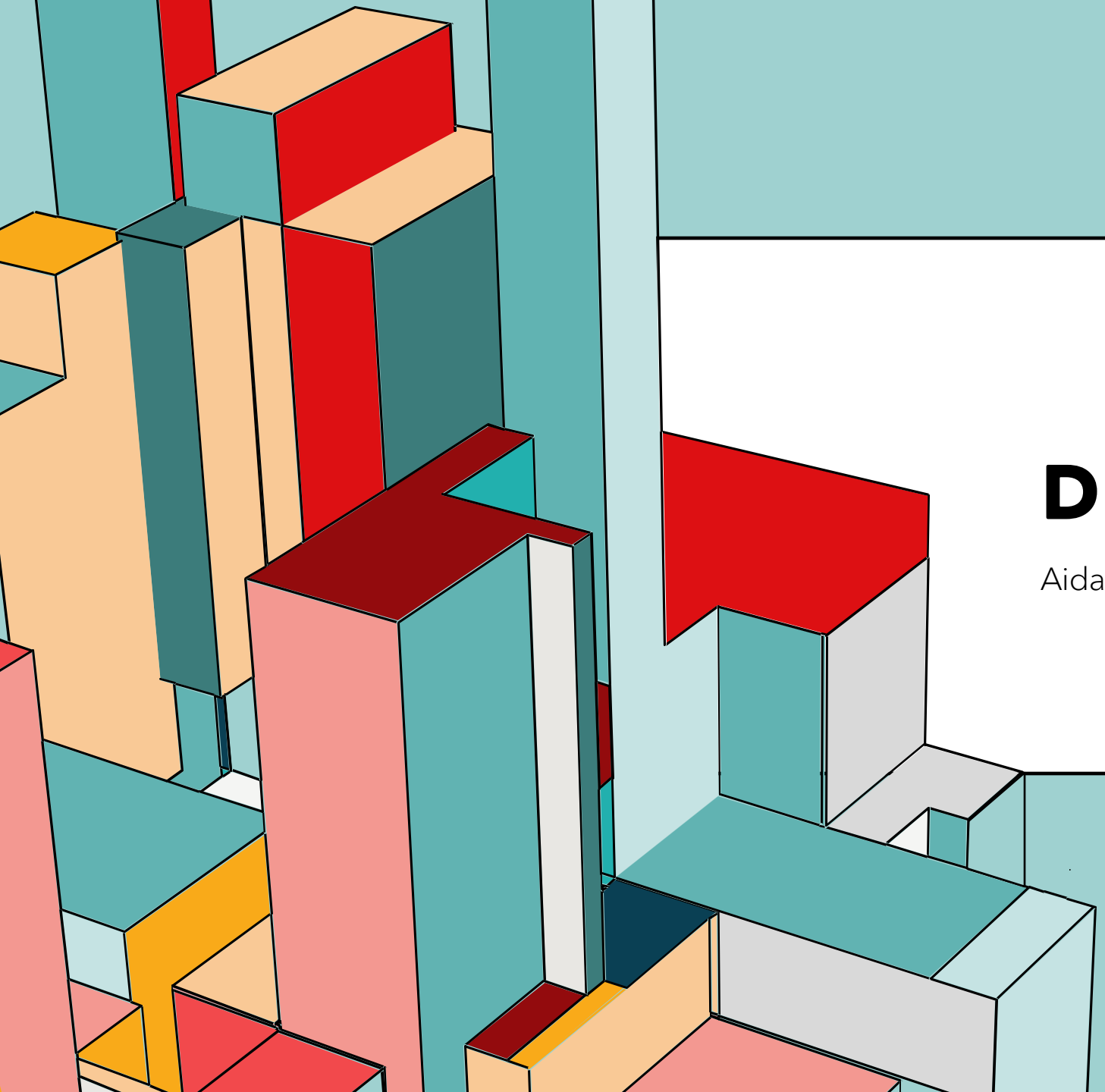


DESIGN PATTERNS

Aidan, Johanna, Lisa, Luca, Marcel und Nora

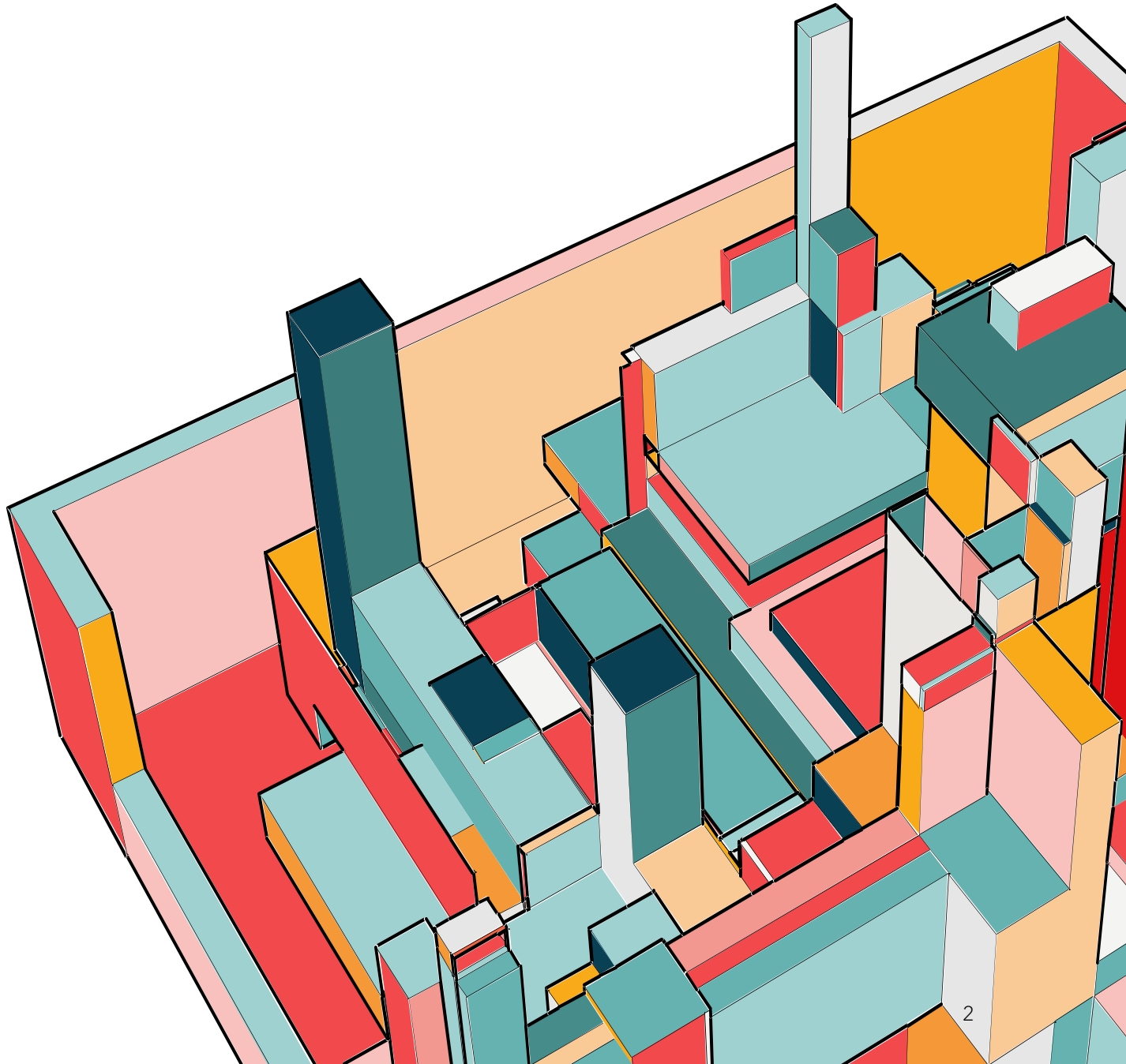
AGENDA

1. ÜBERBLICK DESIGN PATTERN

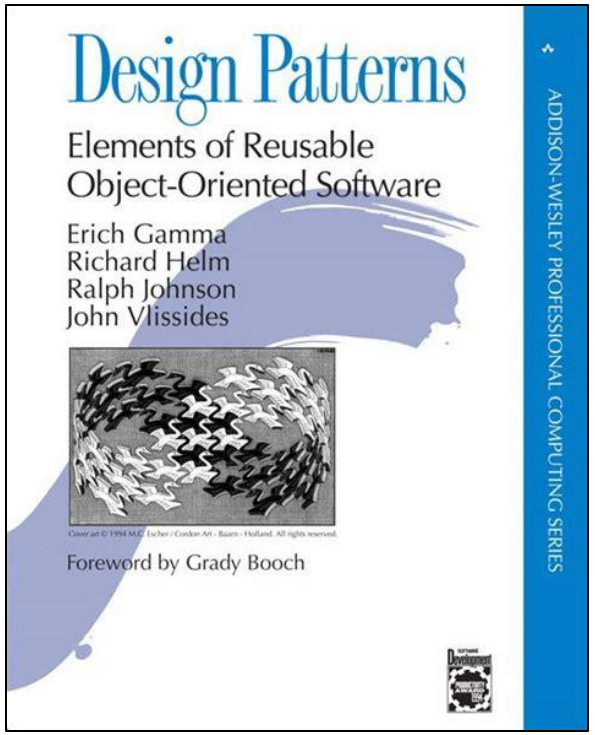
2. CREATIONAL PATTERN
   - ↳ Factory Method

3. STRUCTURAL PATTERN
   - ↳ Composite

4. BEHAVIORAL PATTERN
   - ↳ Strategy



WAS SIND DESIGN PATTERNS?



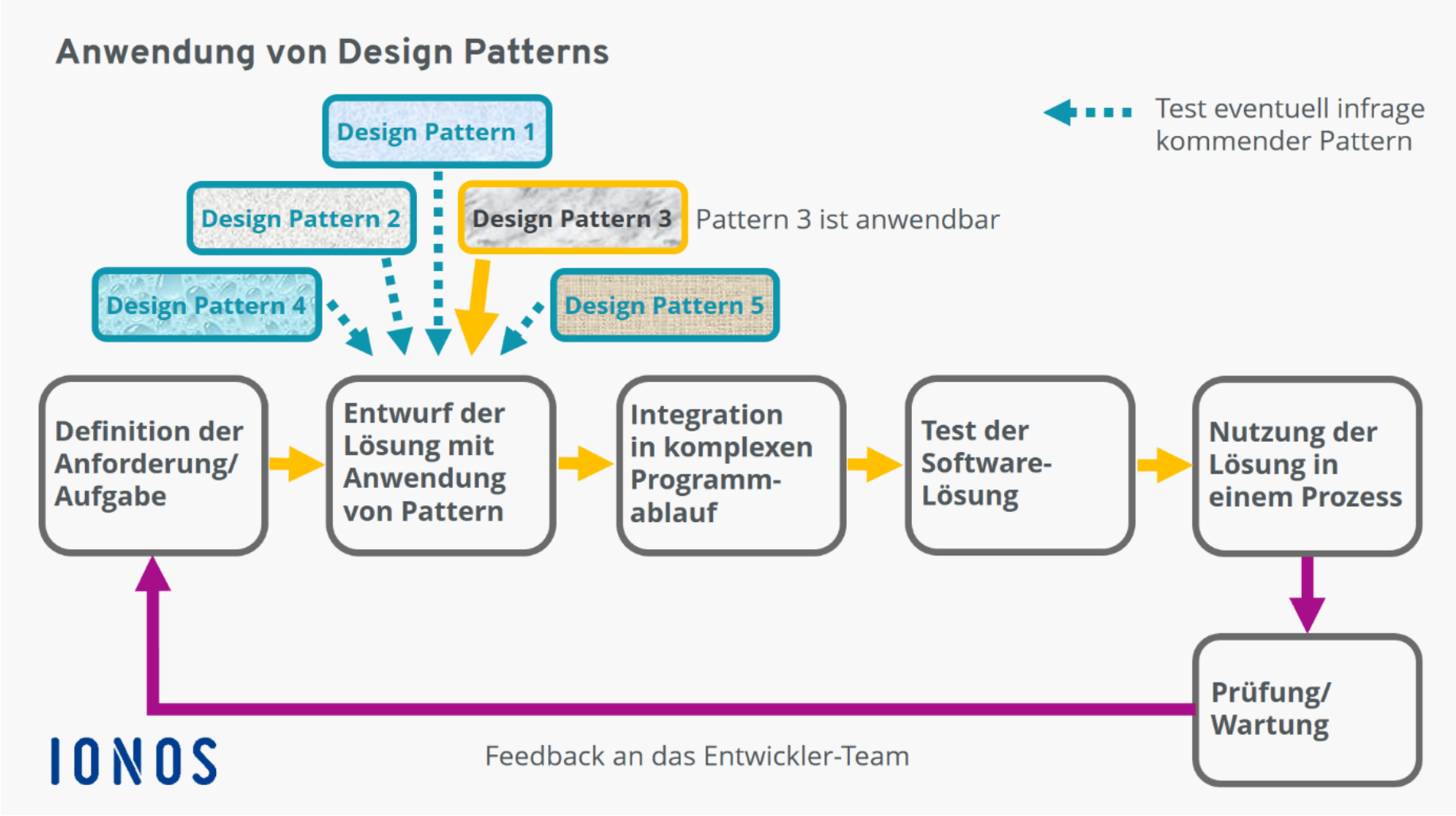
Quelle: https://springframework.guru/gang-of-four-design-patterns/

Ursprung: Gang of Four, 1994

➡ 23 häufige Probleme in OOP und wie man sie lösen kann

- Wiederverwendbare Vorlagen ➡ vereinfachen Entwurfsprozess

➡ müssen nochmal individuell angepasst werden



Quelle: https://www.ionos.de/digitalguide/websites/web-entwicklung/was-sind-design-patterns/

WARUM DESIGN PATTERNS?

Vorteile

- Kosten und Zeitersparnis
- Gemeinsames Vokabular
- Vereinfachte Dokumentation

Nachteile

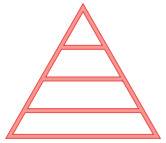
- Umfangreiches Wissen benötigt
- Kreativität einschränkend

KLASSIFIZIERUNG VON DESIGN PATTERNS

CREATIONAL

Erstellungsmechanismen für Objekte

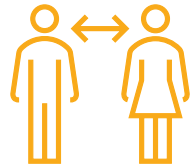
- flexibel und wiederverwendbar



STRUCTURAL

Beziehungen zwischen Klassen

- Bildung einer übergeordneten Struktur
- flexibel und effizient



BEHAVIORAL

Verhalten der Software

- Verantwortlichkeiten zwischen Objekten

KLASSIFIZIERUNG VON DESIGN PATTERNS

CREATIONAL

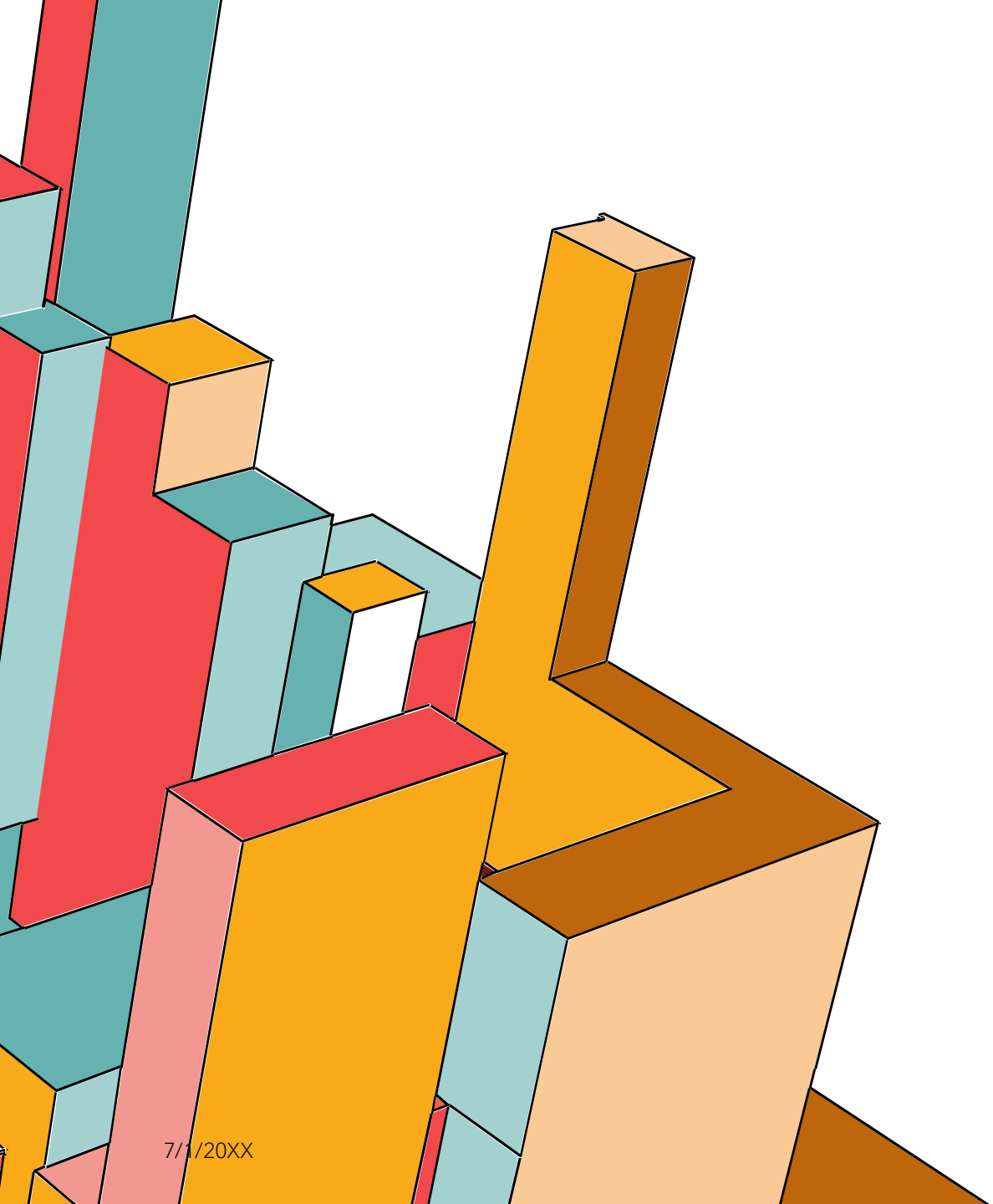
- Singleton
- **Factory Method**
- Abstract Factory
- Builder
- Prototype

STRUCTURAL

- Adapter
- Bridge
- **Composite**
- Decorator
- Facade
- Flyweight

BEHAVIORAL

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- **Strategy**
- Template Method
- Visitor



FACTORY METHOD

Entwurfsmuster, das eine Schnittstelle für die Erstellung von Objekten in einer Oberklasse bietet, es aber Unterklassen ermöglicht, den Typ der erstellten Objekte zu ändern

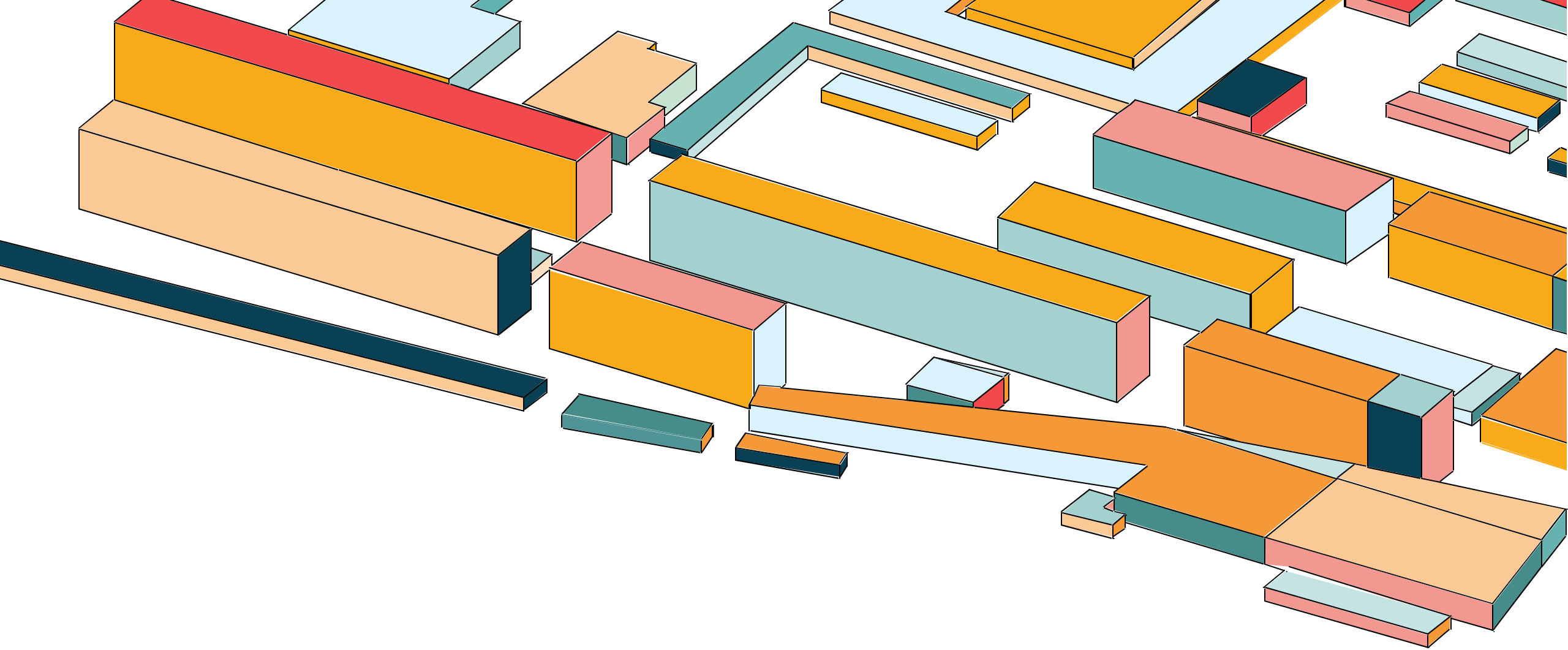
FACTORY METHOD

Problem/Motivation:

Süßigkeitenhersteller, der auf Schokolade spezialisiert ist möchte nun sein Sortiment erweitern.

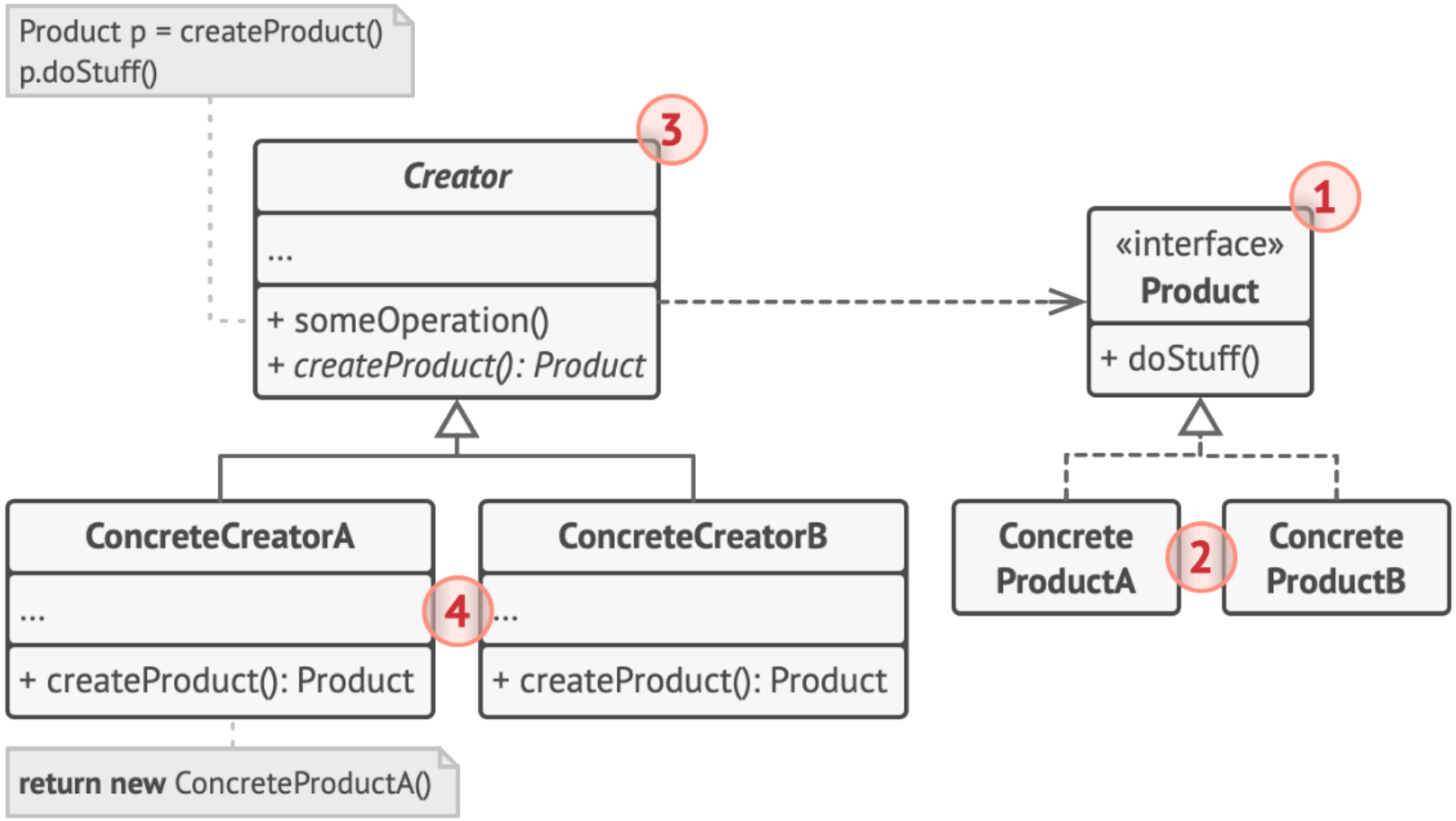
➡ gesamte Codebase betroffen

```
//Creation of the Product-Object
if(productOption.equals(Product.CHOCOLATE)){
    theProduct = new Chocolate();
} else if (productOption.equals(Product.GUMMYBEAR)){
    theProduct = new GummyBear();
} else {
    System.out.println("Wrong entry. Please write C or G the next time.");
}
```

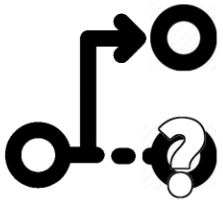


DEMO: FACTORY METHOD

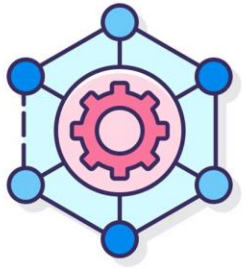
STRUKTUR


Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
3
«interface»
Product
+ doStuff()
1
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
4
Concrete
ProductA
2
Concrete
ProductB
return new ConcreteProductA()

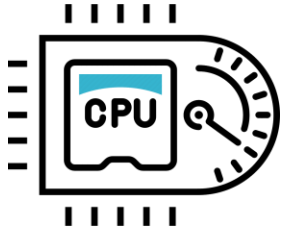
WANN SOLLTE MAN SIE BENUTZEN?



kein Wissen über die Typen und Abhängigkeiten der Objekte



Schnittstelle zur Erweiterung von"Libraries" oder "Frameworks"



Durch Wiederverwendbarkeit Systemressourcen sparen

VOR/NACHTEILE DER FACTORY METHOD

Vorteile

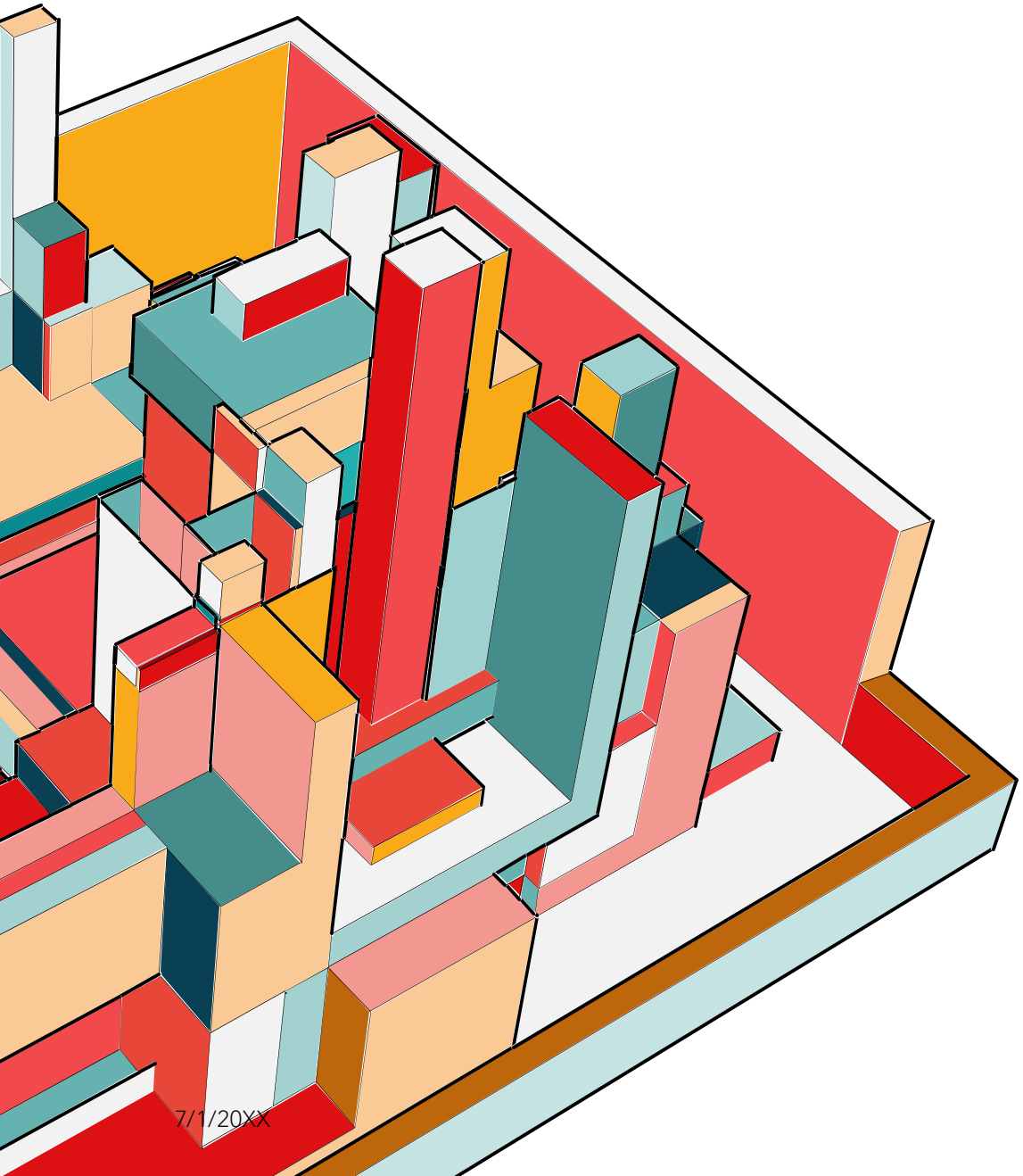
- Code einfacher zu warten
- Neue Typen einfach erstellbar
- Unabhänigkeit zwischen "Creator" und "Product"

Nachteile

- Komplizierte Codebase (viele Klassen)

ÜBERBLICK: STRUCTURAL PATTERNS

- zeigt Beziehungen zwischen Objekten und Klassen
- vereinachte Abbildung der gesamten Hierarchie
- Flexibilität und Effizienz bleiben dennoch erhalten
- organisiert die Klassen- und Objektstruktur



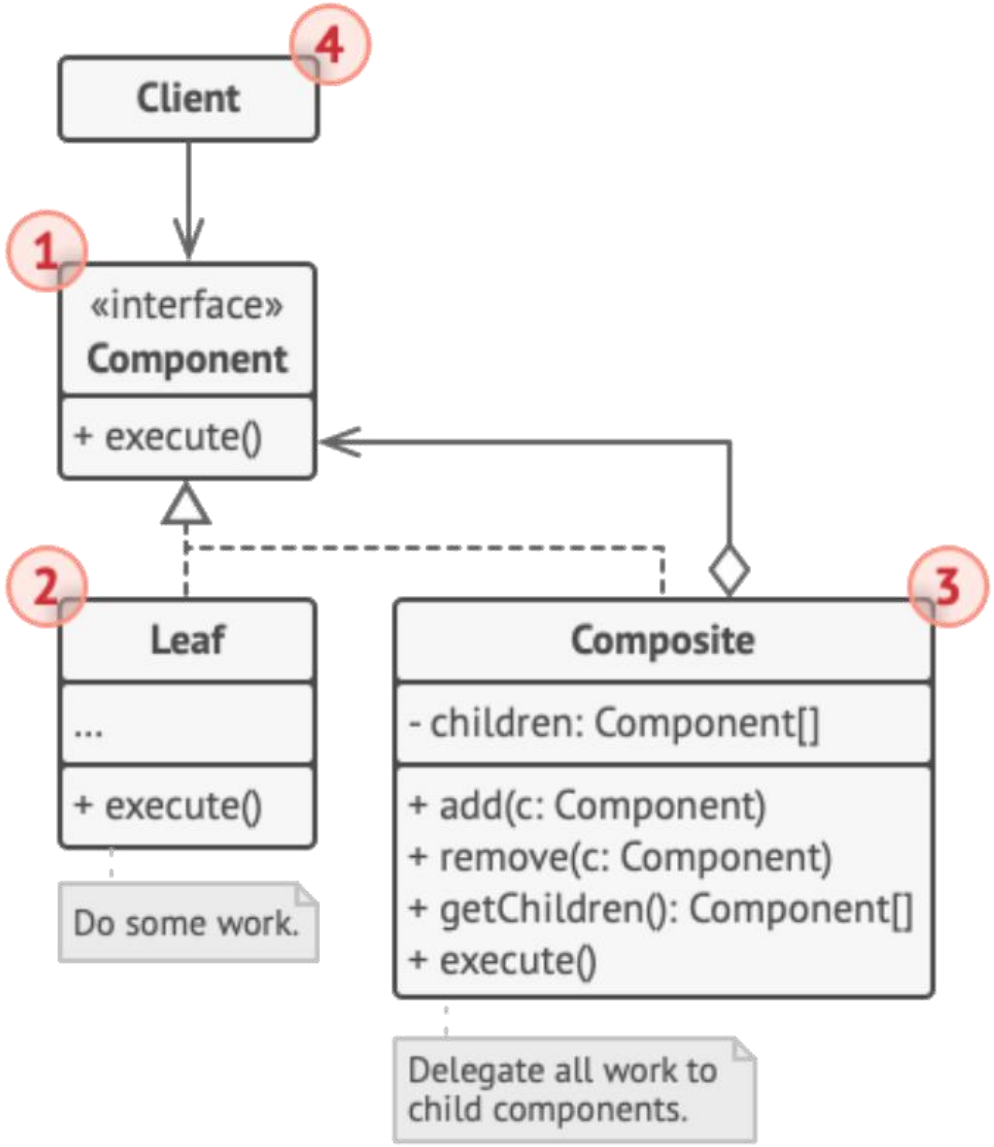
COMPOSITE

Dieses Design-Pattern befasst sich mit der Komposition von Klassen und Objekten, dabei wird das Vererbungskonzept verwendet, um die Beziehungen zueinander darzustellen und zu definieren. Dadurch ist es möglich neue Funktionalitäten zu erhalten.

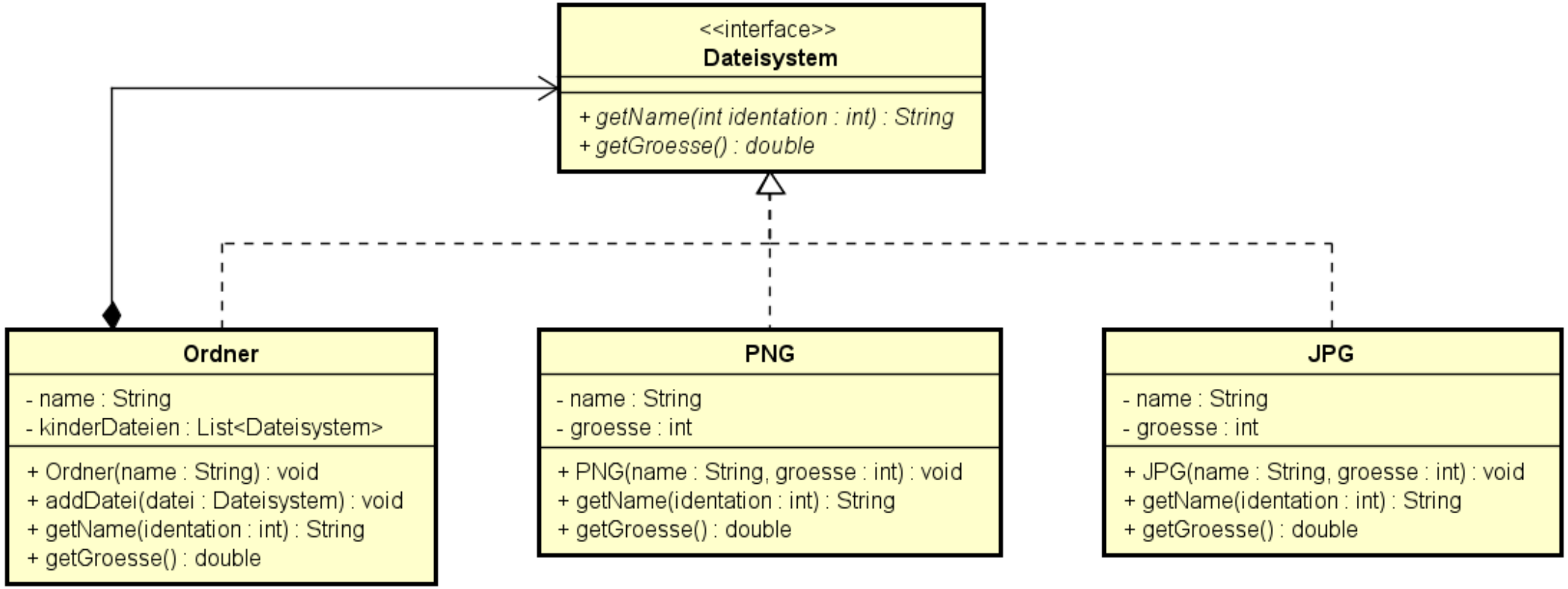
COMPOSITE

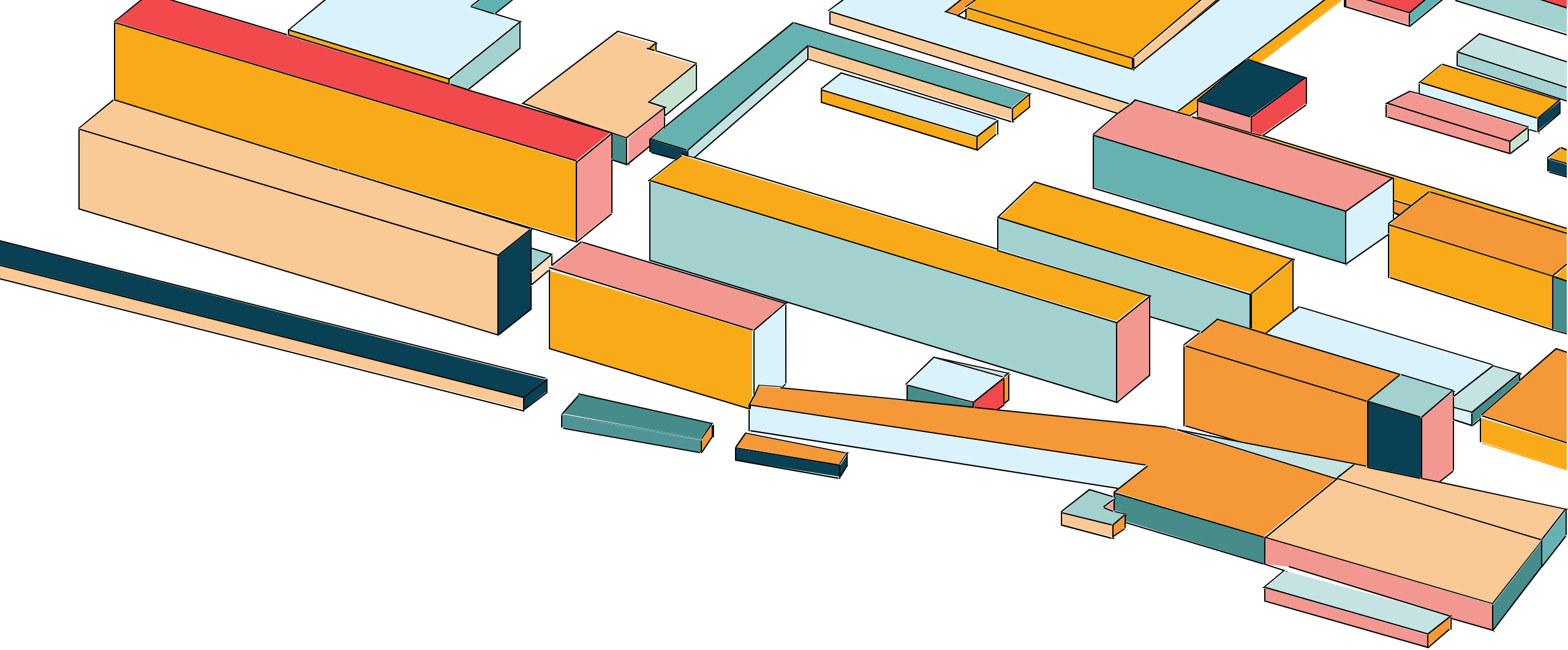
- definieren Klassenhierarchien mit primitive und komplexen Objekten
- aufgebaut in einer Baum-Struktur
- mehrere Objekte stehen in einer Beziehung zueinander
    - können auch als individuelles Objekt genutzt werden

COMPOSITE: STRUKTUR



DEMO: STRUKTUR





DEMO: COMPOSITE

VOR- UND NACHTEILE

- wir verwenden sie, bei der Implementierung einer baum-ähnlichen Struktur
- wenn der Client primitive und komplexe Elemente gleich behandeln soll

VORTEILE

- einfache Umsetzung einer Baum-Struktur
- Open / Closed Principle
- vereinfachter Client-Code (z.B. Fallunterscheidung entfällt)

NACHTEILE

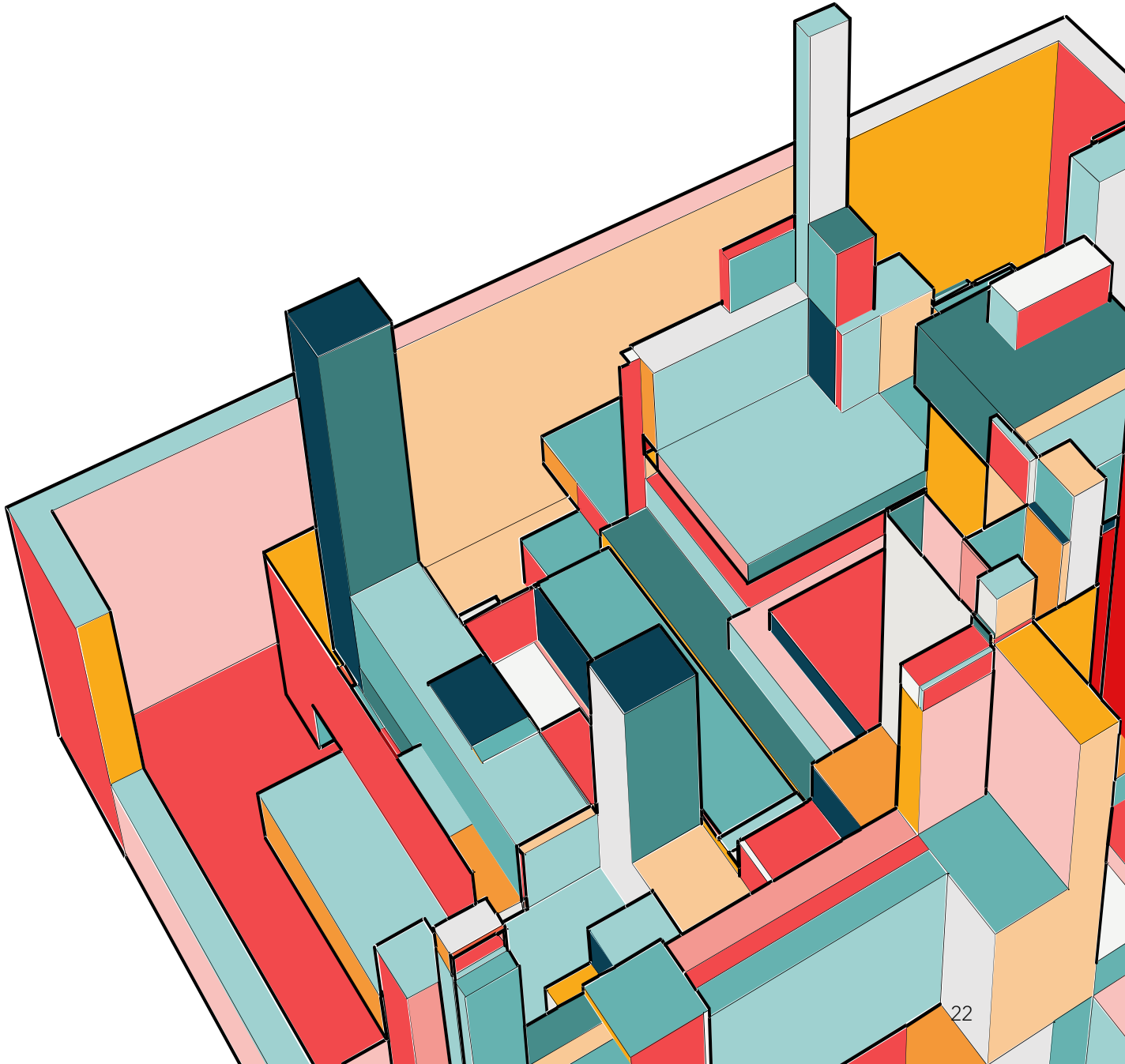
- gemeinsame Interfaces sind schwierig umsetzbar
- bei starker Generalisierung leidet die Verständlichkeit
- anspruchsvolle Definition der allgemeinen Component-Schnittstelle

ÜBERBLICK: BEHAVIORAL PATTERNS

- Interaktion zwischen Objekten und Klassen
- Effektive Kommunikation und Zuweisung von Verantwortlichkeiten zwischen Objekten
- Erhöhung der Flexibilität bei der Durchführung
- Beispiele: Chain of responsibility, Command, Interpreter, Iterator, Visitor etc.

STRATEGY

Entwurfsmuster, mit dem man eine Familie von Algorithmen definieren, jeden von ihnen in einer eigenen Klasse unterbringen und ihre Objekte austauschbar machen kann



STRATEGY

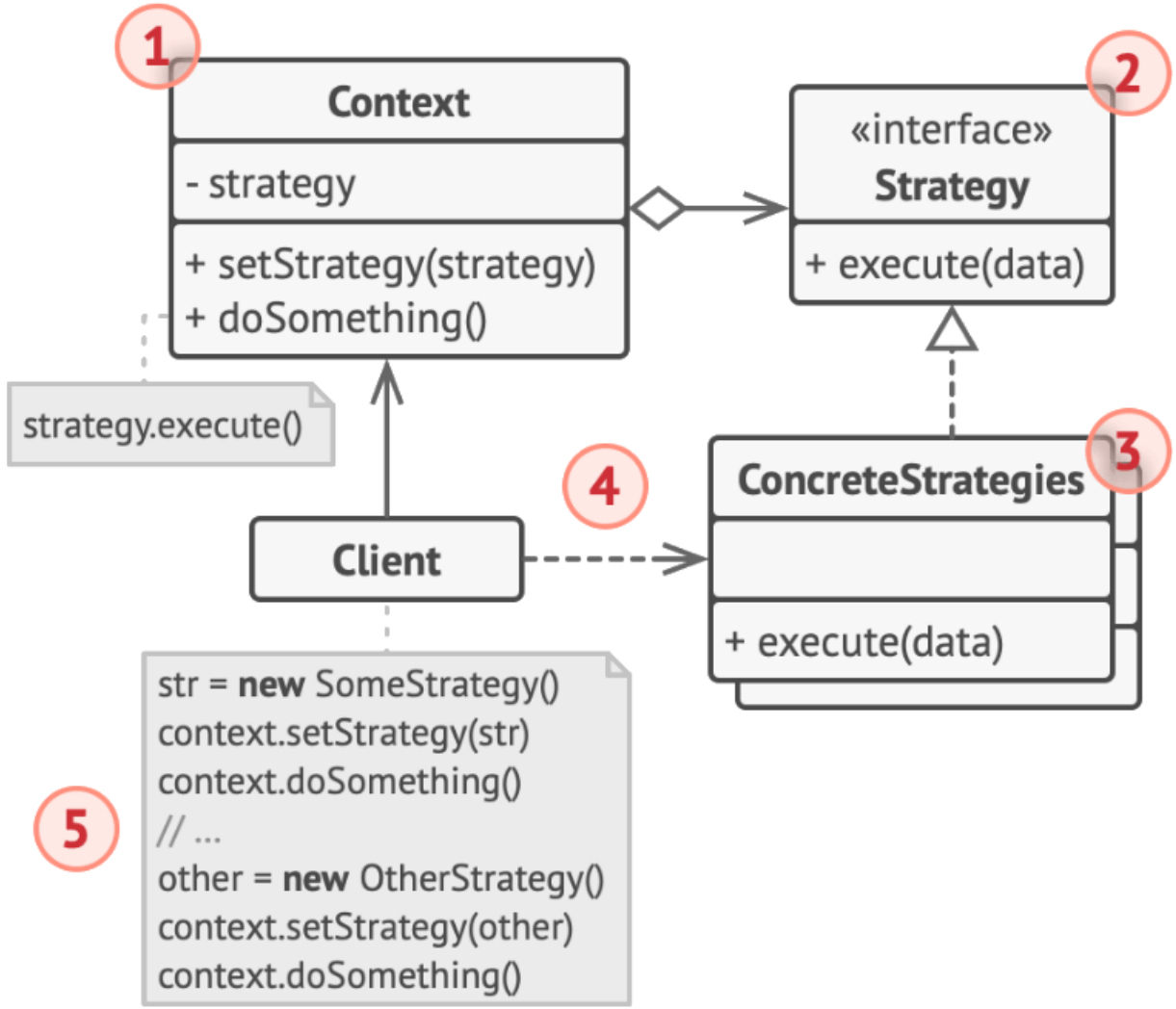
ANWENDUNG

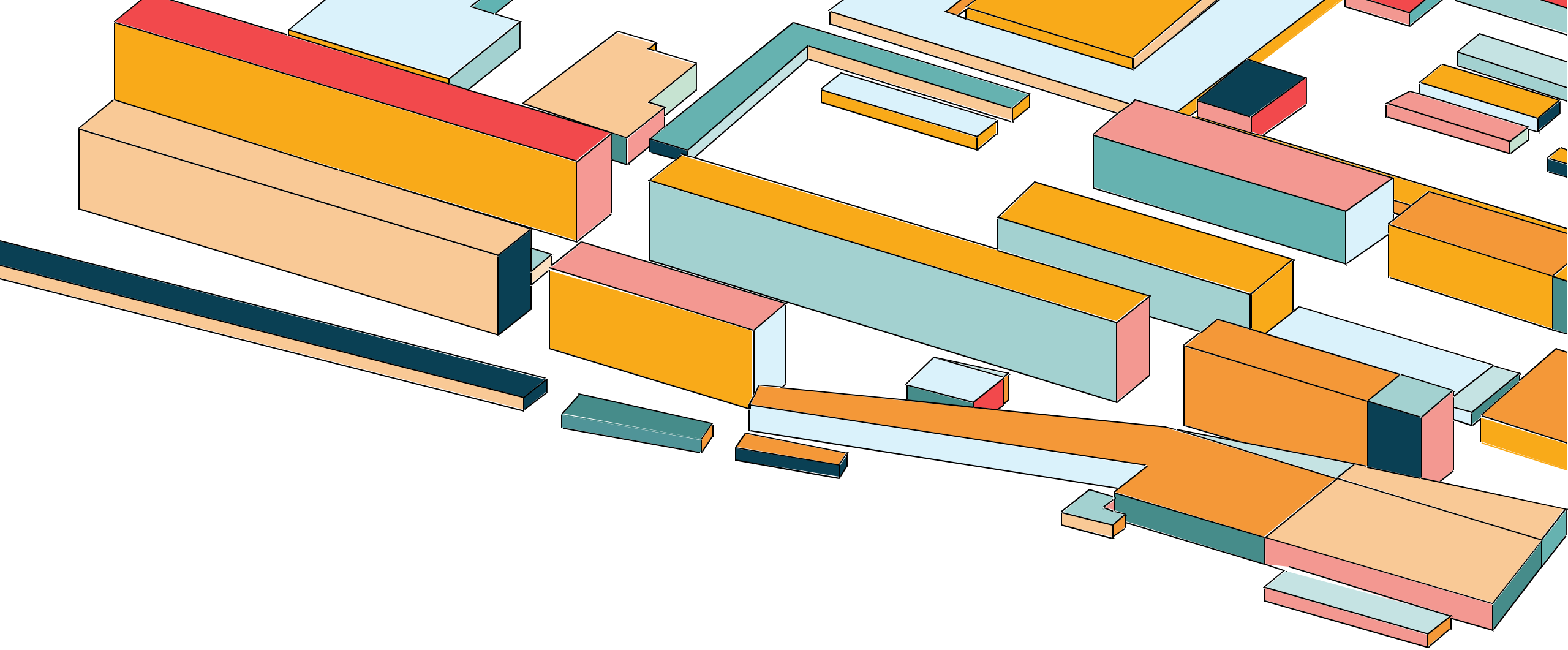
- Elimination bedingter Anweisungen
- Gekapseltes Verhalten in Klassen
- Schwierigkeit beim Hinzufügen neuer Strategien
- Client ist sich der Strategien bewusst
- Client wählt Strategy
- Real Life Beispiel: java.util.Comparator

KONZEPT

- Abstrakte Basisklasse
- Pro Strategie eine konkrete Klasse
- Entfernen von if/else-Bedingungen
- Strategien sind unabhängig

STRUKTUR


1
Context
- strategy
+ setStrategy(strategy)
+ doSomething()
2
«interface»
Strategy
+ execute(data)
strategy.execute()
4
3
ConcreteStrategies
+ execute(data)
Client
5
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()



DEMO: STRATEGY

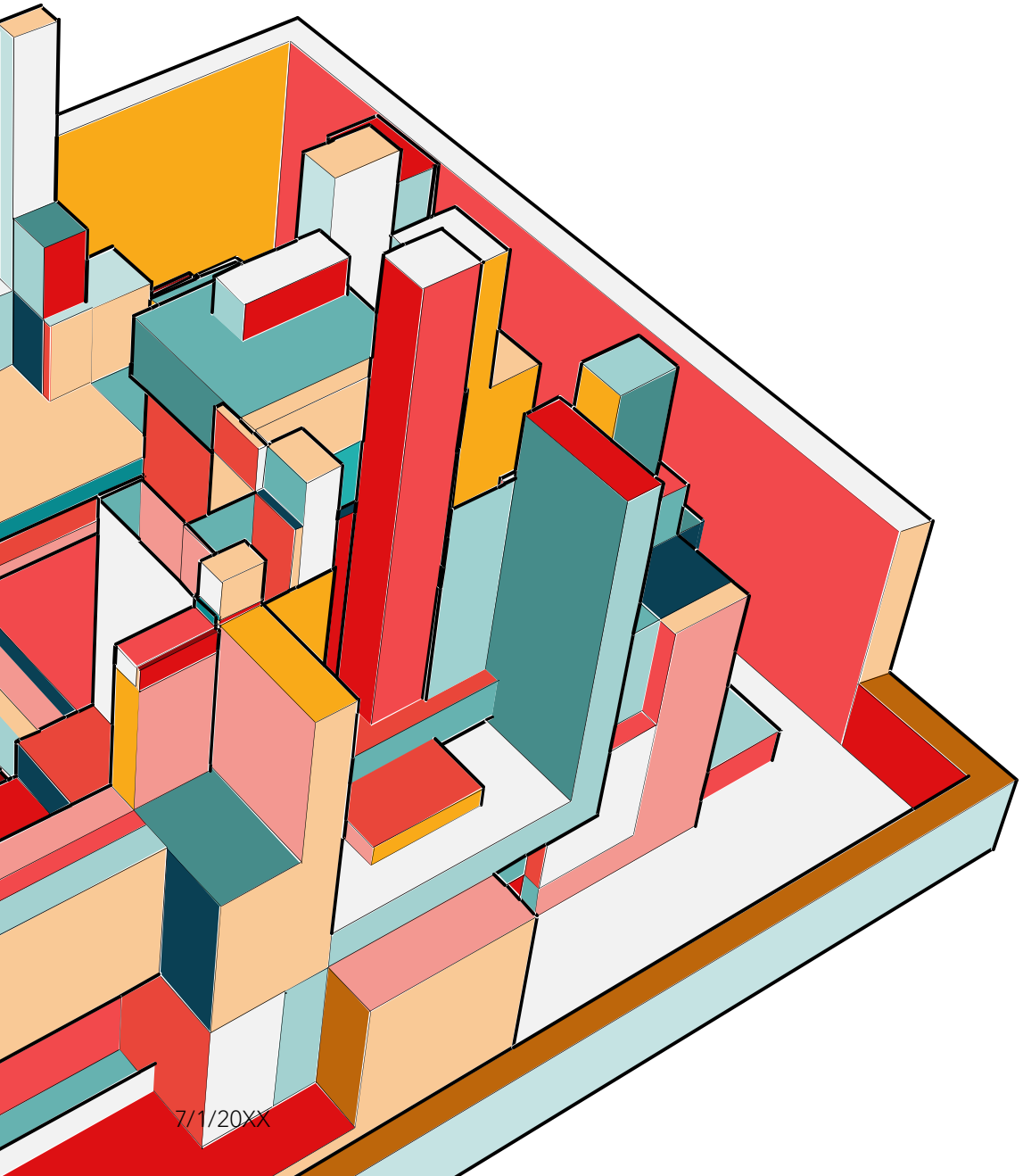
VOR- UND NACHTEILE

VORTEILE

- zur Laufzeit austauschbar
- Algorithmus und Code, der diesen verwendet, sind von einander isolierbar
- Komposition statt Vererbung
- Open/Closed Principle

NACHTEILE

- Wenn nur ein paar Algorithmen, die sich selten ändern
  → kein Grund das Programm zu verkomplizieren
- Clients müssen Unterschiede zwischen Strategien kennen

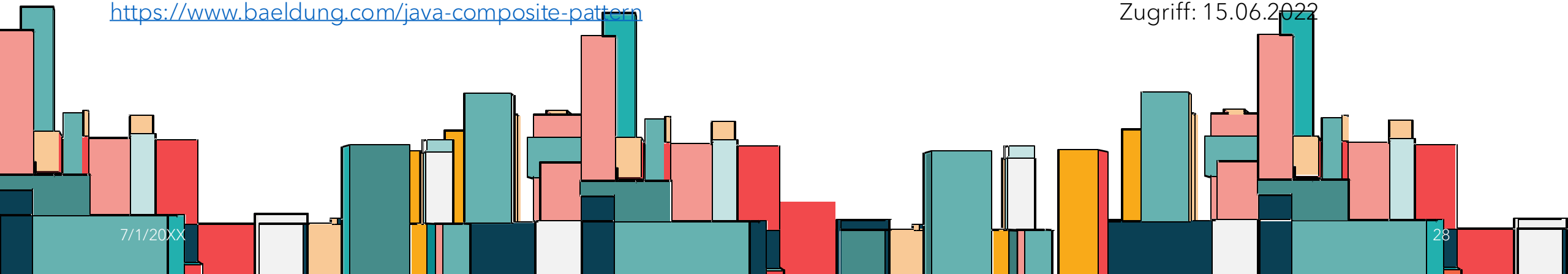


SUMMARY

Kein Plan ob wir diese Folie wirklich brauchen

QUELLEN

https://refactoring.guru/design-patterns/	Zugriff: 05.06.2022
https://app.pluralsight.com/paths/skill/design-patterns-in-java	Zugriff: 30.05.2022
https://www.youtube.com/watch?v=MaLvKBGIQVg&t=3s	Zugriff: 06.06.2022
https://springframework.guru/gang-of-four-design-patterns/	Zugriff: 15.06.2022
https://www.ionos.de/digitalguide/websites/web-entwicklung/was-sind-design-patterns/	Zugriff: 15.06.2022
https://www.youtube.com/watch?v=EdFq_JIThqM	Zugriff: 15.06.2022
https://www.youtube.com/watch?v=ub0DXaeV6hA	Zugriff: 15.06.2022
https://www.javatpoint.com/composite-pattern	Zugriff: 15.06.2022
https://www.baeldung.com/java-composite-pattern	Zugriff: 15.06.2022



VIELEN DANK!