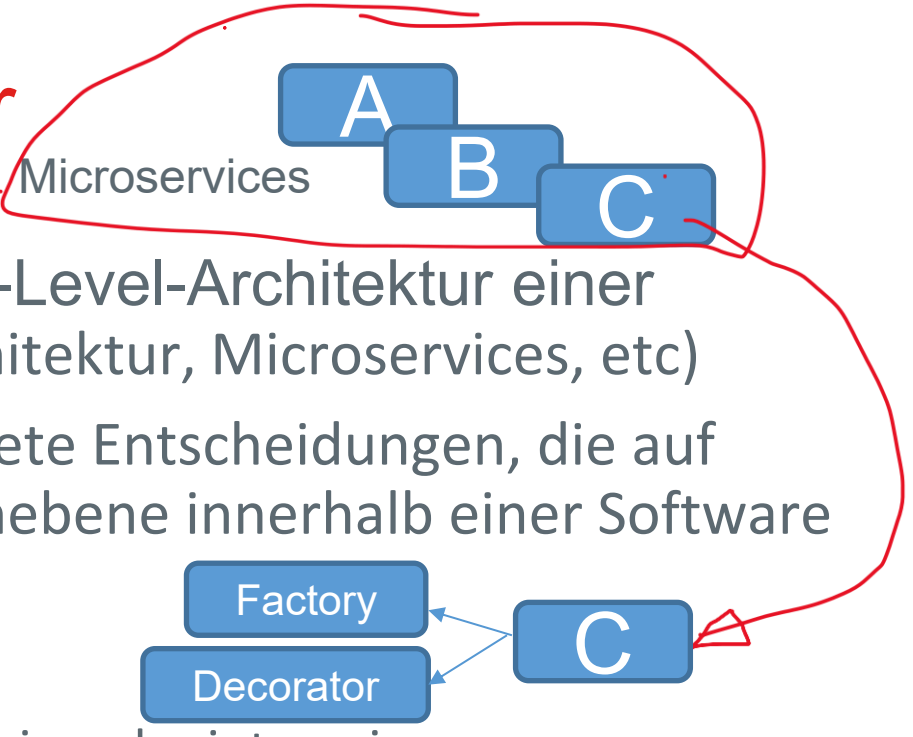
DHBW Stuttgart
Duale Hochschule
Baden-Württemberg

Software Engineering: Design Patterns

Dr. Eugenie Giesbrecht

Softwarearchitektur



- Makroarchitektur - High-Level-Architektur einer Software (Schichtenarchitektur, Microservices, etc)
- Mikroarchitektur - konkrete Entscheidungen, die auf Code- und Komponentenebene innerhalb einer Software getroffen werden:

1. Wie Komponenten miteinander interagieren
2. Welche Entwurfsmuster (Design Patterns) verwendet werden
3. Wie Code organisiert wird (z. B. durch Modulgrenzen)

A Pattern Language

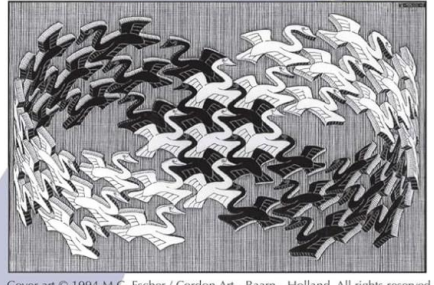
Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

Design Patterns

Elements of Reusable Object-Oriented Software

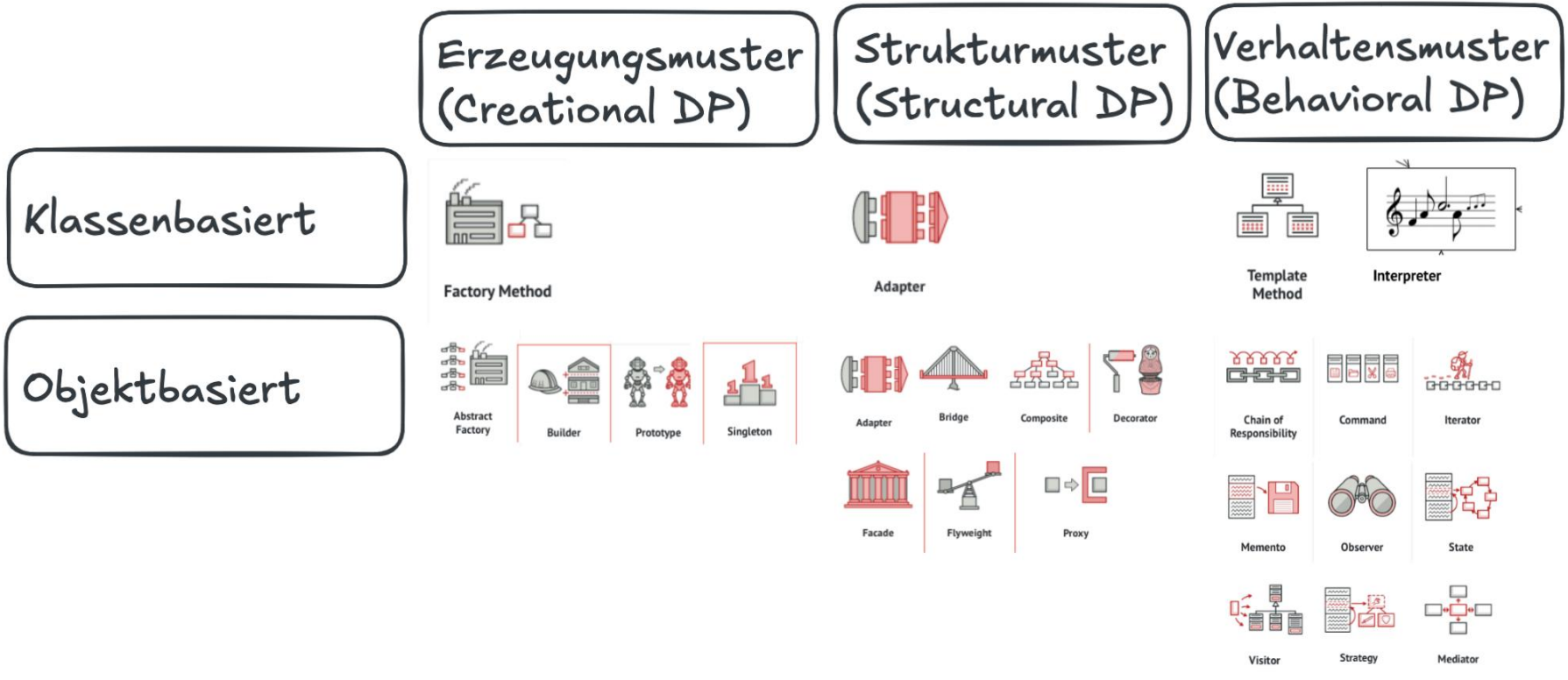
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

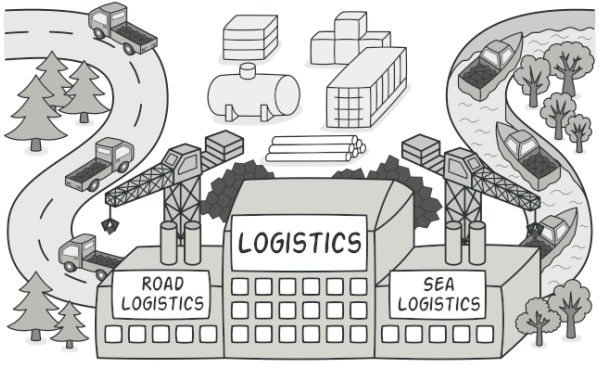
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

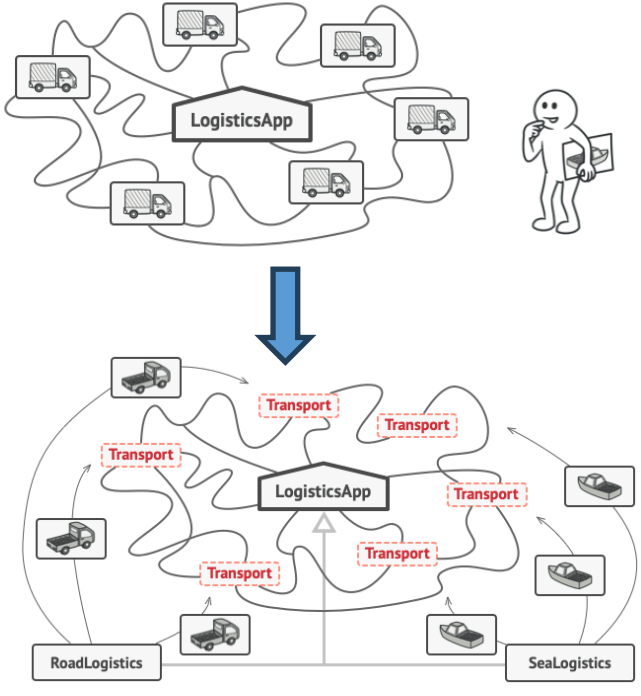
Arten von Design Patterns



Erzeugungsmuster: Factory

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





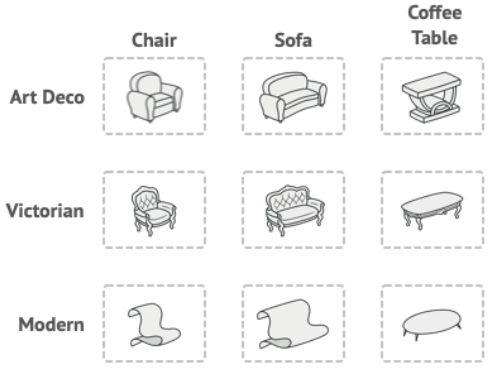
https://refactoring.guru/design-patterns/

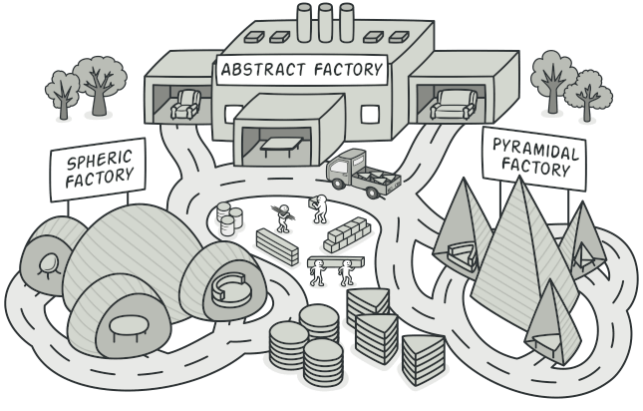
Das Factory Pattern wird verwendet:

- Wenn der genaue Objekttyp erst zur Laufzeit feststeht
- Wenn Nutzer einer Bibliothek oder eines Frameworks eigene Erweiterungen einbauen sollen
- Wenn vorhandene Objekte wiederverwendet werden sollen, um Ressourcen zu sparen

Erzeugungsmuster: Abstract Factory

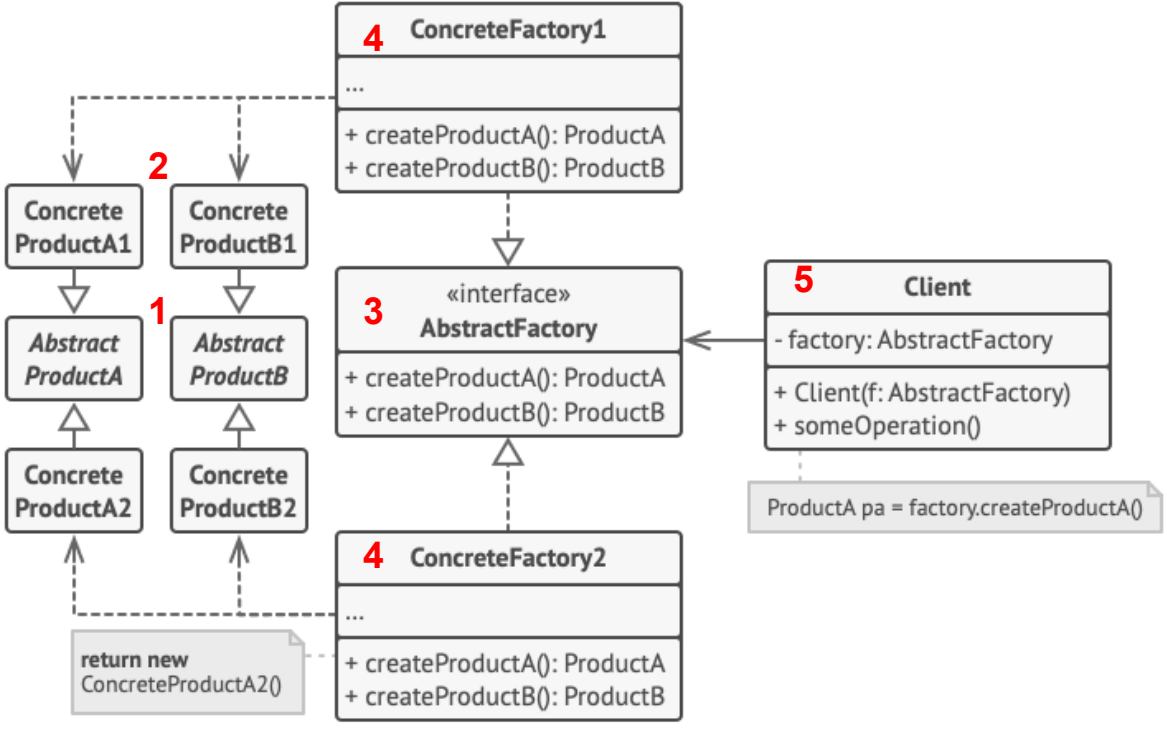
- erlaubt es, Familien verwandter Objekte zu erzeugen, ohne deren konkrete Klassen angeben zu müssen





https://refactoring.guru/design-patterns/abstract-factory

Abstract Factory: Struktur



Übung: ToDo-App mit Factory / Abstract Factory

Sie sollen eine ToDo-App so gestalten, dass Aufgaben flexibel erstellt werden können, ohne dass der Client-Code direkt die konkreten Klassen kennt.

1. Factory-Version

Erstellen Sie ein **Factory Pattern**, das verschiedene **Aufgabentypen** erzeugt:

- TodoTask – normale Aufgabe
- ShoppingTask – Einkaufsliste
- WorkTask – Arbeitsaufgabe

Anforderungen:

- Erstellen Sie eine abstrakte Klasse oder Interface Task mit einer Methode describe().
- Implementieren Sie konkrete Klassen TodoTask, ShoppingTask und WorkTask.
- Erstellen Sie eine TaskFactory mit einer Methode create_task(type: str)

Der Client-Code soll **nur die Factory aufrufen**, z. B.:

```
# Client-Code
factory = TaskFactory()

task1 = factory.create_task("todo")
task2 = factory.create_task("shopping")
task3 = factory.create_task("work")

task1.describe()      # Ausgabe: Dies ist eine allgemeine ToDo-Aufgabe.
task2.describe()      # Ausgabe: Dies ist eine Einkaufsliste-Aufgabe.
task3.describe()      # Ausgabe: Dies ist eine Arbeitsaufgabe.
```

2. AbstractFactory-Version

Anforderungen: Definieren Sie **abstrakte Produkte**: AbstractTask mit describe()

- Implementieren Sie **konkrete Produkte**:
  - SimpleTodoTask, DetailedTodoTask
  - SimpleShoppingTask, DetailedShoppingTask
  - SimpleWorkTask, DetailedWorkTask
- Definieren Sie die **abstrakte Fabrik** AbstractTaskFactory mit Methoden:
  - create_todo_task()
  - create_shopping_task()
  - create_work_task()
- Implementieren Sie **konkrete Fabriken**:
  - SimpleTaskFactory
  - DetailedTaskFactory

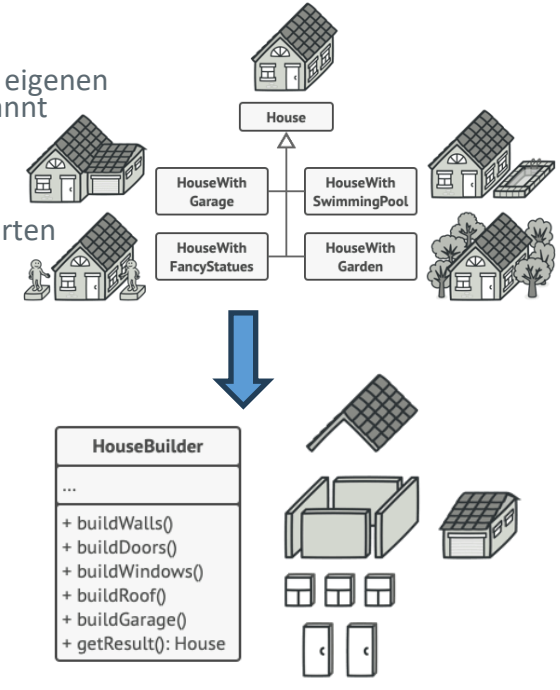
Der Client-Code verwendet die AbstractFactory wie folgt:

```
# Client-Code
factory = DetailedTaskFactory()  # oder SimpleTaskFactory()
todo = factory.create_todo_task()
shopping = factory.create_shopping_task()
work = factory.create_work_task()

todo.describe()         # Detaillierte ToDo-Aufgabe
```

Erzeugungsmuster: Builder

- Erlaubt es, komplexe Objekte Schritt für Schritt zu konstruieren
- Das Builder-Pattern schlägt vor, den Konstruktionscode des Objekts aus seiner eigenen Klasse zu extrahieren und in separate Objekte zu verschieben, die Builder genannt werden



Beispiel Hausbau: Ein Haus hat viele Teile: Wände, Türen, Fenster, Dach, Garage, Garten

Ohne Builder:

```
House house = new House(walls, doors, windows, roof, garage, etc.);
```

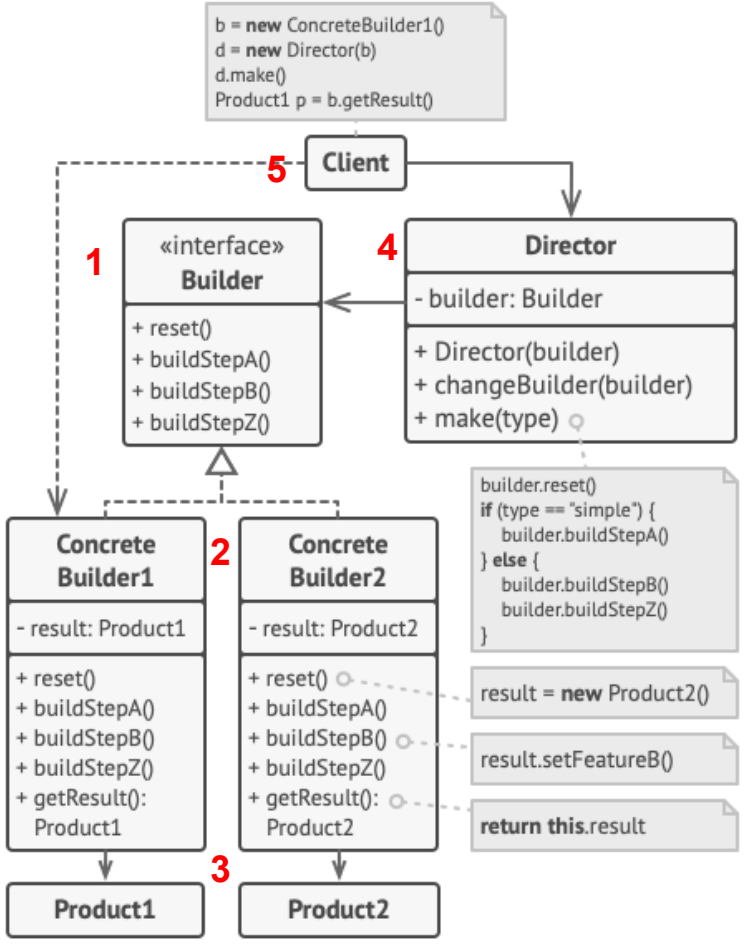
MIT Builder:

```
SchrankBuilder builder = new SchrankBuilder();
        builder.buildWalls();
        builder.buildDoors();
        builder.buildWindows();
```

https://refactoring.guru/design-patterns/

Builder: Struktur

1. Die Builder-Schnittstelle deklariert Produktkonstruktionsschritte
2. Konkrete Builder bieten unterschiedliche Implementierungen der Konstruktionsschritte an
3. Produkte sind resultierende Objekte.
4. Die Director-Klasse definiert die Reihenfolge, in der die Konstruktionsschritte aufgerufen werden
5. Der Client muss eines der Builder-Objekte mit dem Director verknüpfen



Builder-Pattern – Wann verwenden?

1. Komplexe Konstruktoren

- Viele Parameter → unübersichtlich
- Schrittweises Erstellen statt alles auf einmal
- Beispiel: Schrank, Pizza

2. Verschiedene Varianten eines Produkts

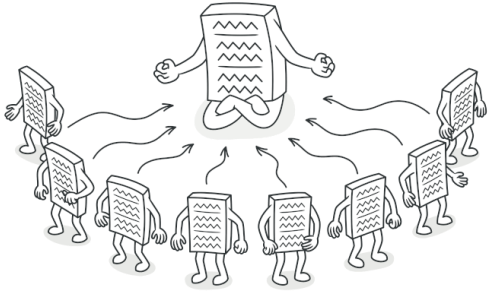
- Gleicher Bauprozess, unterschiedliche Details
- Beispiel: Steinhaus vs. Holzhaus

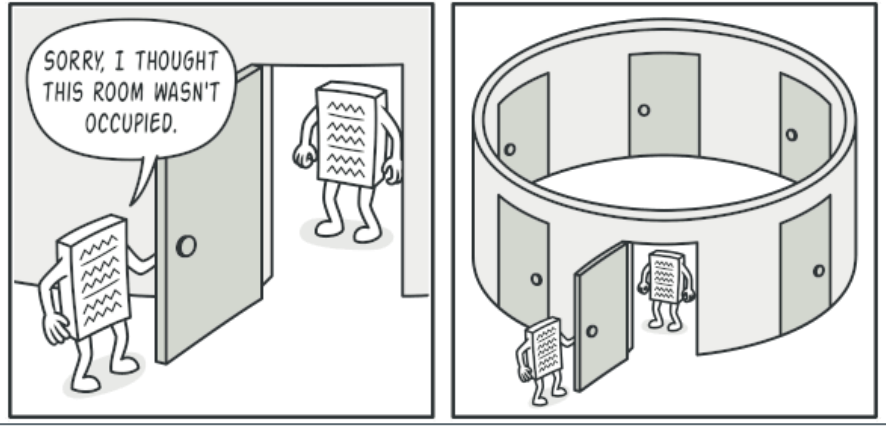
3. Unterschiedliche Schritte bei ähnlichen Prozessen

- Builder trennt Schritte vom Produkt
- Director steuert Reihenfolge
- Beispiel: Ferienhaus (ohne Garage) vs. Familienhaus (mit Garage)

Erzeugungsmuster: Singleton

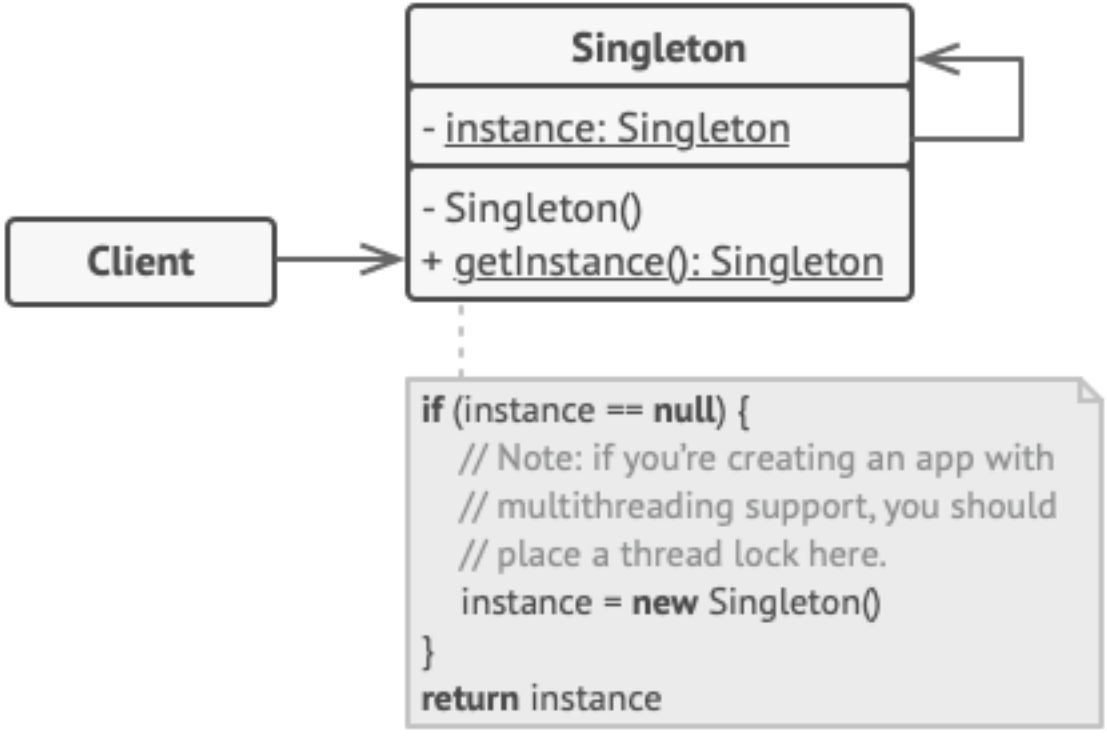
- Singleton ist ein Entwurfsmuster, das verwendet werden kann, um sicherzustellen, dass eine Klasse nur eine Instanz hat. Es bietet einen globalen Zugriffspunkt auf diese Instanz.





https://refactoring.guru/design-patterns/

Singleton: Struktur

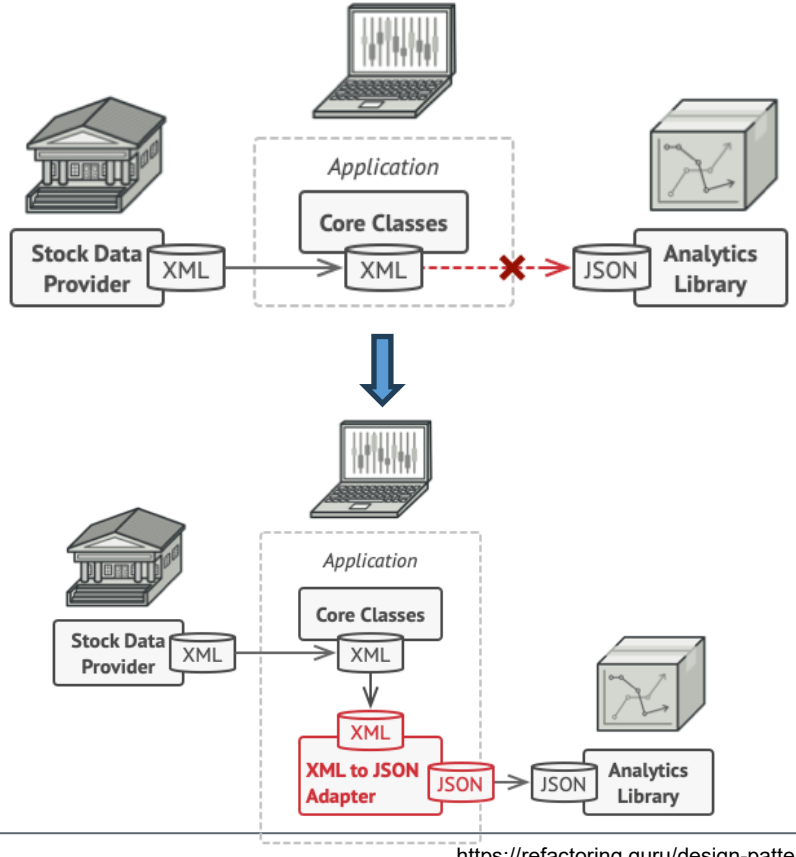


Beispiele für Singleton

- **Datenbankverbindung**
  - Nur ein Objekt verwaltet die Verbindung, damit alle Teile des Programms dieselbe Verbindung nutzen.
- **Logger / Protokollierung**
  - Ein zentrales Logging-Objekt schreibt alle Meldungen, damit Konsistenz und Reihenfolge gewahrt bleiben.
- **Konfigurationsobjekt**
  - Ein Singleton speichert globale Einstellungen, z. B. Sprache, Theme oder Server-URL, die überall im Programm gleich sind.
- **Thread-Pool / Ressourcenmanager**
  - Nur ein Objekt verwaltet die begrenzte Anzahl an Threads oder Systemressourcen.
- **Cache oder Speicher für Zwischenergebnisse**
  - Ein zentrales Objekt hält häufig genutzte Daten bereit, damit sie nicht mehrfach geladen werden müssen.

Strukturmuster: Adapter

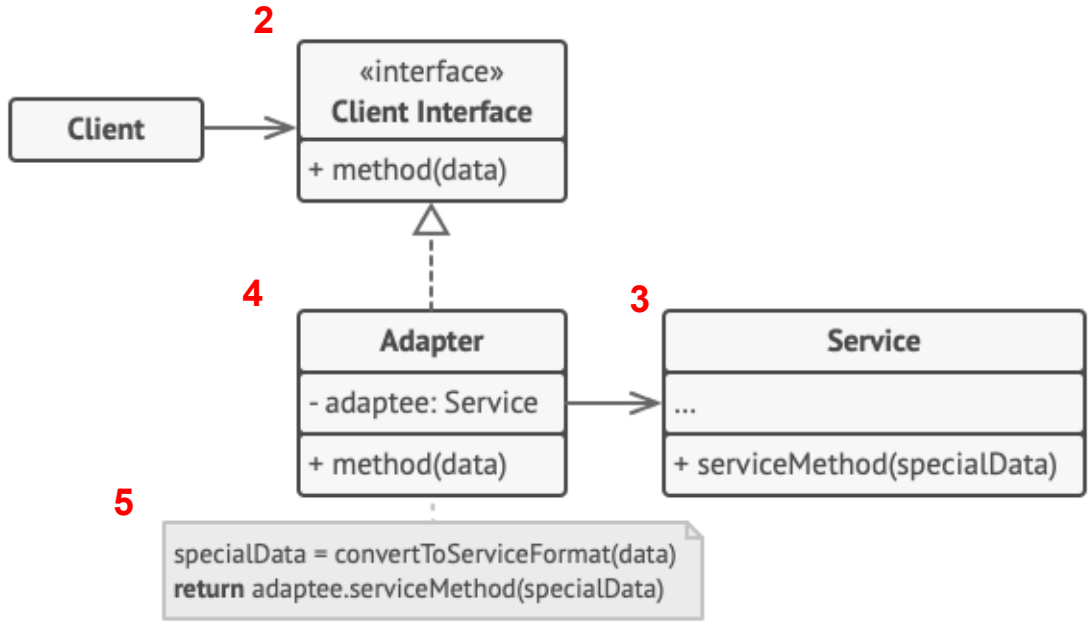
- Verbindet zwei inkompatible Schnittstellen, indem es eine "Adapter"-Klasse bereitstellt
- Beispiel: Ein Adapter, der eine alte Datenbank-Schnittstelle mit einer neuen API verbindet



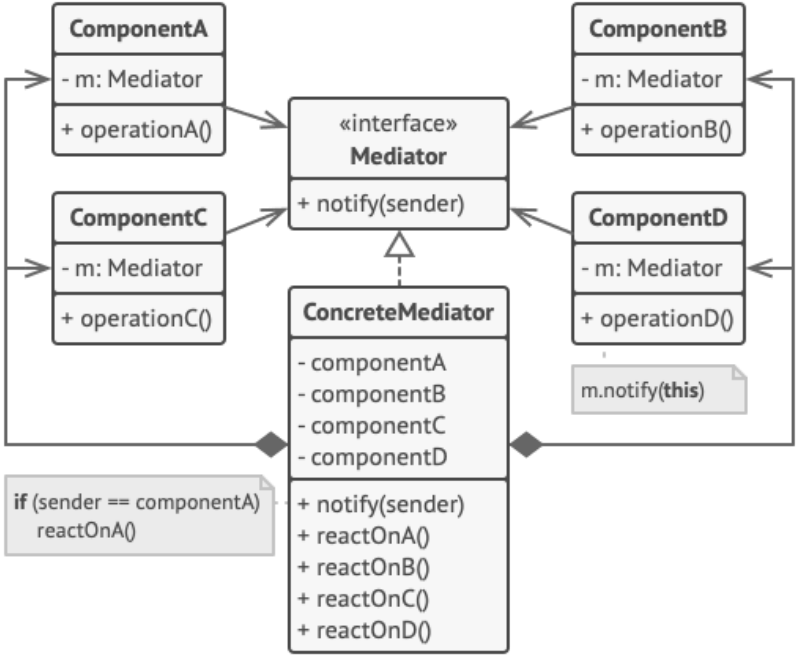
https://refactoring.guru/design-patterns/

Adapter: Struktur

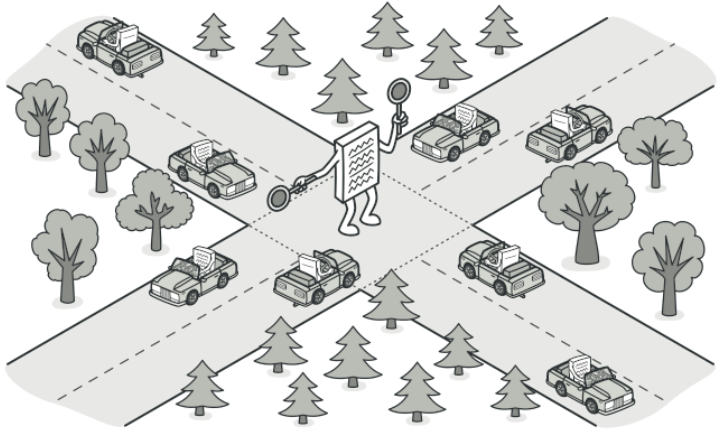
1. Der Client ist eine Klasse, die die bestehende Geschäftslogik des Programms enthält.
2. Die Client-Schnittstelle beschreibt ein Protokoll, dem andere Klassen folgen müssen, um mit dem Client-Code zusammenarbeiten zu können.
3. Der Service ist eine nützliche Klasse mit einer inkompatiblen Schnittstelle.
4. Der Adapter ist eine Klasse, die sowohl mit dem Client als auch mit dem Service zusammenarbeiten kann.
5. Der Client-Code wird nicht an die konkrete Adapterklasse gekoppelt, solange er mit dem Adapter über das Client-Interface arbeitet.



Verhaltensmuster: Mediator



- Mediator ist ein Verhaltensmuster, mit dem Sie chaotische Abhängigkeiten zwischen Objekten reduzieren können.
- Das Muster schränkt die direkte Kommunikation zwischen den Objekten ein und zwingt sie, nur über ein Mediator-Objekt zusammenzuarbeiten.



https://refactoring.guru/design-patterns/

Übung: Adapter- und Mediator-Pattern in einer TODO-App

Implementieren Sie in Ihrer bestehenden TODO-Applikation das **Adapter-Pattern** und das **Mediator-Pattern:**

- Erweitern Sie Ihre TODO-App um eine fiktive externe Aufgabenquelle (eigene mini-API), deren Datenformat nicht Ihrer internen Task-Klasse entspricht.
- **Implementieren Sie einen Adapter**, der externe Aufgabenobjekte in das interne Task-Format Ihrer Anwendung übersetzt, ohne den bestehenden Code der TODO-App zu verändern.

Austauschbarkeit und Komplementarität von Design Patterns

- **Austauschbarkeit:** Mehrere Patterns lösen ähnliche Probleme
- **Komplementarität:** Patterns können sich gegenseitig ergänzen.
- **Praxisnutzen:**
  - Patterns werden selten isoliert eingesetzt.
  - Durch Kombination entsteht **flexibler, wartbarer und erweiterbarer Code**

Zusammenspiel der Design Patterns

```
manager = TaskManager.get_instance()
mediator = TaskMediator(manager)

task1 = TaskFactory.create_task("urgent")
mediator.create_and_add(task1)

factory = WorkTaskFactory()
task2 = factory.create_task()
mediator.create_and_add(task2)

task3 = TaskBuilder().set_title("Prepare slides").set_priority("high").build()
mediator.create_and_add(task3)

external = ExternalTodoService().create_item("Buy milk")
task4 = TaskAdapter().adapt(external)
mediator.create_and_add(task4)
```