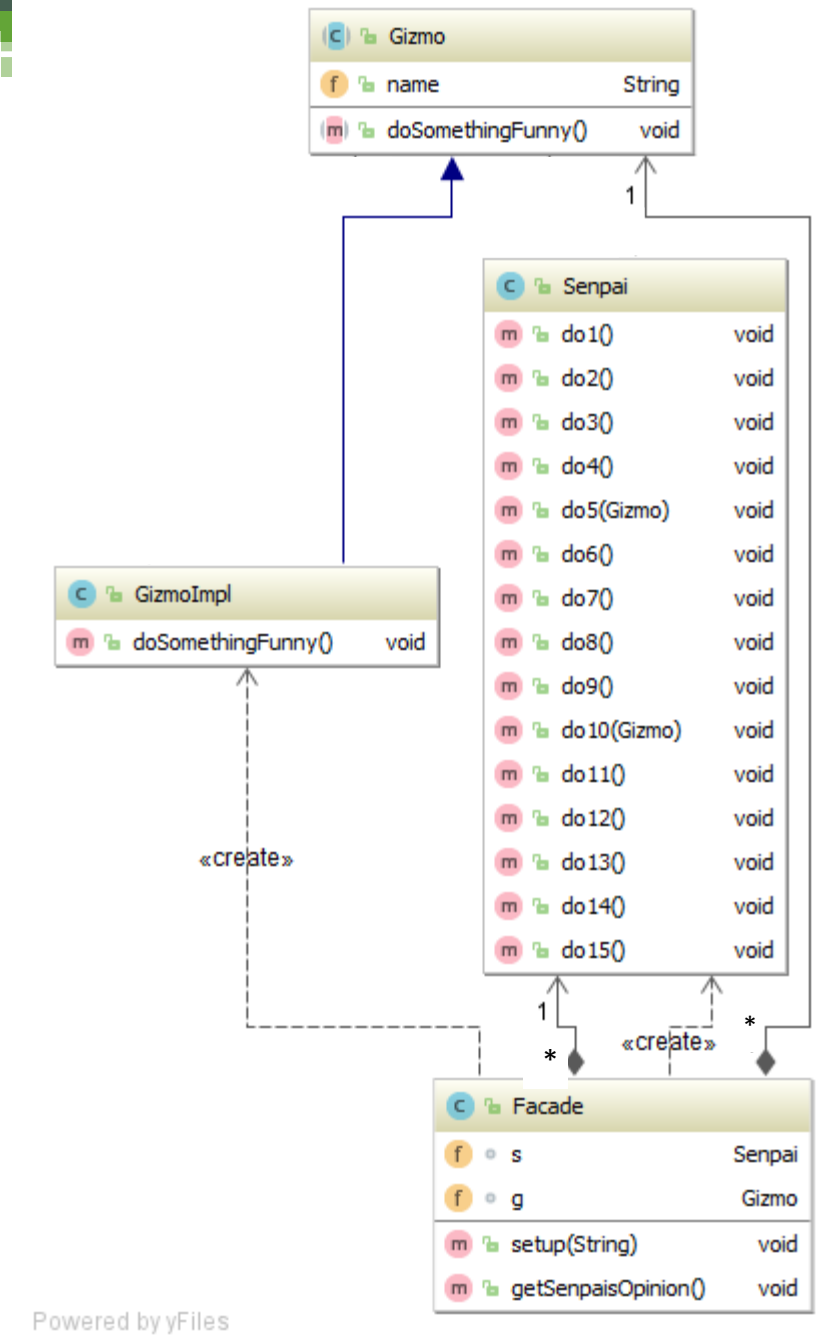
IV. Design Patterns

Facade Pattern: Strukturmuster

Ziel: Vereinfachung / Erstellung einer Subsystem Schnittstelle

```
public static void main(String[] args) {
    Facade f = new Facade();
    f.setup("test");
    f.getSenpaisOpinion();

    f.setup("nyancat");
    f.getSenpaisOpinion();
}
```

Senpai noticed you!
Senpai disagrees!



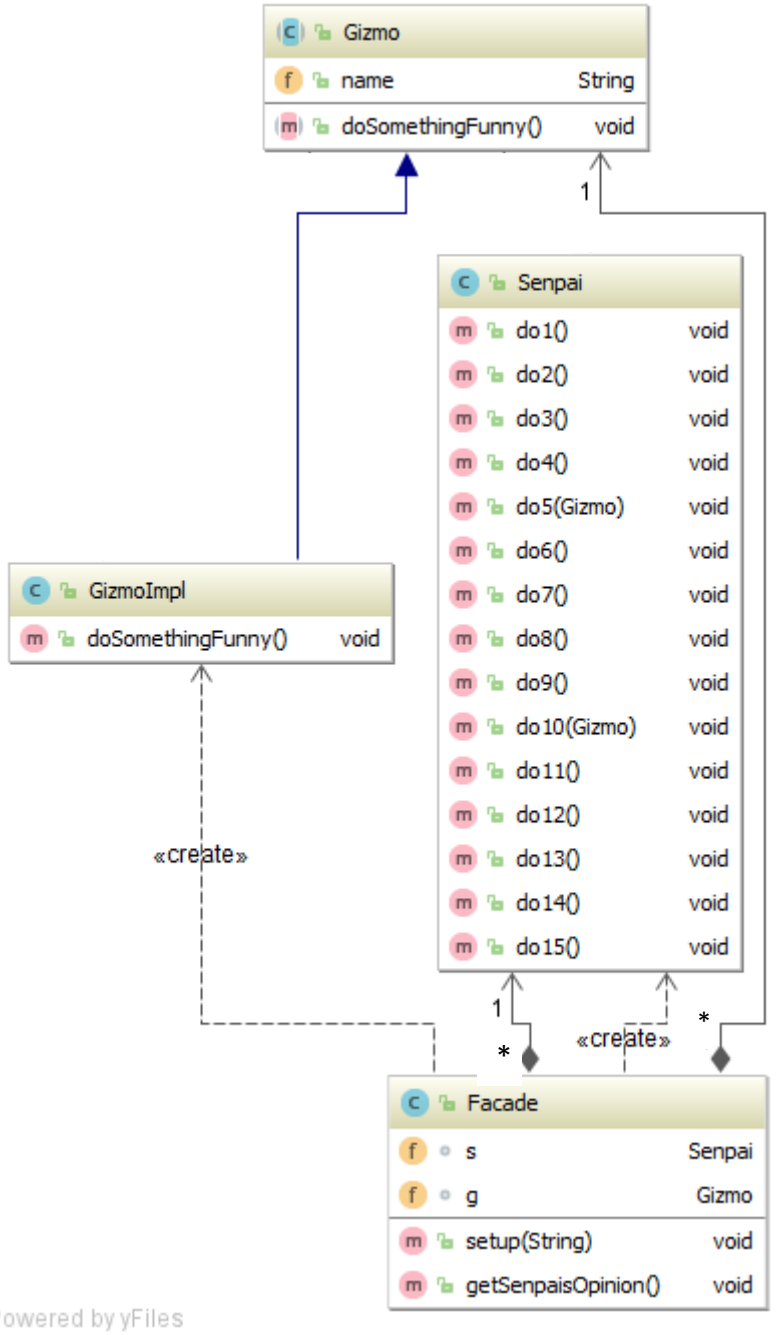
IV. Design Patterns

Facade Pattern: Strukturmuster

Ziel: Vereinfachung / Erstellung einer Subsystem Schnittstelle

Vorteile:
- Definition von Subsystemen mit Hilfe von Facades möglich.
- Reduktion des Komplexität einer Subsytem-Schnittstelle.
- Minimierung von Abhängigkeiten zwischen Subsystem-Komponenten, z.B. muss ‚Senpai' nichts über ‚Gizmo' wissen.
- Minimierung von Schnittstellen Refactoring der Komponenten durch geschickte Nutzung der vorhandenen Schnittstelle in einer Facade.

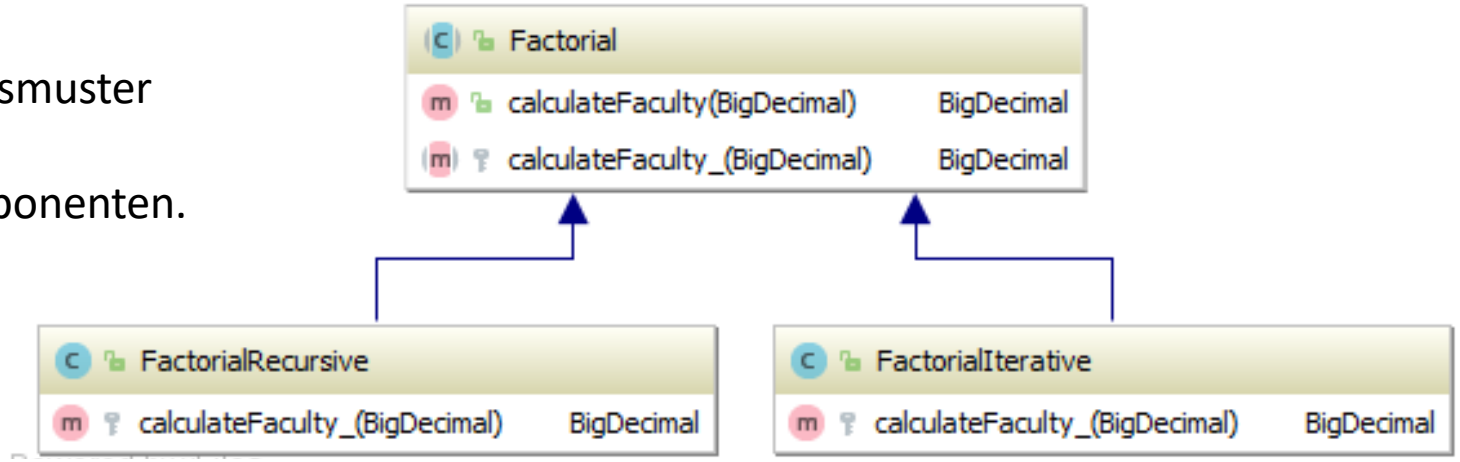
Nachteile:
- Komplexe Subsysteme können / sollten nicht über eine Klasse abgebildet werden.
- Nicht immer sind die Schnittstellen ausreichend generisch um sie auf diese Weise in Subsystemen nutzen zu können. –> Refactoring nötig!



IV. Design Patterns

Template Method Pattern: Verhaltensmuster

Ziel: Modularisierung von algorithmischen Komponenten.

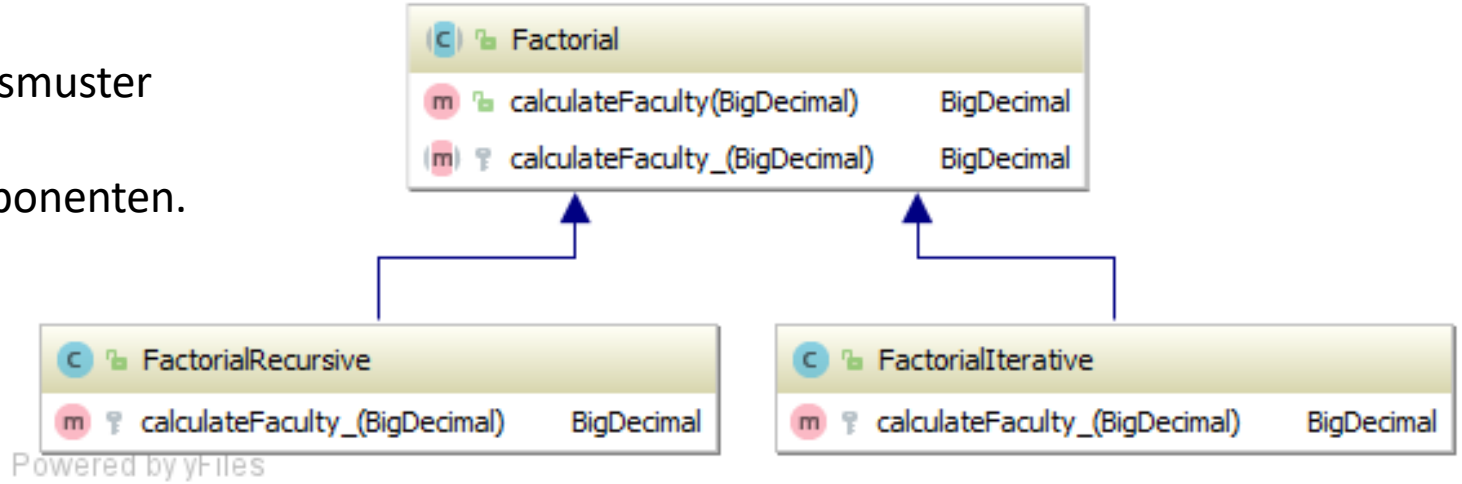



```
public abstract class Factorial {

    public BigDecimal calculateFaculty(BigDecimal n)
    {
        long a = System.currentTimeMillis();
        BigDecimal res = calculateFaculty_(n);
        long b = System.currentTimeMillis();
        System.out.println("Time needed in ms: "+(b-a));
        return res;
    }

    public abstract BigDecimal calculateFaculty_(BigDecimal n);

}
```

IV. Design Patterns

Template Method Pattern: Verhaltensmuster

Ziel: Modularisierung von algorithmischen Komponenten.

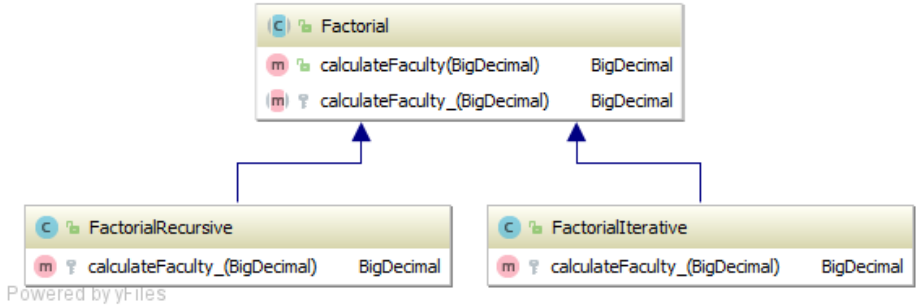



```
public static void main(String[] args) {
    final String value = "30";
    Factorial fa = new FactorialIterative();
    System.out.println("Faculty: " + fa.calculateFaculty(new BigDecimal(value)));

    fa = new FactorialRecursive();
    System.out.println("Faculty: " + fa.calculateFaculty(new BigDecimal(value)));
}
```

IV. Design Patterns

Template Method Pattern: Verhaltensmuster

Ziel: Modularisierung von algorithmischen Komponenten.



Vorteile:

- Die Implementierung eines Algorithmus kann ohne polymorphische Implikationen verändert werden.
- Die Implementierung eines Algorithmus kann ohne Veränderung von Legacy Code verändert werden.
- Minimierung von Boilerplate Code durch algorithmisches Skelett in der Elternklasse.

Nachteile:

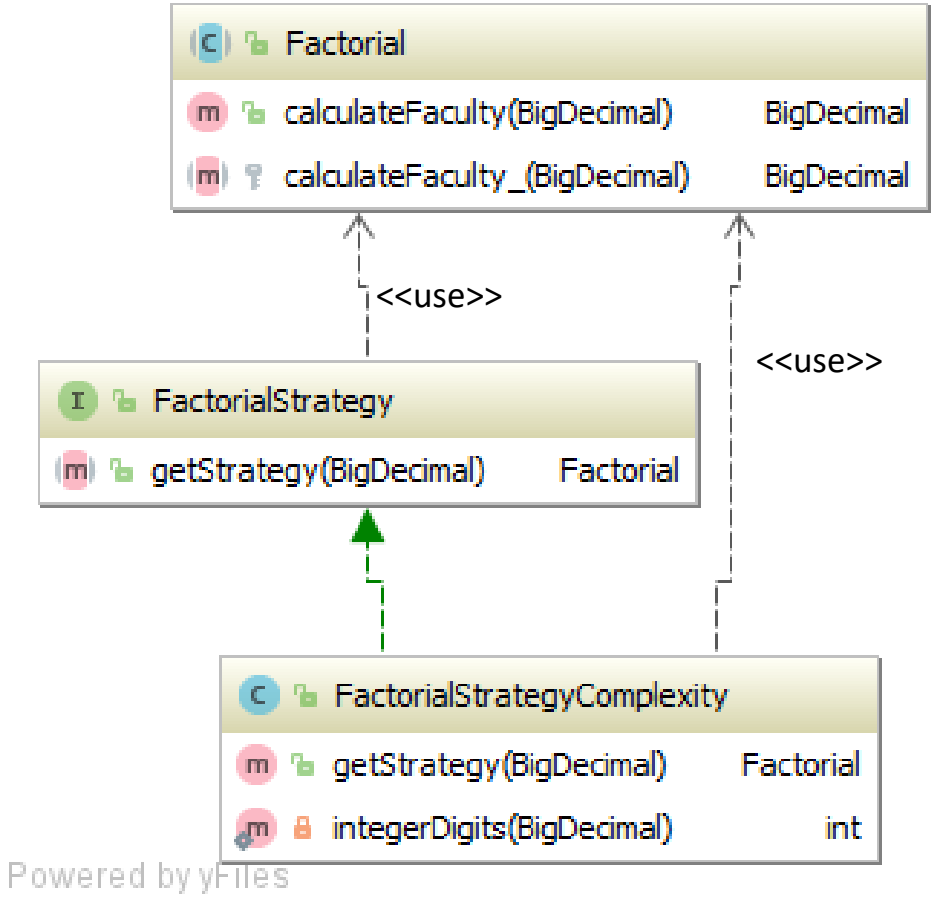
- Laufzeit kann sich bei komplexen Skeletten (d.h. mit vielen Subfunktionen, welche mit Kopien von Werten / Objekten aufgerufen werden) verschlechtern.
- Bei starker Modularisierung droht eine kombinatorische Explosion der Schnittstellen.

IV. Design Patterns

Strategy Pattern: Verhaltensmuster

Ziel: Dynamische Auswahl von geeigneten Verfahren zur Laufzeit

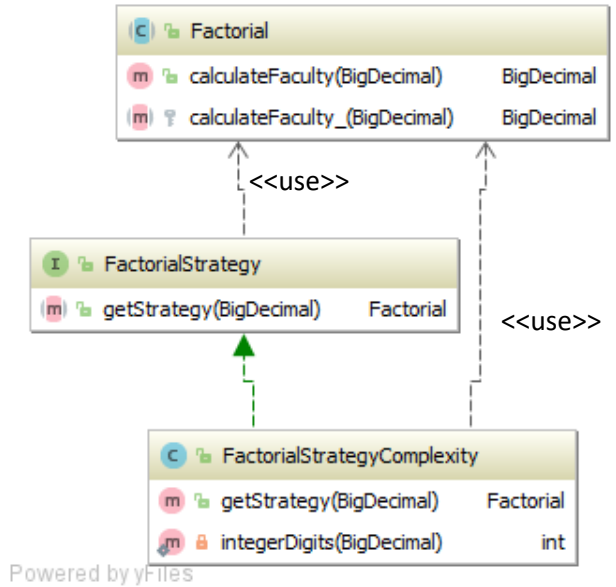
```
public static void main(String[] args) {
    FactorialStrategy strategy =
                new FactorialStrategyComplexity();

    Factorial fa = strategy.getStrategy(
                    new BigDecimal(value));

    System.out.println("Faculty: " +
          fa.calculateFaculty(new BigDecimal(value)));
}
```



IV. Design Patterns

Strategy Pattern: Verhaltensmuster

Ziel: Dynamische Auswahl von geeigneten Verfahren zur Laufzeit



Vorteile:

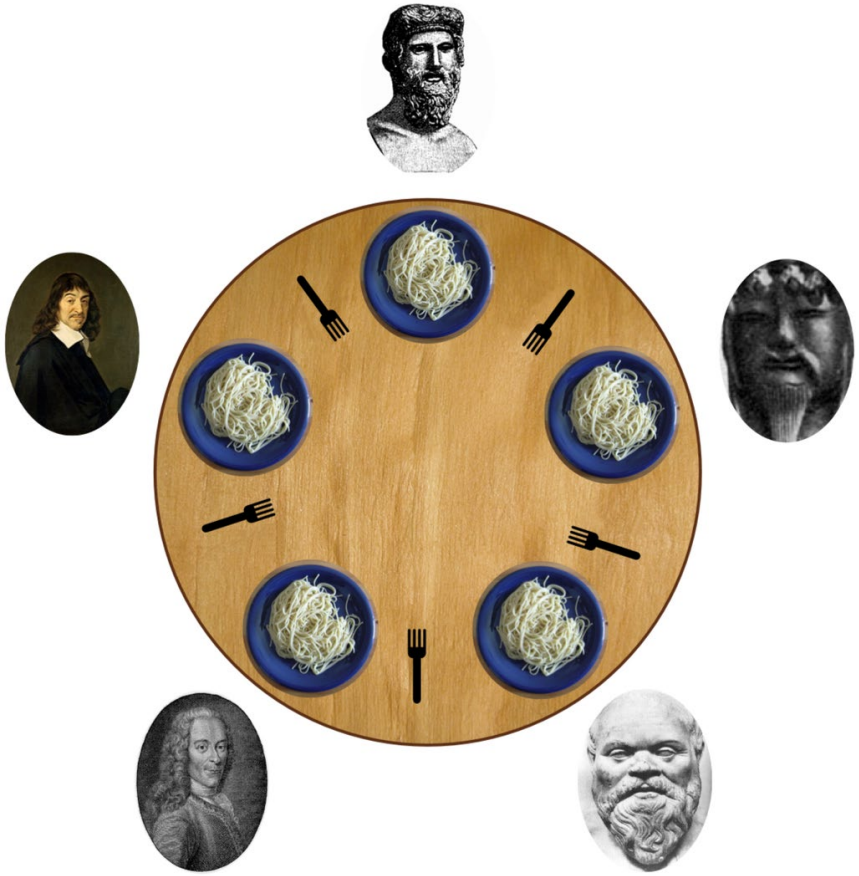
- Mehrere Algorithmen, welche für bestimmte Fälle spezialisiert sind können kombiniert werden um eine kontinuierliche Effizienz zu gewährleisten.
- Verantwortung für Effizienz größtenteils konsolidiert in separaten Strategie-Klassen.
- Reduktion von ‚aufgeblähten' Implementierungen der Algorithmen durch Optimierung für diverse Kontexte (z.B. Systeme oder Parameter).

Nachteile:

- Strategie-Klassen müssen gewartet werden (z.B. bei Änderung der Algorithmen oder Anpassung der Parameter an neue Systeme).

IV. Design Patterns

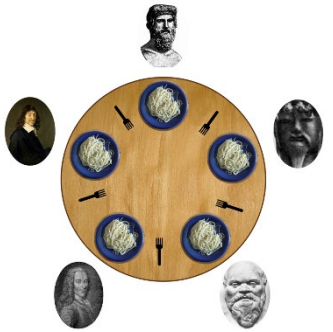
Concurrency Patterns: Parallel-Muster



https://de.wikipedia.org/wiki/Philosophenproblem#/media/File:An_illustration_of_the_dining_philosophers_problem.png

IV. Design Patterns

Concurrency Patterns: Parallel-Muster



https://de.wikipedia.org/wiki/Philosophenproblem#/media/File:An_illustration_of_the_dining_philosophers_problem.png

Dining Philosophers Problem:

- 5 Philosophen sitzen am Tisch.
- Jeder handelt wie folgt: Denken, Essen, Denken, Essen, ….
- Beim Essen nimmt der jeweilige Philosoph zunächst die Gabel links von ihm, dann jene rechts von ihm.
- Zwischen jedem Philosoph liegt genau eine Gabel. -> Somit 5 Gabeln und leider eine zu wenig ☹
- Nach dem Essen legt der Philosoph die Gabeln wieder zurück, erst die rechte Gabel und dann die linke Gabel!

Was ist die triviale Lösung dieses Problems?

IV. Design Patterns

Nächste Vorlesung:

Flynn‘sche Taxonomie, Concurrency Pattern, Maven, Hibernate

+ Abschluss von Design Patterns ☺