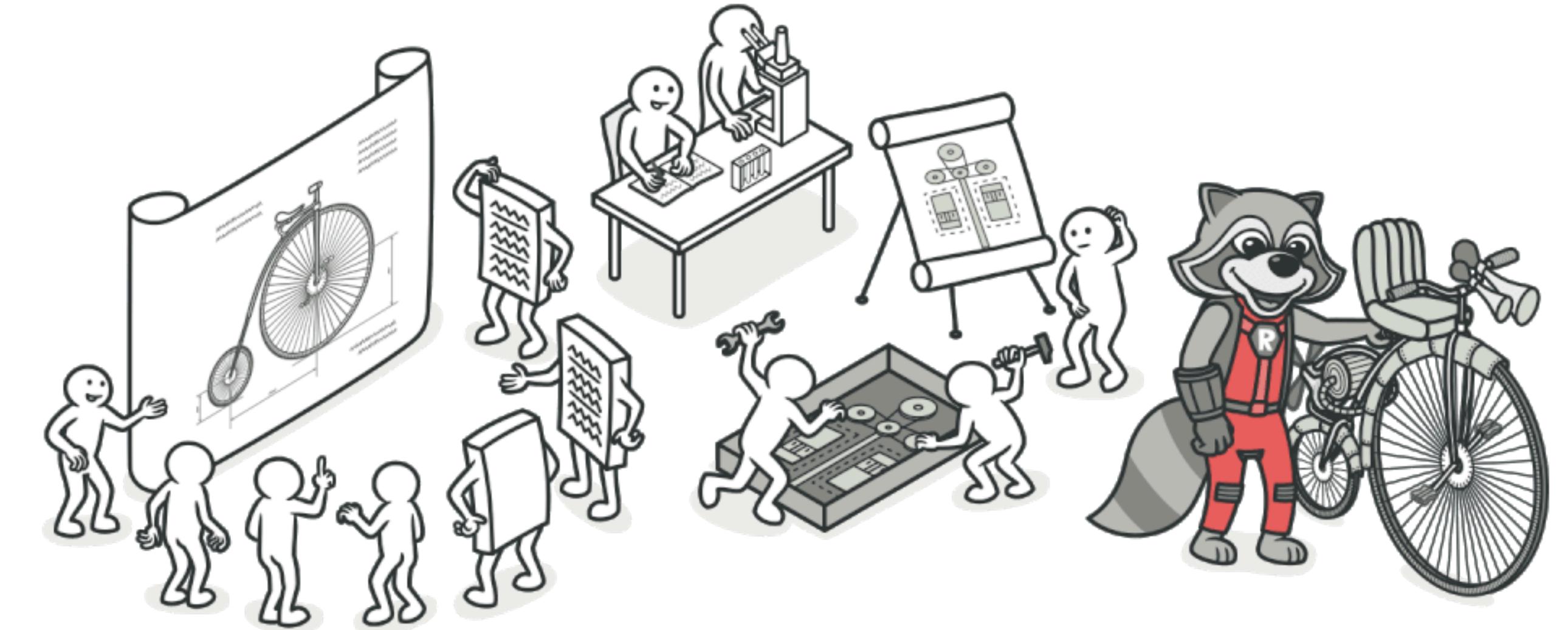


Verhaltensmuster

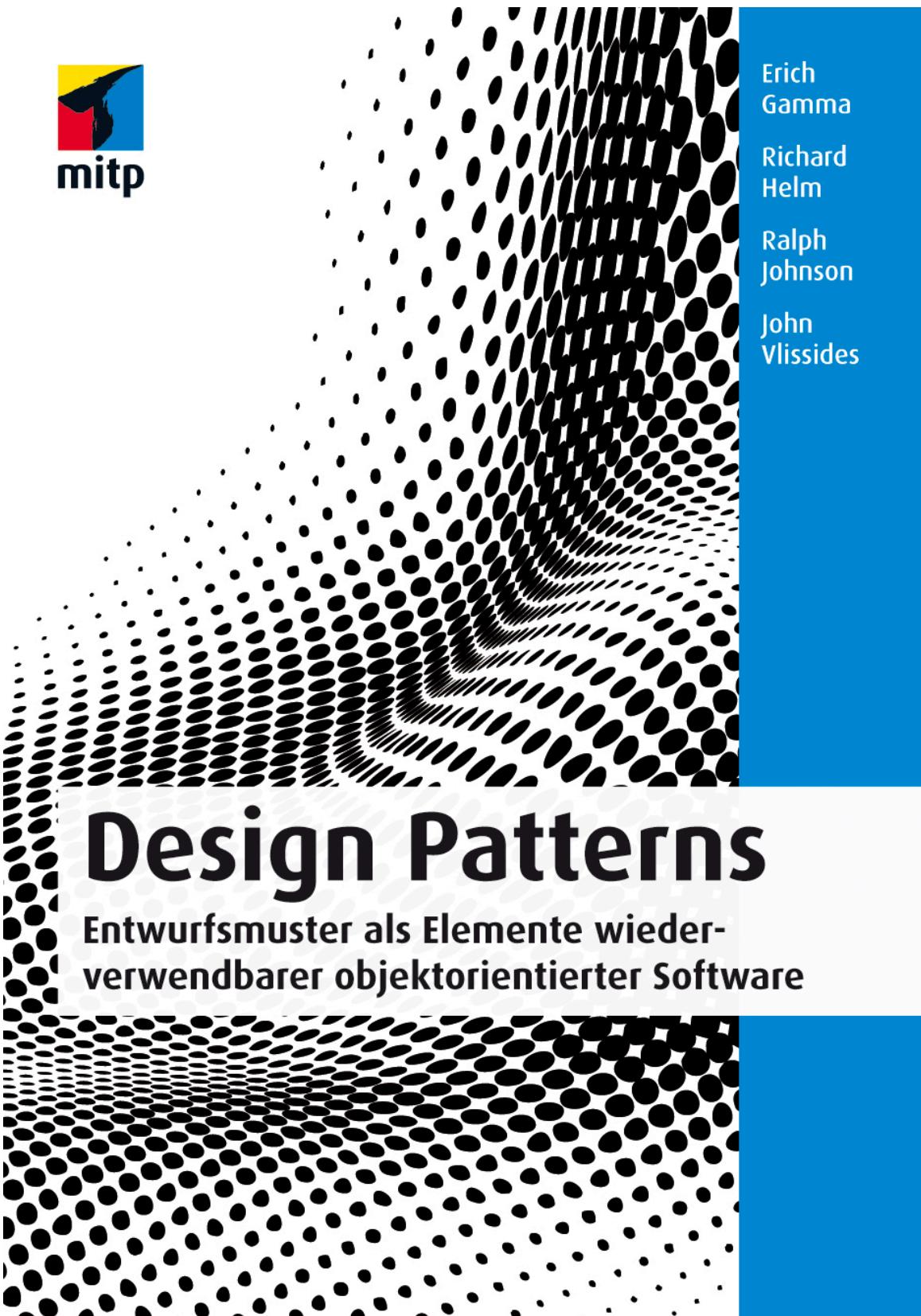
(Behavioral Patterns)

Patrick Creutzburg, 09. Dezember 2020



Quelle: <https://refactoring.guru/>

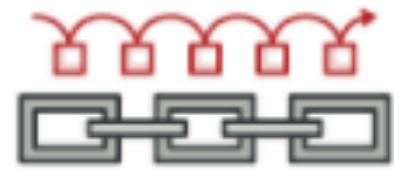
Resources



<https://www.mitp.de/IT-WEB/Software-Entwicklung/Design-Patterns.html>



<https://refactoring.guru/>



Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



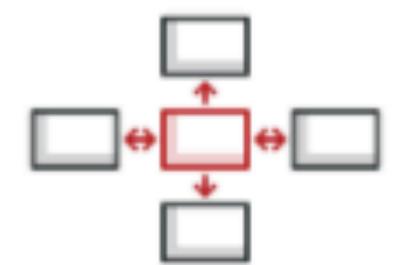
Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.



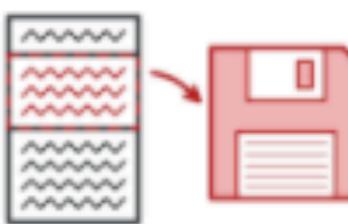
Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



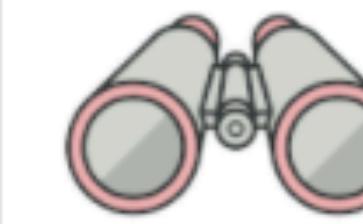
Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



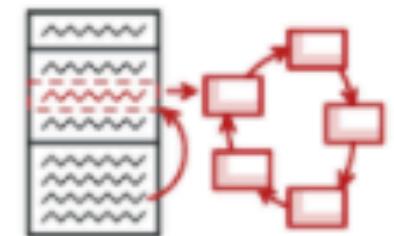
Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.



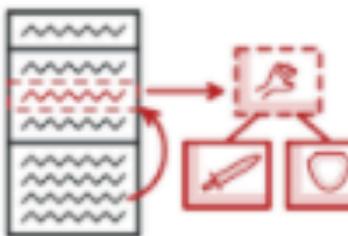
Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



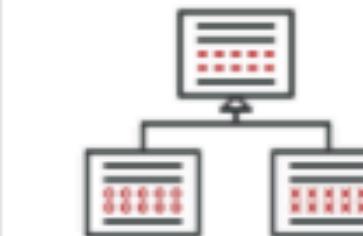
State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



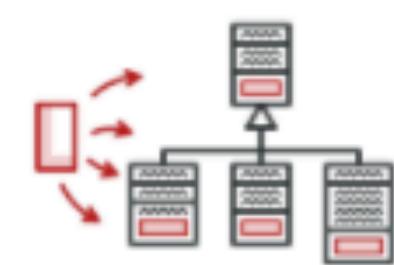
Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.



Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



Visitor

Lets you separate algorithms from the objects on which they operate.

Bedeutung der Entwurfsmuster

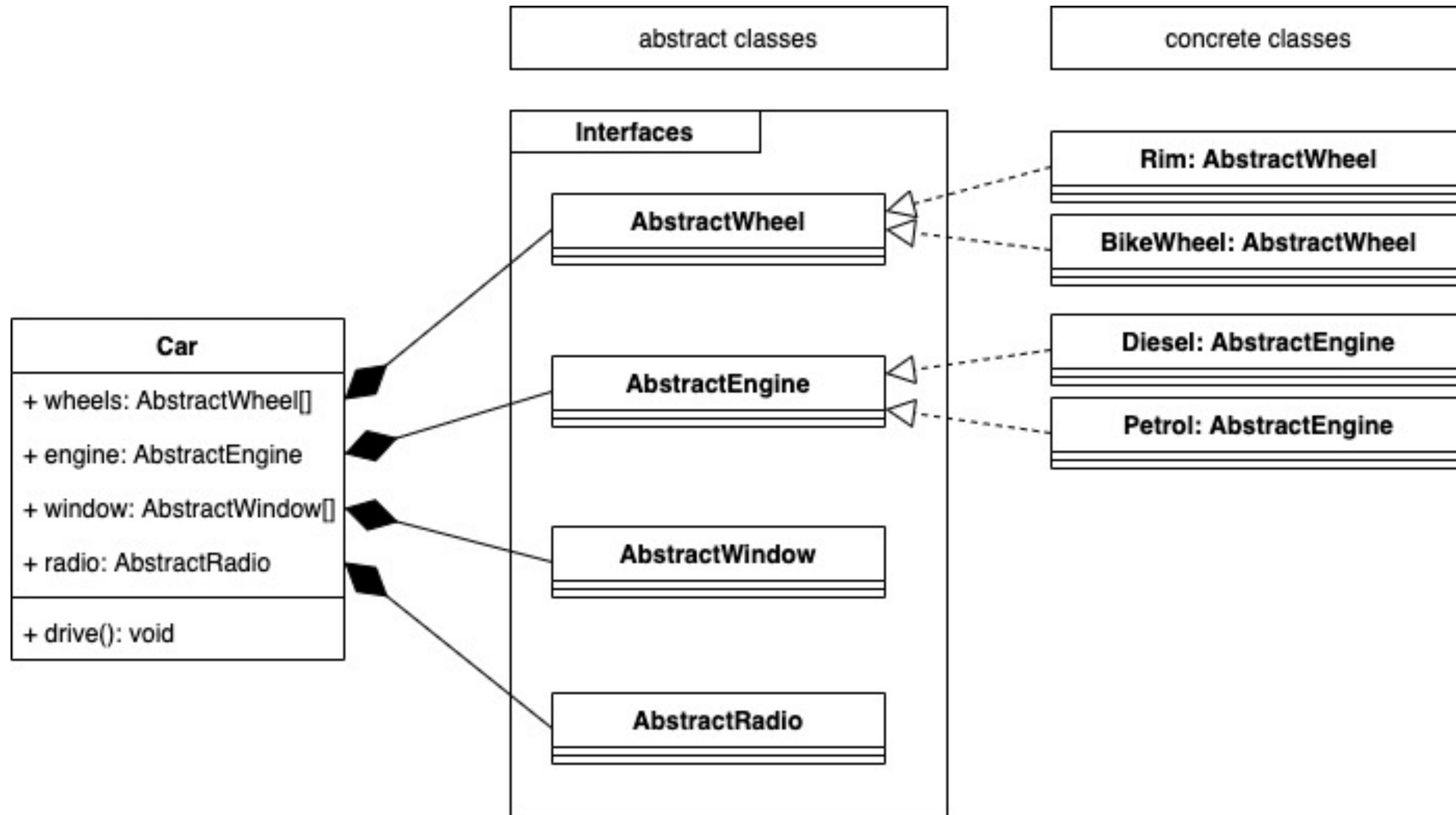
- beschreibt eine “generische” Lösung für ein Entwurfsproblem - wiederverwendbare Vorlage zur Problemlösung
- **Entwickler sollen einheitliche Sprache über Systemdesign sprechen**
- Garantie, dass das System unabhängig von Generierung, Komposition und Darstellung seiner Objekte funktioniert
- Verwendung von Objektkomposition und Dependency Inversion
- Implementierung gegen Schnittstellen (Interfaces od. abstrakte Klassen)
- Bewahren des Open-Closed-Prinzips
- flexibles Design, geringer Wartungsaufwand, hohe Wiederverwendbarkeit

Bedeutung der Verhaltensmuster

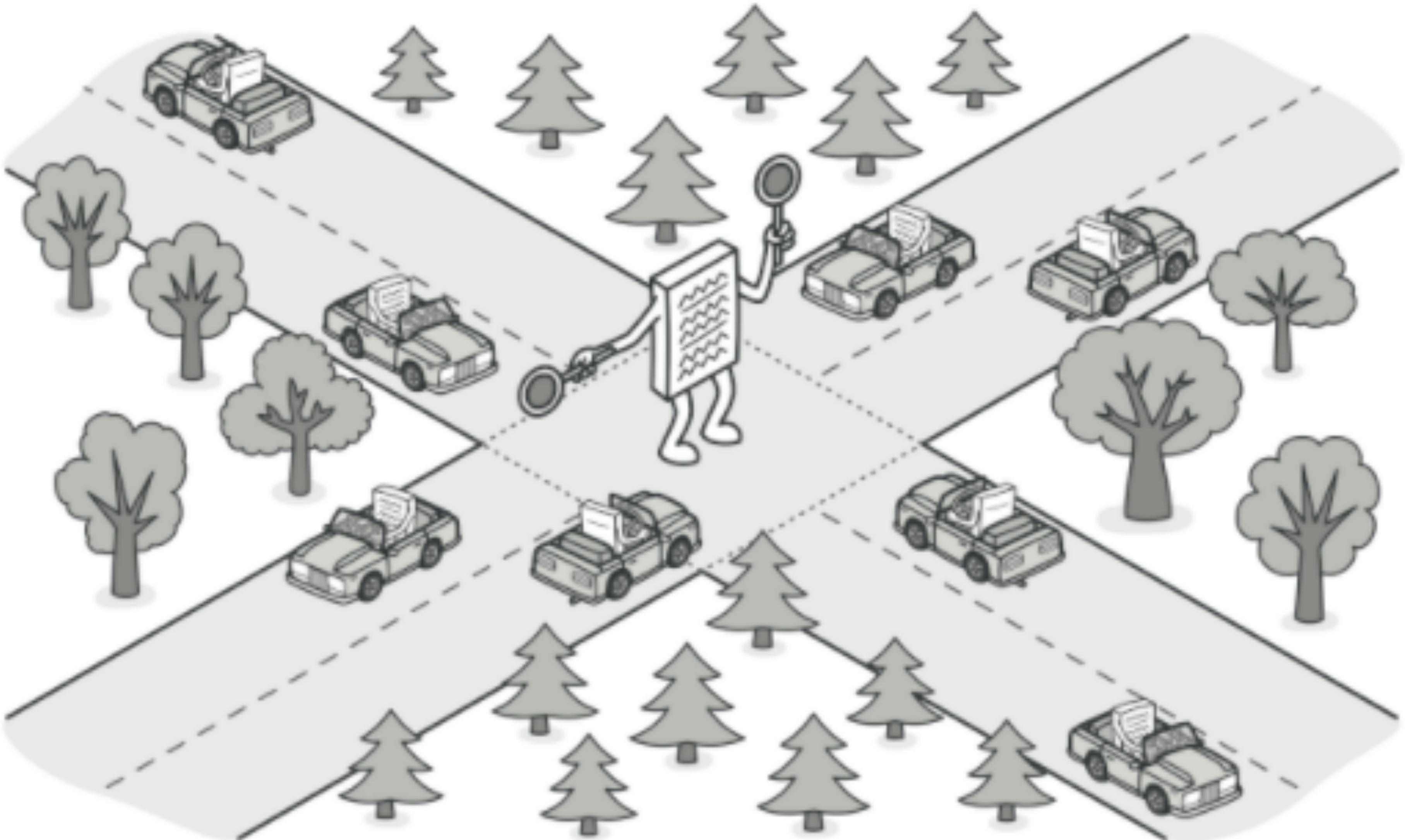
- beschäftigen sich mit Algorithmen und Zuweisung von Zuständigkeiten an Objekte
- beschreiben wechselseitige Kommunikationsmuster
- Erfassung komplexer Programmabläufe

„Software entities ... should be open for extension, but closed for modification.“

Object Composition & Dependency Inversion

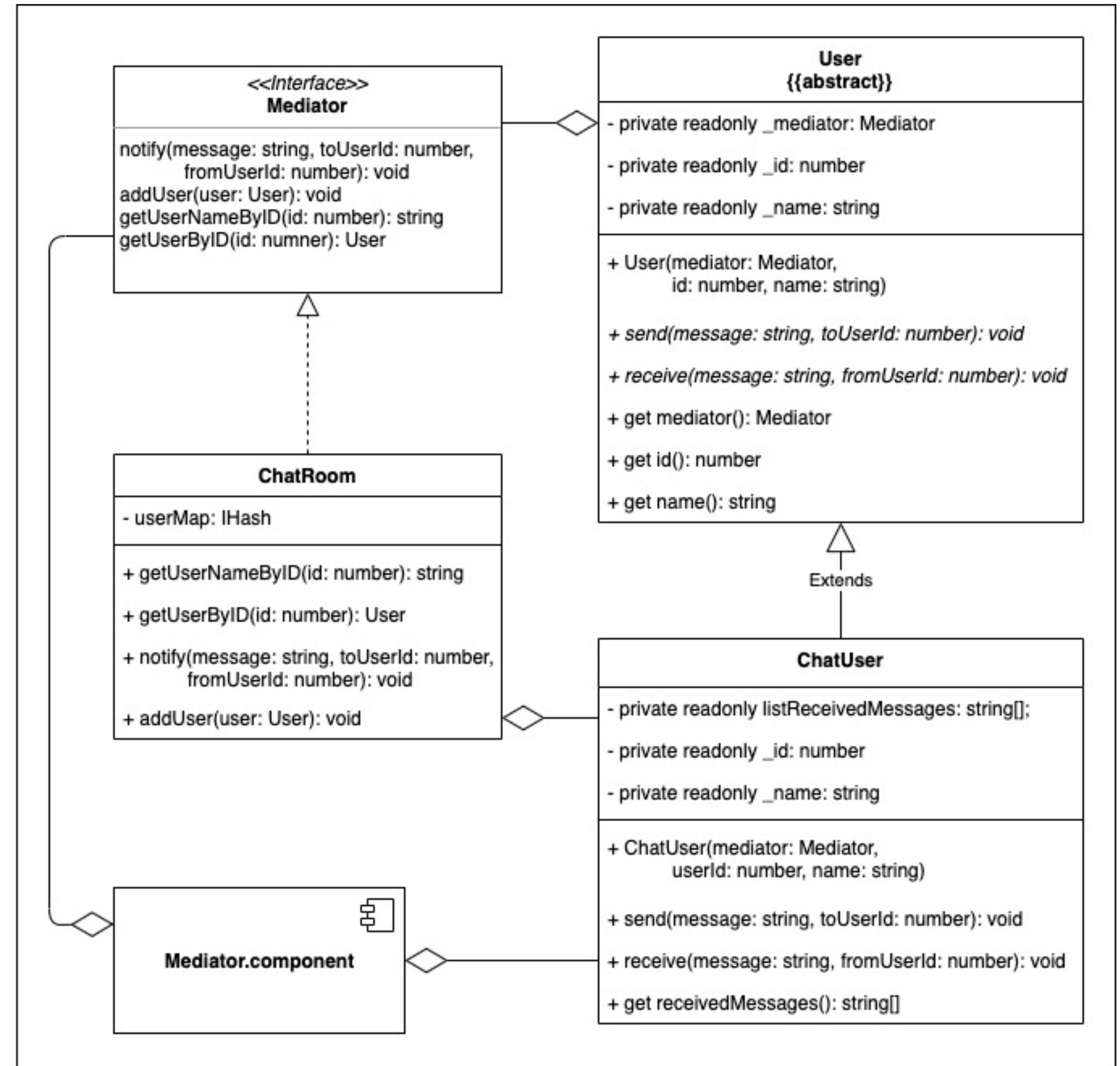


Mediator Vermittler



Quelle: <https://refactoring.guru/design-patterns/mediator>

Demo



Zweck

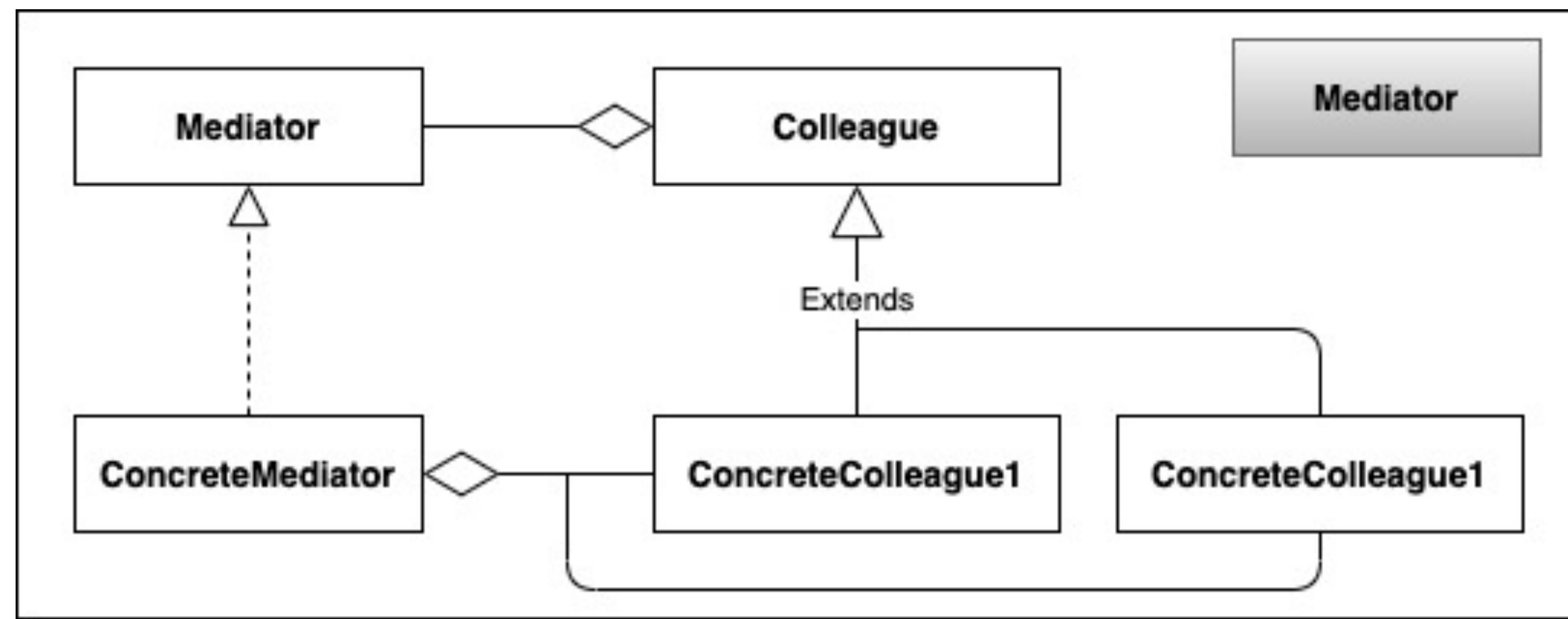
- Definition eines Objektes, welches die Interaktionsweise von Objektgruppen steuert
- Mediator begünstigt lose Kopplung, indem es explizite Referenzierung der Teilnehmer unterbindet und individuelle Steuerung ihrer Interaktionen ermöglicht

Anwendbarkeit (geeignet, wenn...)

- wenn Kommunikation innerhalb einer Objektgruppe wohldefiniert, aber auch auf komplexe Weise erfolgt (Abhängigkeiten sind unstrukturiert und schwer zu verstehen)
- Wiederverwendbarkeit von Objekten ist schwierig, weil es zahlreiche andere Objekte referenziert und mit diesen kommuniziert
- Anpassbarkeit über mehrere Klassen verteiltes Verhalten soll gewährleistet werden
- Bsp. VerneMQ (MQTT Broker)

Konsequenzen (Vor- und Nachteile)

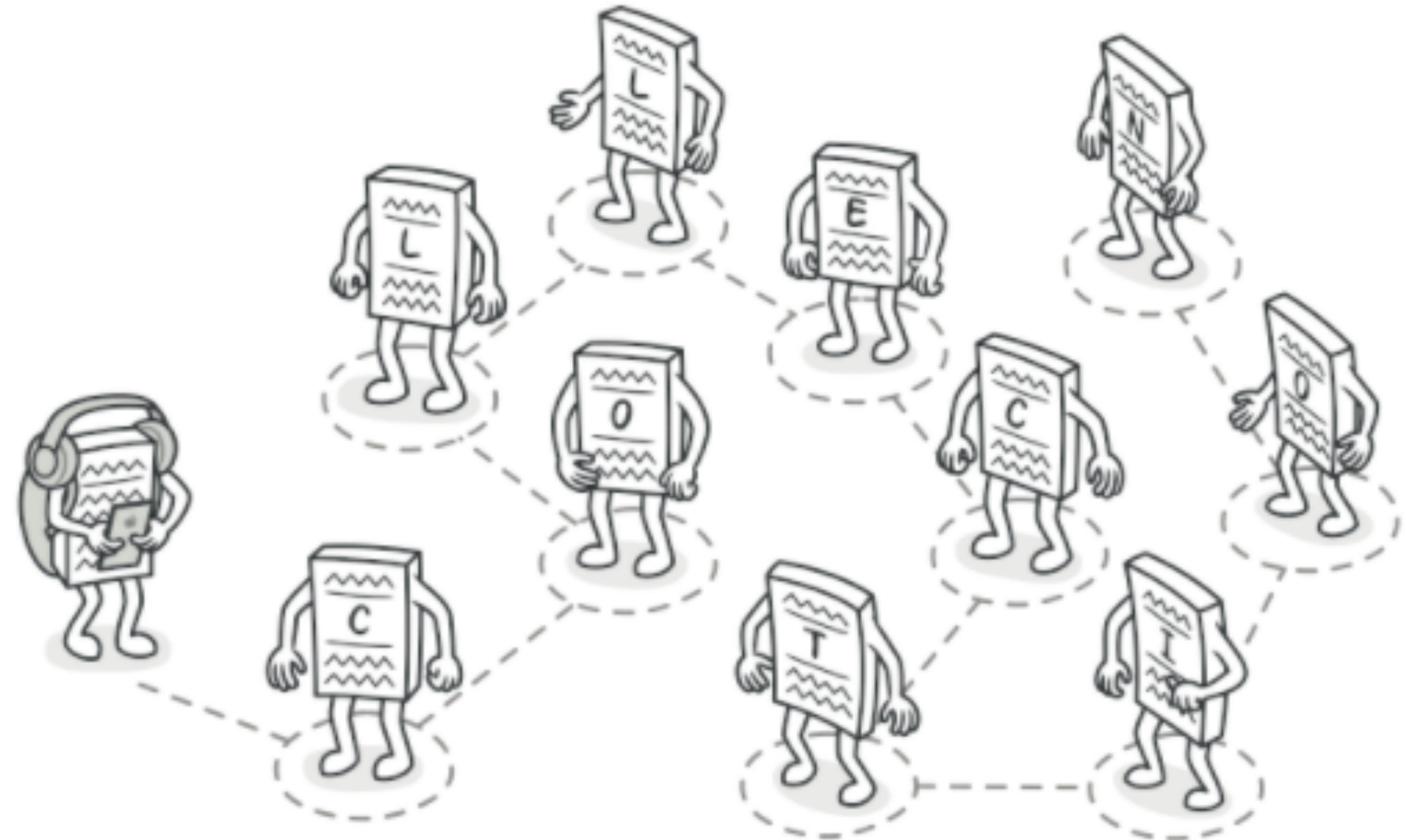
- Eingeschränkte Unterklassenbildung des Mediators für Verhaltensänderung
- Entkopplung von Teilnehmerobjekten - dadurch können sie unabhängig voneinander variiert und wiederverwendet werden
- Vereinfachung der Objektkontrolle (n:n wird zu 1:n) - einfache Objektverwaltung (Verständnis, Erweiterung, etc)
- Abstrahierung der Objektkooperation
- Zentralisierte Steuerung - Komplexität der Interaktionen wird gegen Komplexität des Mediators ausgetauscht (monolithischer Ansatz)



Code-Beispiel

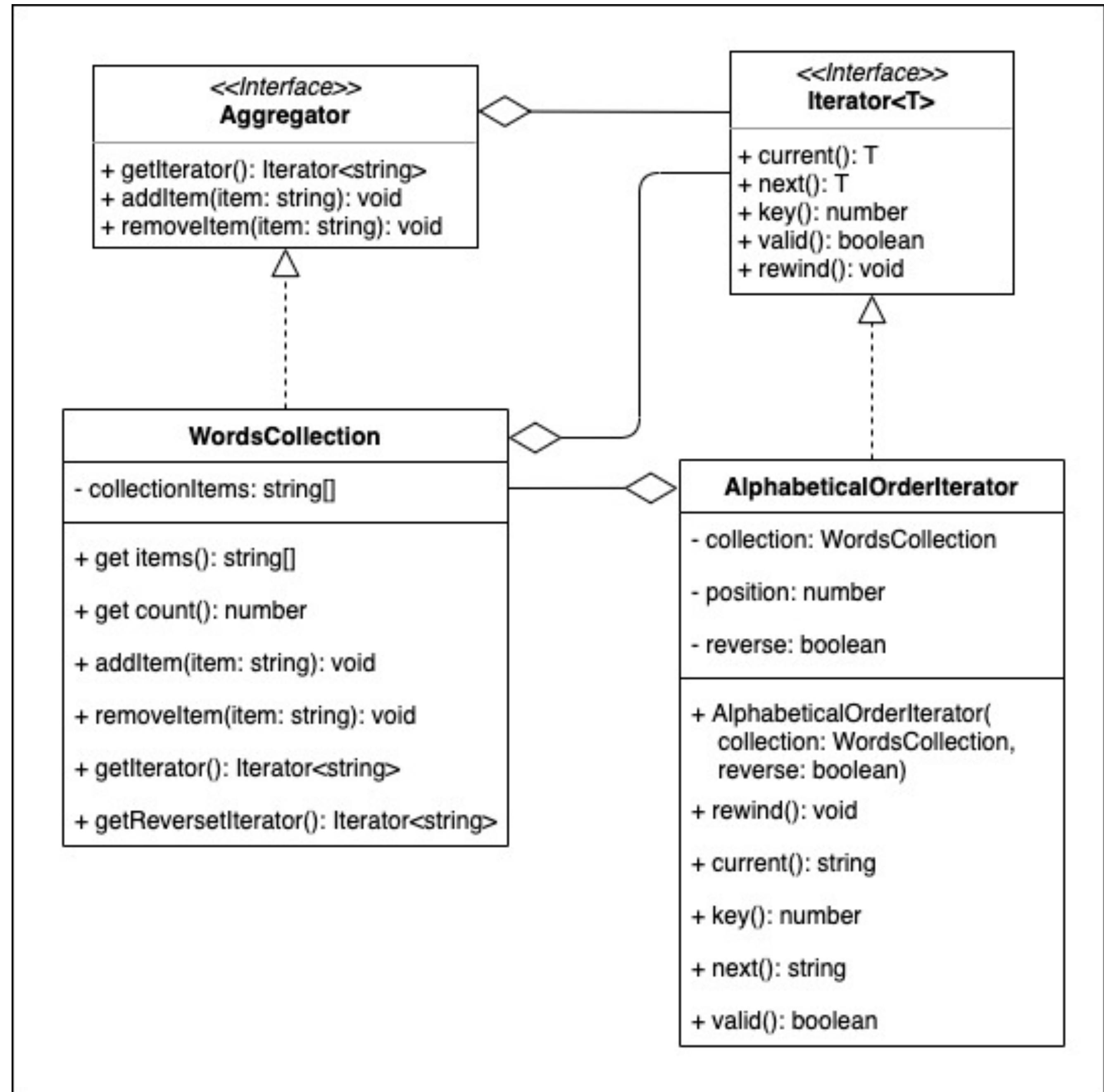
Iterator

Iterator



Quelle: <https://refactoring.guru/design-patterns/iterator>

Demo



Zweck

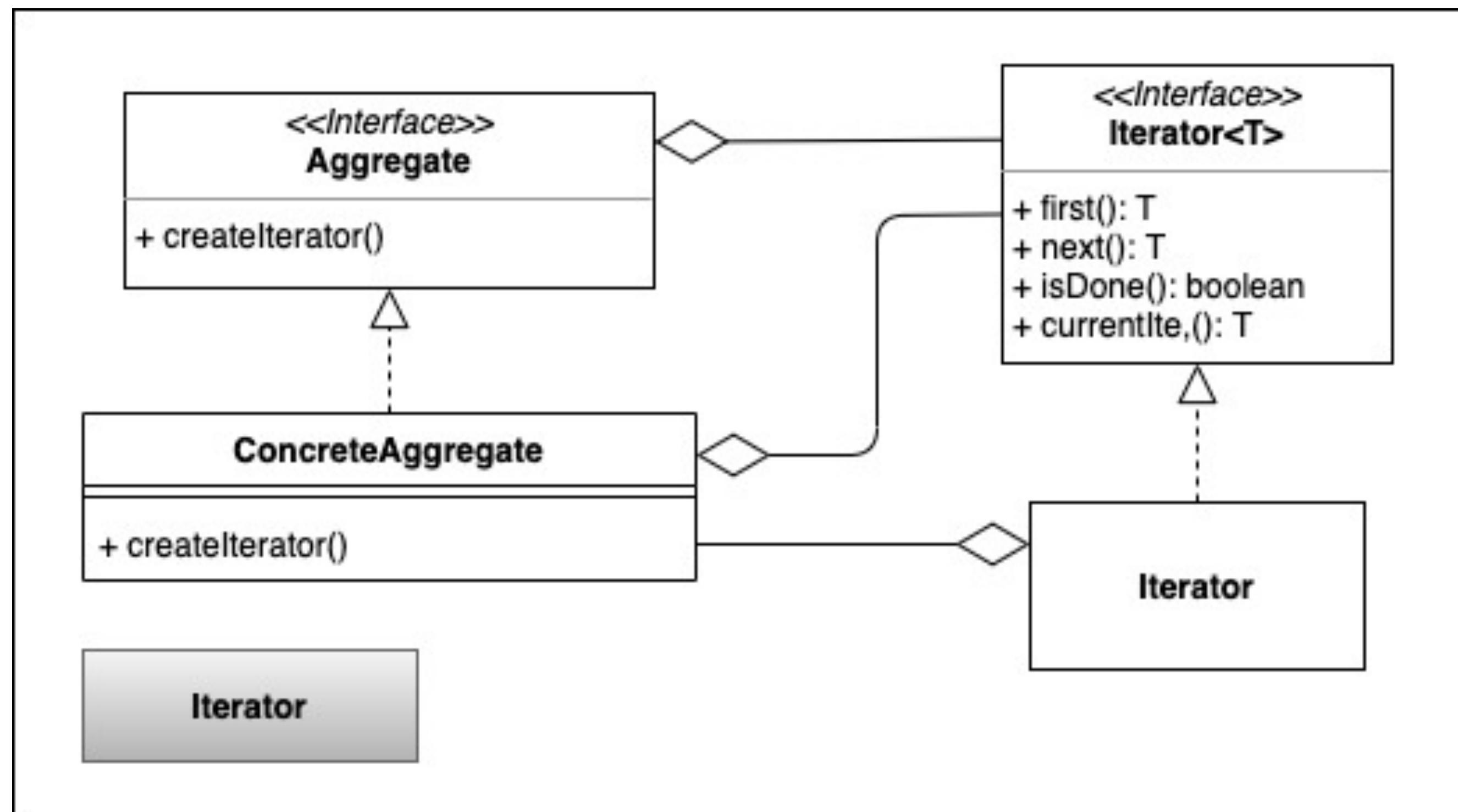
- Bereitstellung eines sequenziellen Zugriffs auf Elemente eines Aggregierten Objekts, ohne dessen zugrunde liegende Struktur offenzulegen
- (Bsp.: auf Arrays kann nicht mehr via getter zugegriffen werden)

Anwendbarkeit (geeignet, wenn...)

- Zugriff auf Inhalte eines Aggregate-Objektes, ohne dessen interne Darstellung preiszugeben, gewährleistet werden soll
- Unterstützung mehrerer zeitgleicher Traversierenden von aggregierten Objekten
- Bereitstellung einer einheitlichen für die Traversierung unterschiedlicher zusammengesetzter Strukturen (z.b. polymorphe Iteration)

Konsequenzen (Vor- und Nachteile)

- Unterstützung verschiedener Traversierungsvarianten (Wechsel von Traversierungsalgorithmen) z.b. durch Generics
- Vereinfachung der Aggregatschnittstelle
- Parallel Ausführung mehrerer Traversierungsvorgänge (Iterator überwacht seinen Traversierungszustand selbst und somit können mehrere gleichzeitig durchgeführt werden)



Code-Beispiel

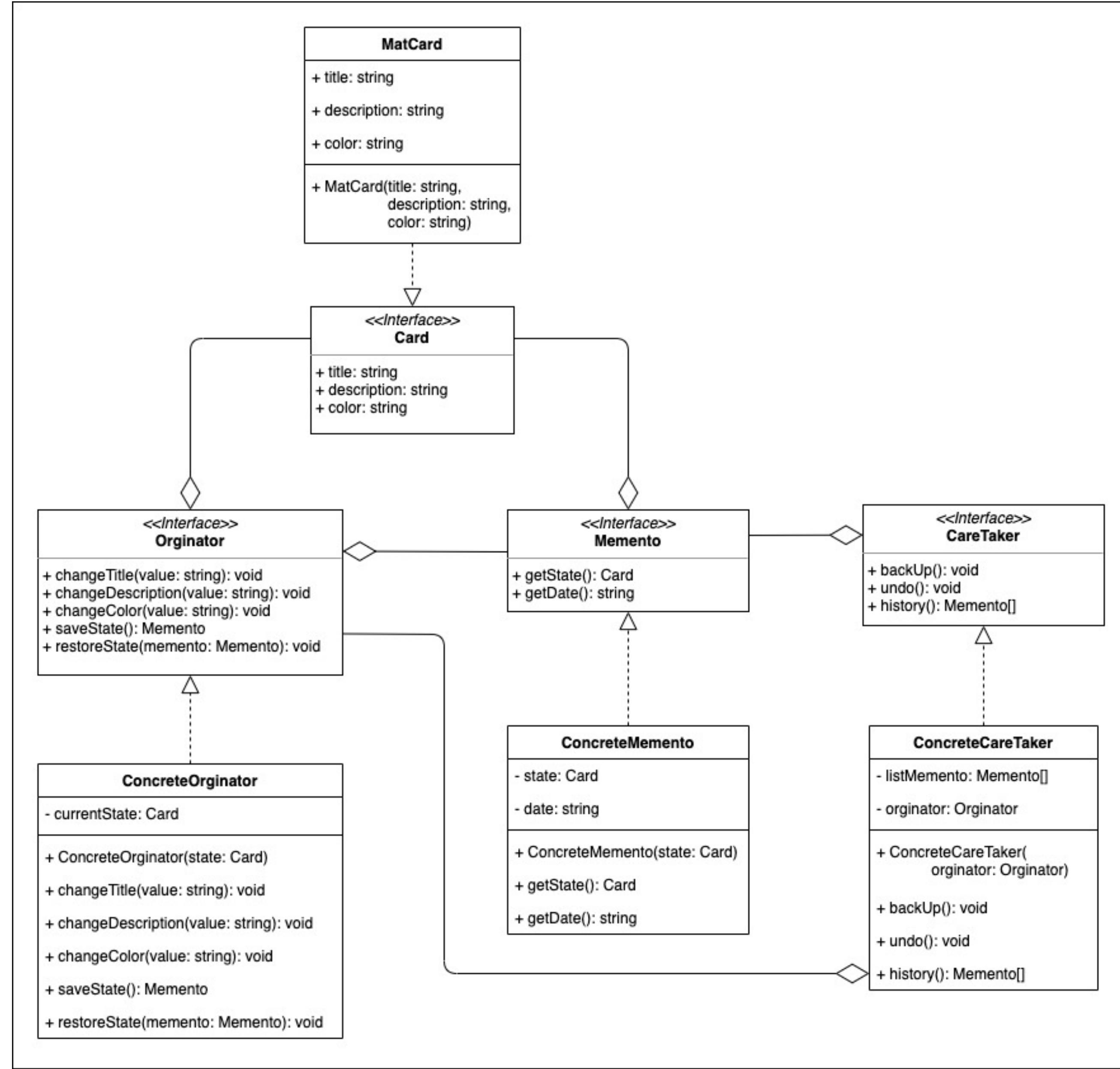
Memento

Memento



Quelle: <https://refactoring.guru/design-patterns/memento>

Demo



Zweck

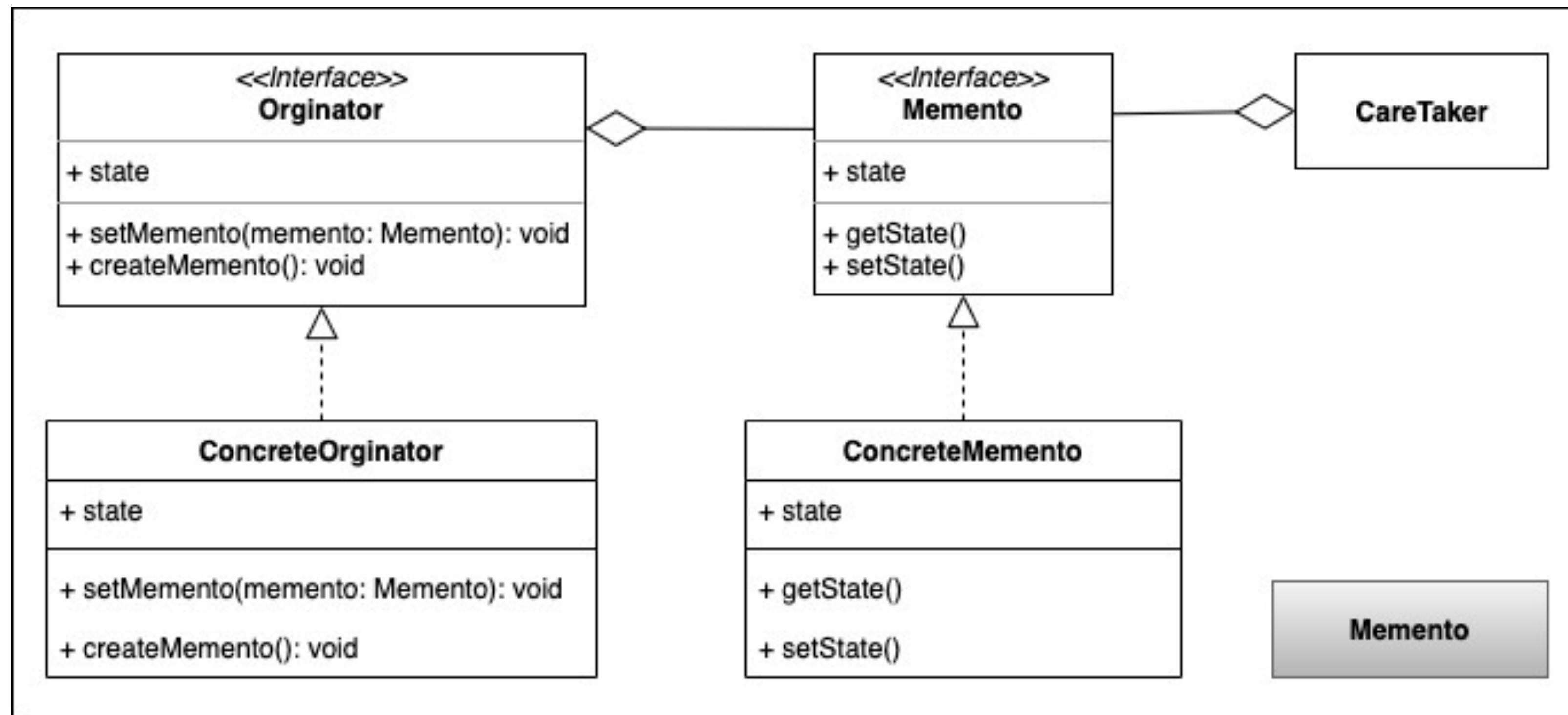
- Erfassung und Externalisieren des internen Zustands eines Objektes, ohne dessen Kapseln zu beeinträchtigen
- kann später wieder in anderen Zustand zurück versetzt werden

Anwendbarkeit (geeignet, wenn...)

- der Zustand eines Objektes (oder ein Teil davon) gespeichert werden soll, damit es später wieder hergestellt werden kann
- eine direkte Schnittstelle zum Abrufen des Zustands Implementierungsdetails enthüllen und die Kapseln des Objektes beeinträchtigen würde

Konsequenzen (Vor- und Nachteile)

- Beibehaltung der Kapselungsgrenzen - vermeidet die Freigabe von Informationen nach außen
- Vereinfachung des Urhebers - Last der Speicherverwaltung bei Urheber und somit Informationskontrolle
- Verwendung von Mementos kann aufwendig sein (beim Kopieren großer Mengen von Informationen)
- Versteckter Aufwand bei der Verwaltung von Mementos beim CareTaker (z.B. große Speicherverwaltungslast beim Speichern von Mementos)



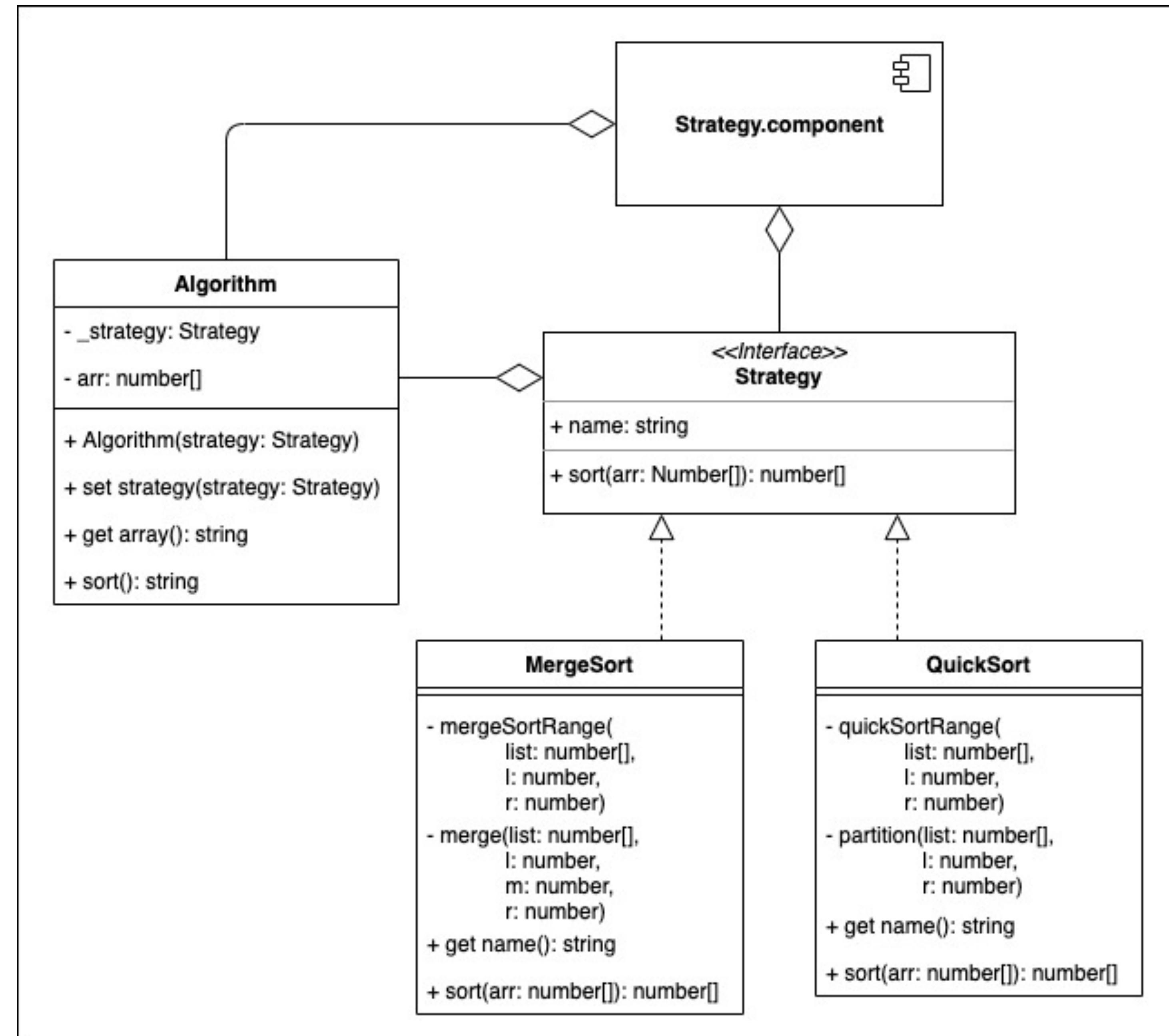
Code-Beispiel

Strategy Strategie



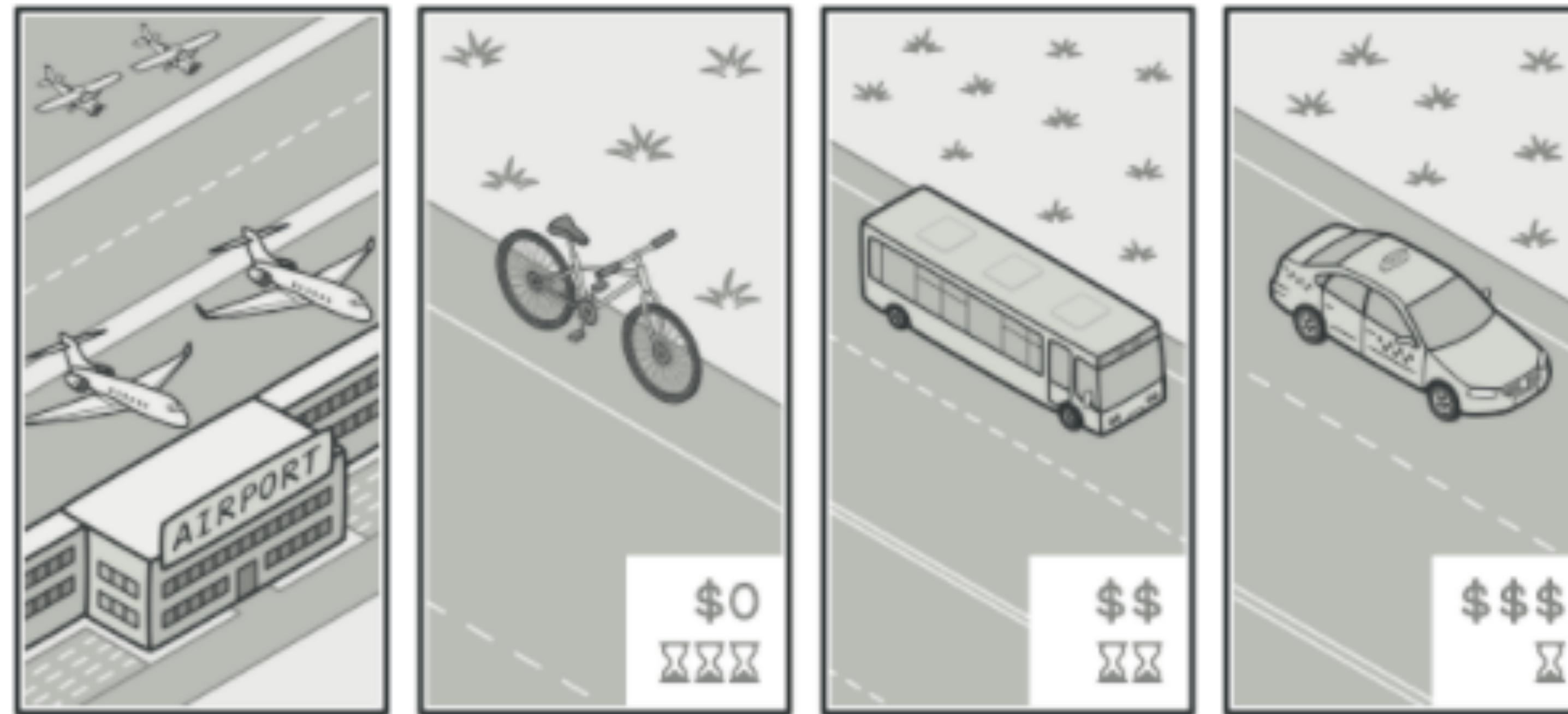
Quelle: <https://refactoring.guru/design-patterns/strategy>

Demo



Zweck

- Definition einer Familie von einzeln gekapselten, austauschbaren Algorithmen
- ermöglicht variable und von Client unabhängige Nutzung eines Algorithmus



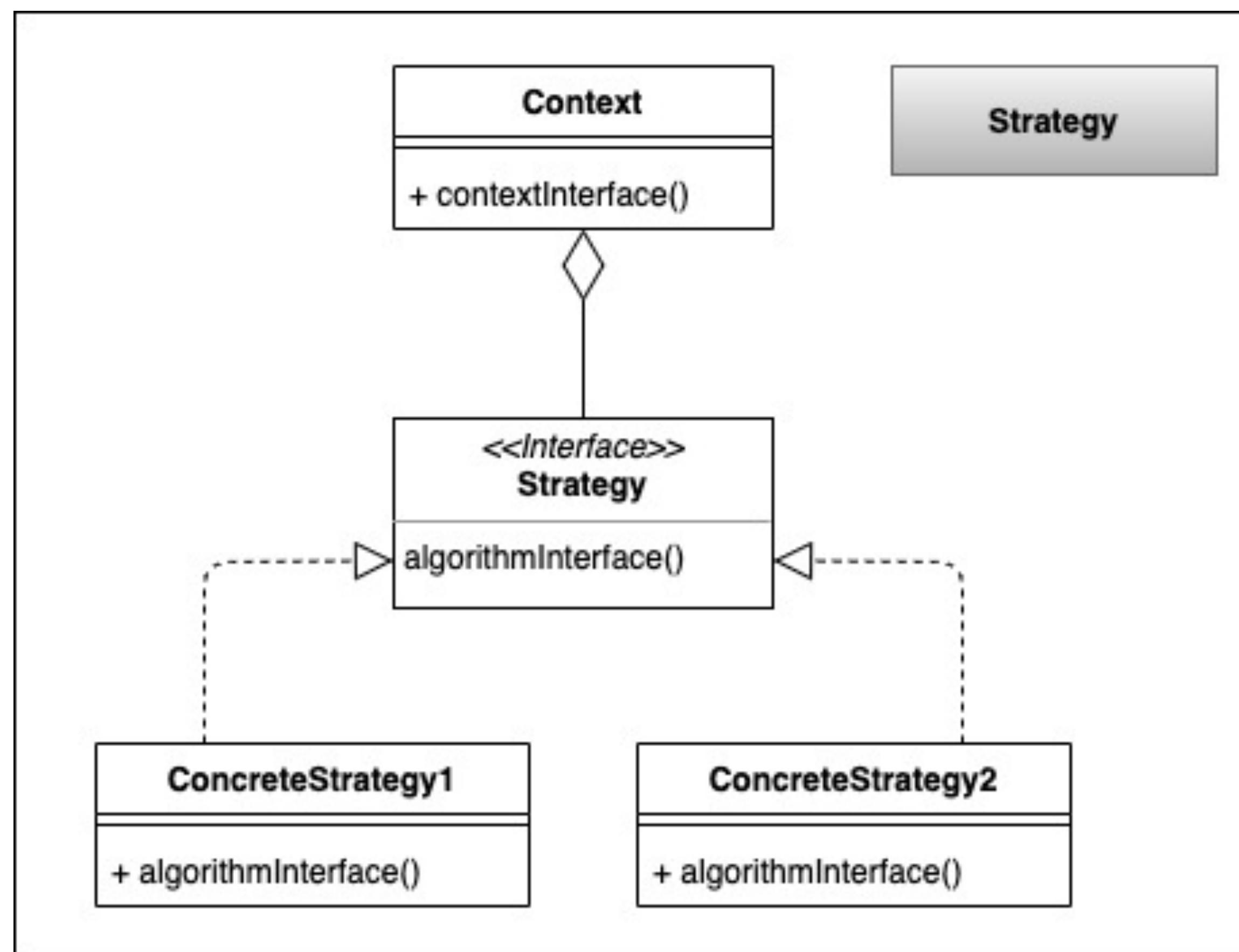
Quelle: <https://refactoring.guru/images/patterns/content/strategy/strategy-comic-1-en.png>

Anwendbarkeit (geeignet, wenn...)

- stellt Möglichkeit zur Verfügung, eine Klasse mit einer von vielen verschiedenen Verhaltensweisen auszustatten
- verschiedene Varianten eines Algorithmus notwendig sind (z.B. unterschiedliche Kompromisse zwischen Speicherbedarf und Rechenzeit)
- Implementation verschiedener Varianten einer Klassenhierarchie von Algorithmen
- wenn Algorithmus Daten verwendet, von dem der Client keine Kenntnis hat
- bedingte Anweisungen in eine eigene Klasse verpackt werden soll

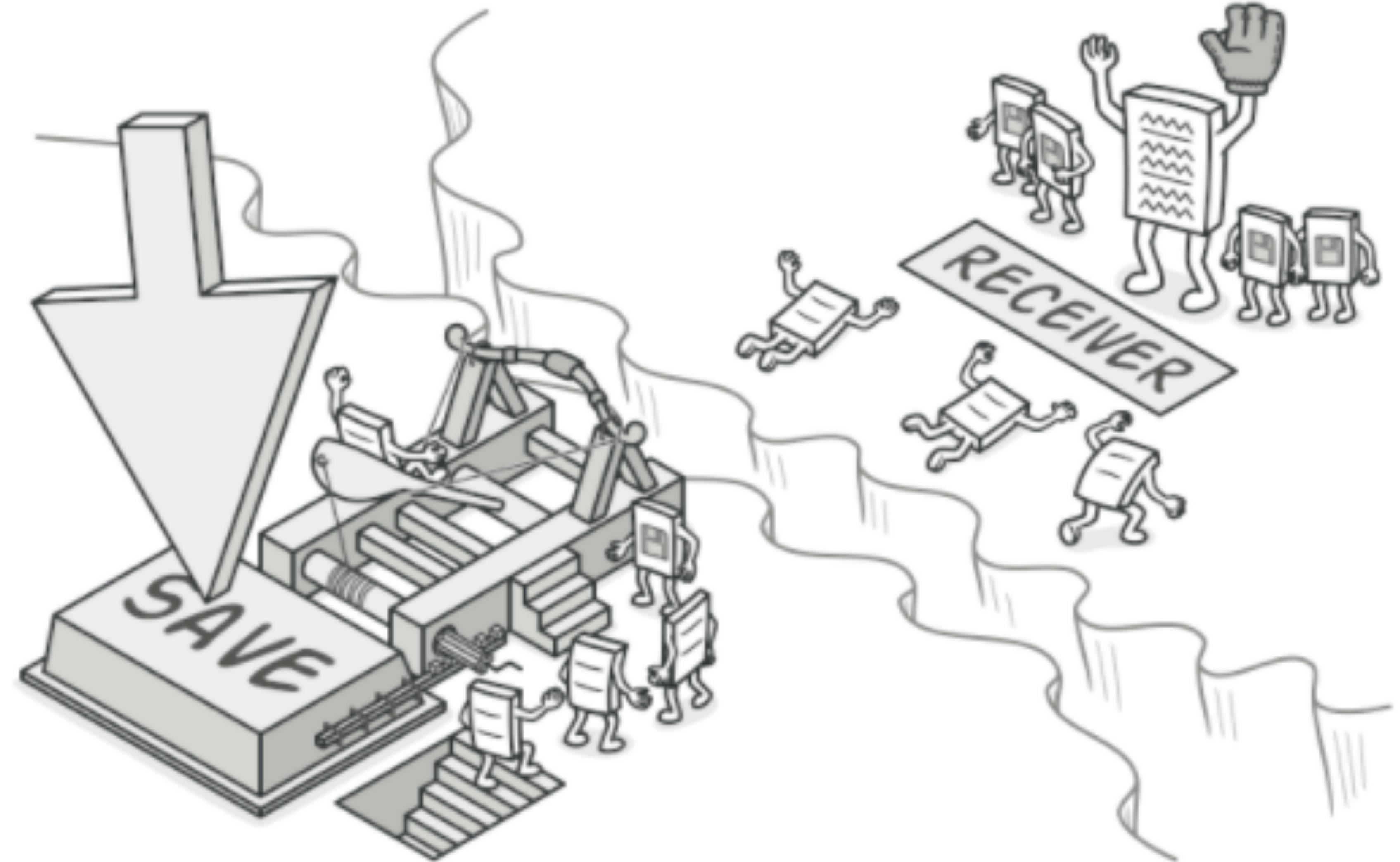
Konsequenzen (Vor- und Nachteile)

- Entstehung einer Familie verwandter Algorithmen (oder Verhaltensweisen)
- Alternative zur Erstellung von Unterklassen (durch Komposition wird der Algorithmus leicht veränder-, austausch- und erweiterbar)
- beseitigt das Erfordernis bedingter Anweisungen durch Polymorphismus (z.B. beim Refactoring von Switch Statements)
- verschiedene Auswahl von Implementierungen desselben Verhaltens (Kompromiss zwischen Speicherbedarf und Rechenzeit)
- Client muss wissen, wie sich die Strategien voneinander unterscheiden (bedingte Enthüllung von Implementierungen)
- Mehraufwand bei der Kommunikation zwischen *Strategy* und *Context*
- erhöhte Anzahl von Objekten



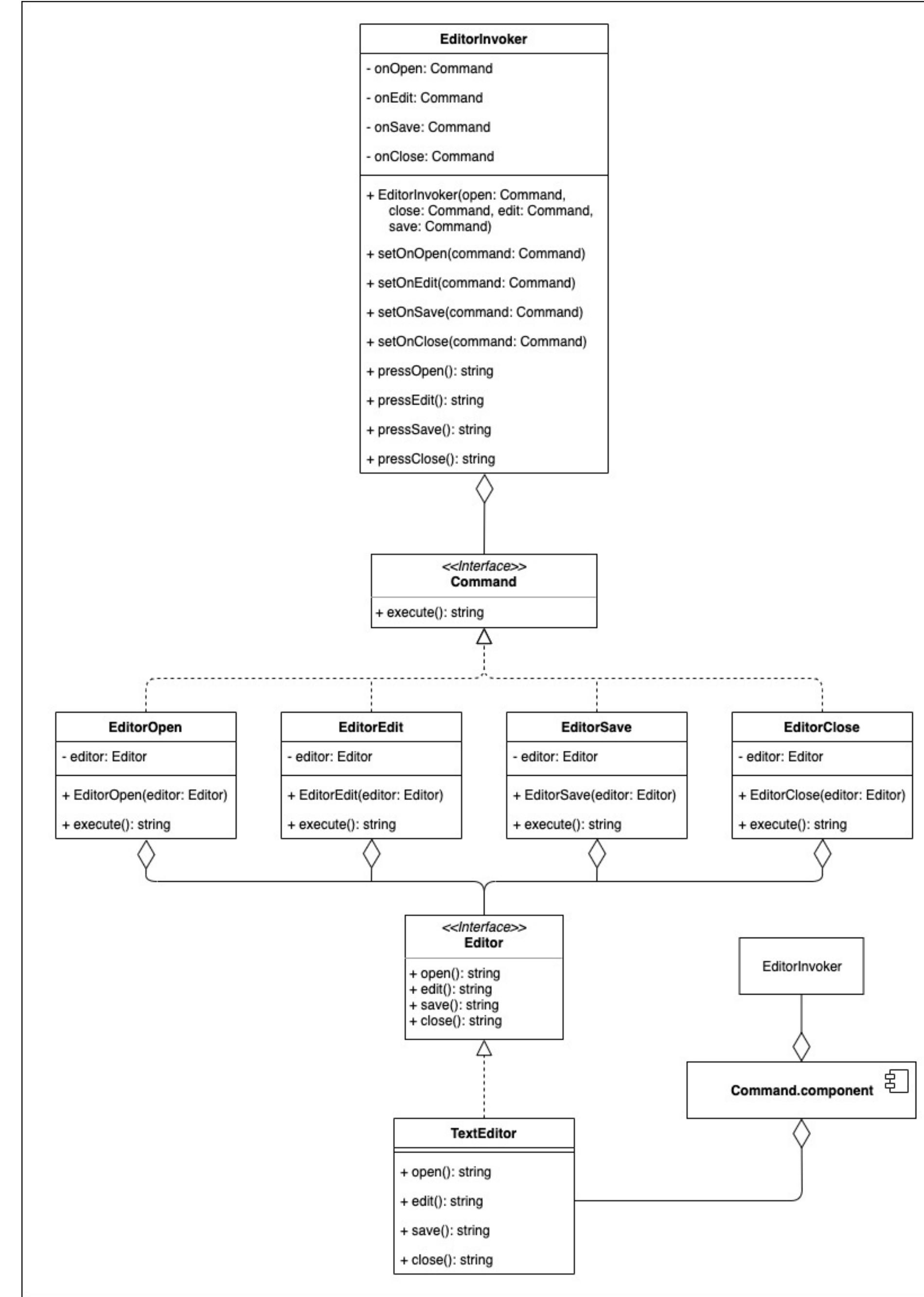
Code-Beispiel

Command Befehl



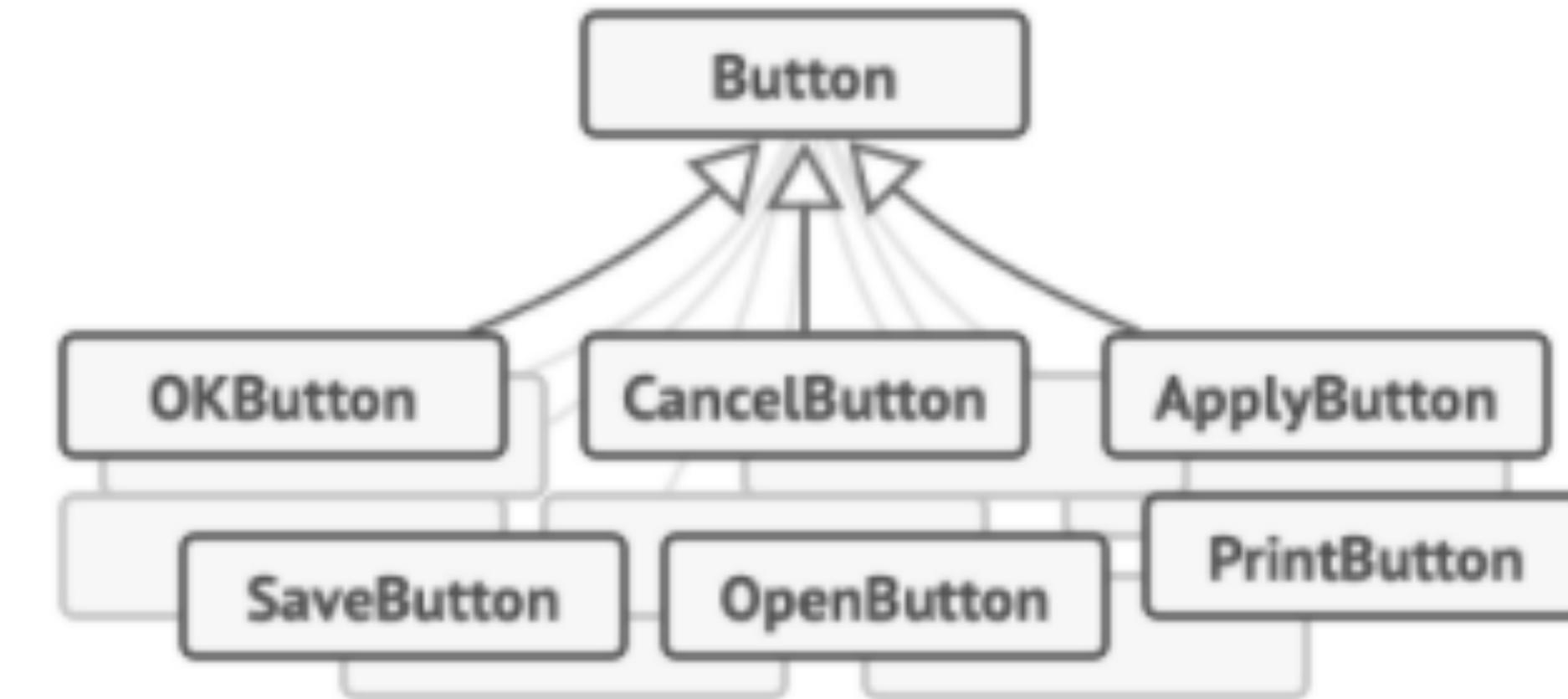
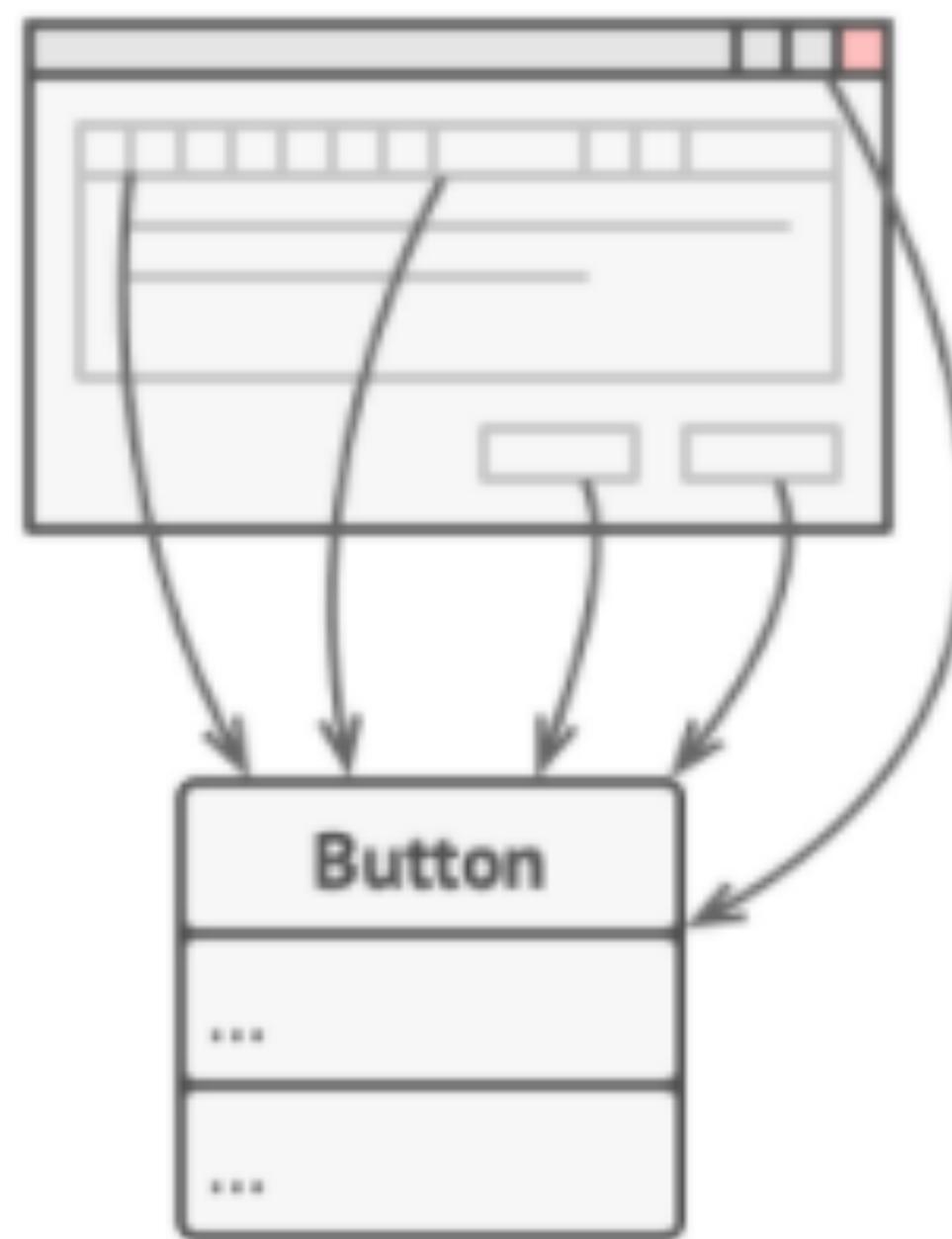
Quelle: <https://refactoring.guru/design-patterns/command>

Demo



Zweck

- Kapselung eines Requests als Objekt, um so Parametrisierung von Clients mit verschiedenen Requests, Warteschlangen- oder Logging-Operationen zu ermöglichen (+ Rückgängigmachen)



Quelle: <https://refactoring.guru/images/patterns/diagrams/command/problem2.png>

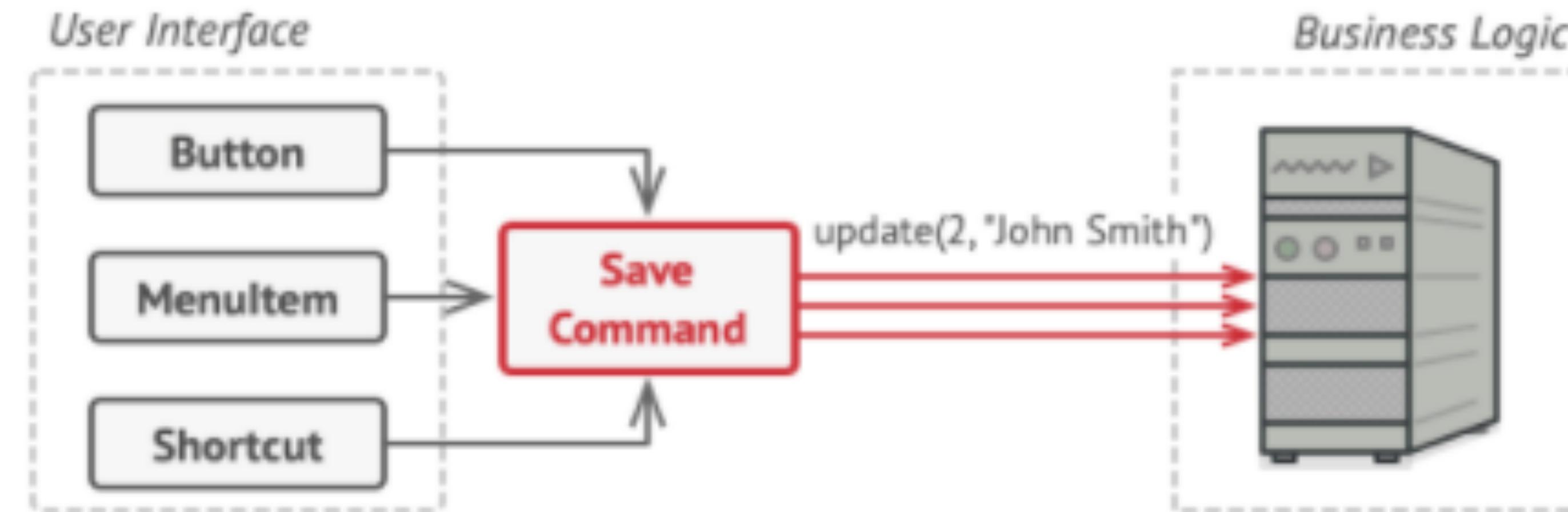
Quelle: <https://refactoring.guru/images/patterns/diagrams/command/problem1.png>

Anwendbarkeit (geeignet, wenn...)

- Objekte mit einer ausführenden Operation parametrisiert werden sollen (z.B. durch Callback in prozeduralen Sprachen)
- Requests spezifiziert, in Warteschlange gestellt und zu versch. Zeitpunkten ausgeführt werden sollen
- UNDO (Rückgängigmachen von Operationen) unterstützt werden soll (Befehlshistorie)
- Protokollierung vorgenommener Änderungen unterstützt werden soll (z.B. Wiederherstellung nach Systemcrash)
- ein System mit komplexen Operationen strukturiert werden soll, die auf primitiven Operationen aufbaut (z.B. Transaktionen)

Konsequenzen (Vor- und Nachteile)

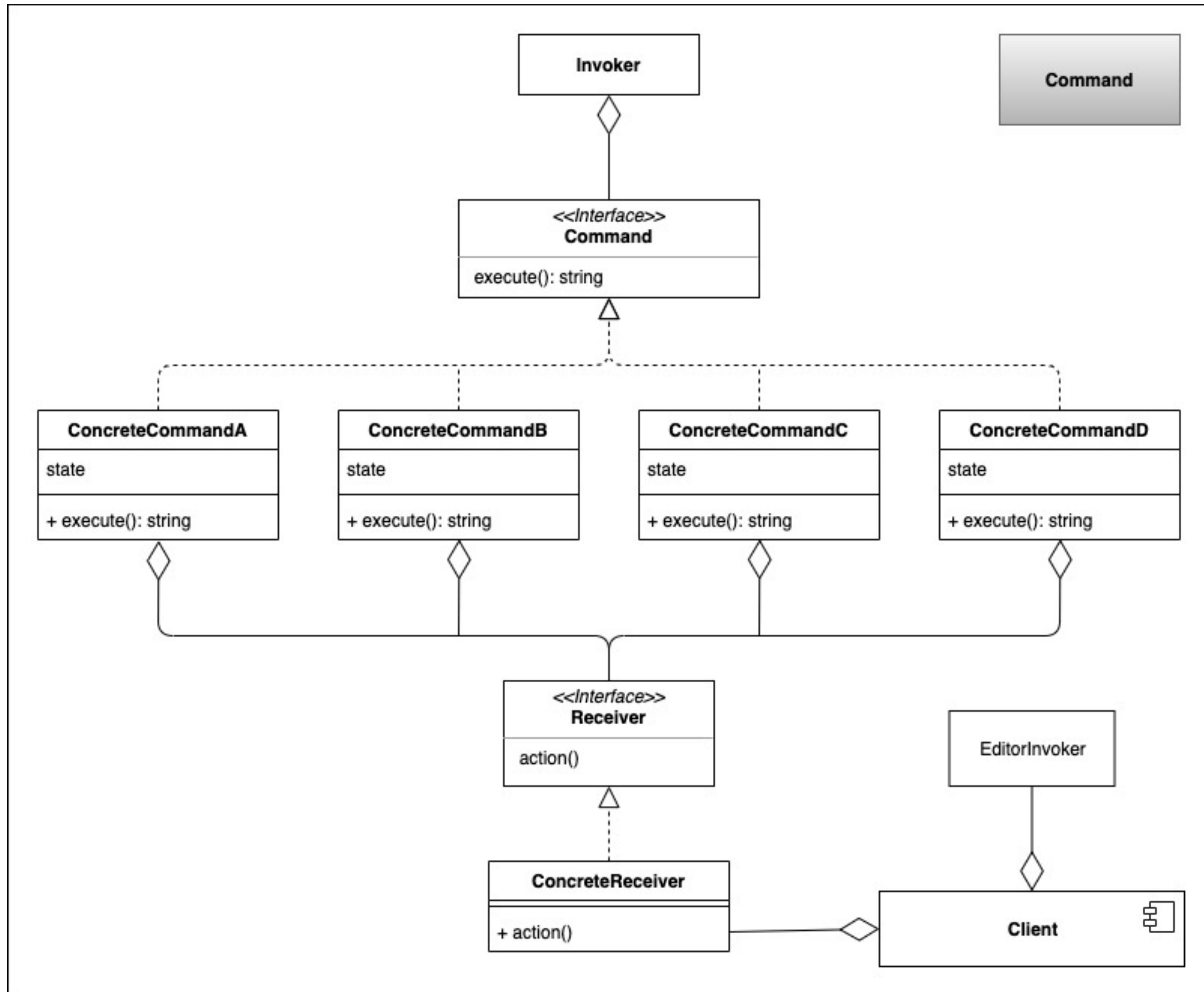
- entkoppelt Objekt, welches die Operation auslöst, von Objekt, welches in der Lage ist, sie auszuführen
- Command-Objekte sind Objekte erster Ordnung, die wie andere Objekte manipuliert und erweitert werden können
- mehrere Befehle können zu einzigen Command-Objekt zusammengefügt werden



🚗 Real-World Analogy

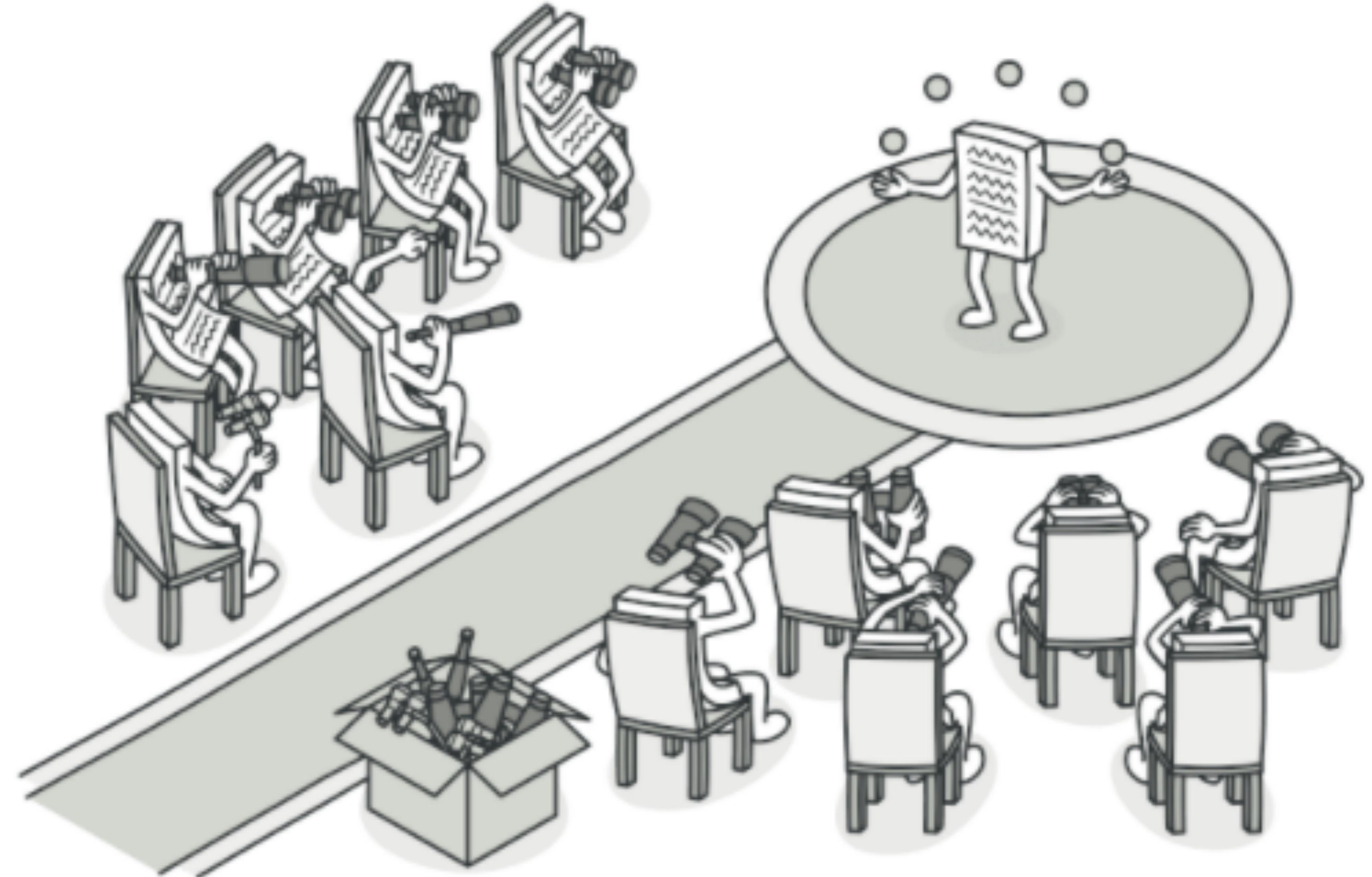


Quelle: <https://refactoring.guru/images/patterns/content/command/command-comic-1.png>



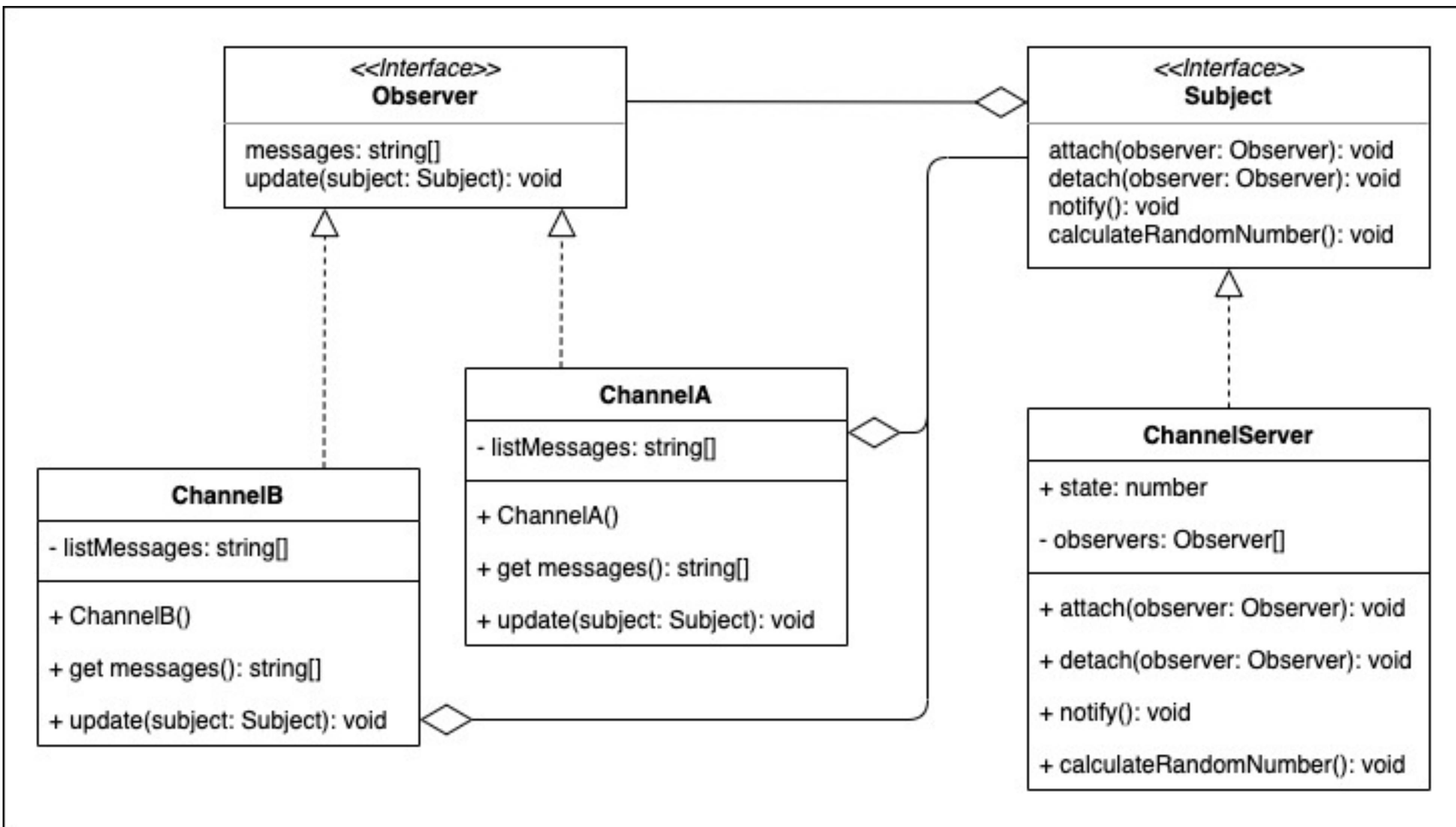
Code-Beispiel

Observer Beobachter



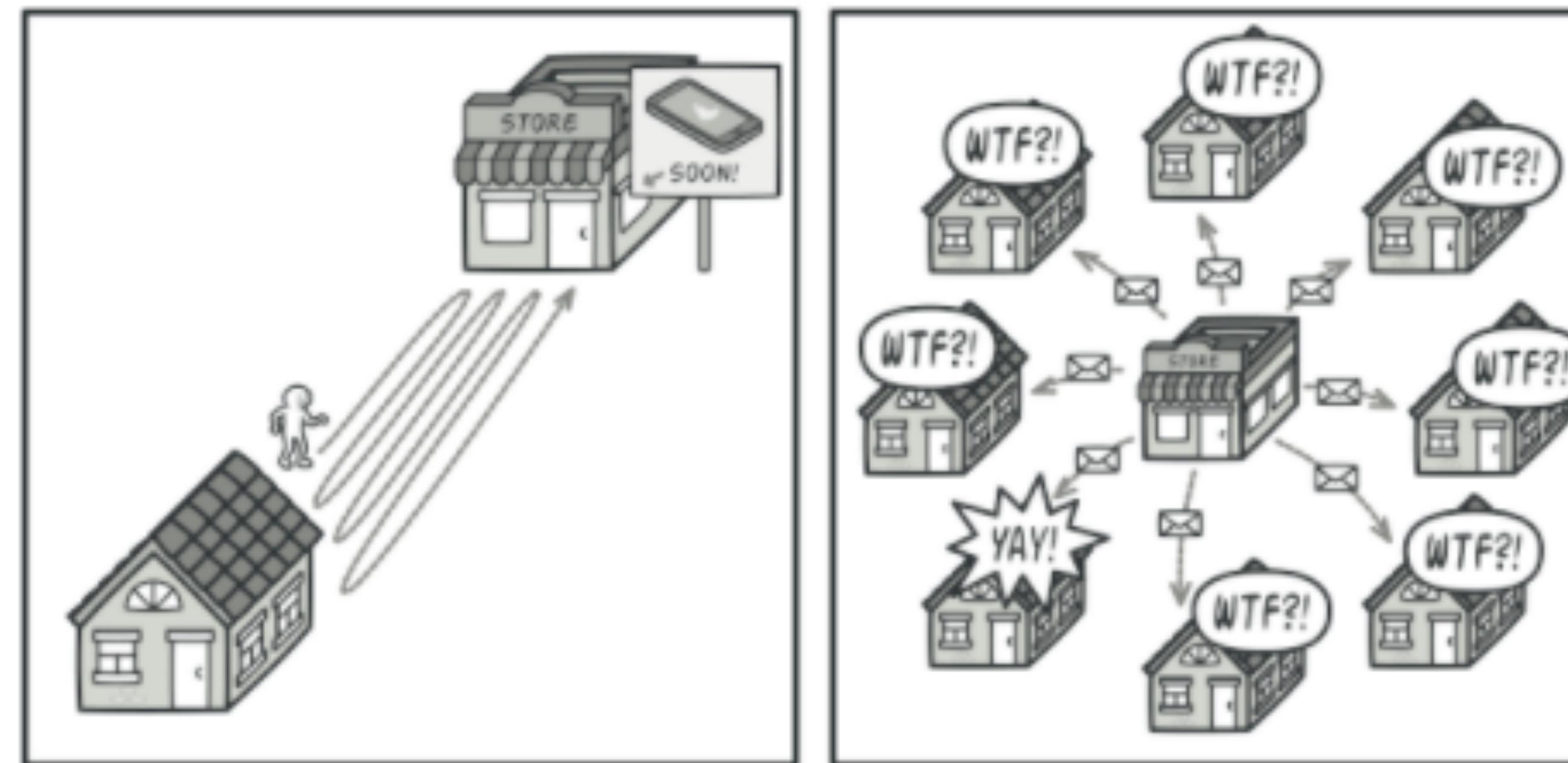
Quelle: <https://refactoring.guru/design-patterns/observer>

Demo



Zweck

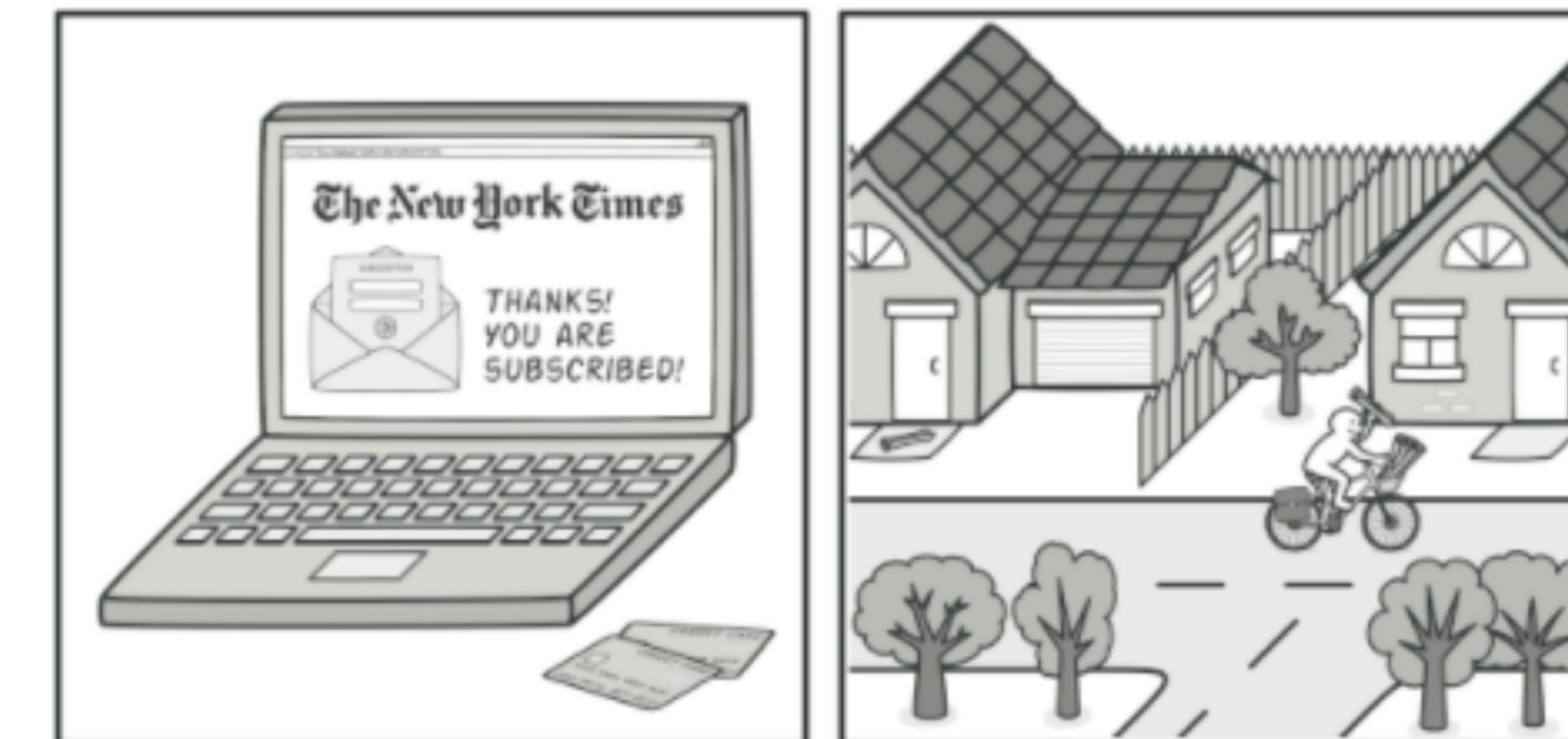
- Definition einer 1:n-Abhangigkeit zwischen Objekten, damit nach Zustandsnderungen eines Objektes alle davon abhangigen Objekte benachrichtigt und automatisch aktualisiert werden



Quelle: <https://refactoring.guru/images/patterns/content/observer/observer-comic-1-en.png>

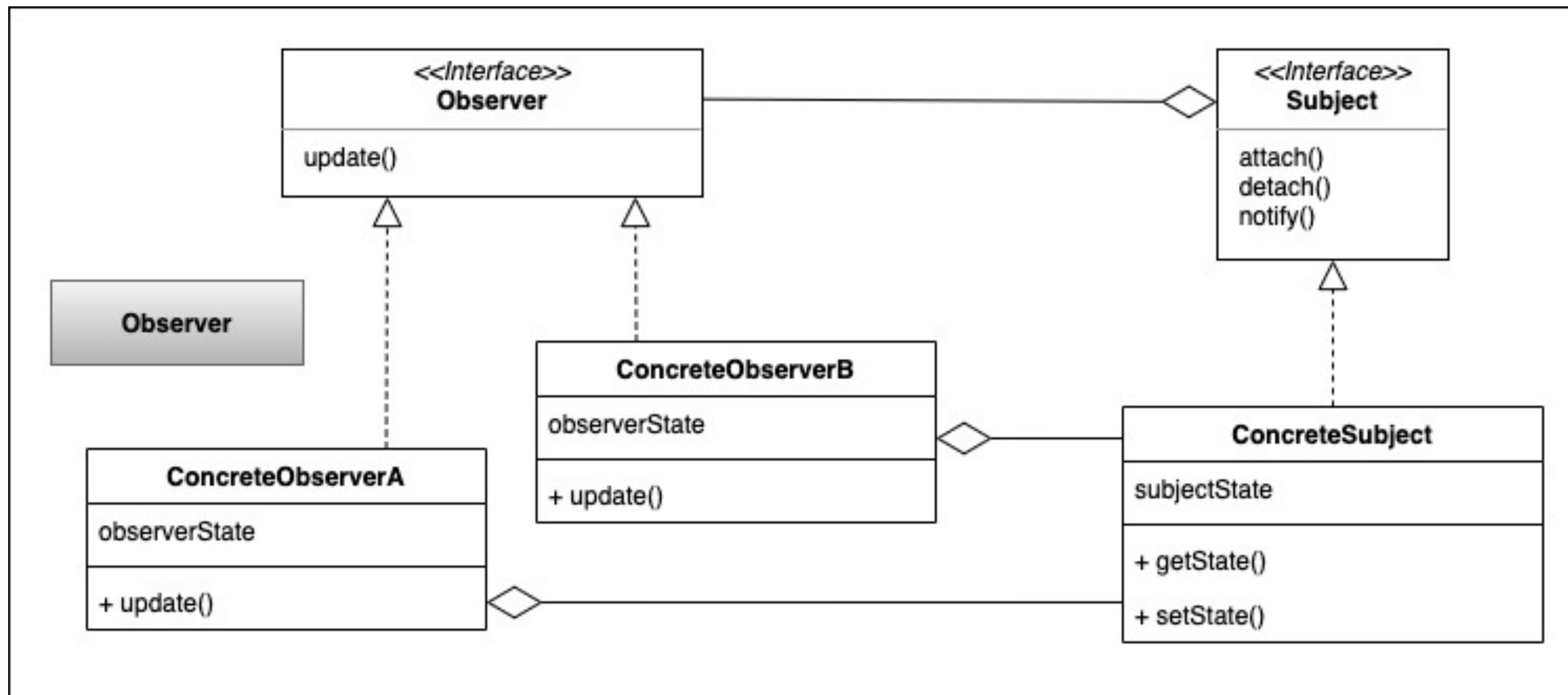
Anwendbarkeit (geeignet, wenn...)

- Abstraktion besitzt zwei Aspekte und einer davon ist vom anderen abhängig.
Kapseln durch Observer in verschiedene Objekte gestattet unabhängige Änderung und Wiederverwendung
- Modifikation eines Objektes erfordert Änderung anderer Objekte - es ist nicht bekannt wie viele
- ein Objekt soll andere Objekte benachrichtigen können (ohne zu wissen, um welche Objekte es sich handelt)



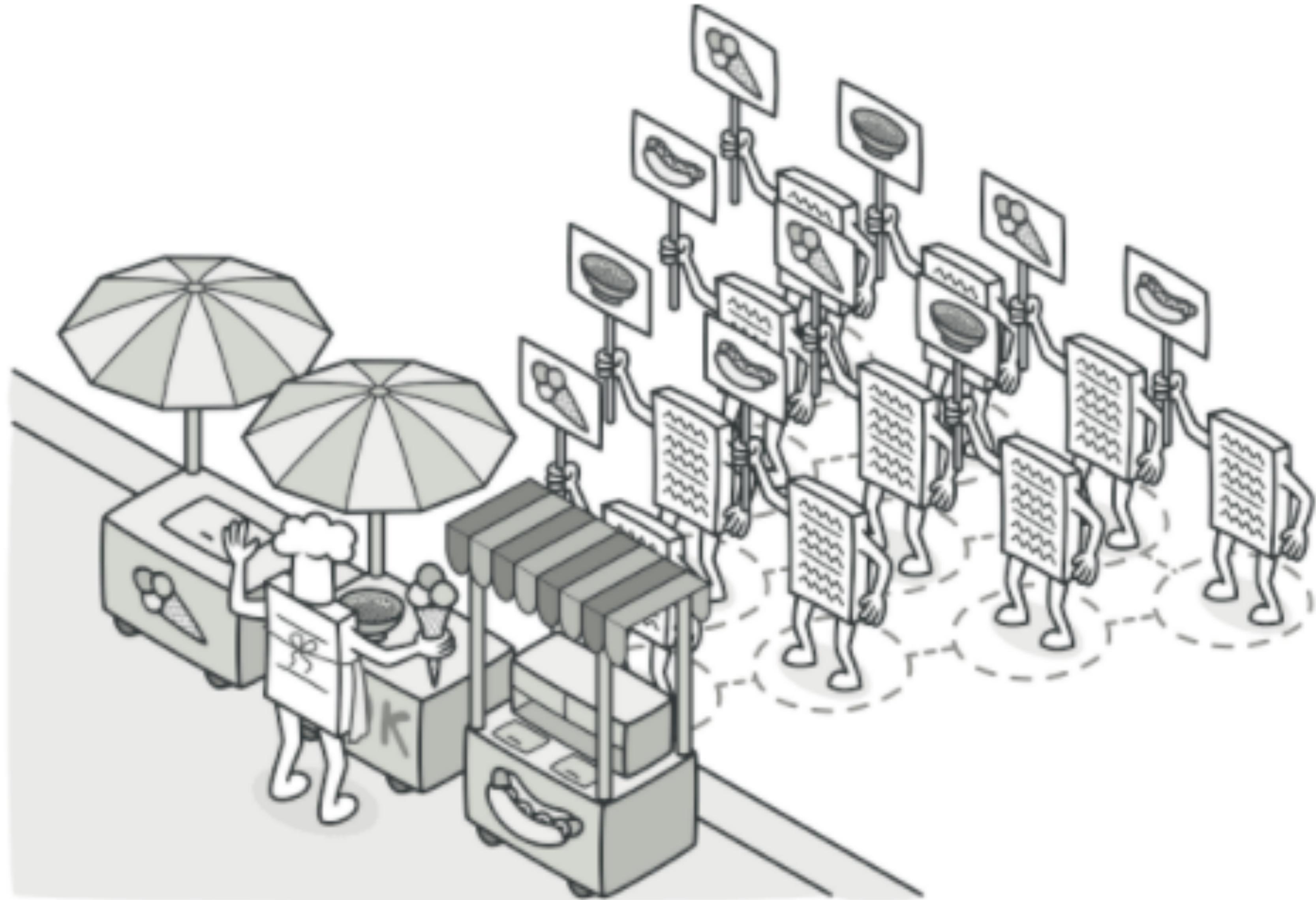
Konsequenzen (Vor- und Nachteile)

- unabhängige Änderung, Hinzufügen oder Entfernen von Subjekten und Observern
- Abstrakte Kopplung von Subjekt und Observer
- Unterstützung von Broadcast-Kommunikation (keine Angabe eines Empfängers vom Subjekt - automatische Weiterleitung an alle Interessenten)
- unerwartete oder fehlerhafte Aktualisierungen (Observer kennen keine anderen Observer - nicht wohldefinierte oder ordentliche Module können zu zweifelhaften Aktualisierungen führen)



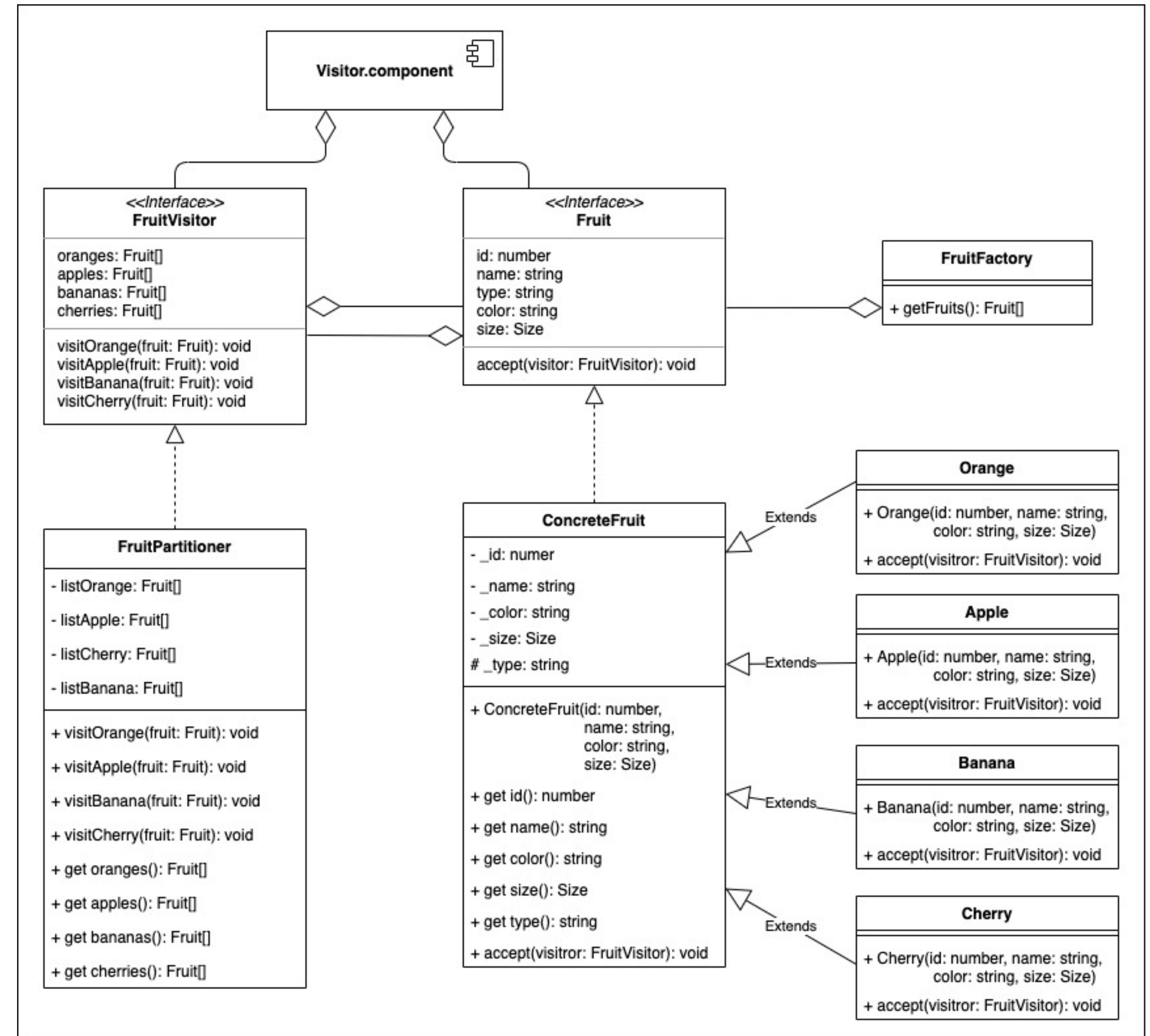
Code-Beispiel

Visitor Besucher



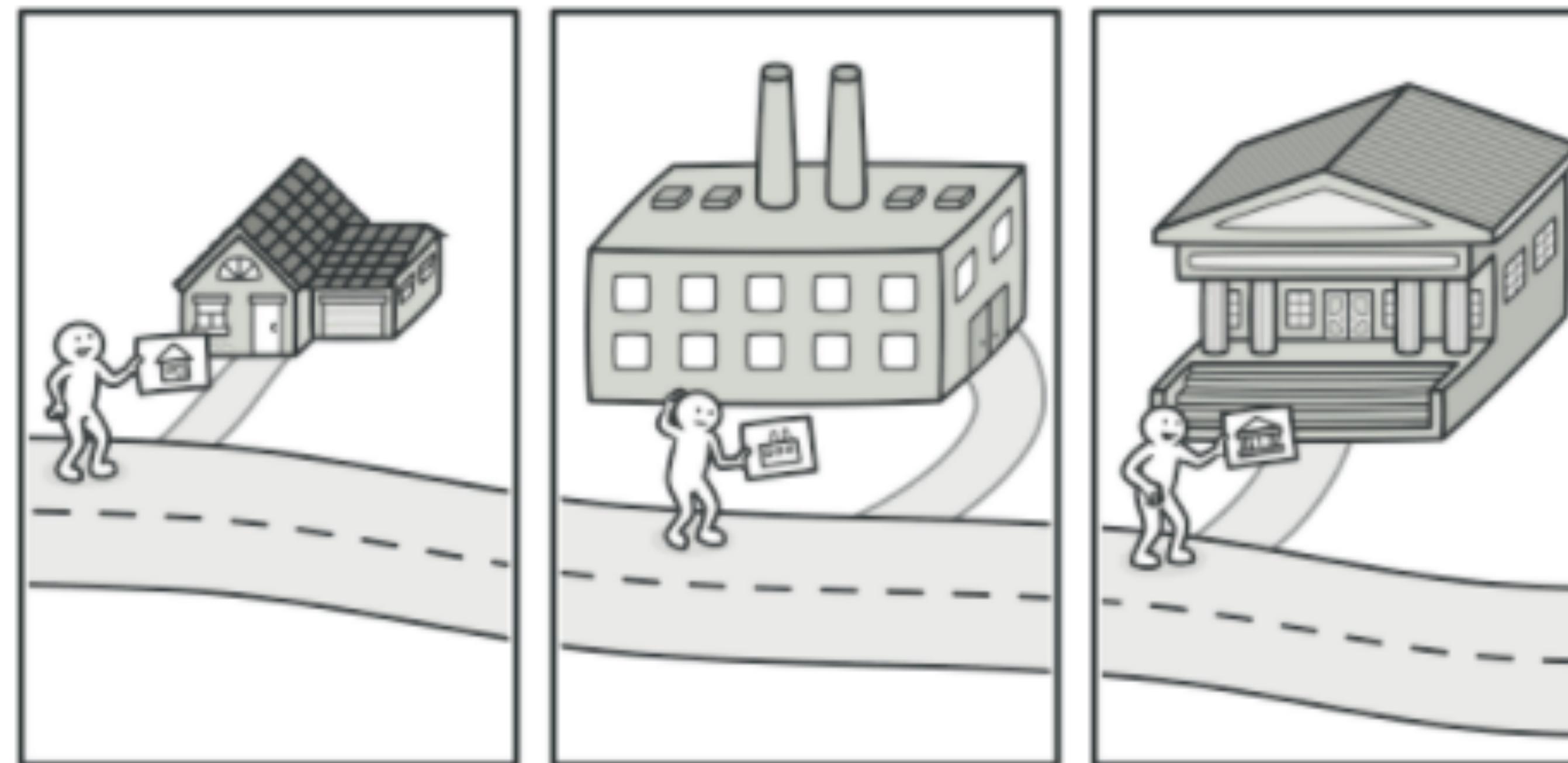
Quelle: <https://refactoring.guru/design-patterns/visitor>

Demo



Zweck

- Darstellung einer auf die Elemente einer Objektstruktur anzuwendenden Operation
- Definition neuer Operation, ohne Klasse der von ihr bearbeiteten Elemente zu verändern



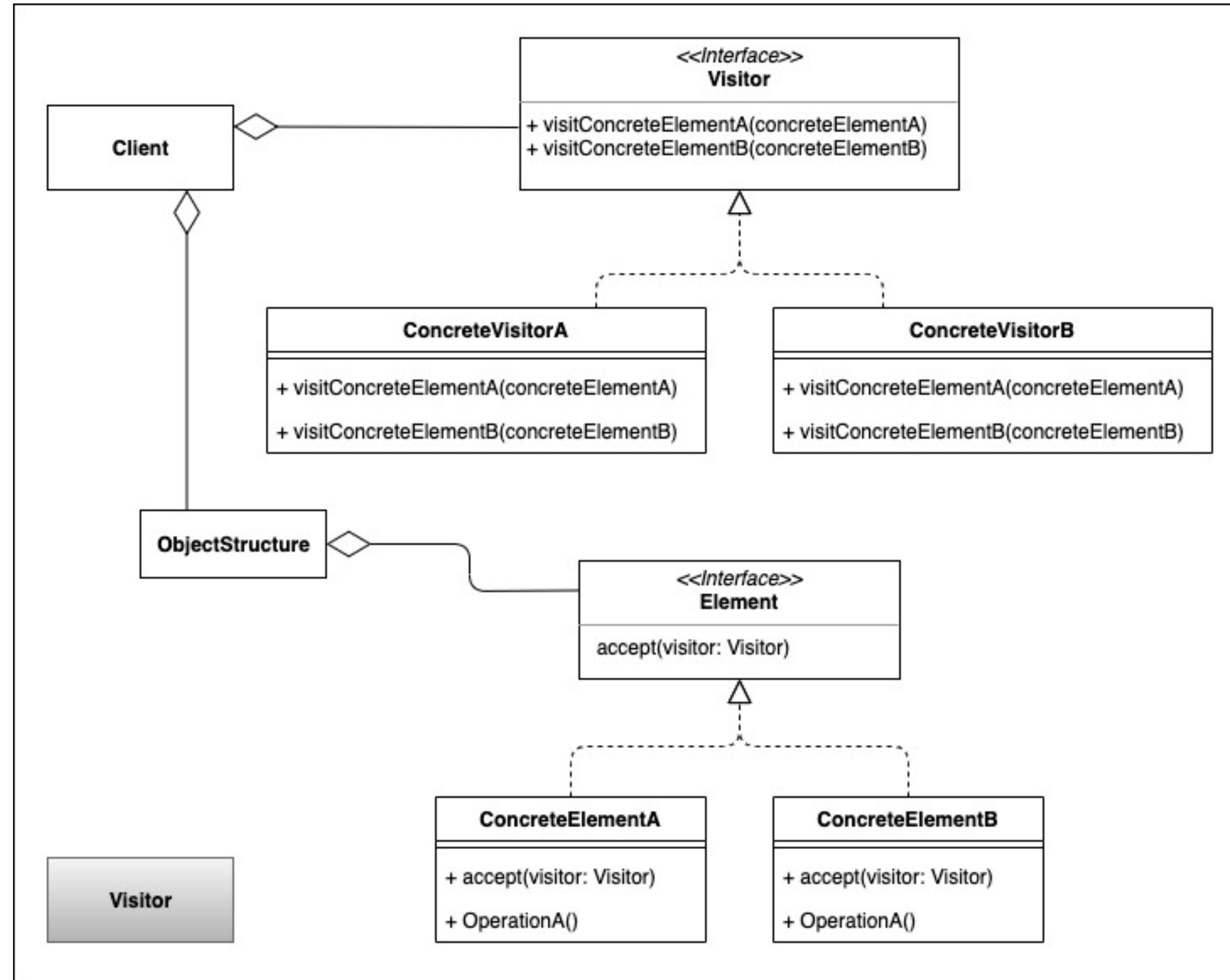
Quelle: <https://refactoring.guru/images/patterns/content/visitor/visitor-comic-1.png>

Anwendbarkeit (geeignet, wenn...)

- Objektstruktur enthält viele Klassen von Objekten mit unterschiedlichen Schnittstellen und Objektoperationen, die von den konkreten Klassen abhängen
- viele individuelle und verschiedenartige Operationen mit Objekten einer Objektstruktur durchgeföhrten werden
- Operationen werden in derselben Klasse definiert
- seltene Änderungen der Klassen innerhalb der Objektstruktur, aber häufig neue Operationen dafür eingerichtet werden sollen
- Änderung von Klassen macht Neudefinition aller Visitor-Schnittstellen erforderlich (dann sollten Operationen in der Klasse definiert werden)

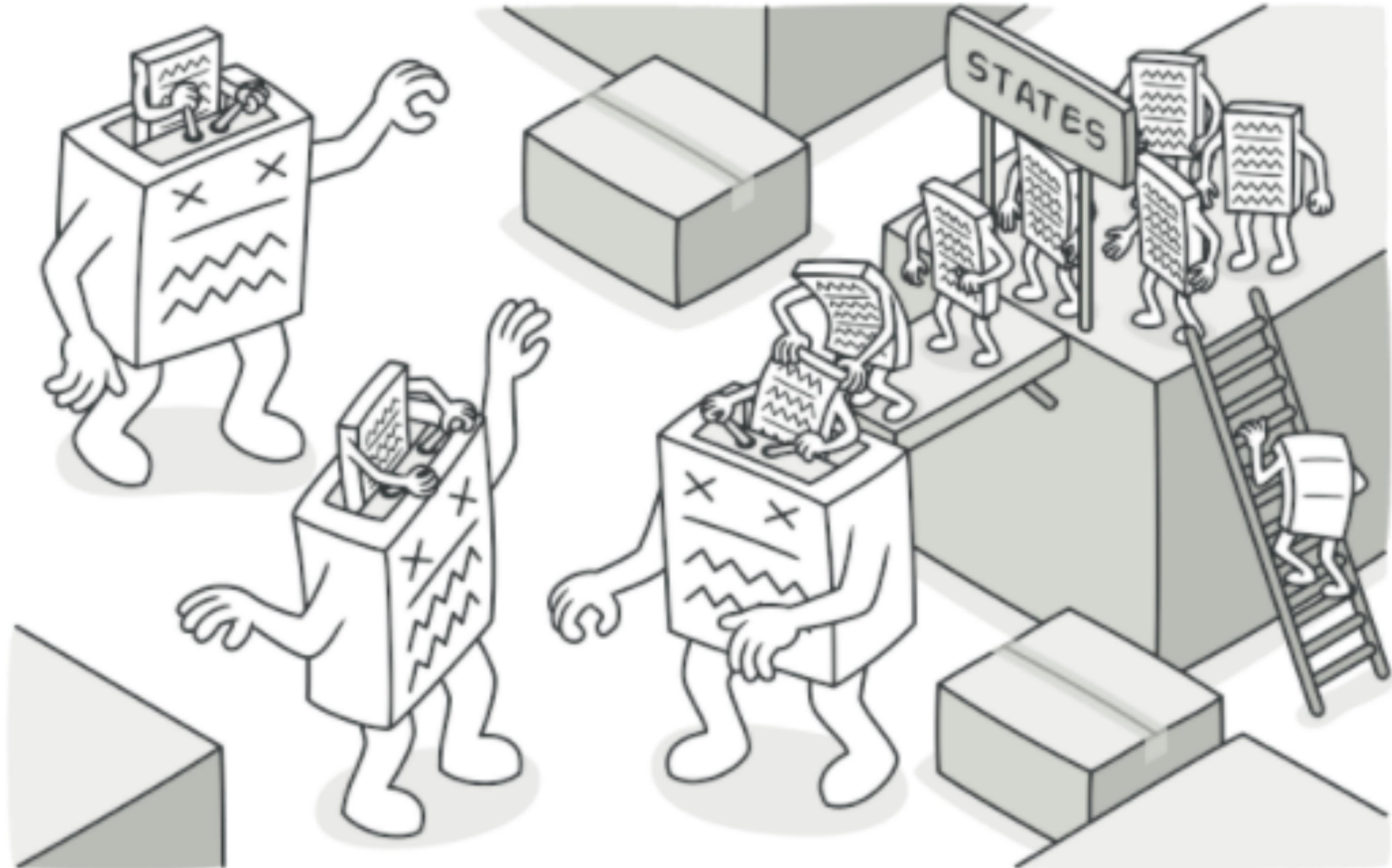
Konsequenzen (Vor- und Nachteile)

- *Visitor*-Objekte erleichtern das Hinzufügen neuer Operationen
- vereint einander ähnliche Operationen und trennt unähnliche voneinander (Verortung im *Visitor*-Objekt)
- Hinzufügen neuer *ConcreteElement*-Klassen ist schwierig, da jedes Element zu einer neuen abstrakten *Visitor*-Operation führt + Implementierung in jeder *ConcreteVisitor*-Klasse
- Benutzung des Pattern hängt von Häufigkeit der Änderungen der Klassen ab
- Klassenhierarchie-übergreifende Besuche (SubClass, SuperClass)
- Einsammeln von Zuständen
- Verletzung der Kapselung und damit Verletzung von OCP



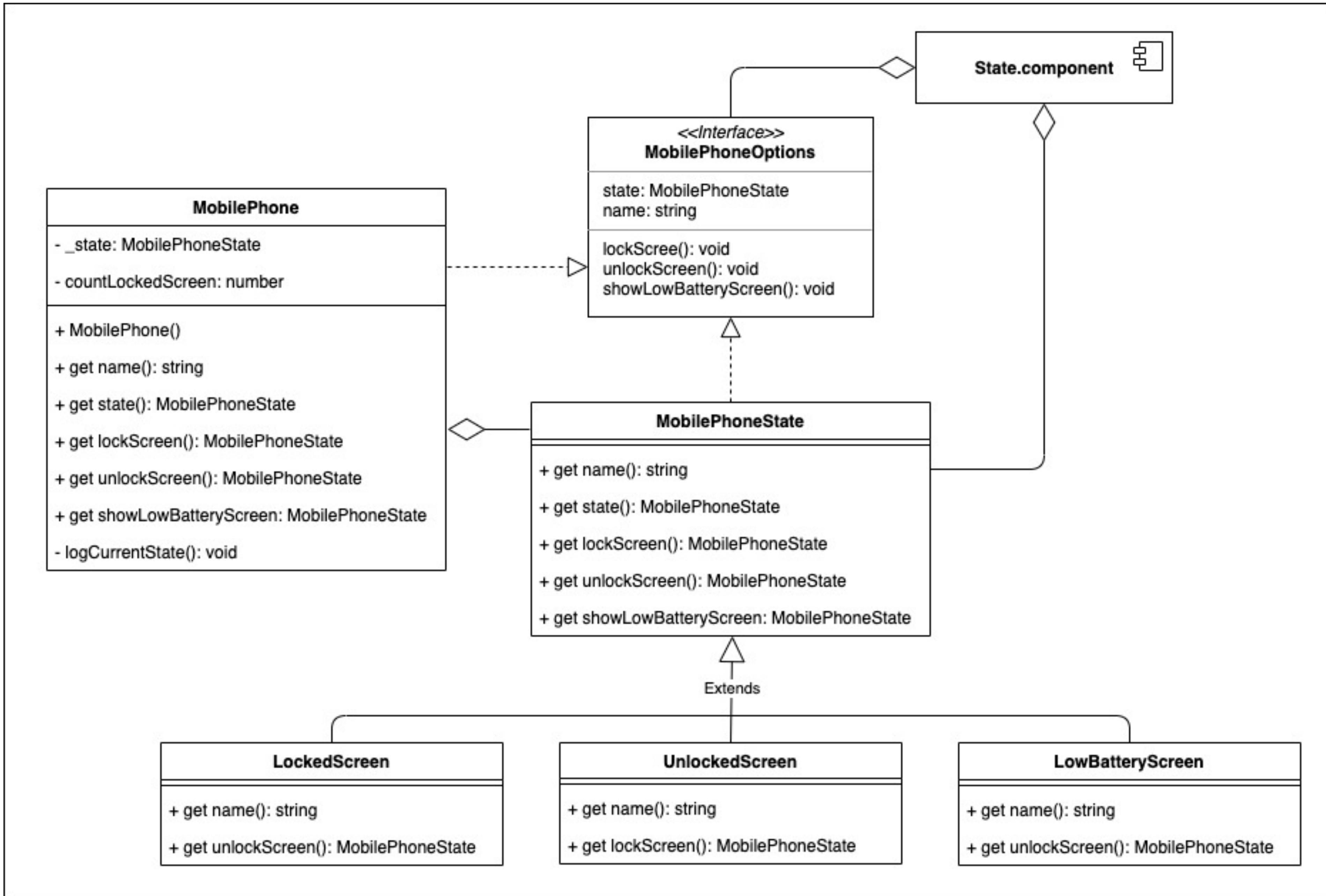
Code-Beispiel

State Zustand



Quelle: <https://refactoring.guru/design-patterns/state>

Demo



Zweck

- Anpassung der Verhaltensweise eines Objektes im Fall einer internen Zustandsänderung (Wirkung wie Klassenwechsel)



Quelle: <https://refactoring.guru/images/patterns/diagrams/state/problem2-en.png>

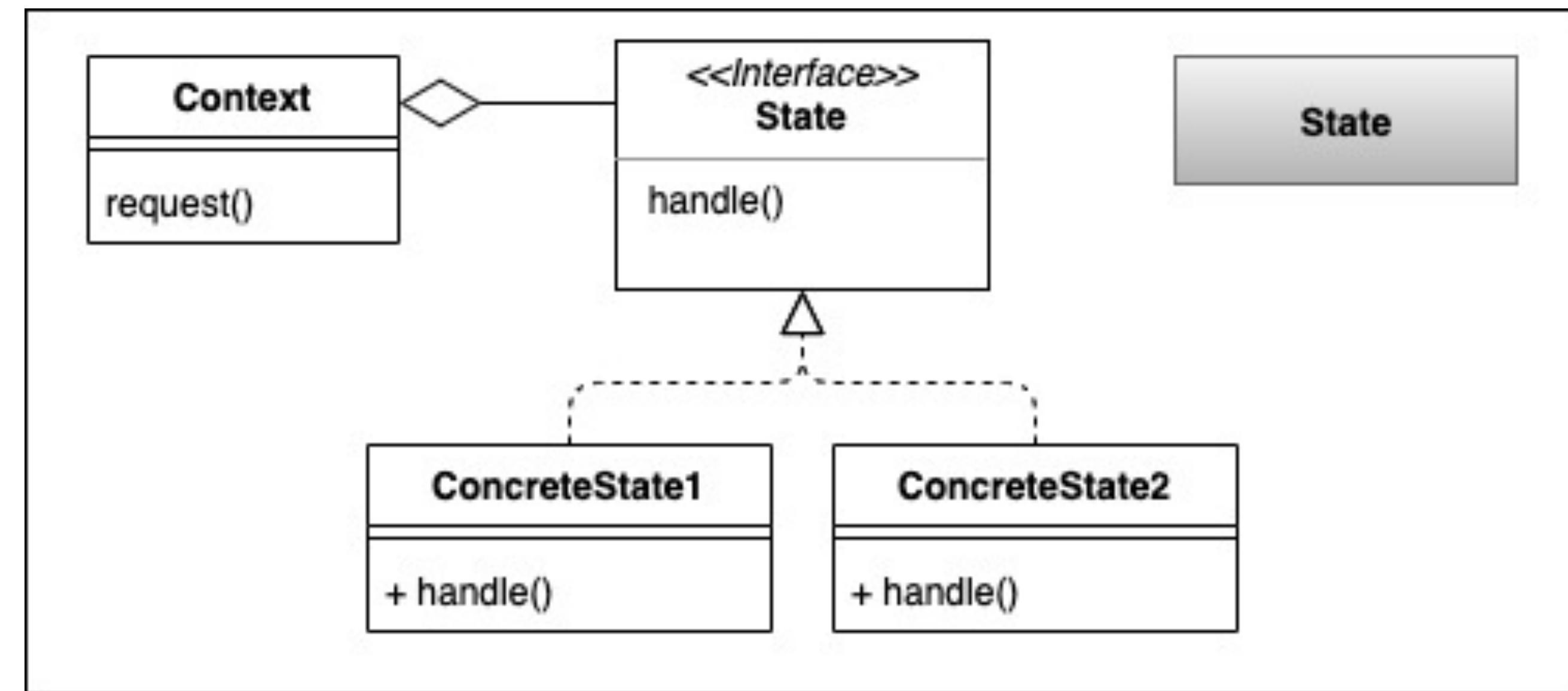
Anwendbarkeit (geeignet, wenn...)

- Verhalten eines Objektes hängt von seinem Zustand ab und muss seine Verhaltensweisen zur Laufzeit von diesem Zustand ändern
- Operationen vom Zustand des Objektes abhängen (e.g. Switch Statement)

```
class Document {
    field state: string
    // ...
    method publish() {
        switch (state) {
            "draft": {
                state = "moderation"
                break
            }
            "moderation": {
                if (currentUser.role == 'admin')
                    state = "published"
                break
            }
            "published": {
                // Do nothing.
                break
            }
        }
    }
}
```

Konsequenzen (Vor- und Nachteile)

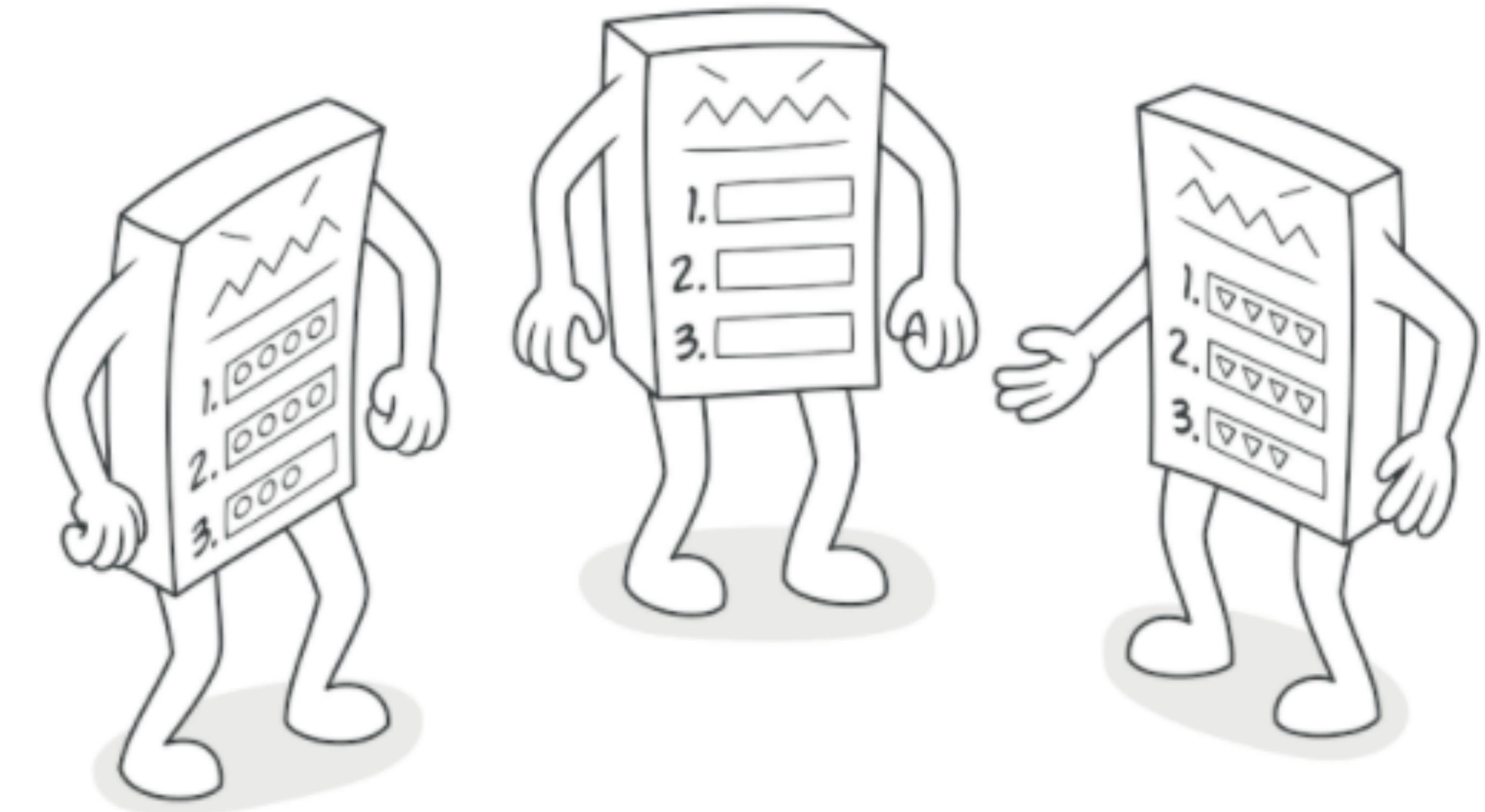
- verortet zustandspezifische Verhaltensweisen + verteilt sie auf verschiedene Zustände (zustandsspezifischer Code findet sich in *State*-Unterklasse und damit einfache Erweiterung)
- Zustandsänderungen werden deutlich hervorgehoben
- *State*-Objekte können gemeinsam genutzt werden



Code-Beispiel

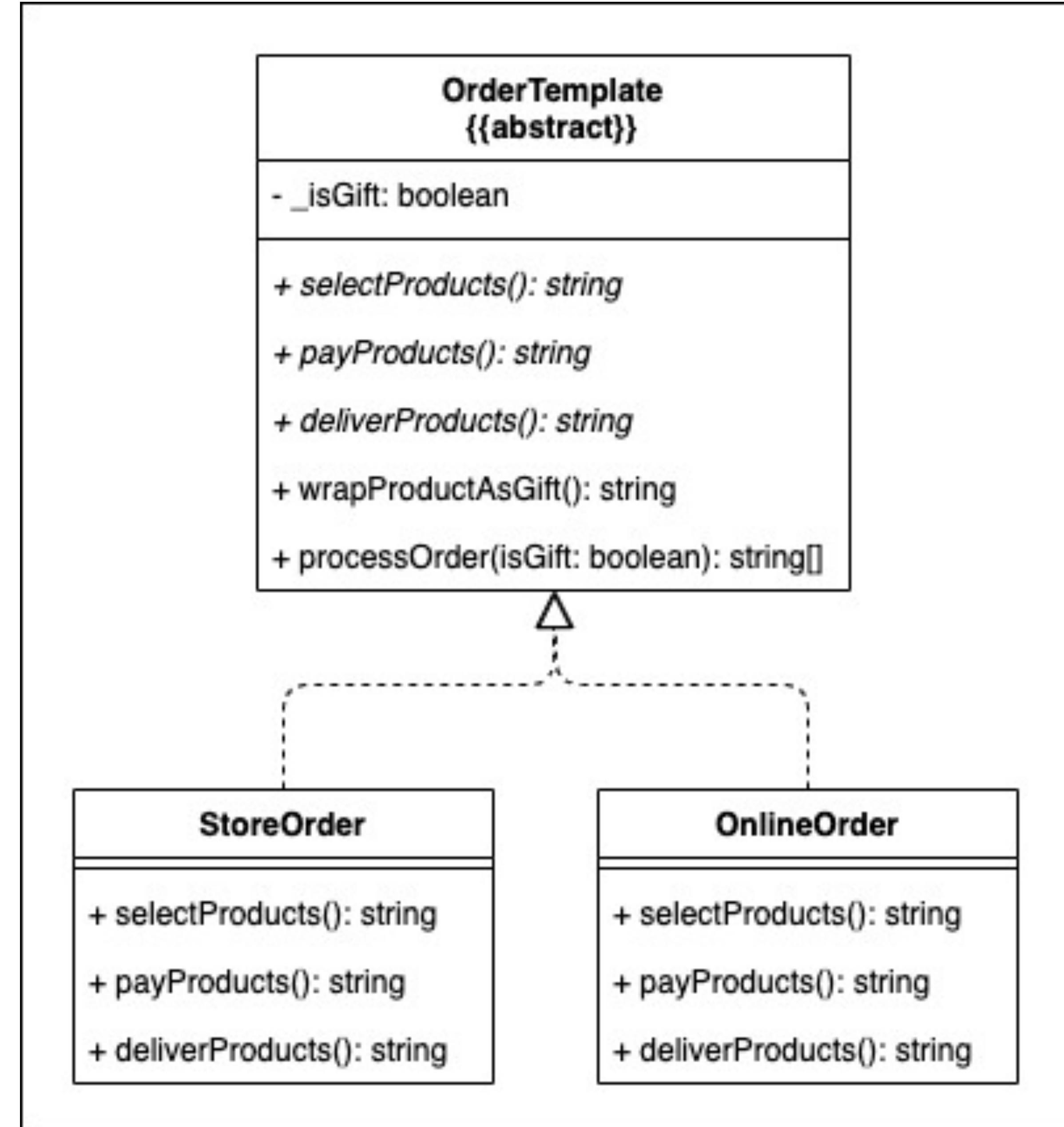
Template Method

Schablonenmethode



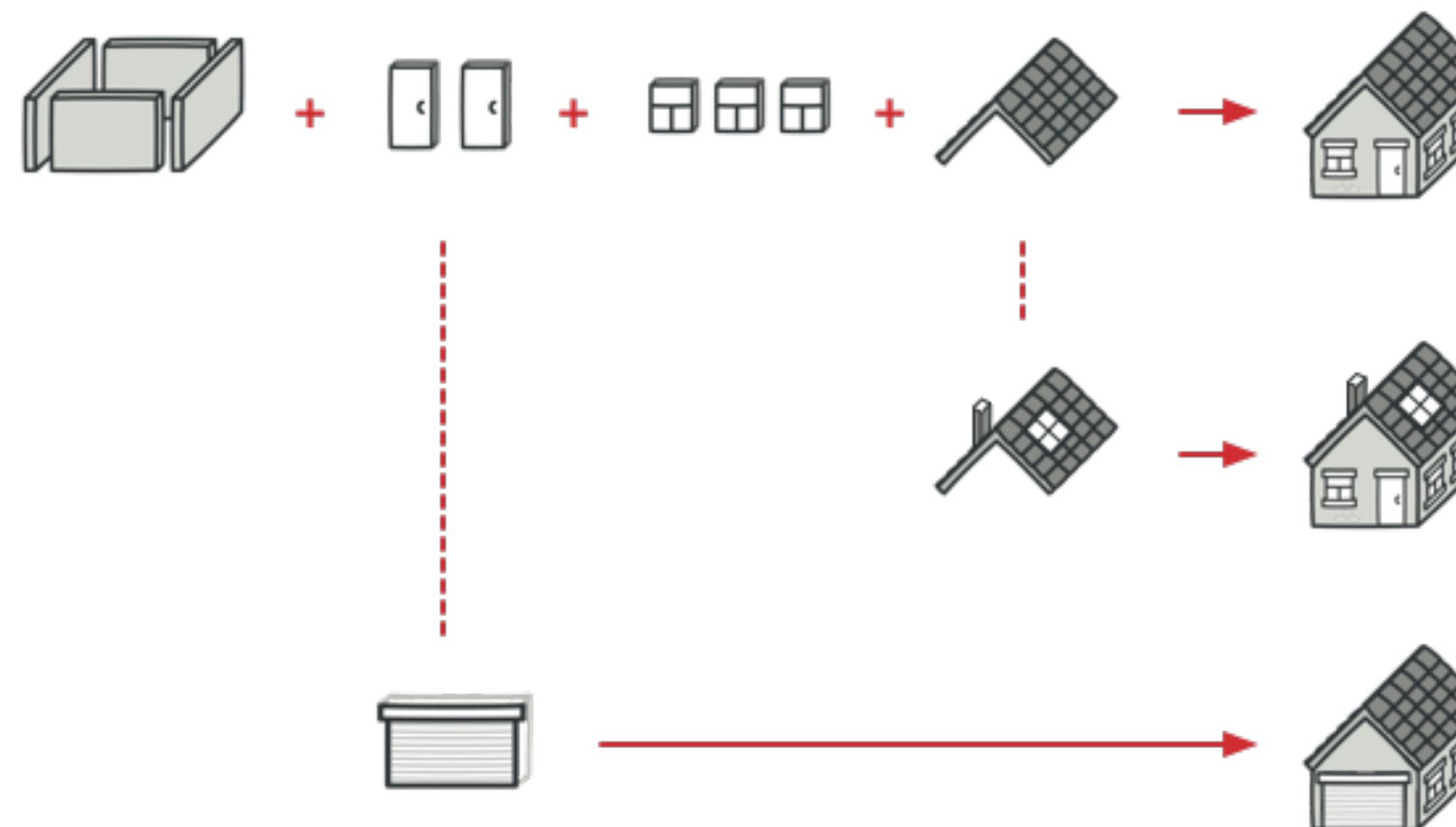
Quelle: <https://refactoring.guru/design-patterns/template-method>

Demo



Zweck

- Definition der Grundstruktur eines Algorithmus in einer Operation sowie Delegation einiger Ablaufschritte an Unterklassen
- ermöglicht Unterklassen bestimmte Schritte eines Algorithmus zu überschreiben, ohne dessen Struktur zu ändern



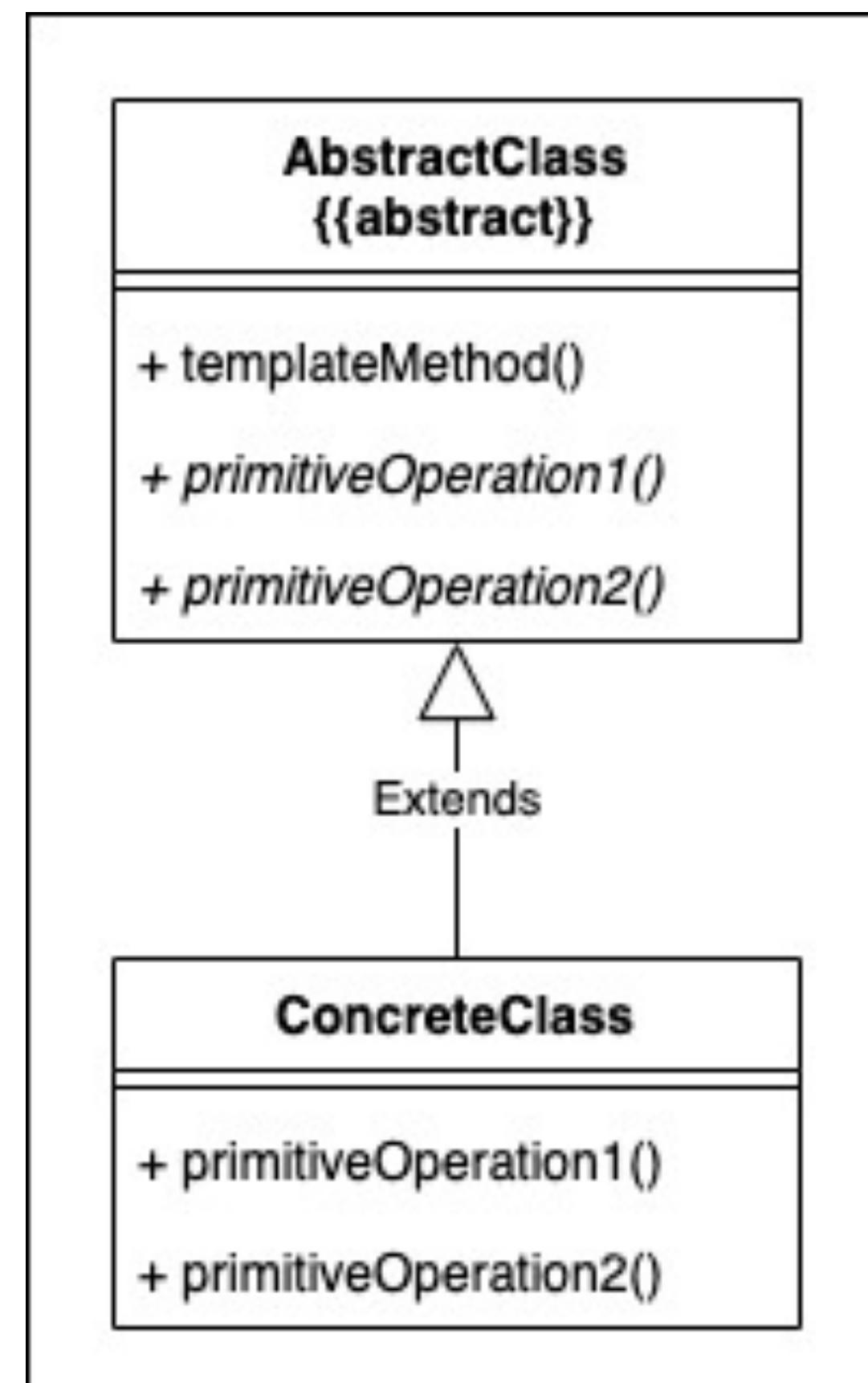
Quelle: <https://refactoring.guru/images/patterns/diagrams/template-method/live-example.png>

Anwendbarkeit (geeignet, wenn...)

- um unveränderlichen Teil eines Algorithmus einmalig zu implementieren und Unterklassen überlassen, Verhaltensklassen zu implementieren, die veränderlich sind
- gemeinsame Verhaltensweisen in Unterklassen ausgelagert werden sollen (*refactoring to generalize*)
- Regulation der Erweiterung von Unterklassen (*Hook*)

Konsequenzen (Vor- und Nachteile)

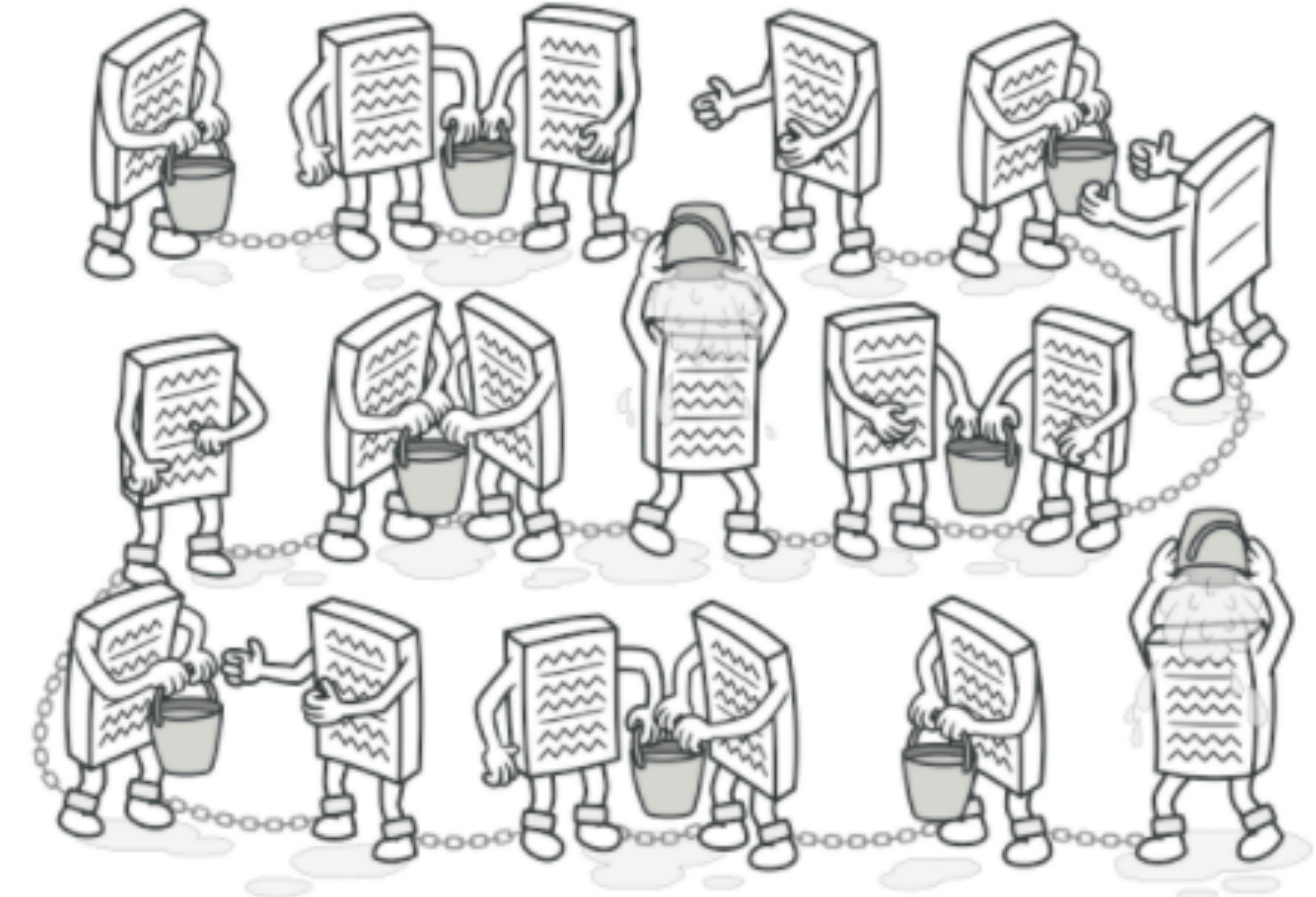
- grundlegende Technik der Wiederverwendung
- besonderen Wert in Klassenbibliotheken
- *Cleanere Vererbung*
- wird auch als **Hollywood-Principle** bezeichnet (“Rufen Sie uns nicht an, wir rufen Sie an.”) -> Basisklasse ruft Operation der Unterklasse auf



Code-Beispiel

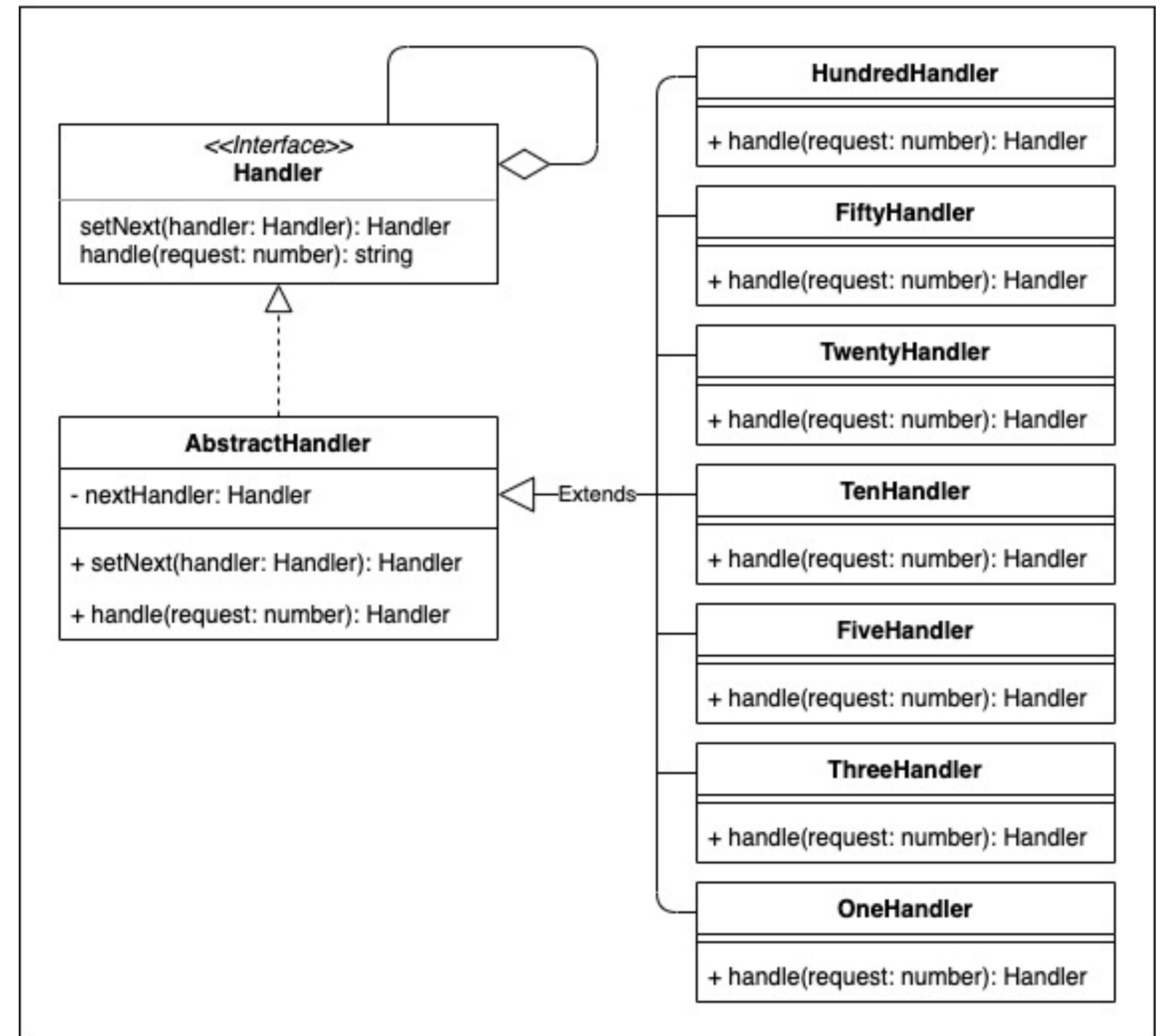
Chain Of Responsibility

Zuständigkeitskette



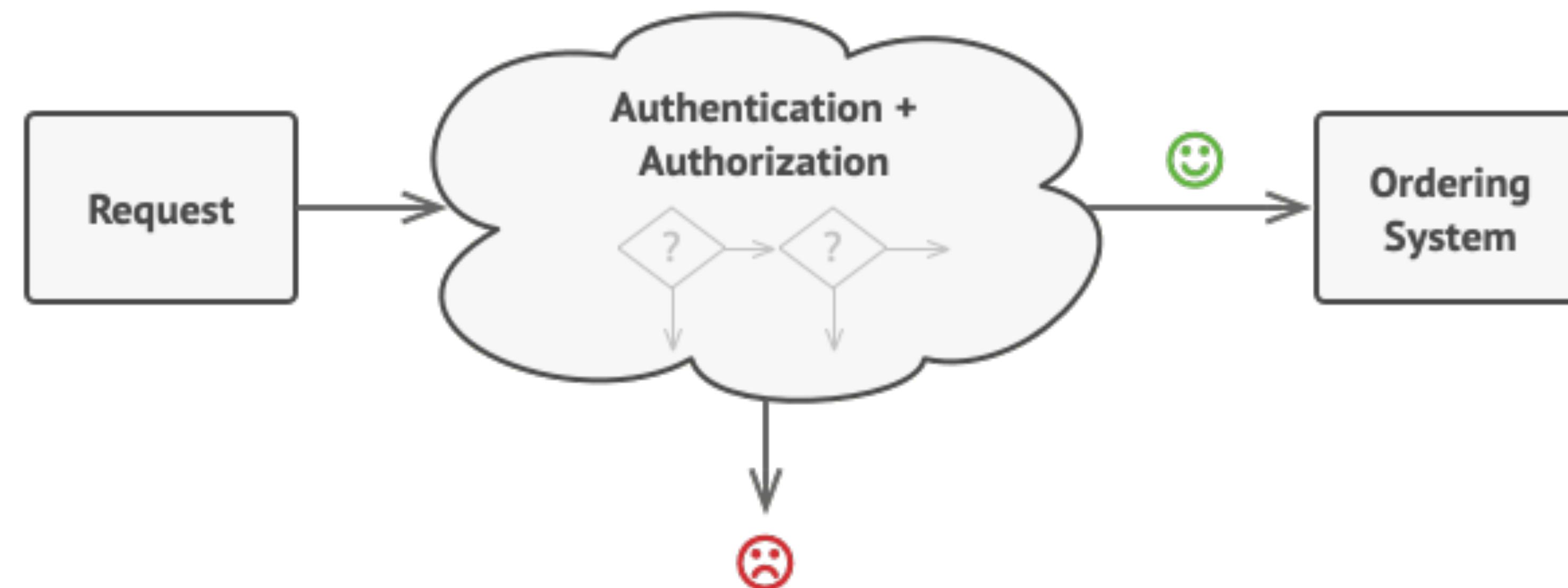
Quelle: <https://refactoring.guru/design-patterns/chain-of-responsibility>

Demo



Zweck

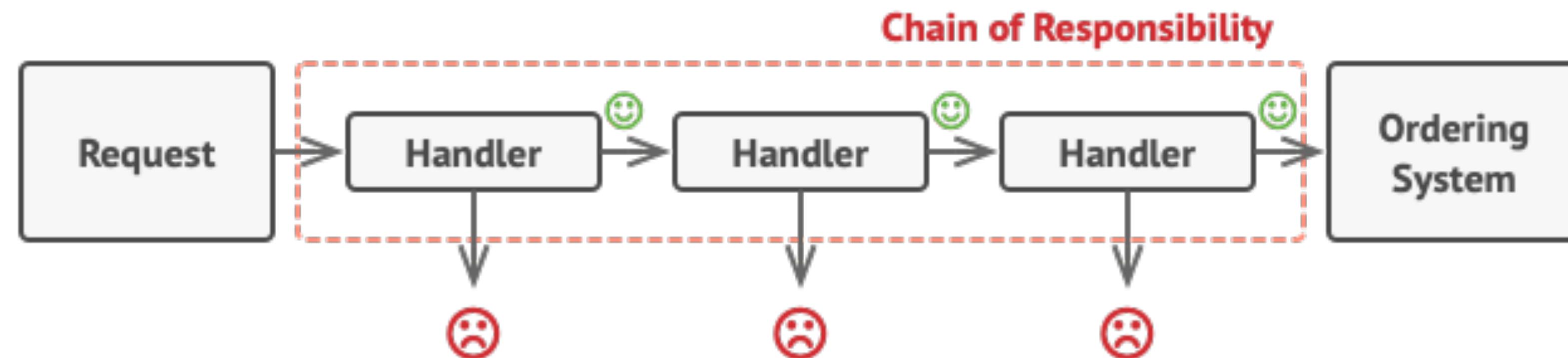
- Vermeidung der Kopplung eines Request-Absenders mit dessen Empfänger, indem mehrere Objekte die Möglichkeit haben, den Request zu bearbeiten
- empfangene Objekte werden miteinander verkettet und Request wird weitergeleitet, bis er angenommen und bearbeitet wird



Quelle: <https://refactoring.guru/images/patterns/diagrams/chain-of-responsibility/problem1-en.png>

Anwendbarkeit (geeignet, wenn...)

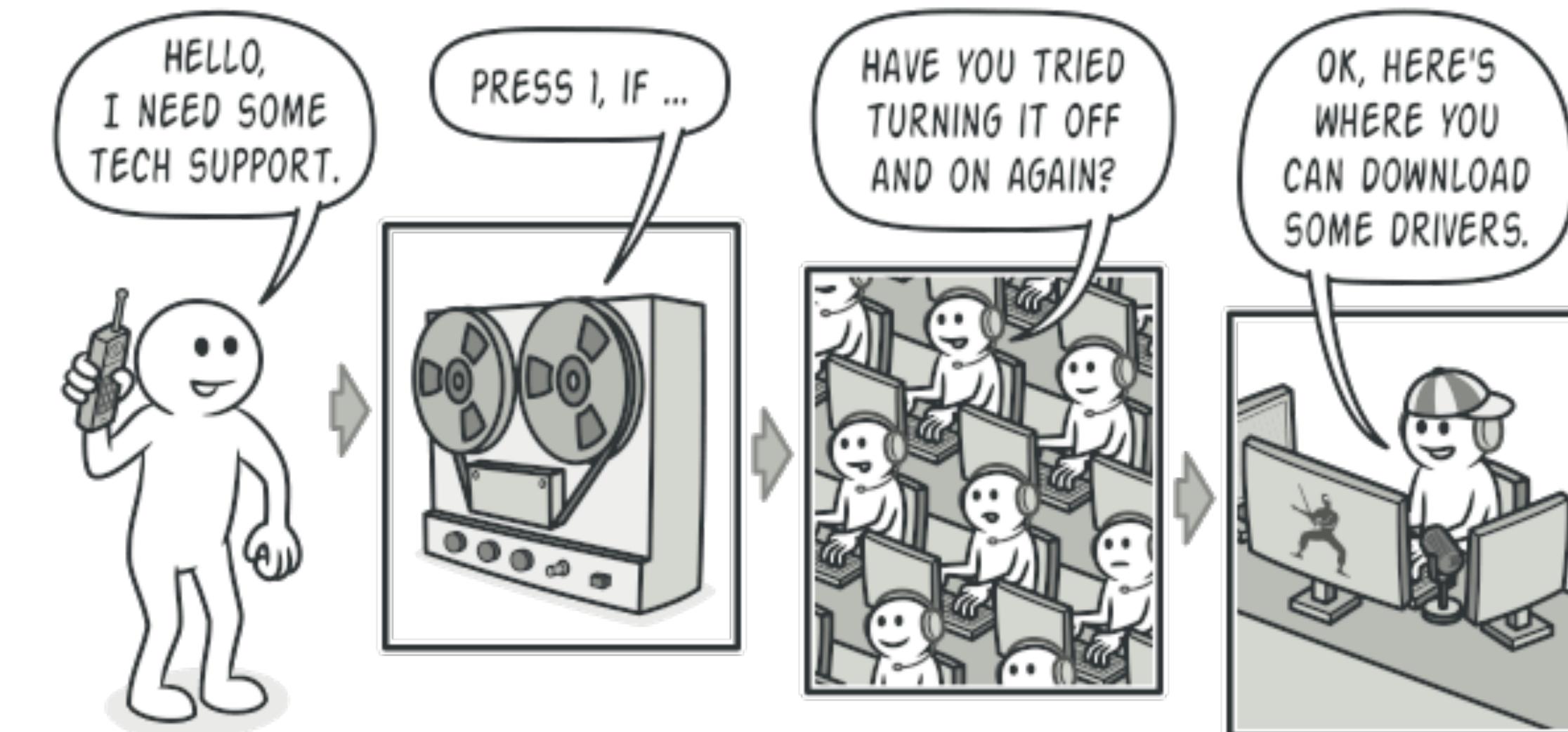
- Requests von mehreren Objekte bearbeitet werden können und Handler (Bearbeiter) von vornherein nicht bekannt ist - automatisch zur Laufzeit bestimmt
- ein Request ohne explizite Angabe des Empfängers an eins von mehreren Objekten gerichtet werden soll
- Objektsatz, der Request bearbeiten kann, dynamisch bestimmt wird



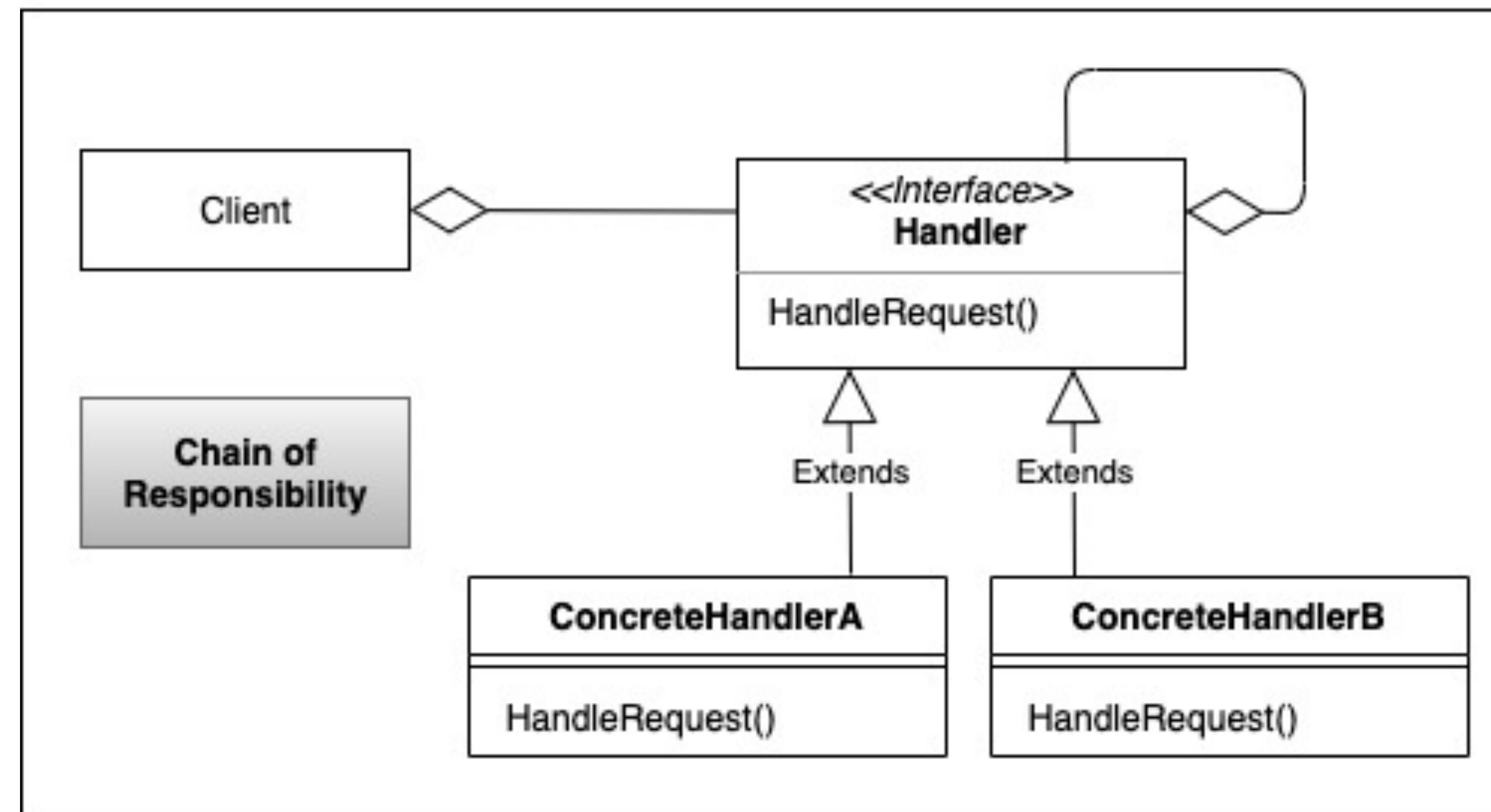
Quelle: <https://refactoring.guru/images/patterns/diagrams/chain-of-responsibility/solution1-en.png>

Konsequenzen (Vor- und Nachteile)

- Reduzierte Kopplung (Objekt der Bearbeitung kennt Absender nicht) und damit unabhängige Implementierung
- Referenz auf Nachfolgeobjekt anstatt auf alle Objektkandidaten
- Zusätzliche Flexibilität bei der Zuweisung (Reihenfolge, Unterklassenbildung)
- keine Bearbeitungsgarantie



Quelle: <https://refactoring.guru/images/patterns/content/chain-of-responsibility/chain-of-responsibility-comic-1-en.png>



Code-Beispiel

Interpreter
Interpreter

Zweck

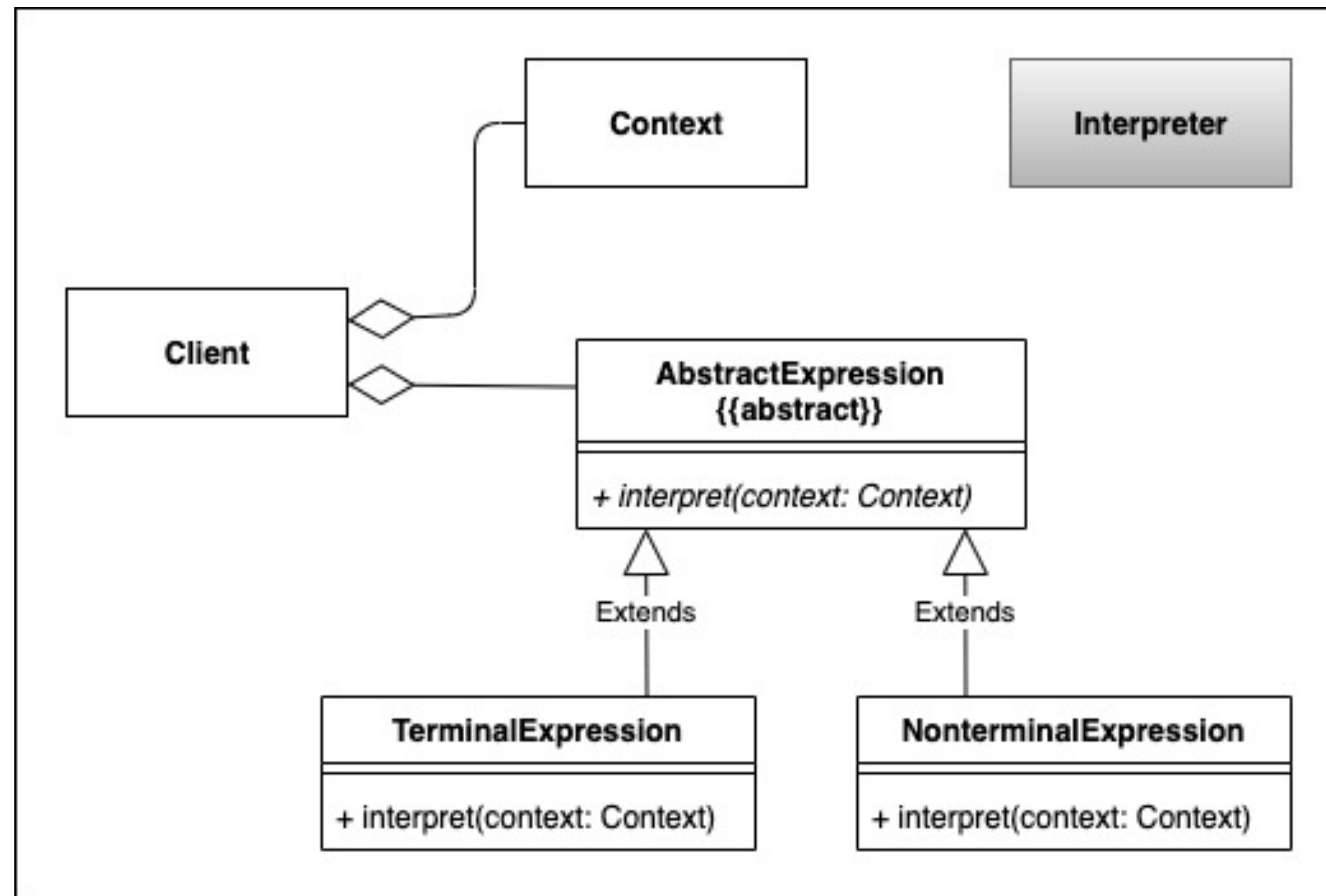
- Definition einer Repräsentation der Grammatik einer Sprache sowie Bereitstellung eines Interpreters, die diese Grammatik nutzt, um Sätze zu interpretieren

Anwendbarkeit (geeignet, wenn...)

- Grammatik einfach gehalten ist
- Faktor Effizienz keine vorrangige Rolle spielt (z.B. Transformation von endlichen Automaten)

Konsequenzen (Vor- und Nachteile)

- Einfache Änderung und Erweiterung der Grammatik
- Einfache Implementierung der Grammatik
- Schwierige Verwaltung komplexer Grammatiken
-



Code-Beispiel

Fragen?

Happy Coding! :)