

Strukturmuster

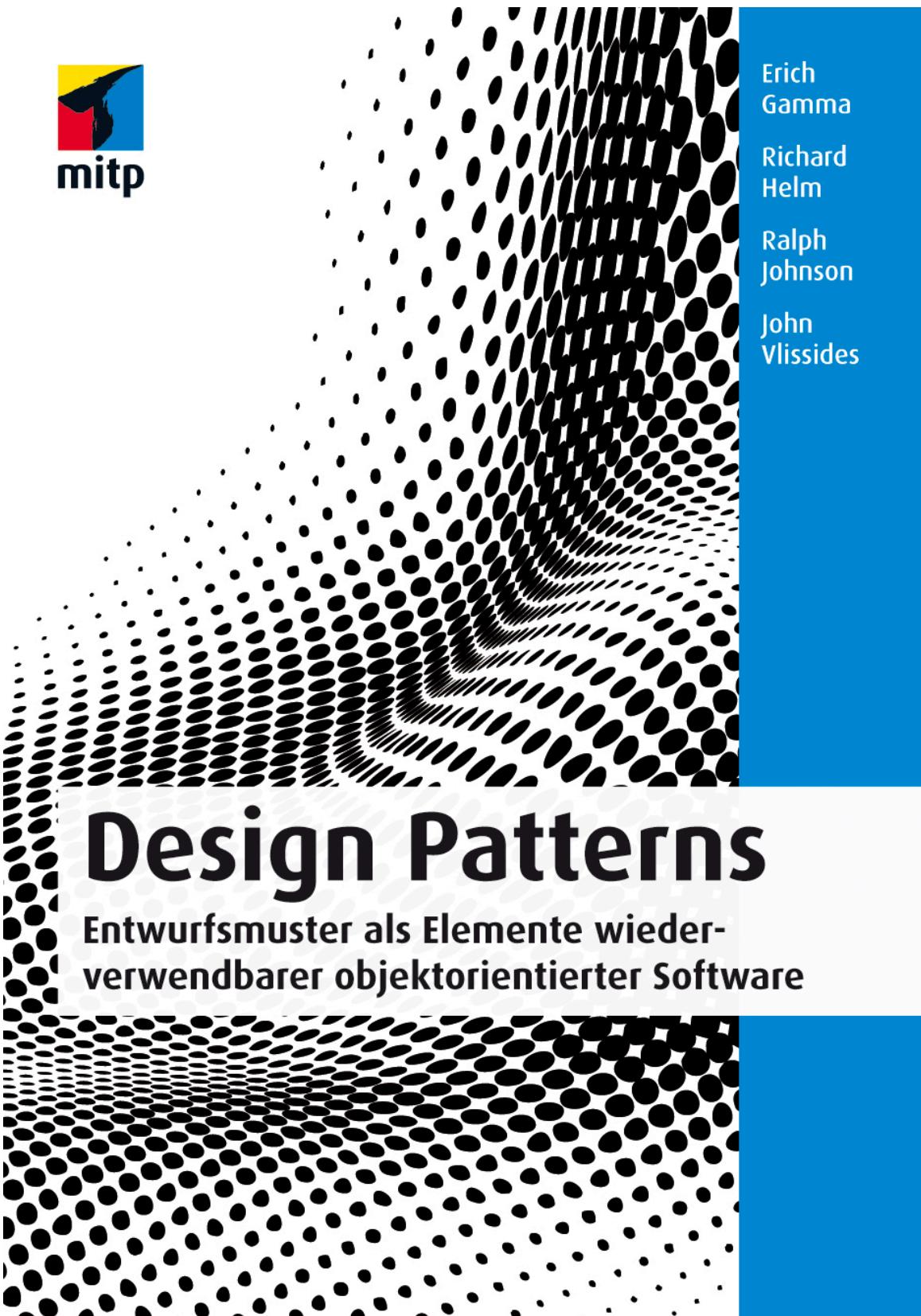
(Structural Patterns)

Patrick Creutzburg, 24. November 2020
Twitter: @Itchimonji



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

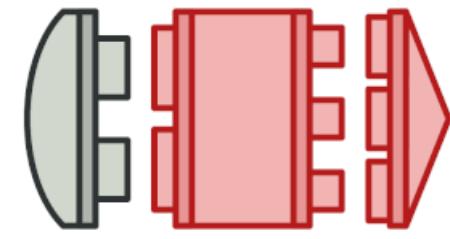
Resources



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

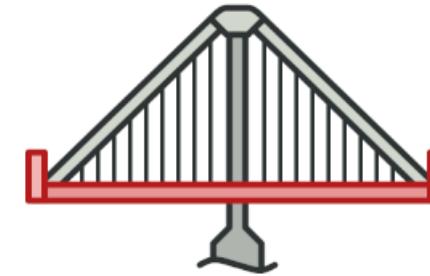


<https://refactoring.guru/>



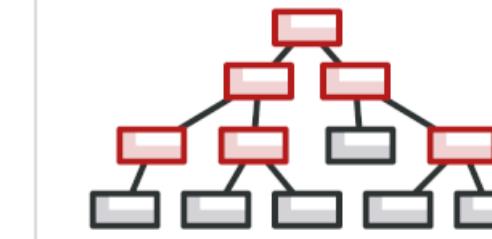
Adapter

Allows objects with incompatible interfaces to collaborate.



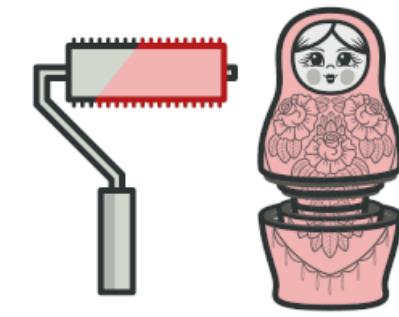
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



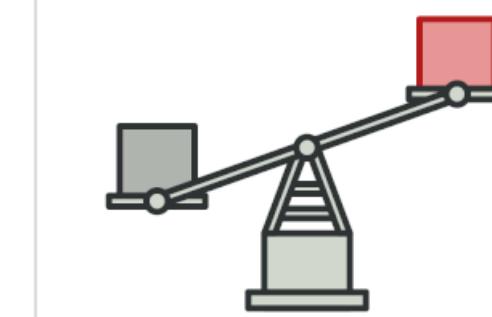
Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



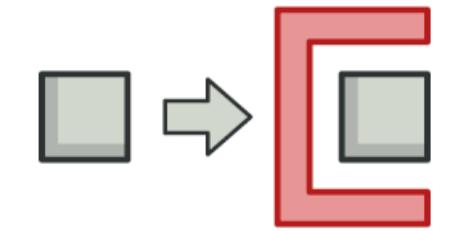
Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Proxy

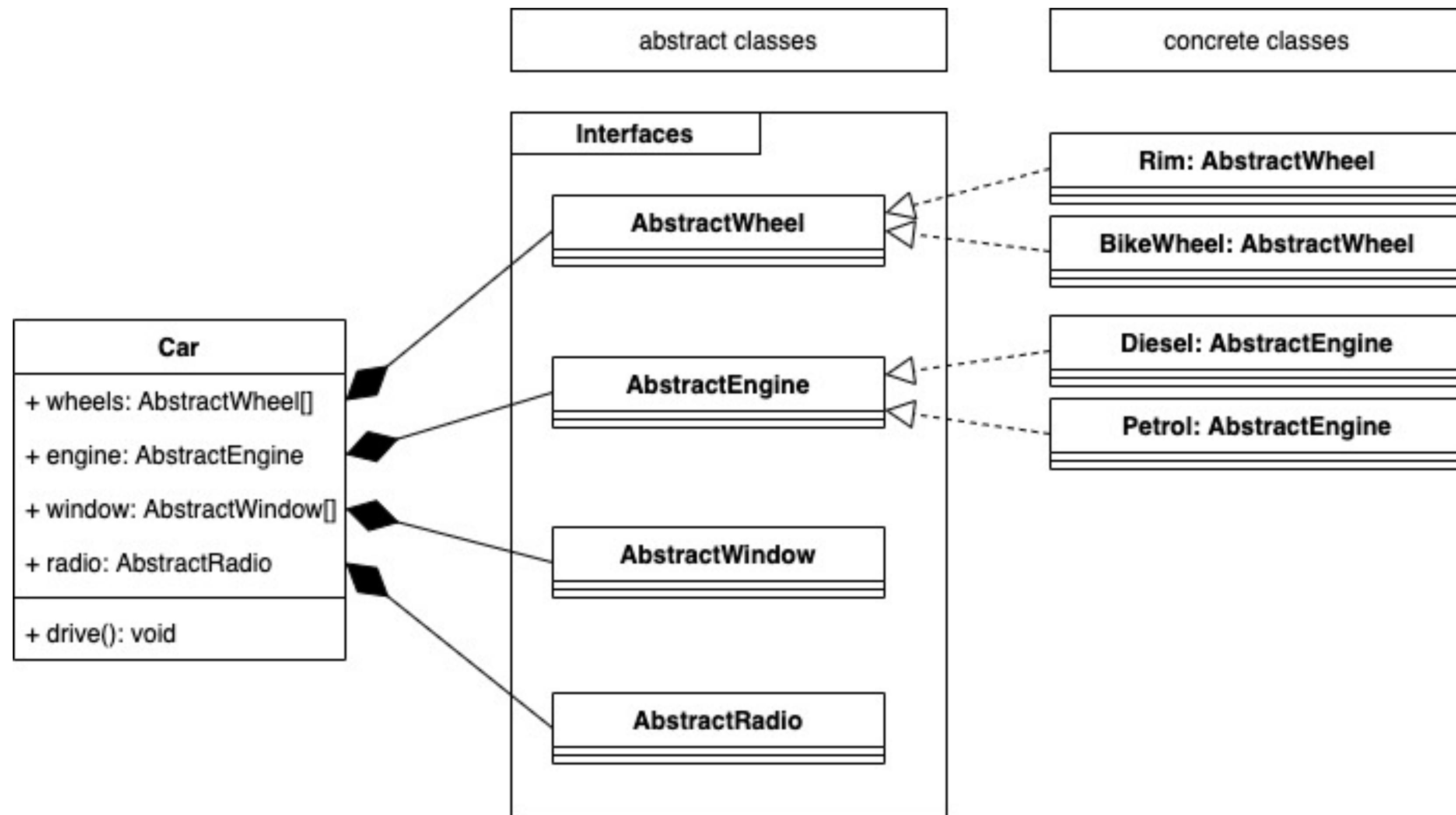
Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Bedeutung

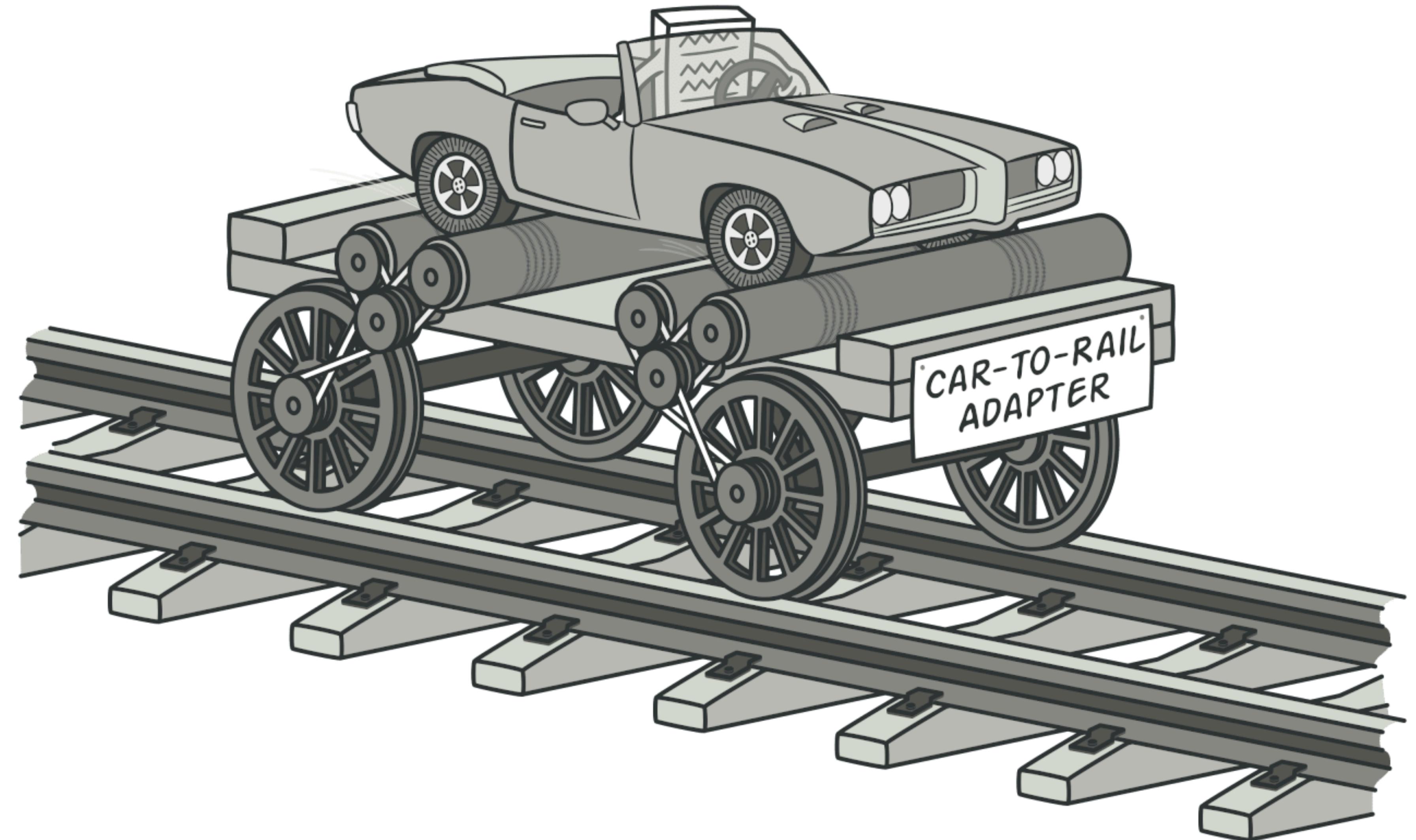
- Komposition von Klassen und Objekten zur Errichtung umfassender Strukturen
- **Kombination von Objekten um neue Funktionalität entstehen zu lassen**
- gewährt Flexibilität durch laufzeitbedingtes veränderbares Kompositionsgefüge
- Steigerung von Effizienz und Konsistenz
- Anwendung von Dependency Inversion
- Implementierung gegen Schnittstellen (Interfaces od. abstrakte Klassen)
- Bewahren des Open-Closed-Prinzips
- flexibles Design, geringer Wartungsaufwand, hohe Wiederverwendbarkeit

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

Object Composition & Dependency Inversion

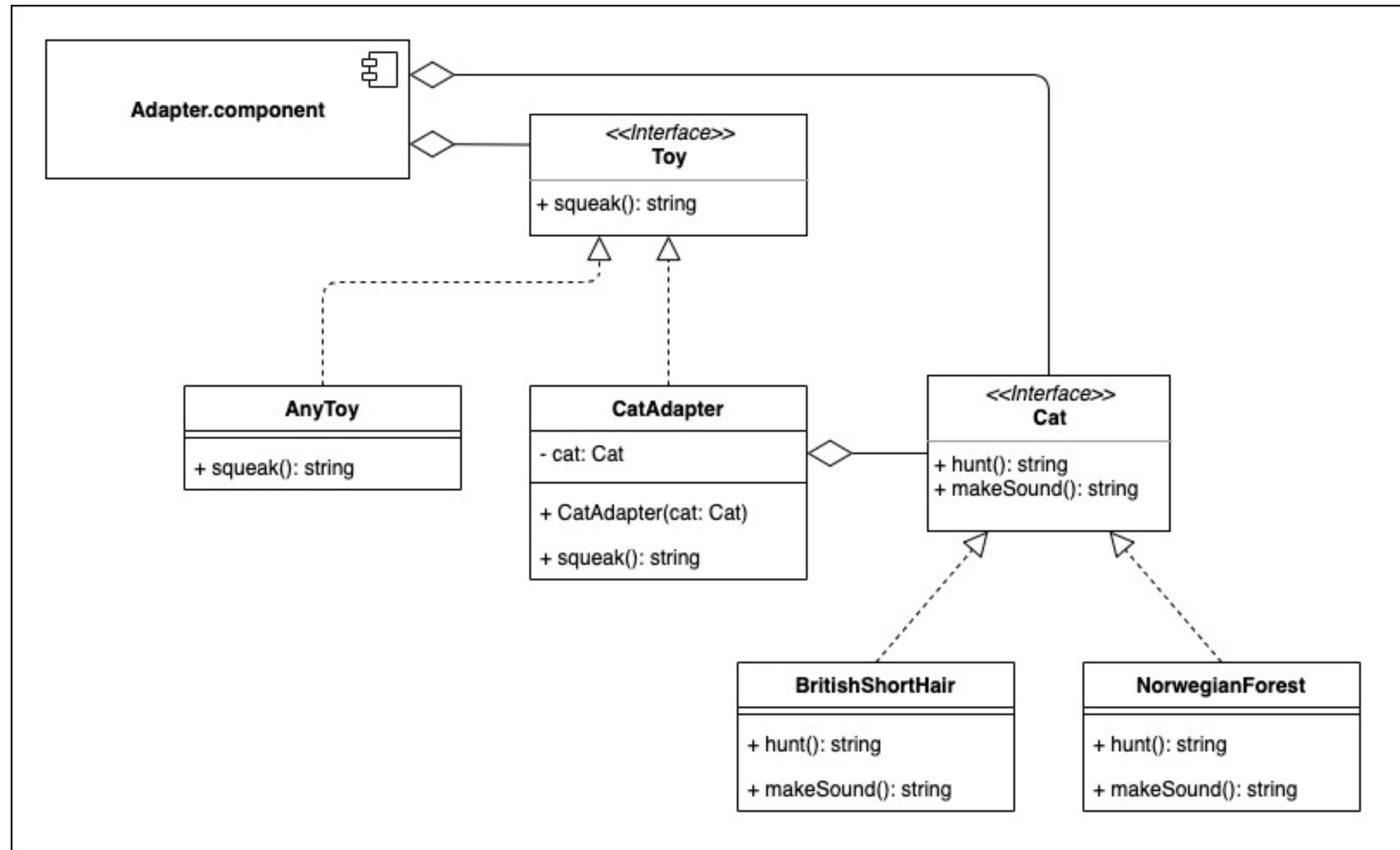


Adapter Adapter



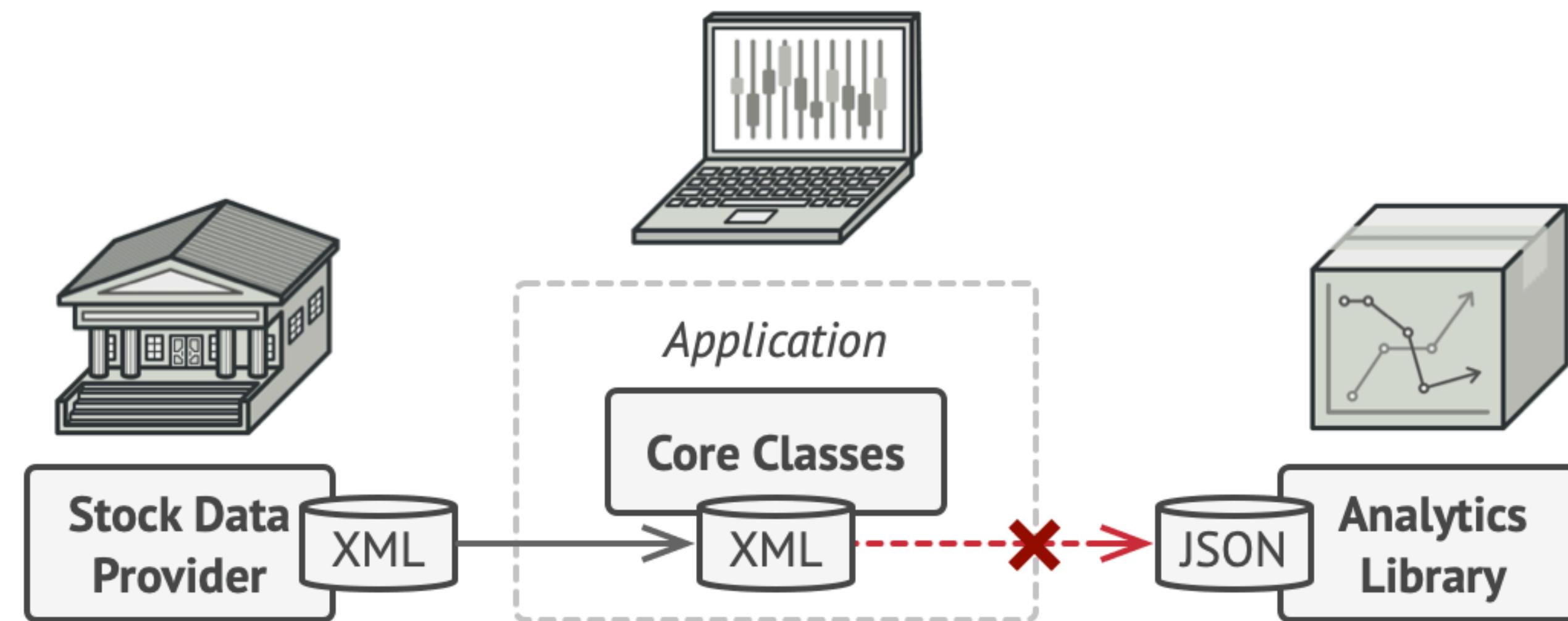
Quelle: <https://refactoring.guru/design-patterns/adapter>

Demo



Zweck

- Anpassung der Schnittstelle einer Klasse an ein anderes erwartetes Interface
- ermöglicht Zusammenarbeit von Klassen, die aufgrund von Inkompatibilität nicht dazu in der Lage wären



You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

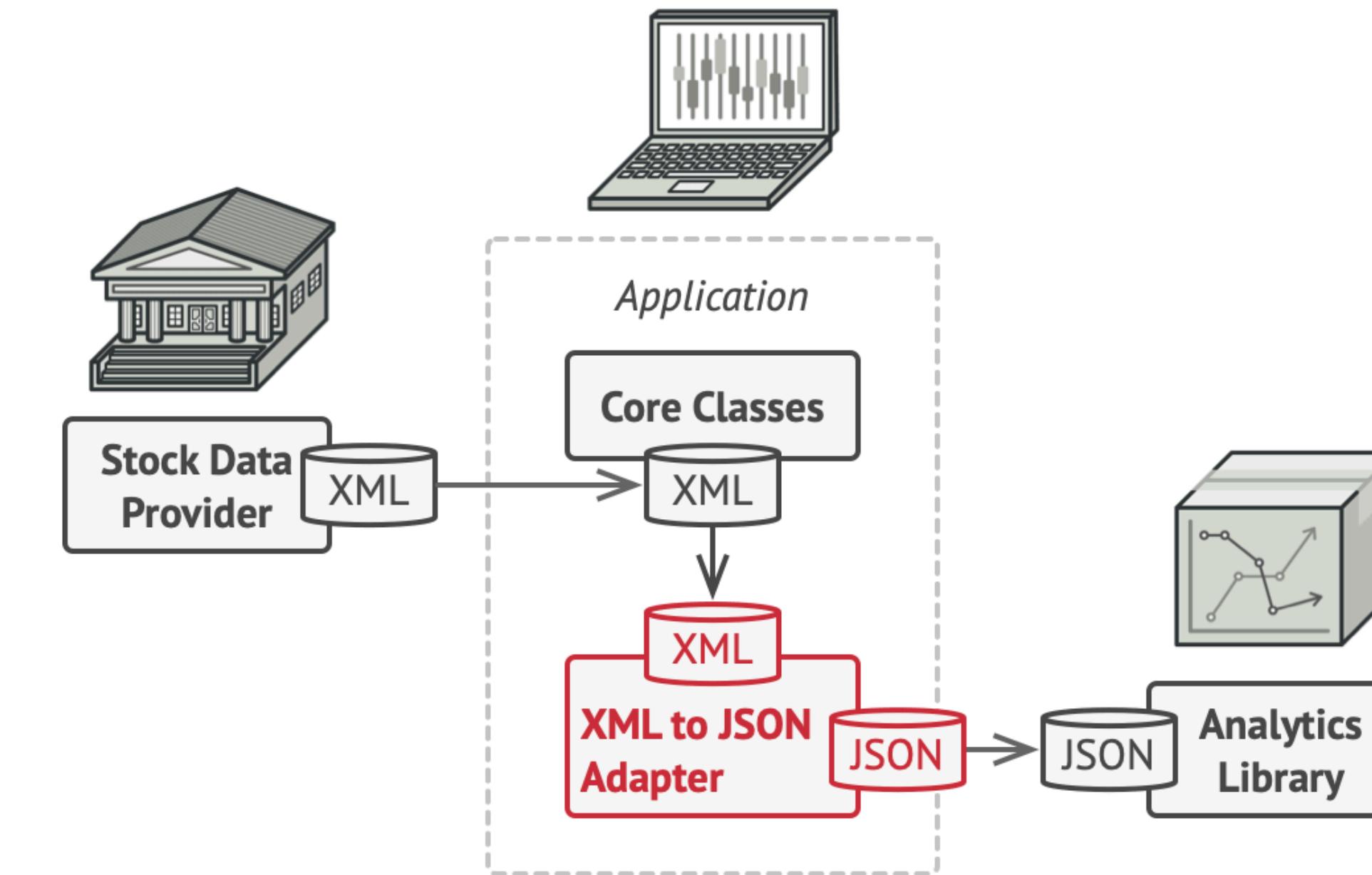
Anwendbarkeit

- wenn Schnittstelle verwendet werden soll, die den aktuellen Anforderungen nicht entspricht
- wenn wiederverwendbare Klasse erzeugt wird, die mit vorhandenen Klassen oder Schnittstellen nicht kompatibel ist
- wenn Schnittstellen der Basisklassen adaptiert werden sollen
- Klassenadapter -> falls Mehrfachvererbung möglich
- ansonsten Objektadapter

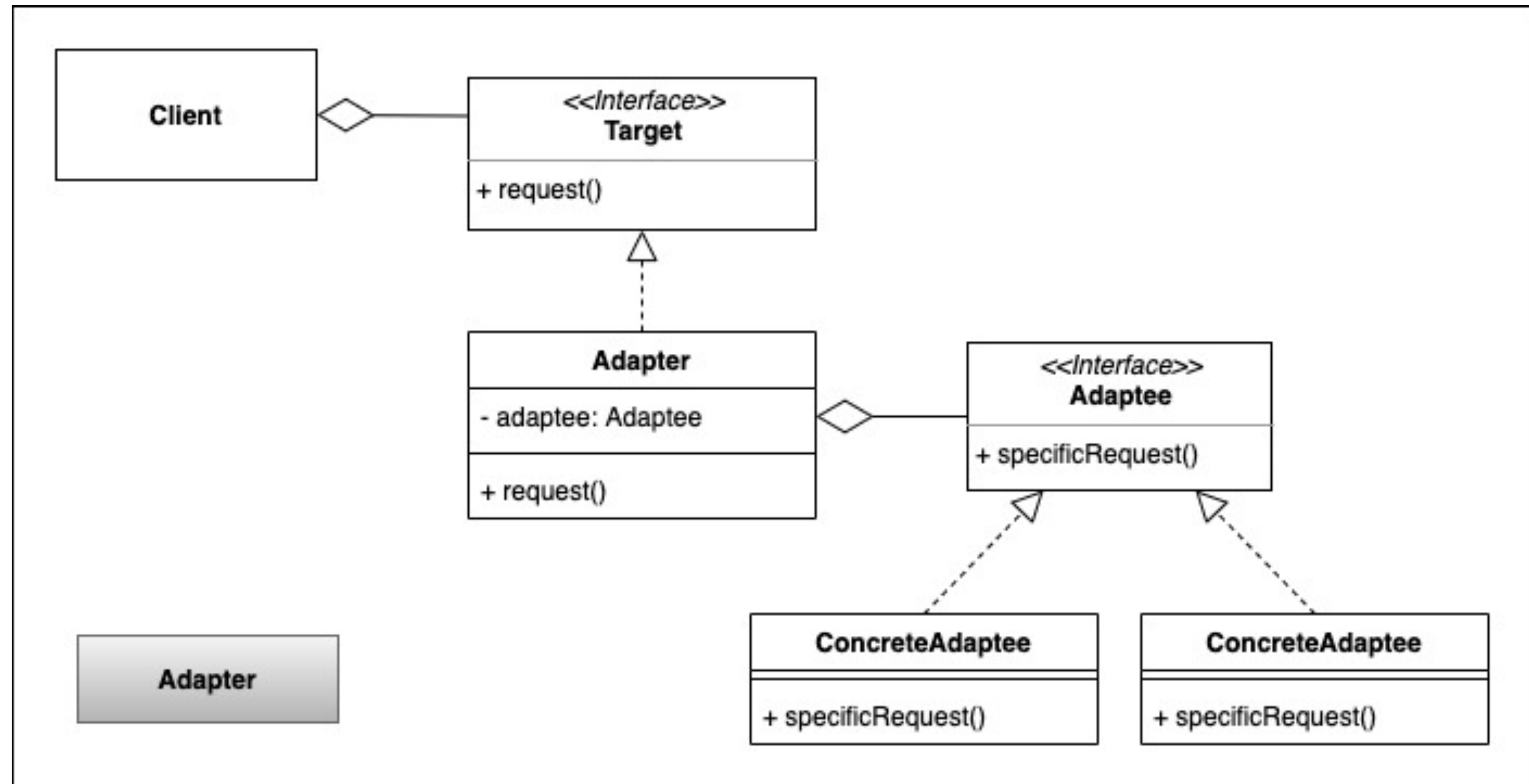
Konsequenzen (Vor- und Nachteile)

- (Klassenadapter) funktionieren nicht, wenn Klassen mitsamt ihren Unterklassen adaptiert werden sollen, da das adaptierte Objekt durch Zuweisung die Adaptee-Klasse an die Target-Schnittstelle angepasst wurde
- (Klassenadapter) nutzt nur 1 Objekt
- (Objektadapter) ermöglicht einzelnen Adapter die Zusammenarbeit mit mehreren adaptierten Objekten
- erschwert das Überschreiben des Verhaltens des Adapter-Objektes

- unterschiedlicher Anpassungsaufwand beim Einsatz eines Adapters (von gering bis komplex)
- Steckbare Adapter - direkte Integrierung in wiederzuverwendenden Klassen derselben Schnittstelle
- 2-Wege-Adapter, falls unterschiedliche Clients gleiche Transparenz in beide Richtungen benötigen

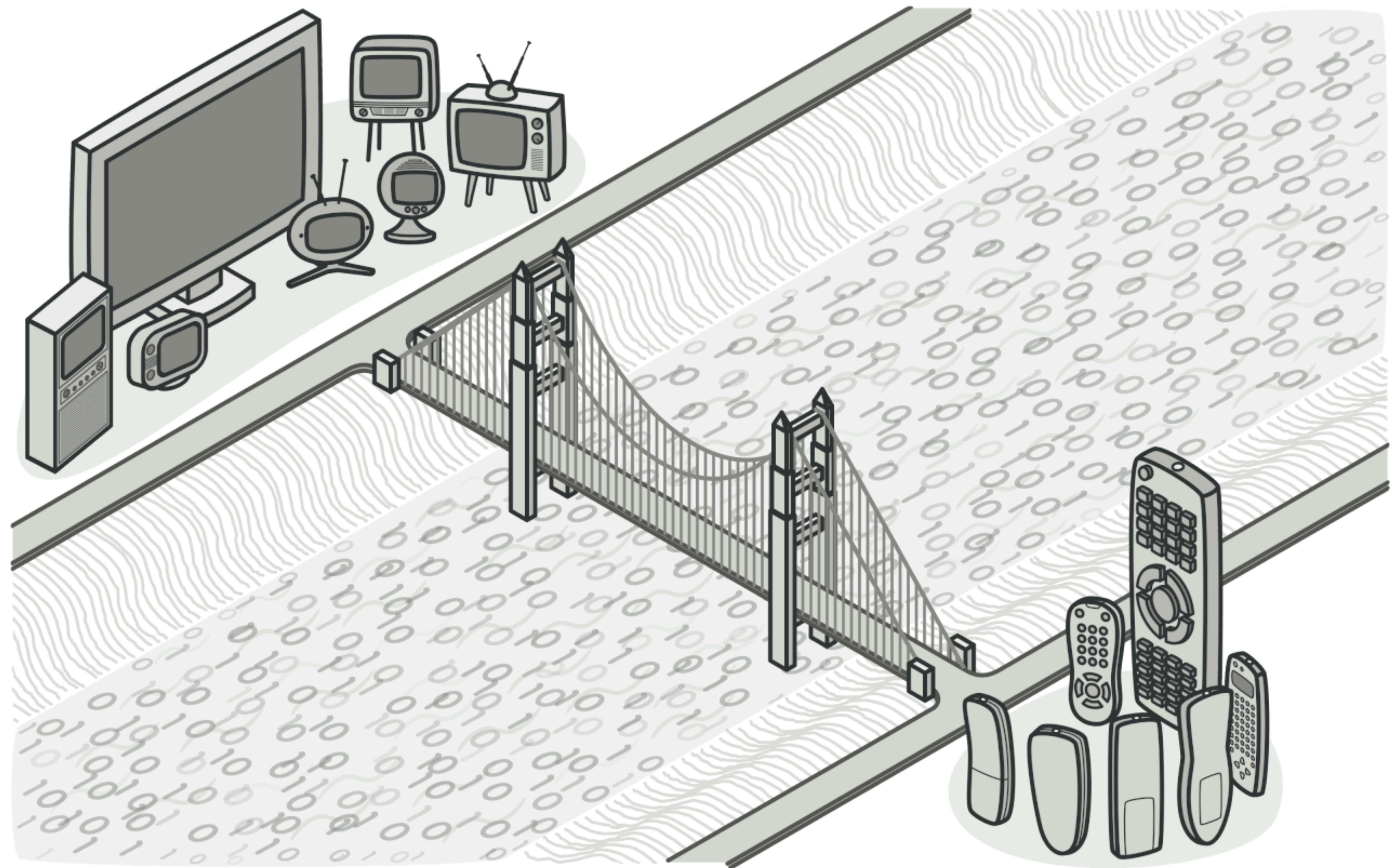


Resource: <https://refactoring.guru/images/patterns/diagrams/adapter/solution-en.png>



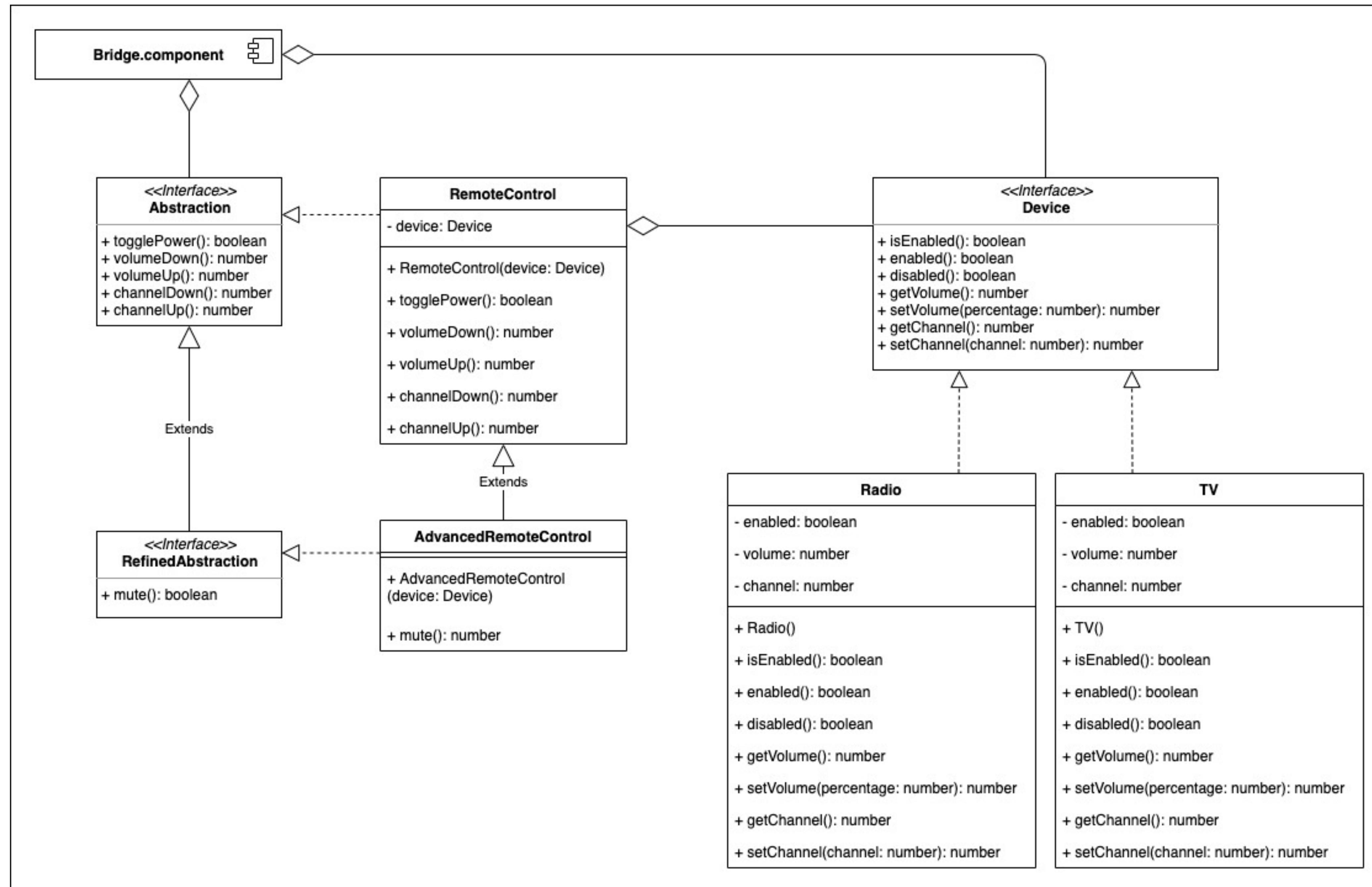
Code-Beispiel

Bridge Brücke



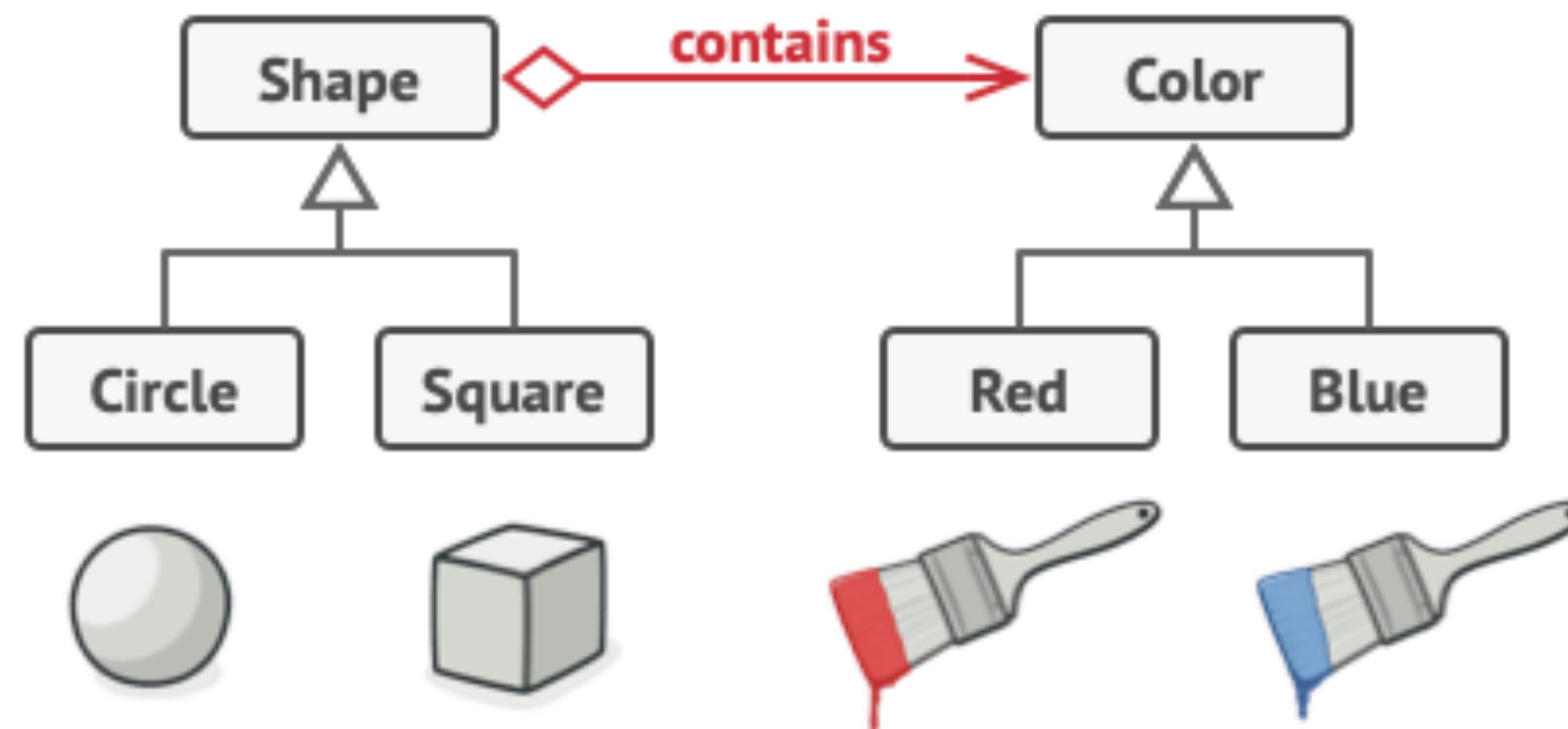
Quelle: <https://refactoring.guru/design-patterns/bridge>

Demo



Zweck

- Entkopplung einer Abstraktion von ihrer Implementierung
- unabhängige Variation von Abstraktion und Implementierung



Resource: <https://refactoring.guru/images/patterns/diagrams/bridge/solution-en.png>

Anwendbarkeit

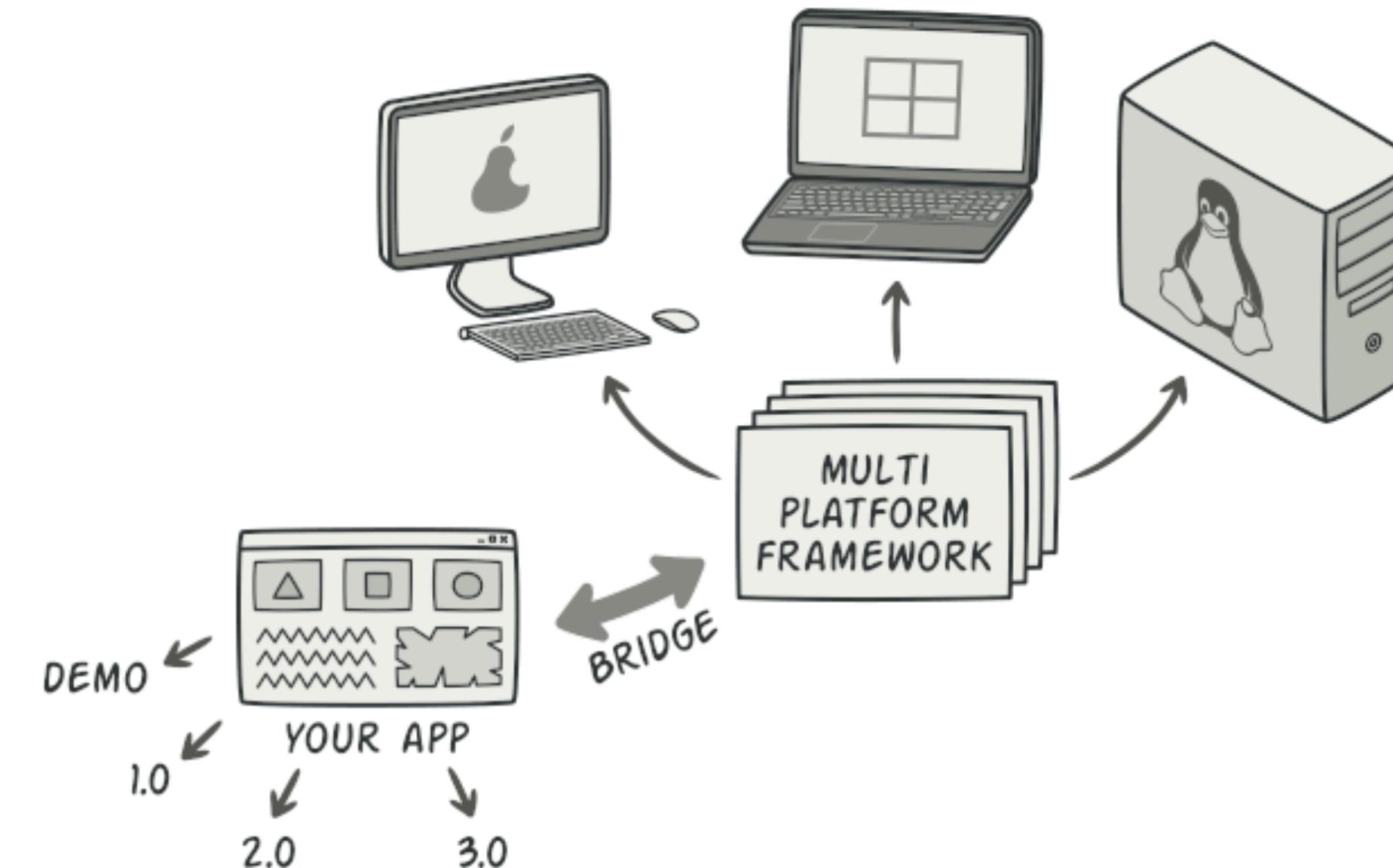
- wenn permanente Bindung zwischen Abstraktion und Implementierung verhindert werden soll (ermöglicht laufzeitbedingte Wechsel der Implement.)
- wenn Abstraktion und Impl. durch Unterklassen erweitern werden sollen
- wenn Modifizierungen an der Abstraktion keine Auswirkung auf den Client haben soll
- wenn zahlenmäßiger Klassenanstieg eintritt. *Rumbaugh* bezeichnet solche Klassenhierarchien als “Nested Generalizations”
- wenn Implementierung von mehreren Objekten gemeinsam genutzt werden soll (bsp. Reference Counting)

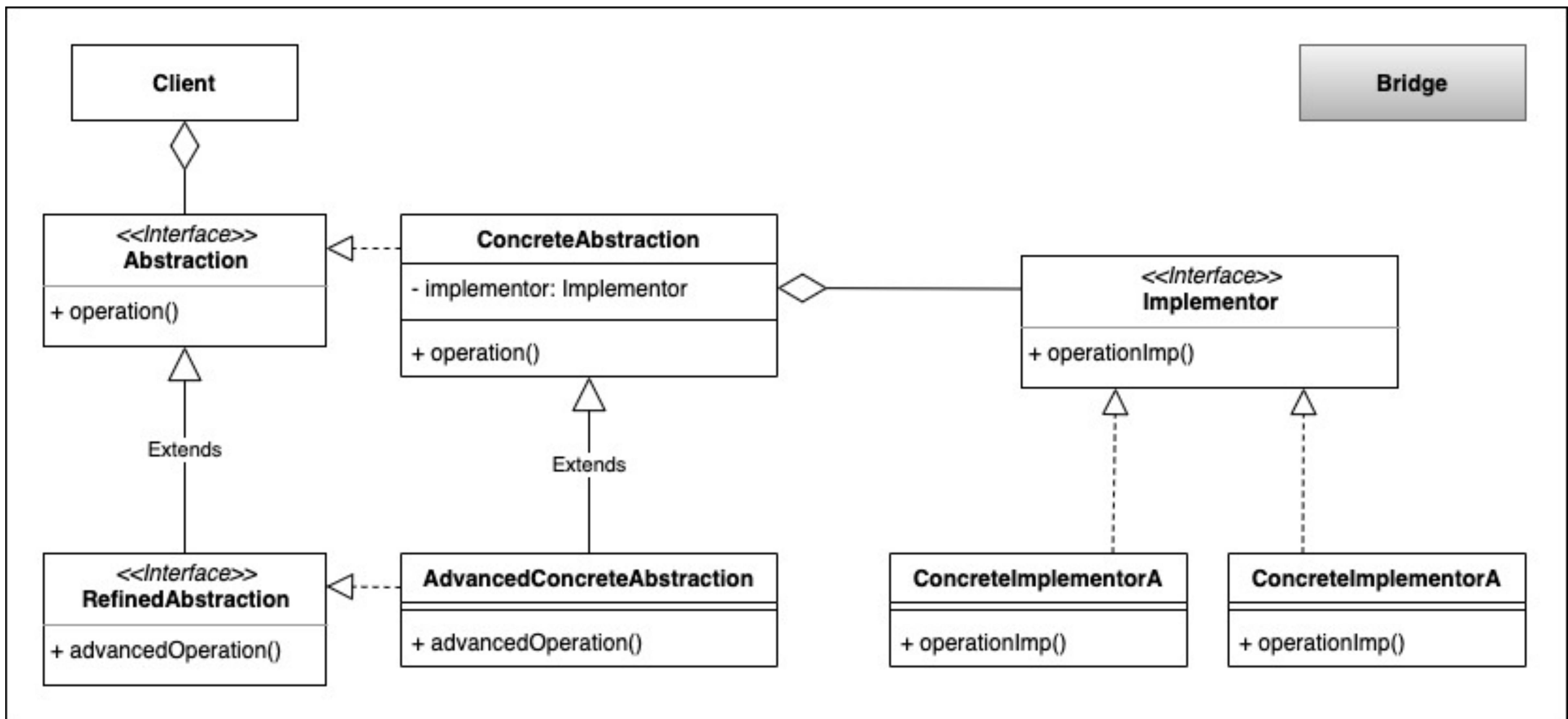
Konsequenzen (Vor- und Nachteile)

- Entkopplung von Schnittstelle und Implementierung: Objekte können zur Laufzeit ihre Implementierung wechseln - hohe Dynamik und Flexibilität zur Laufzeit
- begünstigt *Layering* (Schichtenbildung), was Systemstrukturierung begünstigt
- bessere Erweiterbarkeit, da *Abstraction-* und *Implementor*-Hierarchien unabhängig voneinander **erweitert** werden können
- Verbergen der Implementierungsdetails vor dem Client

Unterschied zum Adapter Pattern

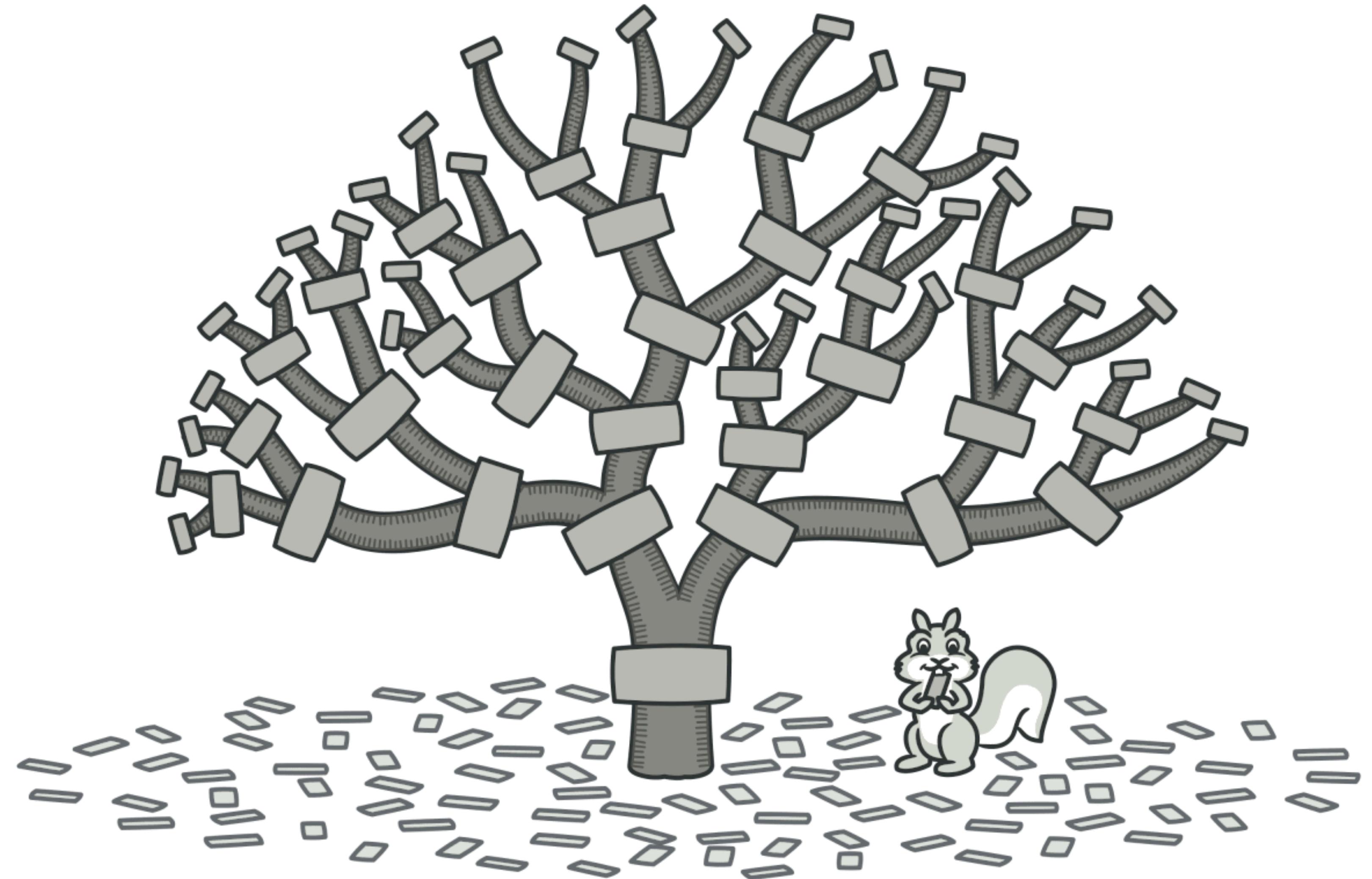
- das Adapter Pattern wird nach Fertigstellung des Systemdesigns angewendet, um die Zusammenarbeit nicht miteinander verwandter Klassen zu ermöglichen (Auflösung von Inkompatibilitäten)
- das Bridge Pattern fließt dagegen von Anfang an in das Systemdesign ein





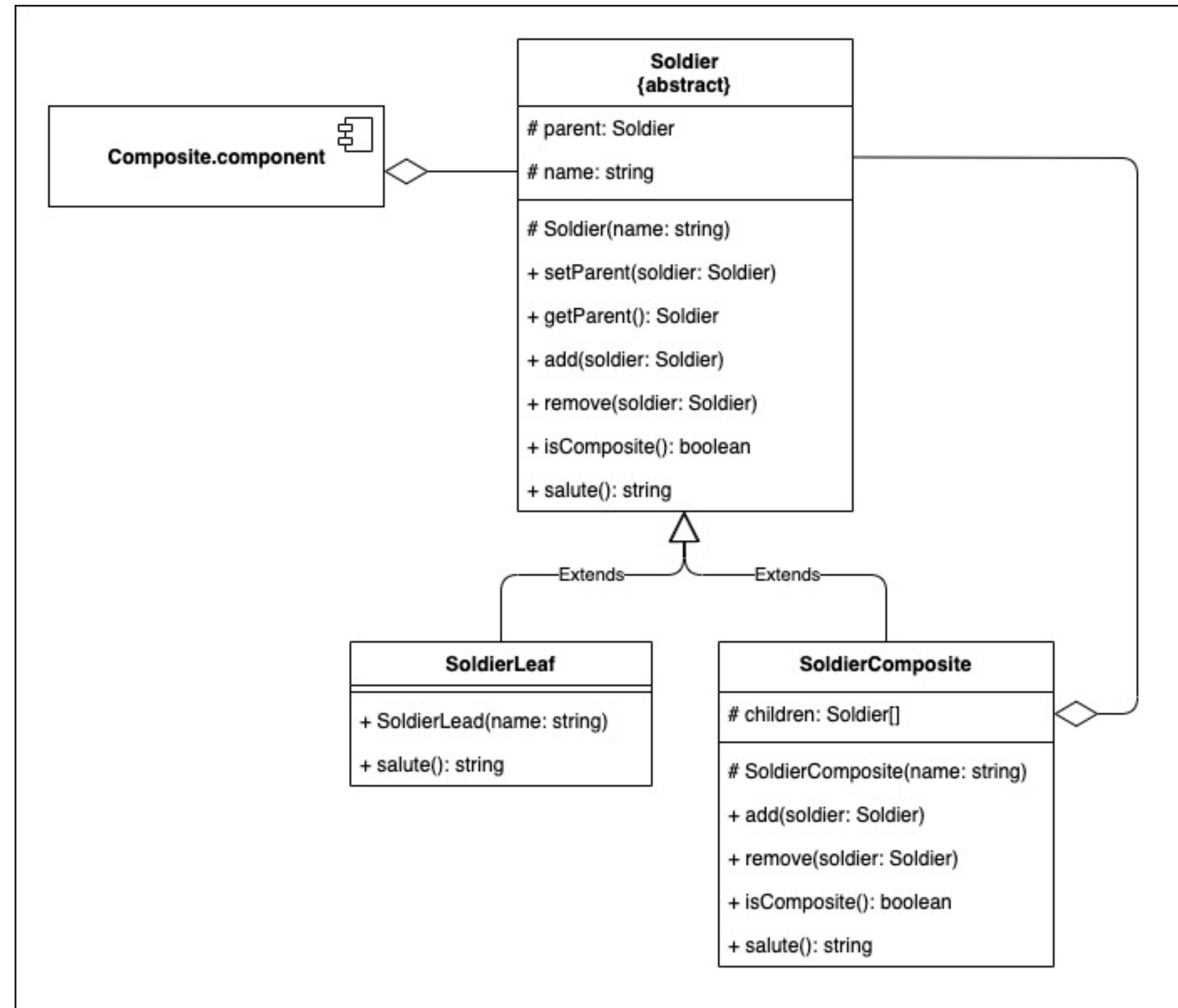
Code-Beispiel

Composite Kompositum



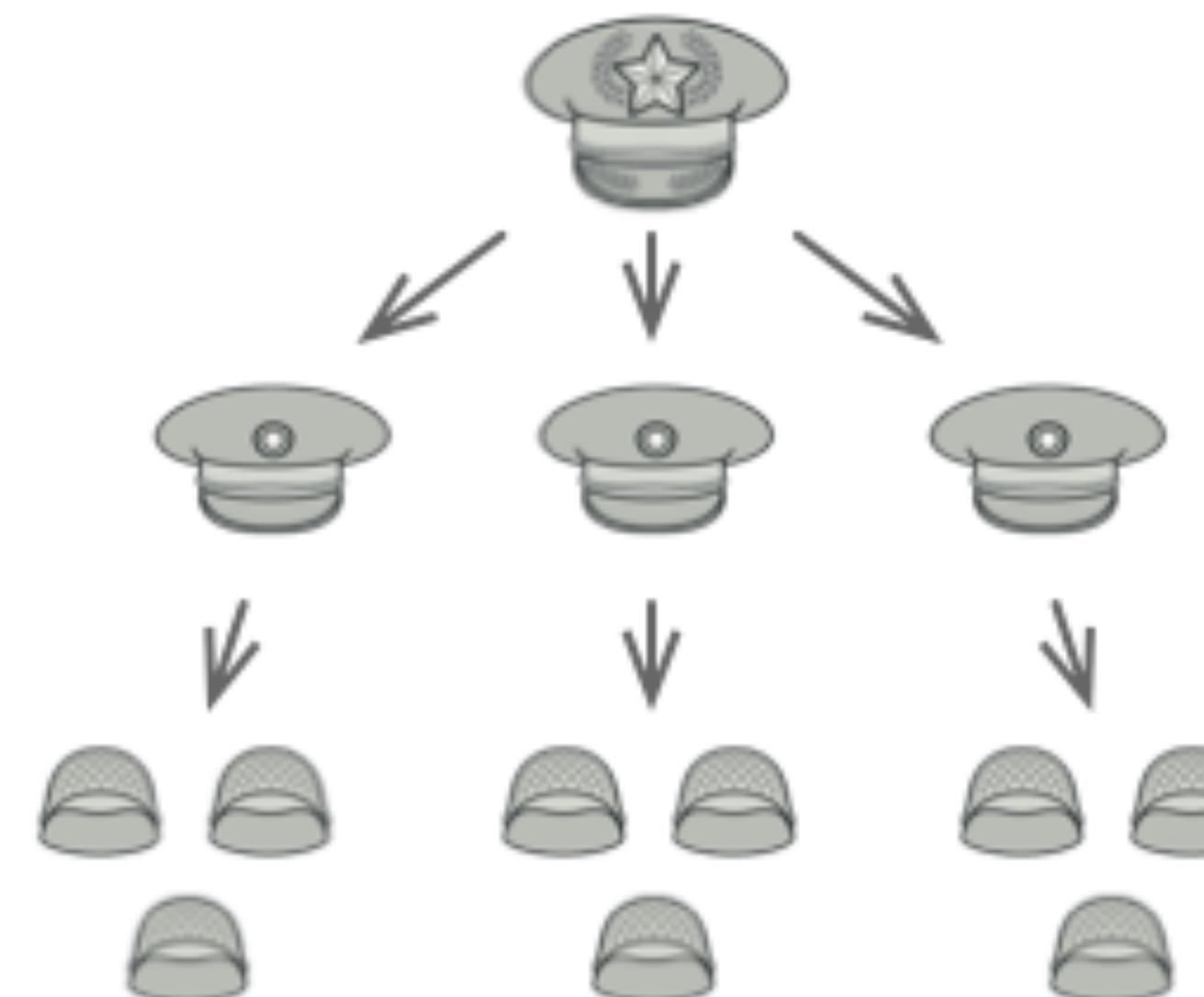
Quelle:<https://refactoring.guru/design-patterns/composite>

Demo



Zweck

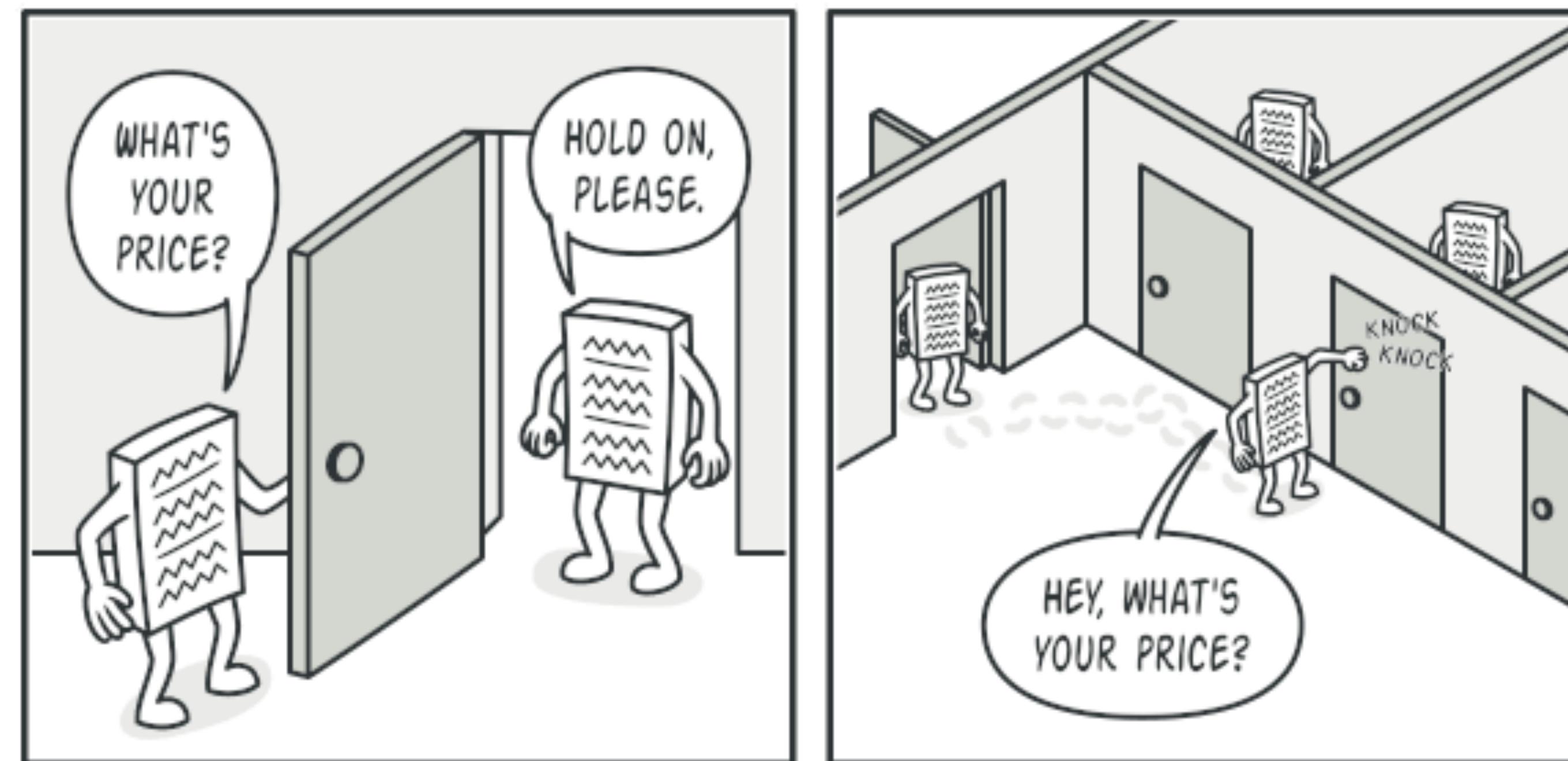
- Komposition von Objekten in Baumstrukturen zur Abbildung von Teil-Ganzes-Hierarchien
- einheitlicher Umgang mit individuellen Objekten als auch mit Objektkompositionen



Resource: <https://refactoring.guru/images/patterns/diagrams/composite/live-example.png>

Anwendbarkeit

- wenn Teil-Ganzes-Hierarchien von Objekten dargestellt werden sollen
- wenn der Client nicht zwischen individuellen Objekten und Objektkompositionen unterscheiden soll (sondern einheitliche Behandlung)

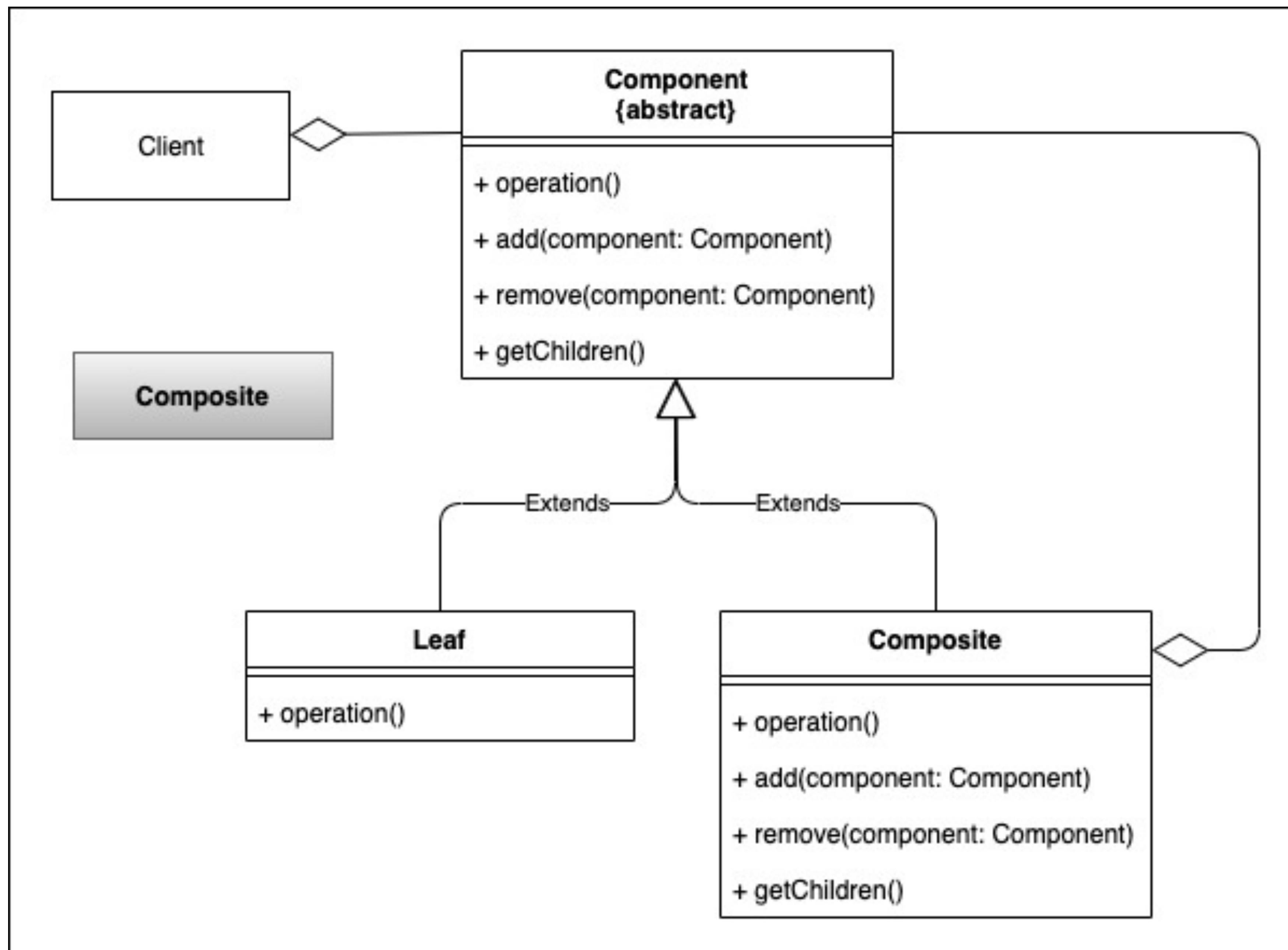


Konsequenzen (Vor- und Nachteile)

- definiert aus zusammengesetzten Objekten bestehende Klassenhierarchien
- Client weiß nicht, ob es sich um ein individuelles Objekt (*Leaf*) oder zusammengesetzte Komponente (*Component*) handelt
- erleichtert das Hinzufügen neuer Komponententypen. Neu definierte *Composite*- oder *Leaf*-Klassen arbeiten automatisch mit vorhandenen Strukturen zusammen
- Systemdesign wird allgemeingültig, da die Struktur nicht begrenzt werden kann (nur mit Laufzeitüberprüfungen) im Gegensatz zur Komposition mit bestimmten Komponenten

Erweiterbarkeit

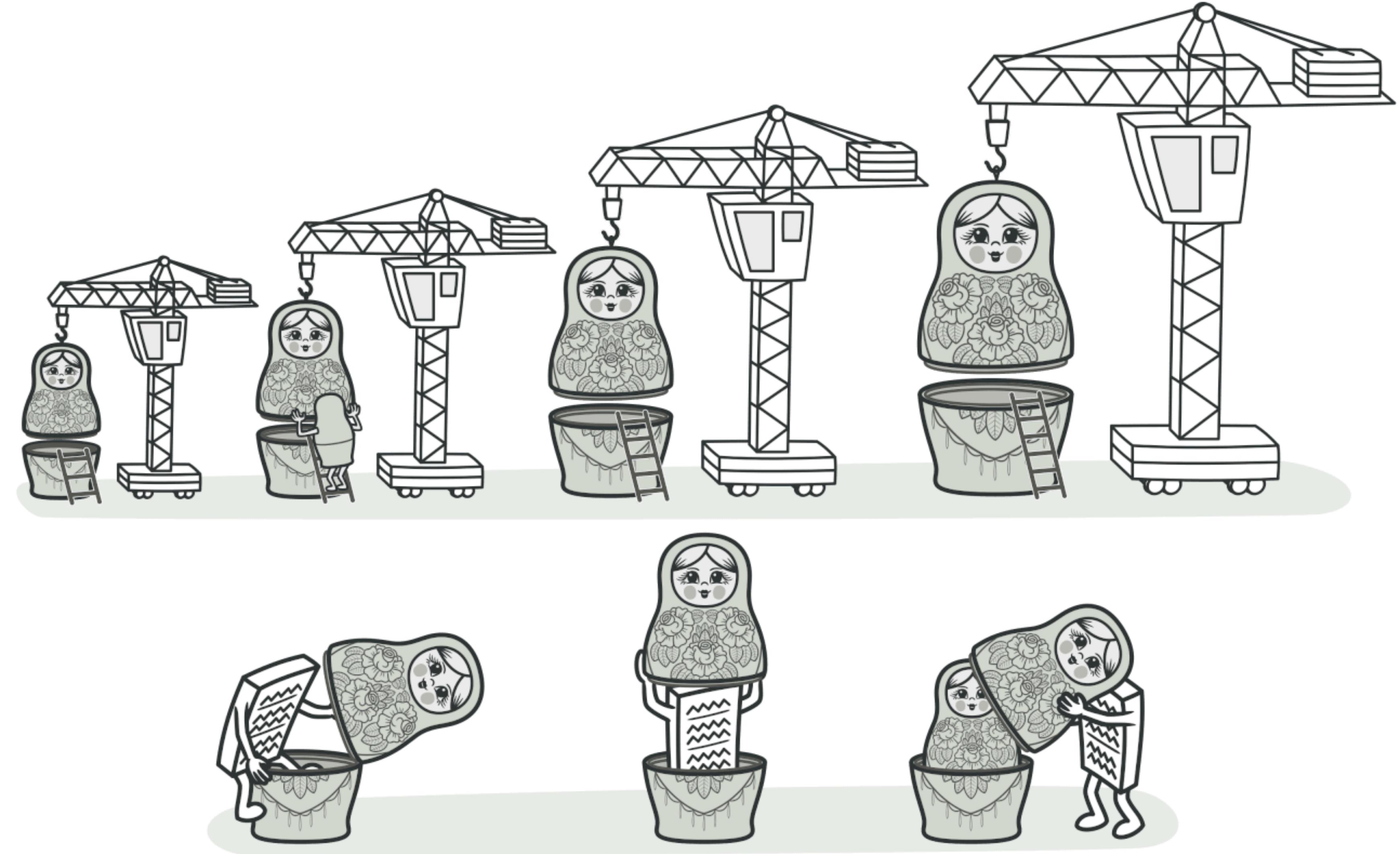
- Explizite Referenzen auf Elternobjekte: einfachere Traversierung und Verwaltung der Struktur durch bessere Navigation
- Gemeinsame Nutzung von Komponente: um Speicherbeanspruchung zu reduzieren (Kombination mit *Singleton* oder *Flyweight*)
- Ordnung der Kindobjekte (z.B. Front-to-back-Ordnung) durch *Iterator*
- Nutzung von Caching bei häufiger Traversierung
- Leaf-Objekt können auch variieren von der Klasseninstanz
- Beispiel: Ordnerstruktur der OS



Code-Beispiel

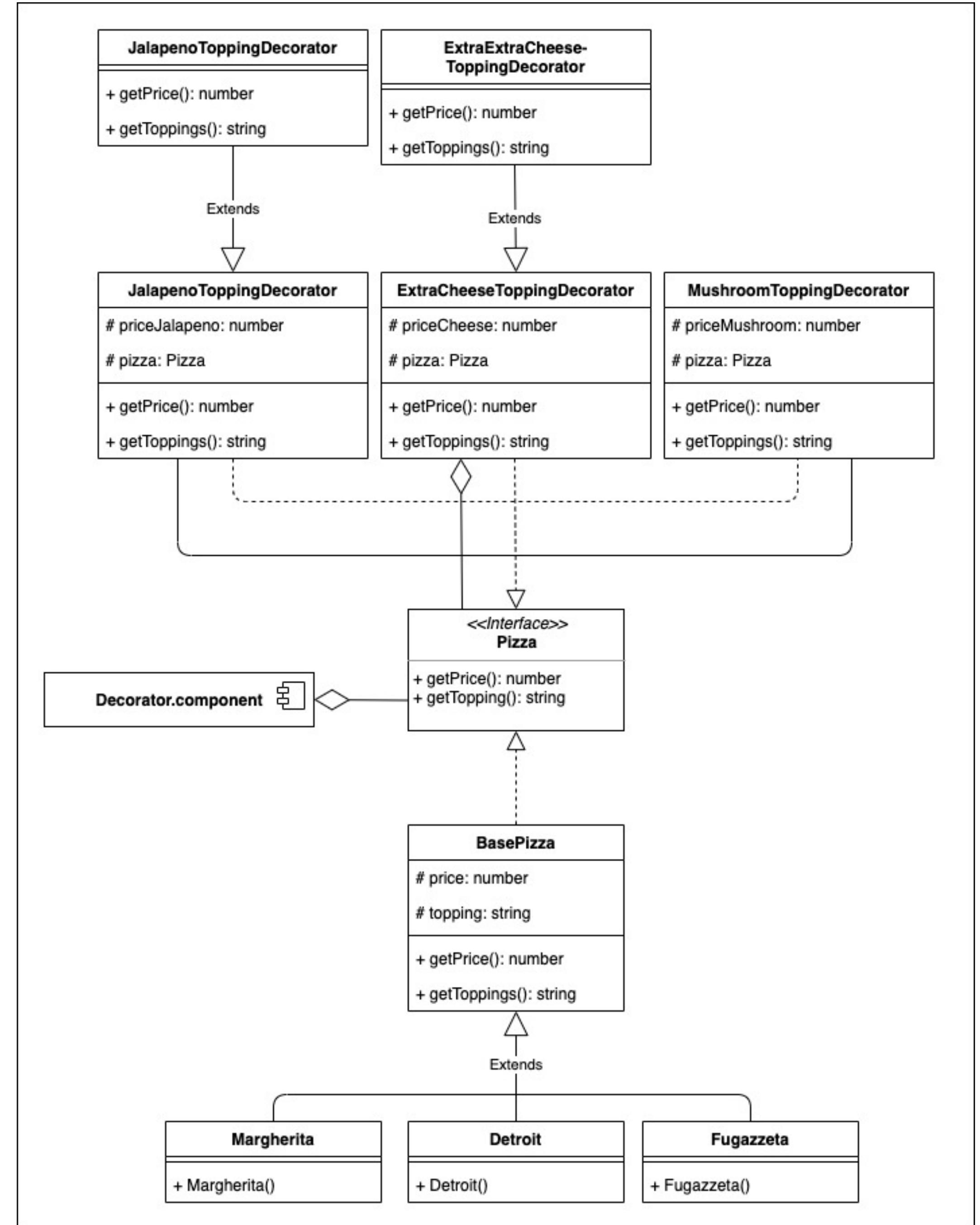
Decorator

Dekorierer



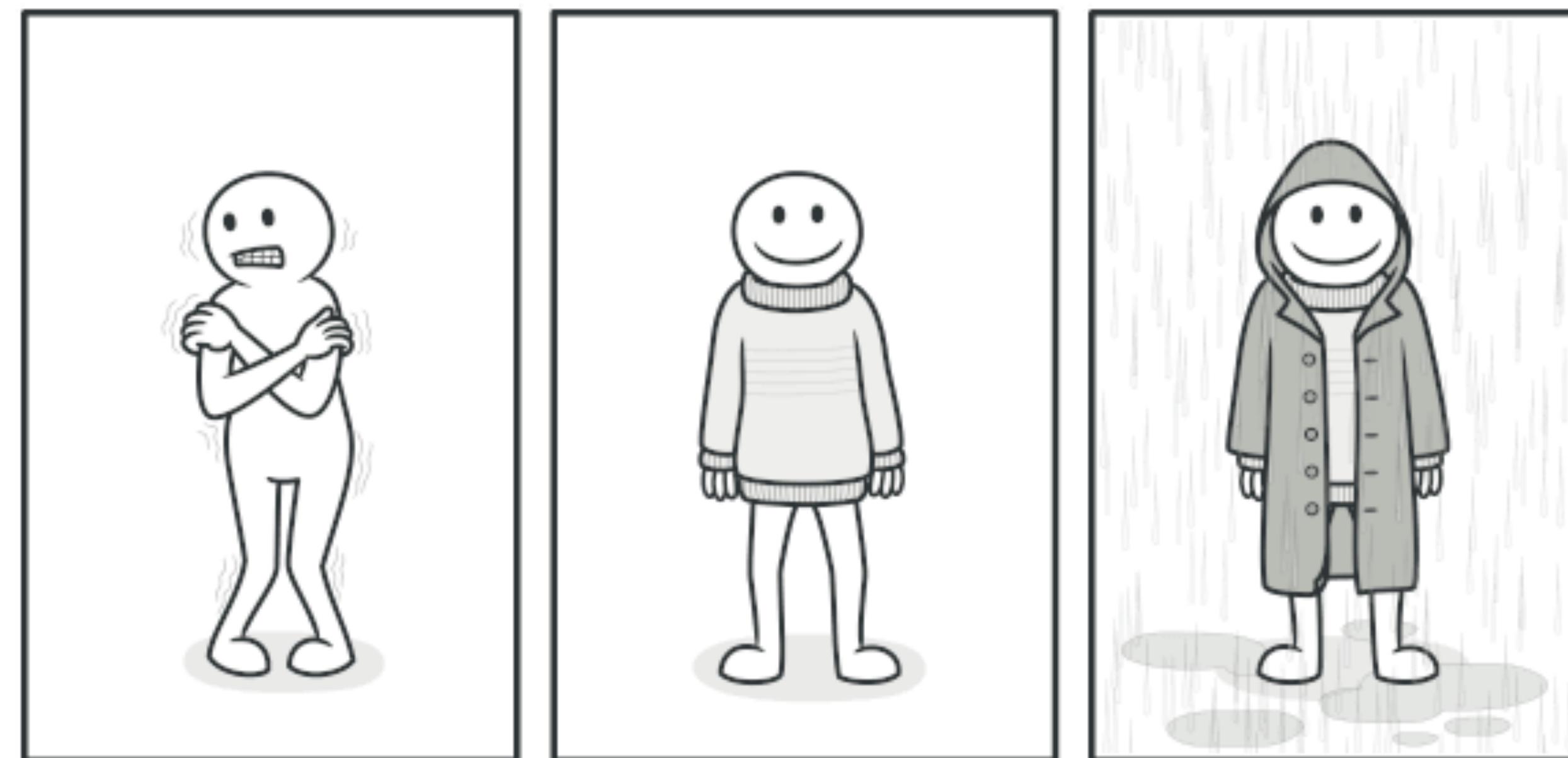
Quelle: <https://refactoring.guru/design-patterns/decorator>

Demo



Zweck

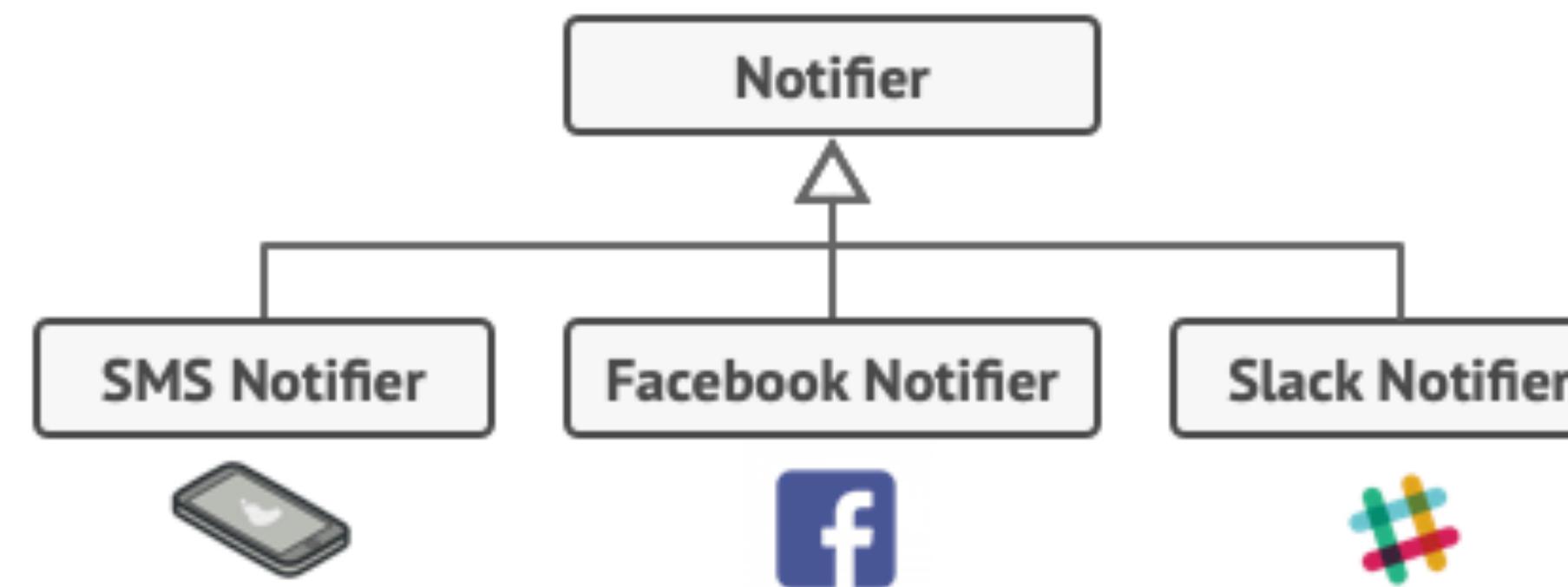
- dynamische Erweiterung der Funktionalität eines Objektes
- flexible Alternative zur Unterklassenbildung
- auch Wrapper genannt



Resource: <https://refactoring.guru/images/patterns/content/decorator/decorator-comic-1.png>

Anwendbarkeit

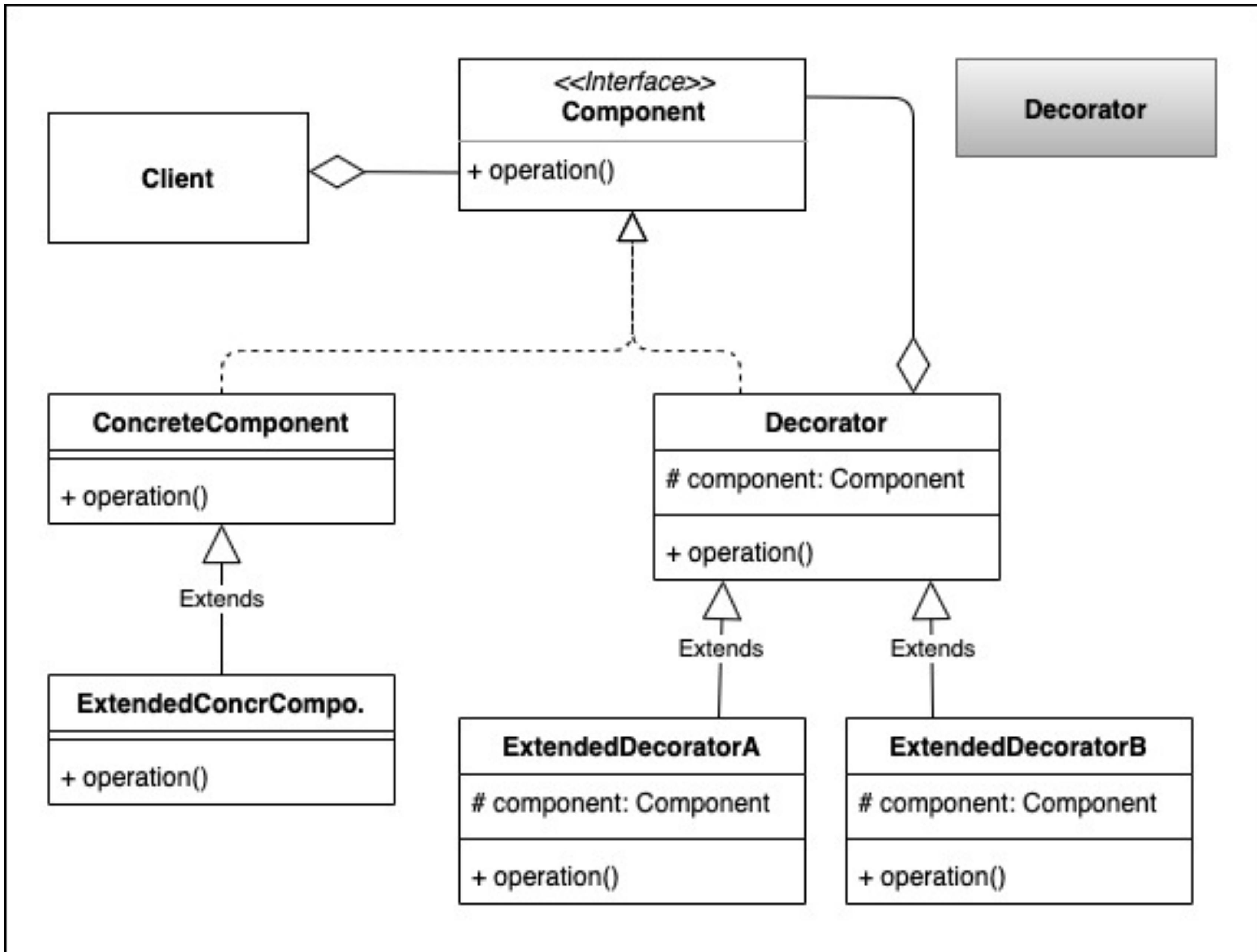
- wenn einzelne Objekte dynamisch mit weiterer Funktionalität ausgestattet werden sollen
- wenn Funktionalität ergänzt und entfernt werden soll
- wenn Funktionserweiterung nicht durch Unterklassenbildung möglich ist - und falls doch, dies zu explosiven zahlenmäßigen Anstieg von Unterklassen führen würde (aufgrund hoher Kombinationsrate)



Resource: <https://refactoring.guru/images/patterns/diagrams/decorator/problem2.png>

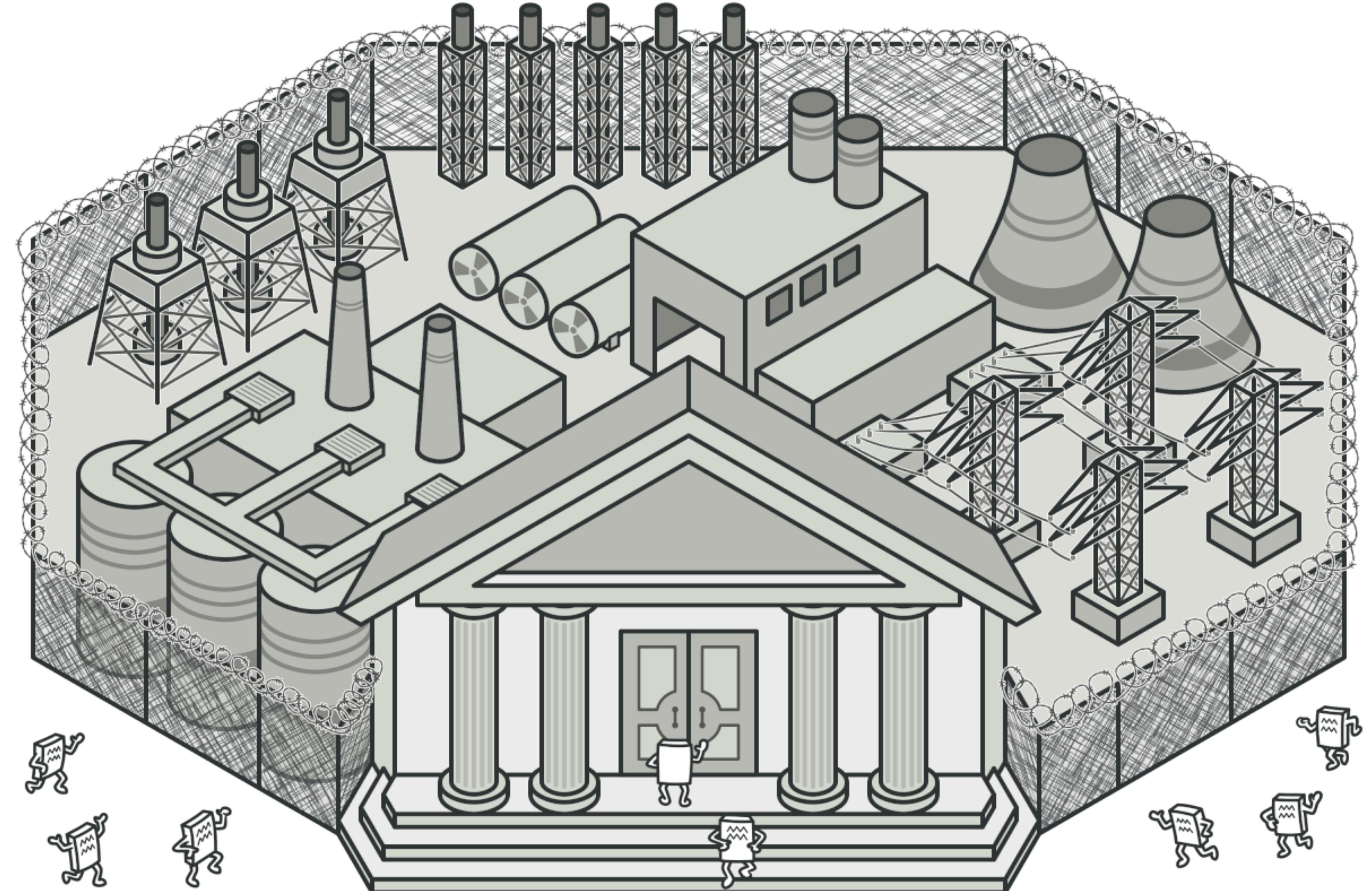
Konsequenzen (Vor- und Nachteile)

- Mehr Flexibilität als bei statischer Vererbung durch dynamisches Hinzufügen oder Entfernen von Funktionen sowie Kombination verschiedener *Decorator*-Objekte (Schokoeis mit Streuseln oder Schokostückchen ;-))
- Vermeidung funktionsüberladener Klassen, da Funktionen erst ergänzt werden können, wenn sie gebraucht werden (durch einfache Kombination) - *YAGNI Principle*
- *Decorator*-Objekt ist nicht mit seiner Komponente identisch, sondern dient ihr als Umhüllung (*Transparent Enclosure*)
- System besteht aus vielen kleinen Objekten, was die klare Übersicht einschränkt



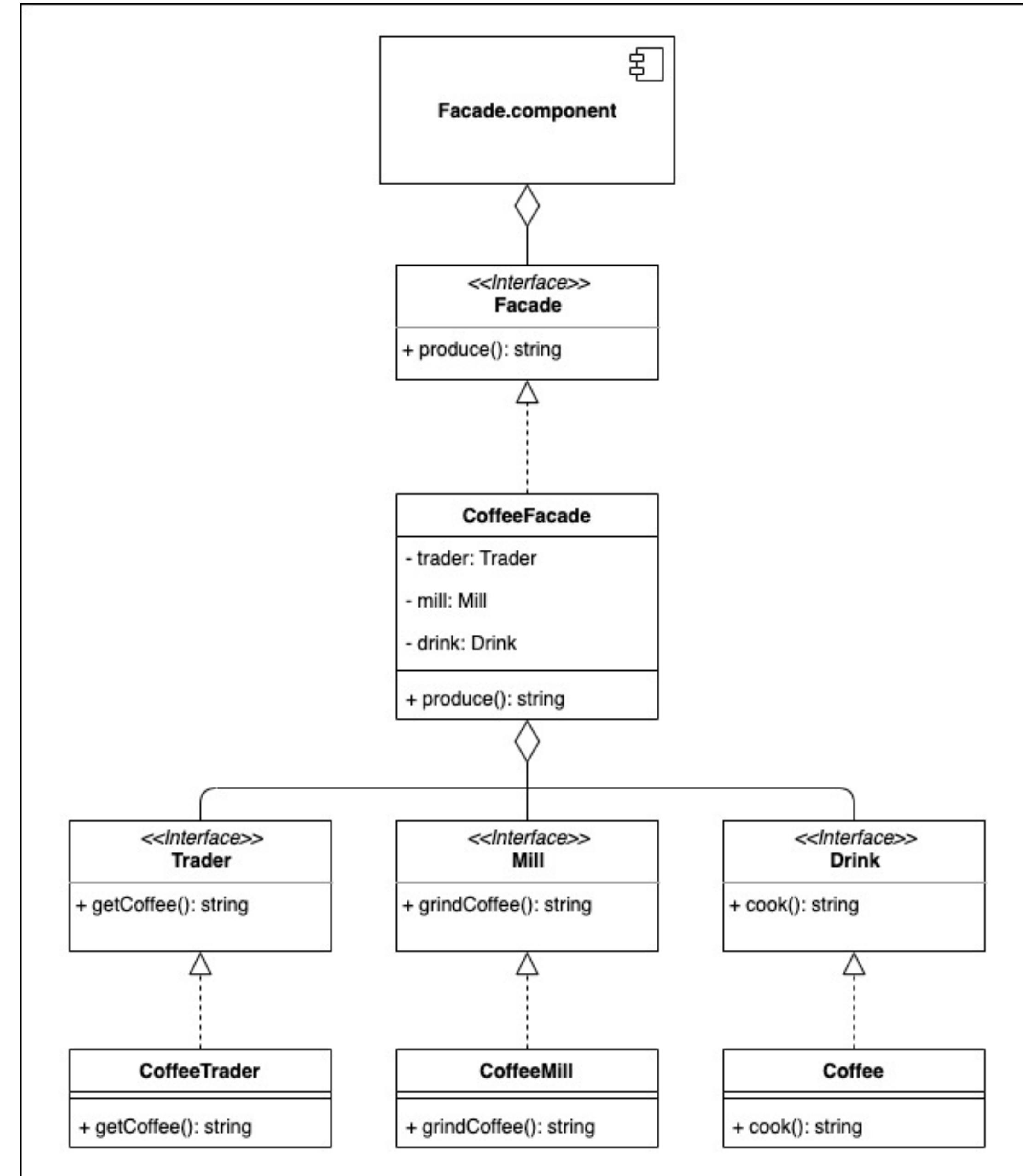
Code-Beispiel

Facade Fassade



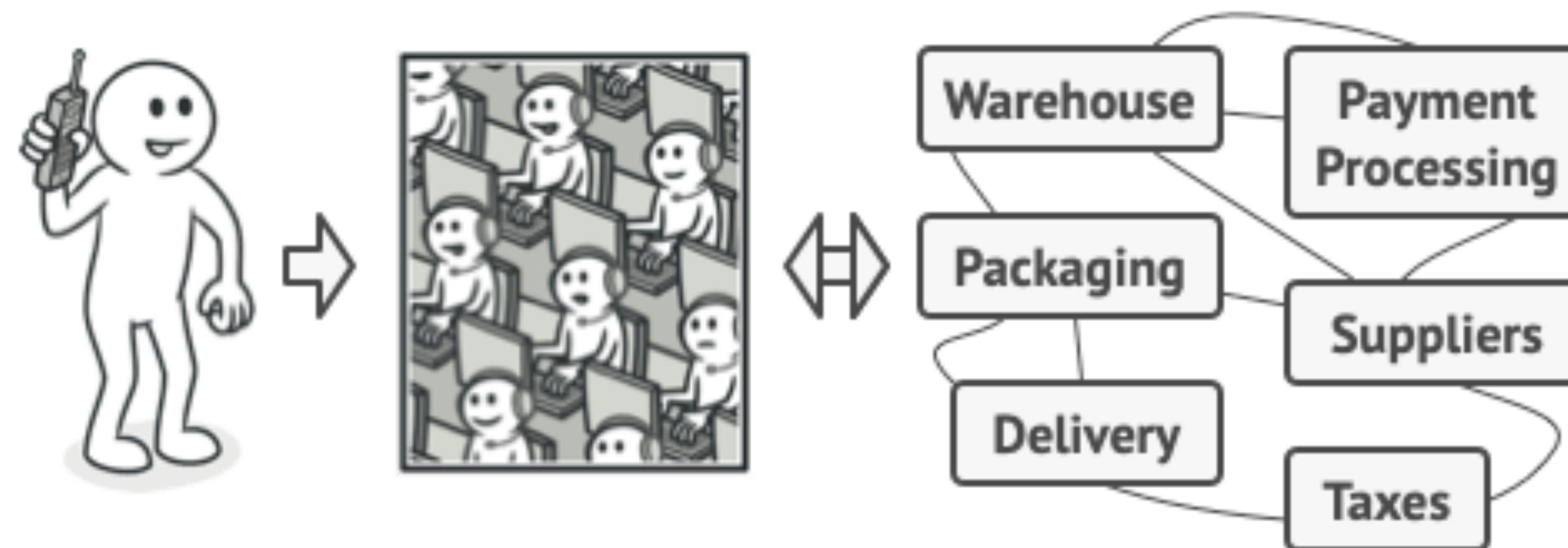
Quelle:<https://refactoring.guru/design-patterns/facade>

Demo



Zweck

- Bereitstellung einer Schnittstelle zu einem Subsystem
- definiert eine Schnittstelle höherer Ebene, um die Nutzung eines Subsystems zu vereinfachen



Resource: <https://refactoring.guru/images/patterns/diagrams/facade/live-example-en.png>

Anwendbarkeit

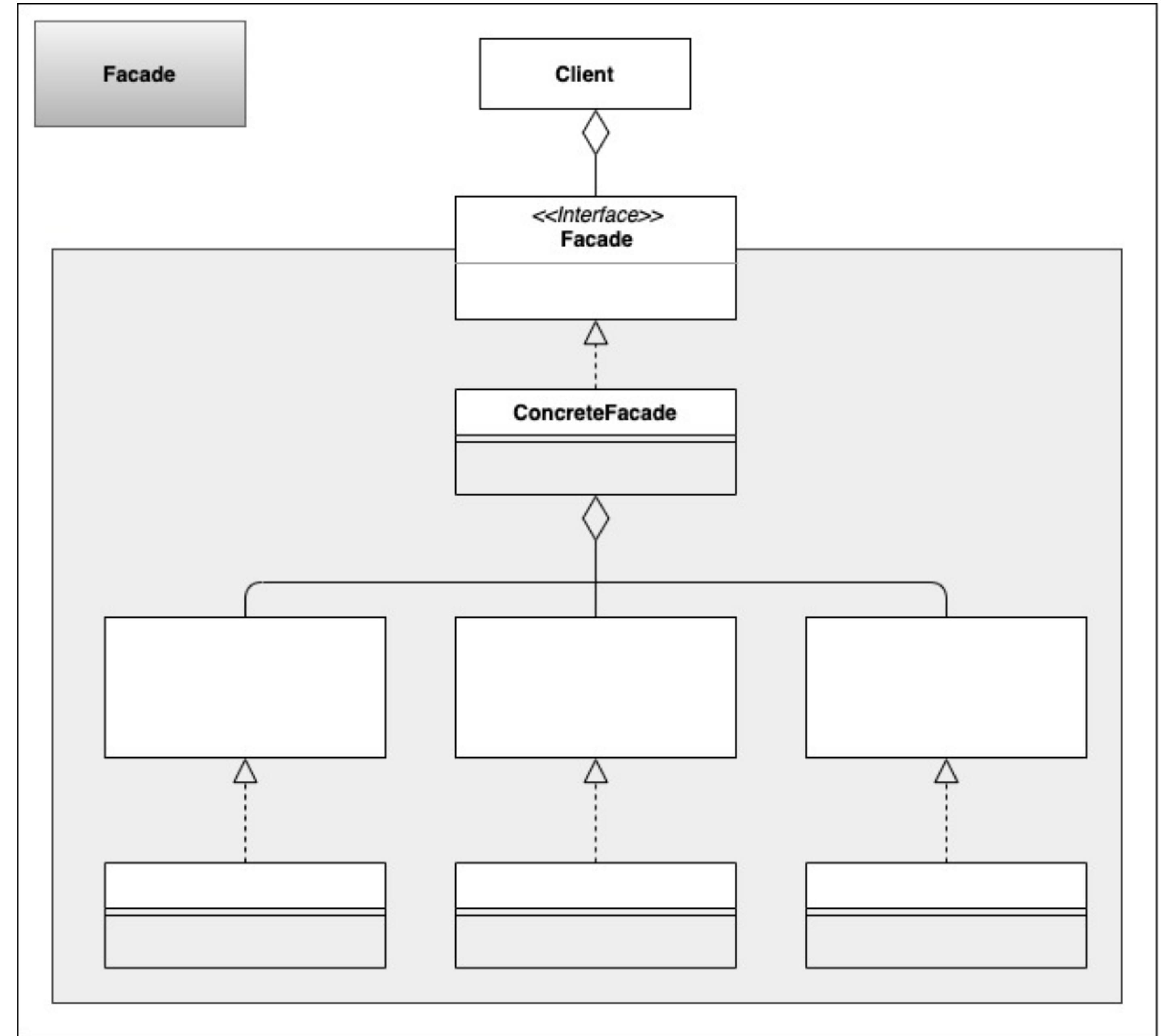
- wenn eine einfache Schnittstelle zu einem komplexen Subsystem bereitgestellt werden soll - solche Subsysteme können im Laufe des Entwicklungsprozesses zunehmend komplizierter werden
- wenn der Client von einem Subsystem entkoppelt werden soll (begünstigt Unabhängigkeit und Portabilität des Subsystems)
- wenn Kommunikation eines Subsystem zum Client oder anderen Subsystemen nur über eine Schnittstelle erfolgen soll

Konsequenzen (Vor- und Nachteile)

- Abschirmung des Clients von den Komponenten des Subsystems - Minimierung der Anzahl der Objekte, die vom Client verwaltet werden müssen
- lose Kopplung zwischen Client und Subsystem; somit können Komponenten im Subsystem problemlos ausgetauscht werden, ohne dass der Client davon Kenntnis hat
- Aufhebung zirkulärer Abhängigkeiten
- Unabhängige Implementierung von Client und Subsystem (geringerer Kompilierungsaufwand)

Verwandte Patterns

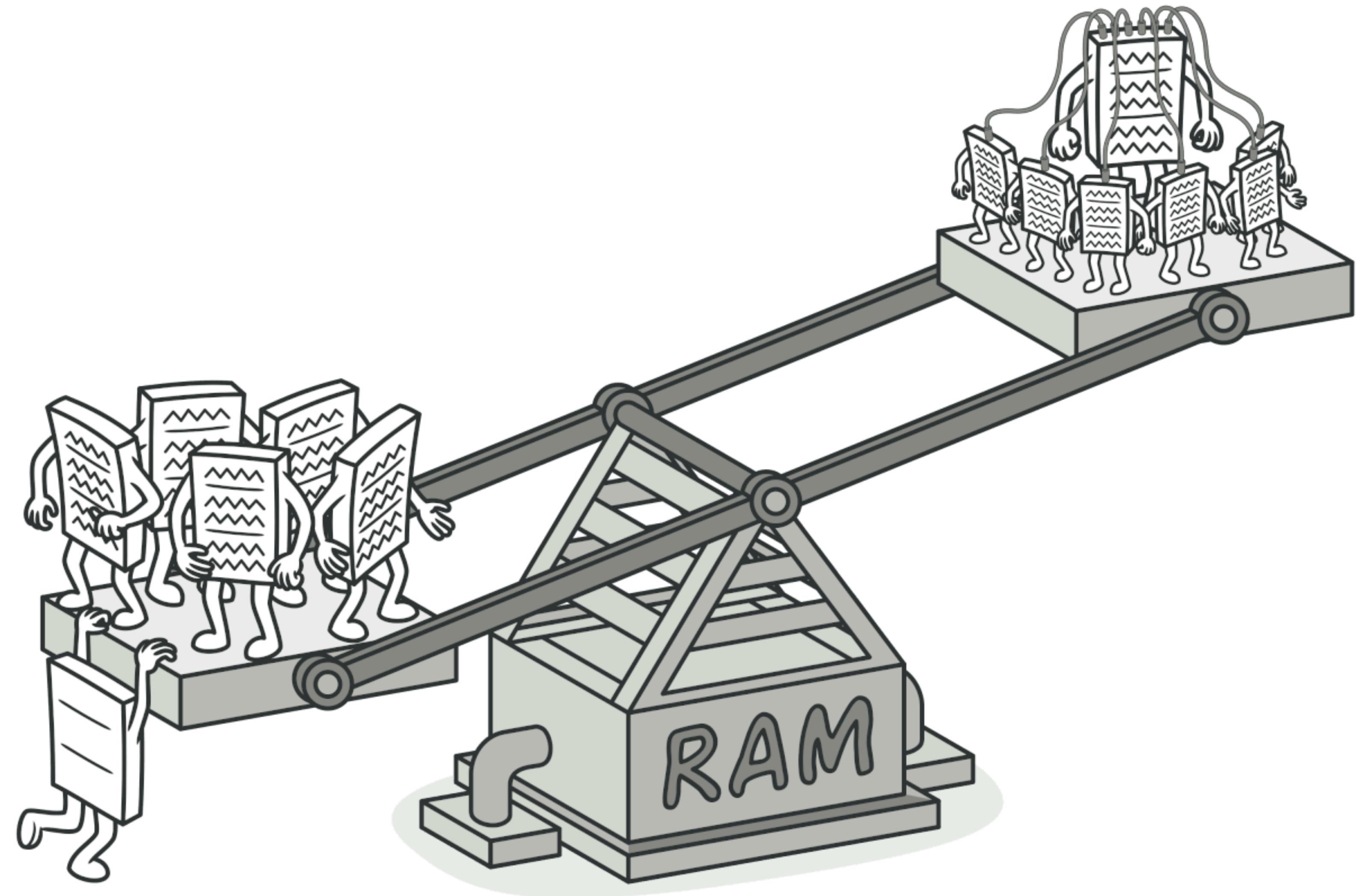
- Wird sehr oft in Kombination sowie als Alternative zur *Abstract Factory* verwendet
- wird oft mit *Mediator* verwechselt bzw. wird ein Kombination aus beiden Patterns gewählt
- Facade ist meistens einzigartig und somit ein *Singleton*



Code-Beispiel

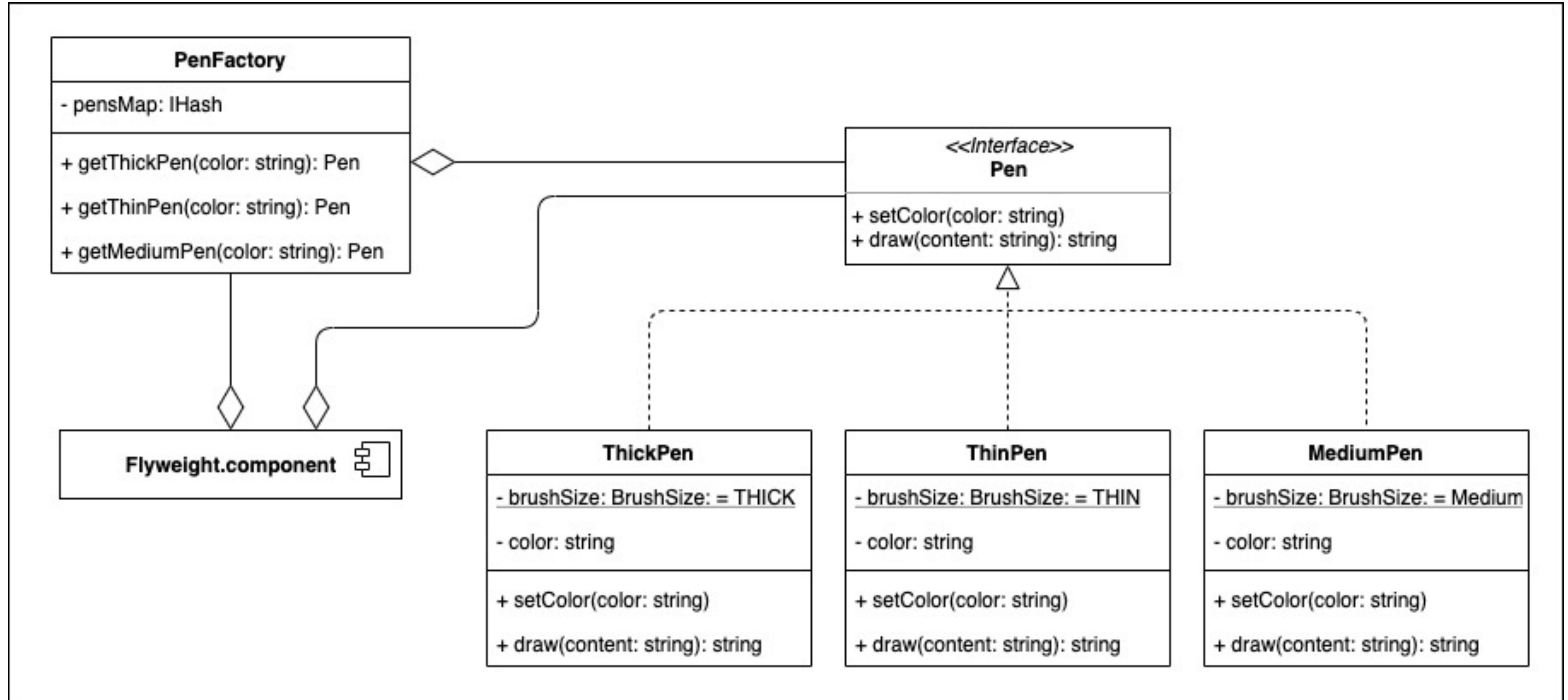
Flyweight

Fliegengewicht



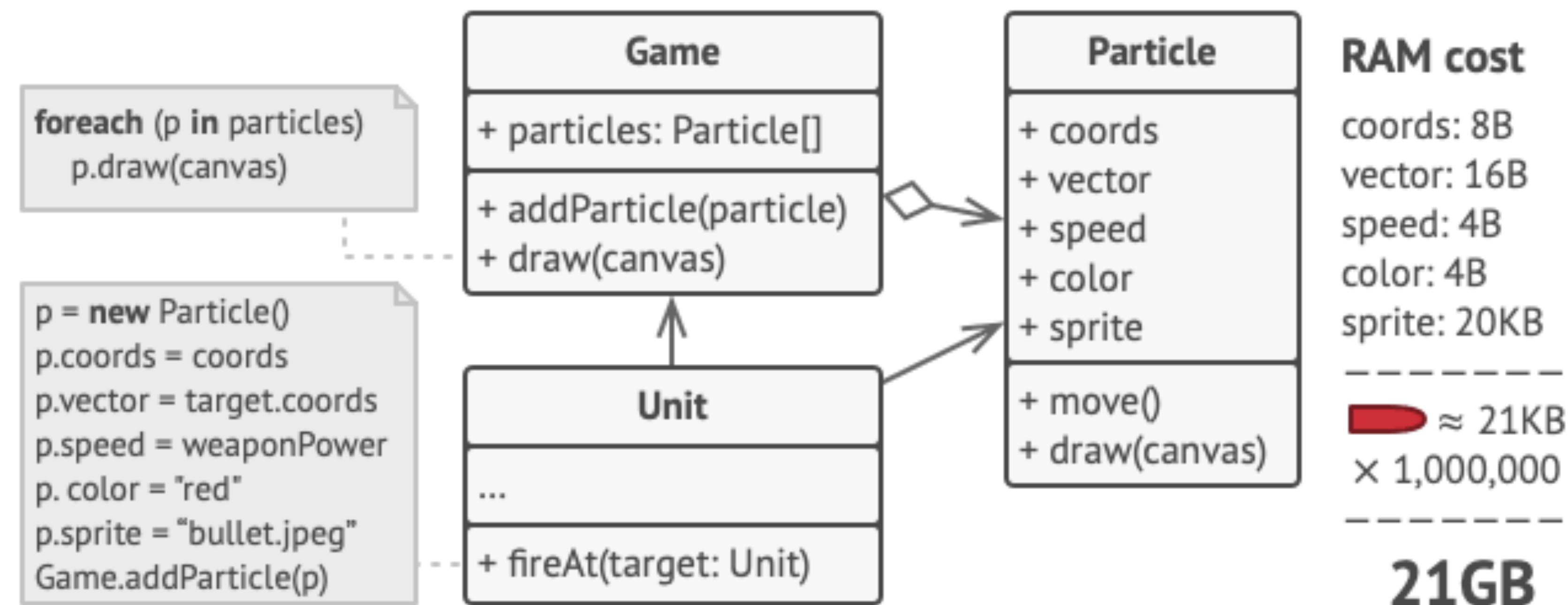
Quelle: <https://refactoring.guru/design-patterns/flyweight>

Demo



Zweck

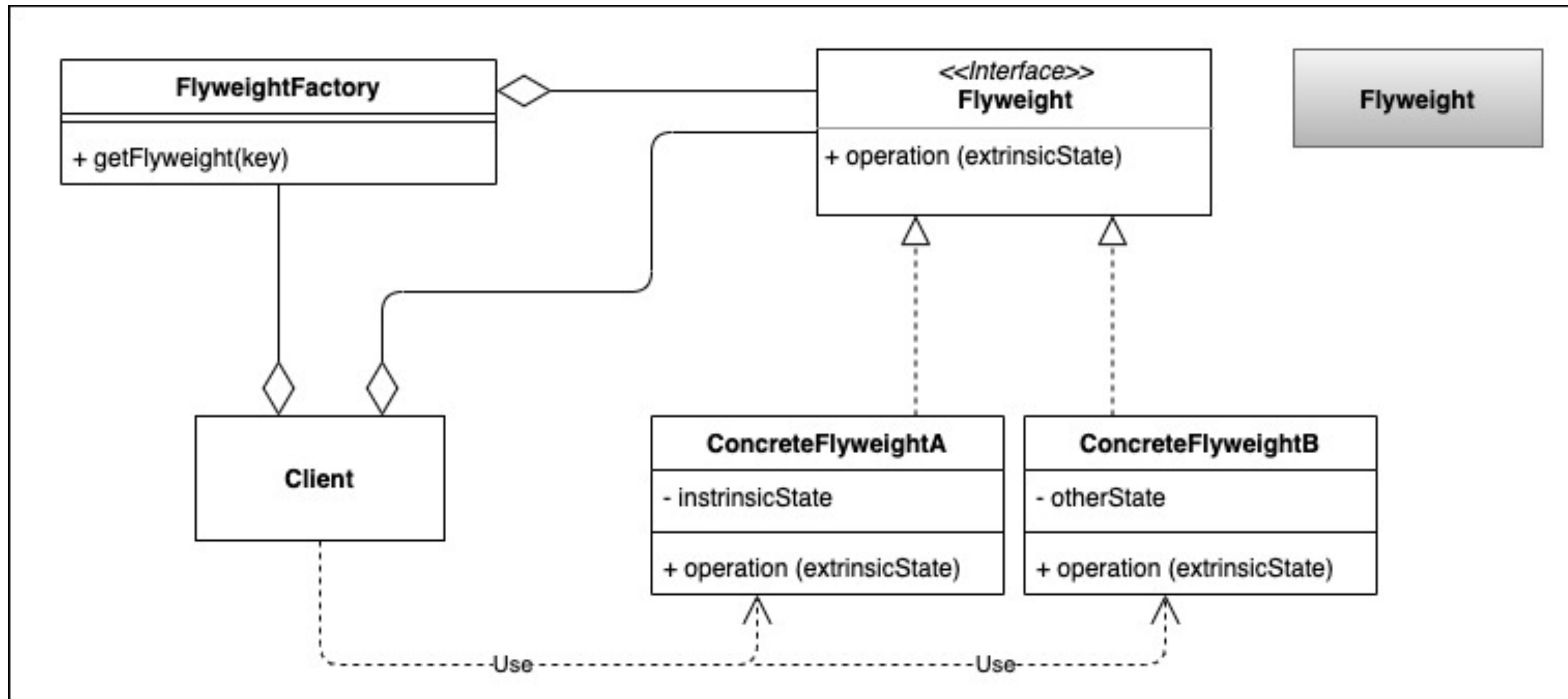
- gemeinsame Nutzung feingranulärer Objekte, um sie in großer Anzahl effizient einsetzen und wiederverwenden zu können



Resource: <https://refactoring.guru/images/patterns/diagrams/flyweight/problem-en.png>

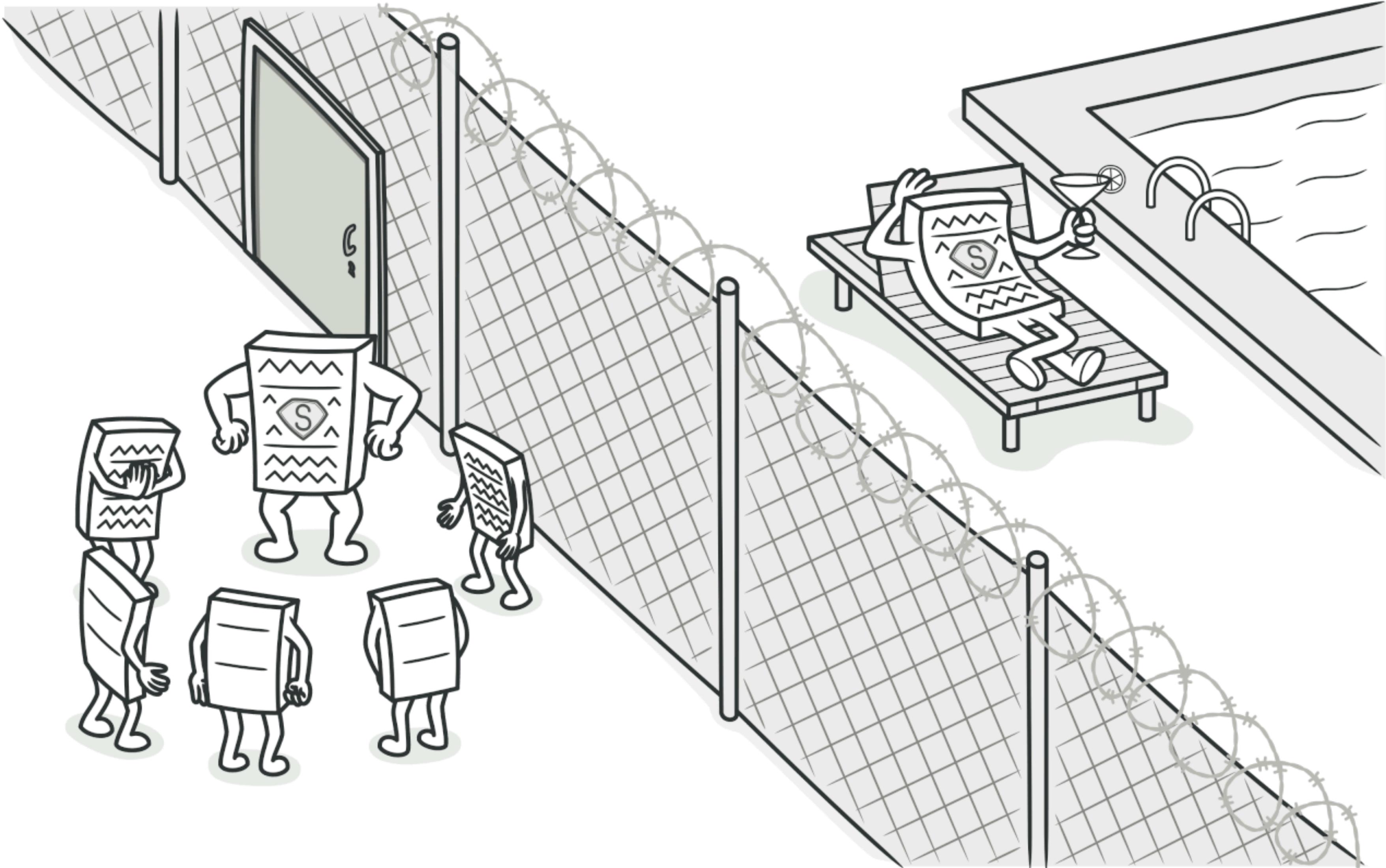
Anwendbarkeit

- sollte nur angewendet werden, wenn alle nachfolgenden Punkte zum eigenen System passen:
- wenn zu hohe Speicheranforderungen aufgrund zahlreicher verwendeter Objekte sind
- wenn Objektzustand extrinsisch gemacht werden kann
- wenn viele Objektgruppen nach Beseitigung des extrinsischen Zustands durch wenig gemeinsam genutzte Objekte ersetzt werden können
- wenn die Anwendung nicht von der Objektidentität abhängig ist



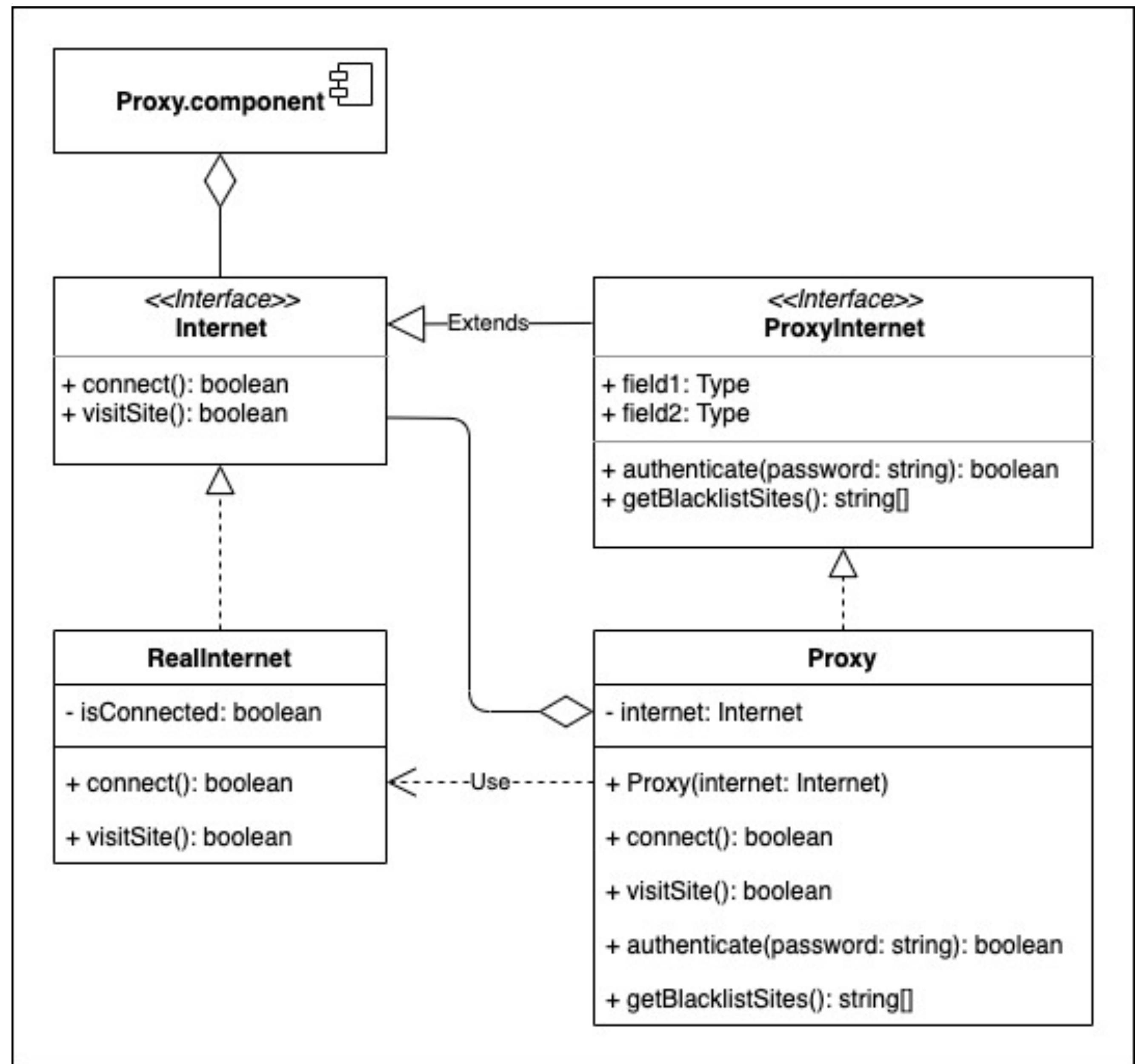
Code-Beispiel

Proxy Proxy



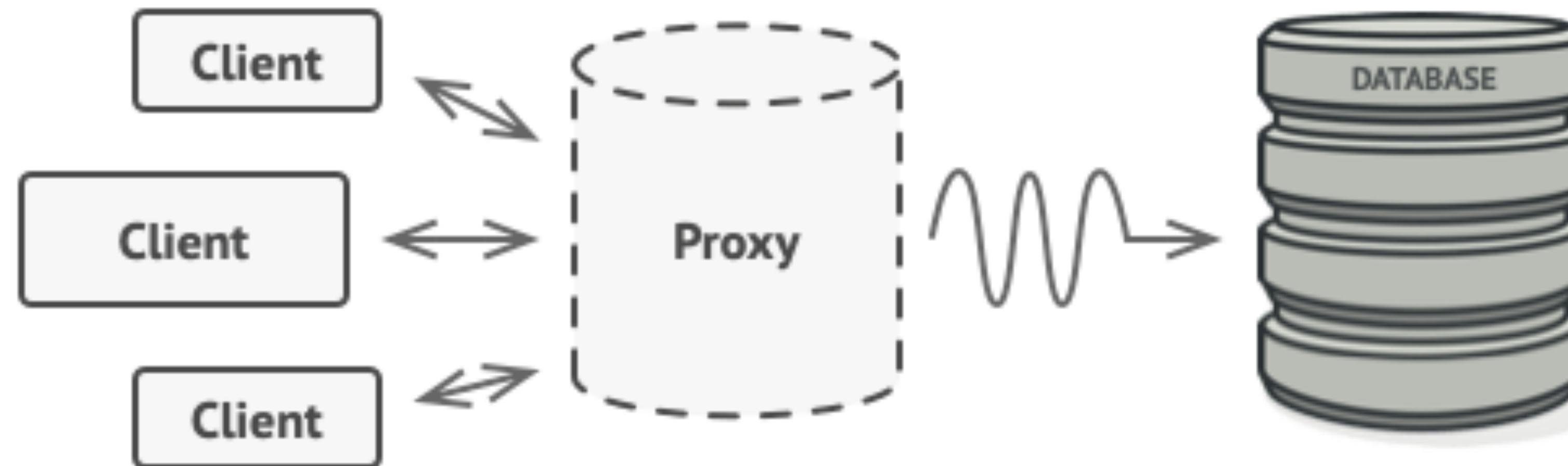
Quelle: <https://refactoring.guru/design-patterns/proxy>

Demo



Zweck

- Bereitstellung eines vorgelagerten Stellvertreterobjekts bzw. Platzhalter zwecks Zugriffssteuerung eines Objekt



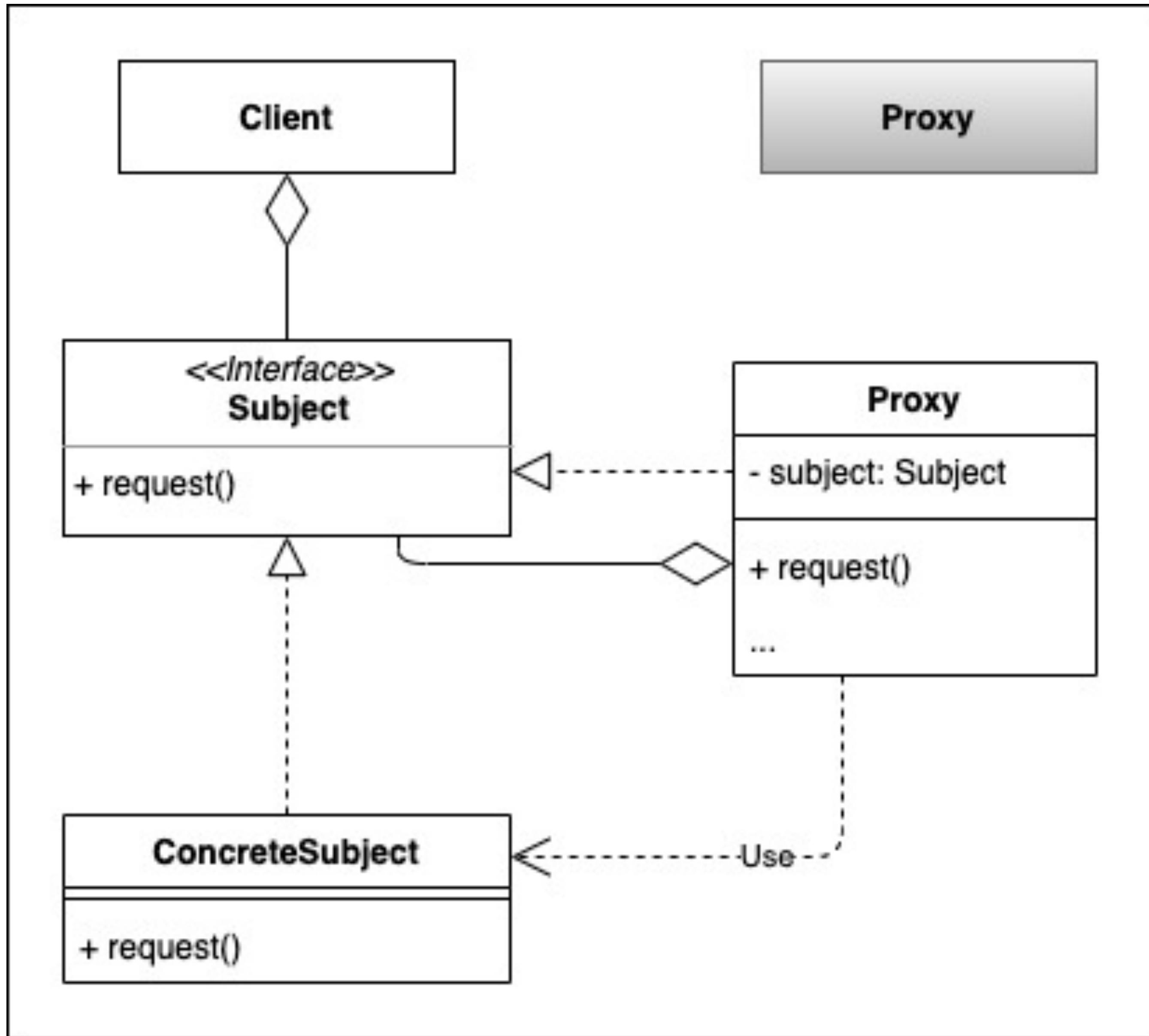
Resource: <https://refactoring.guru/images/patterns/diagrams/proxy/solution-en.png>

Anwendbarkeit

- wenn über einen simplen Pointer hinausgehender Bedarf nach anpassungsfähigeren u. intelligenteren Referenzierung eines Objektes besteht
- Remote Proxy: lokaler Stellvertreter für ein Objekt im anderen Adressbereich
- Virtuelles Proxy: Erzeugung ressourcenlastiger Objekte auf Anforderung (e.g. Transaction)
- Schutz-Proxy: Kontrolle über Zugriff auf Originalobjekt
- Smart-Reference: Ersatz für einfachen Pointer und führt beim Zugriff auf Objekt zusätzliche Operationen aus, z.B. Reference Counting, Transaction Handling

Konsequenzen (Vor- und Nachteile)

- ergänzt Differenzierungsebene für Zugriff auf Objekt
- Remote-Proxy kann Tatsache verbergen, dass sich ein Objekt in einem anderen Adressbereich befindet
- virtuelles Proxy kann Optimierungen wie die Erzeugung eines Objektes auf Anforderung durchführen
- Schutz-Proxy und Smart-Reference ermöglichen zusätzliche Verwaltungsaufgaben, sobald auf ein Objekt zugegriffen wird
- Beispiel: Objekte können auf internen Zustand geprüft werden und damit ergibt sich deren “Erlaubnis” andere Instanzen zu benutzen



Code-Beispiel

Fragen?

Happy Coding! :)