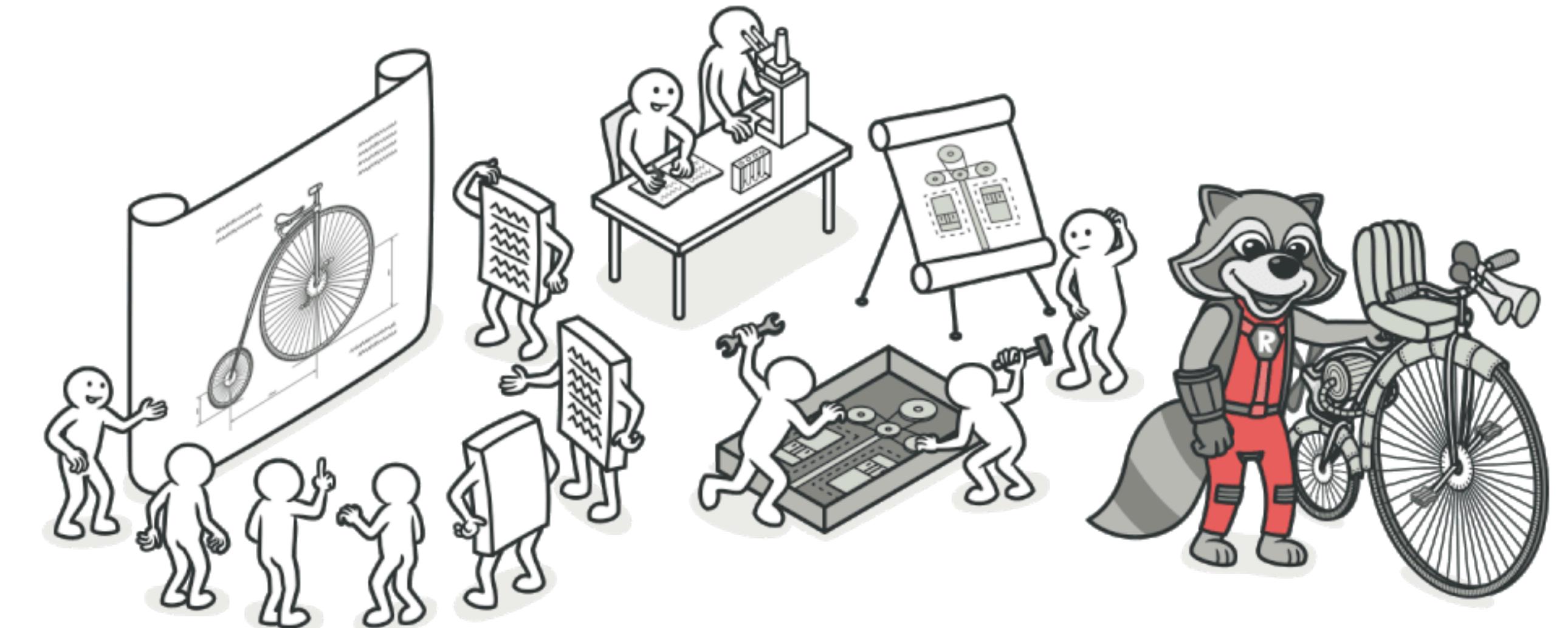


Erzeugungsmuster

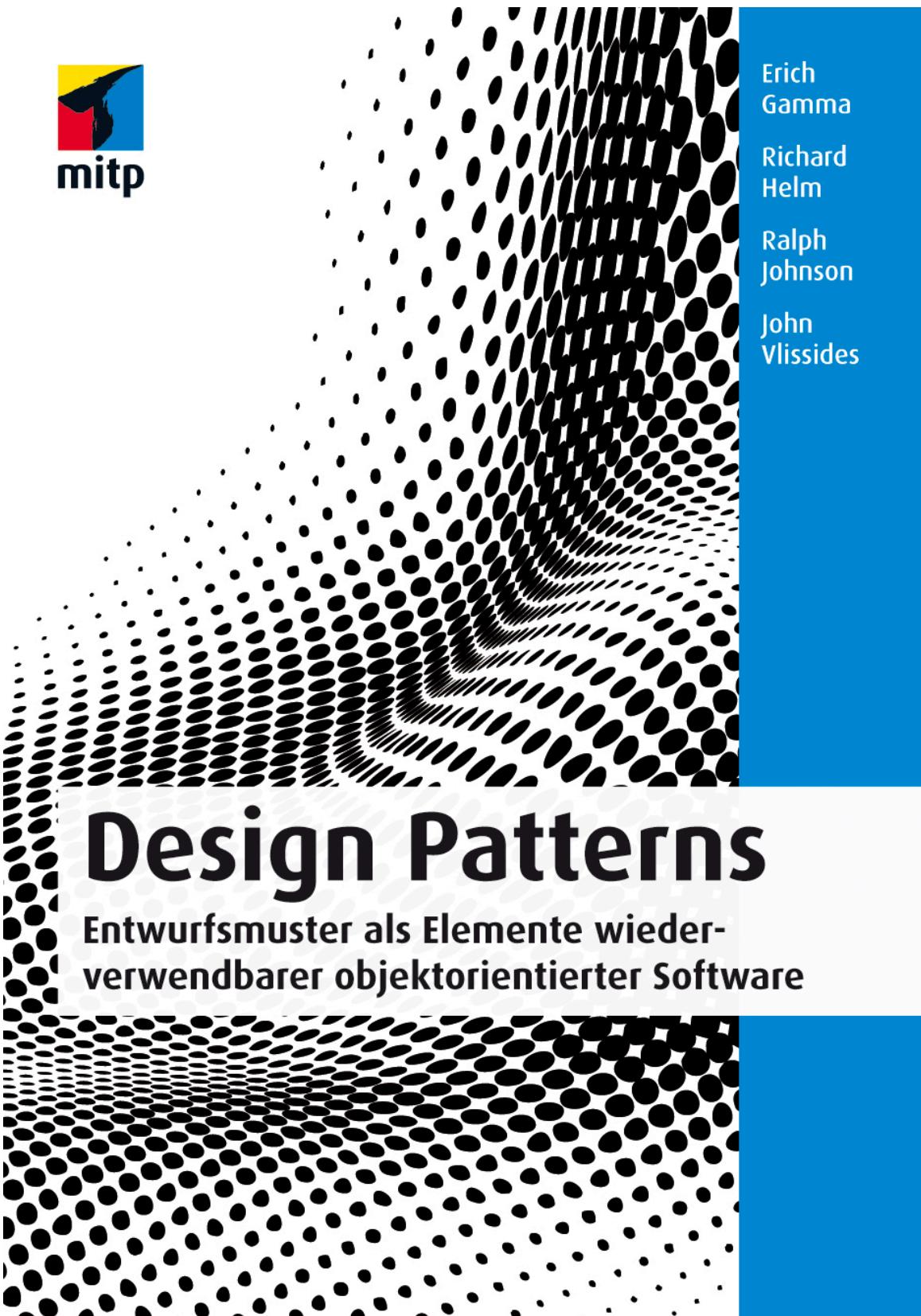
(Creational Patterns)



Patrick Creutzburg, 23. Oktober 2020
Twitter: @Itchimonji

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

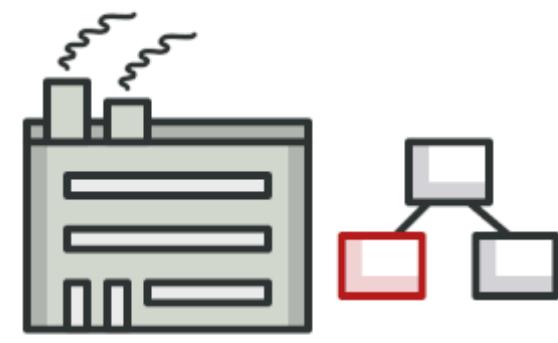
Resources



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

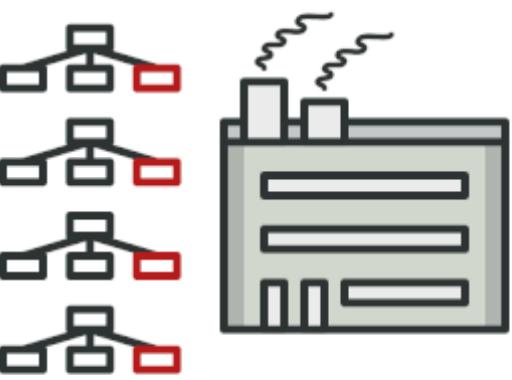


<https://refactoring.guru/>



Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



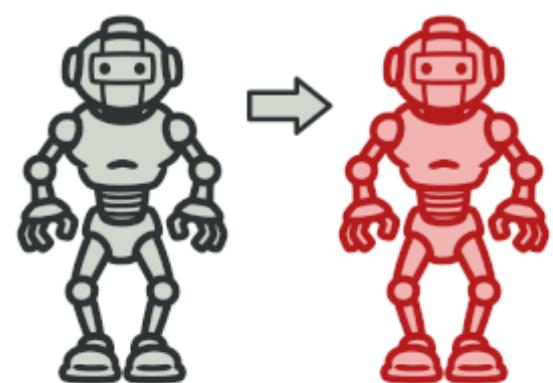
Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.



Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

Lets you copy existing objects without making your code dependent on their classes.



Singleton

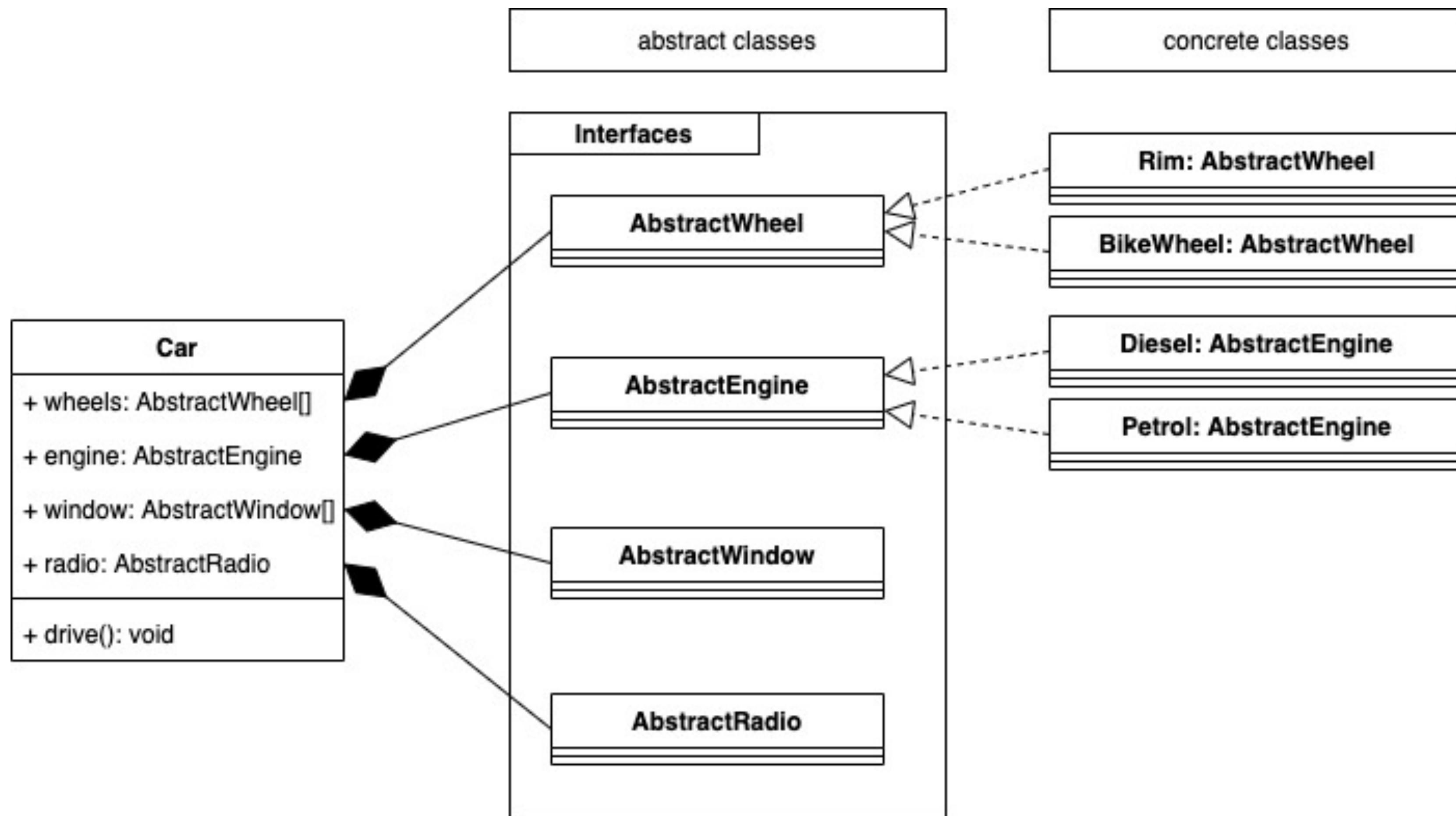
Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Bedeutung

- Abstrahieren des Instanziierungsprozesses
- Garantie, dass das System unabhängig von Generierung, Komposition und Darstellung seiner Objekte funktioniert
- Aufteilung in klassen- und objektbasierte Muster
- Verwendung von Objektkomposition und Dependency Inversion
- Kapselung der konkreten Klassen + Verbergung der Erzeugung
- Implementierung gegen Schnittstellen
- Bewahren des Open-Closed-Prinzips
- flexibles Design, geringer Wartungsaufwand, hohe Wiederverwendbarkeit

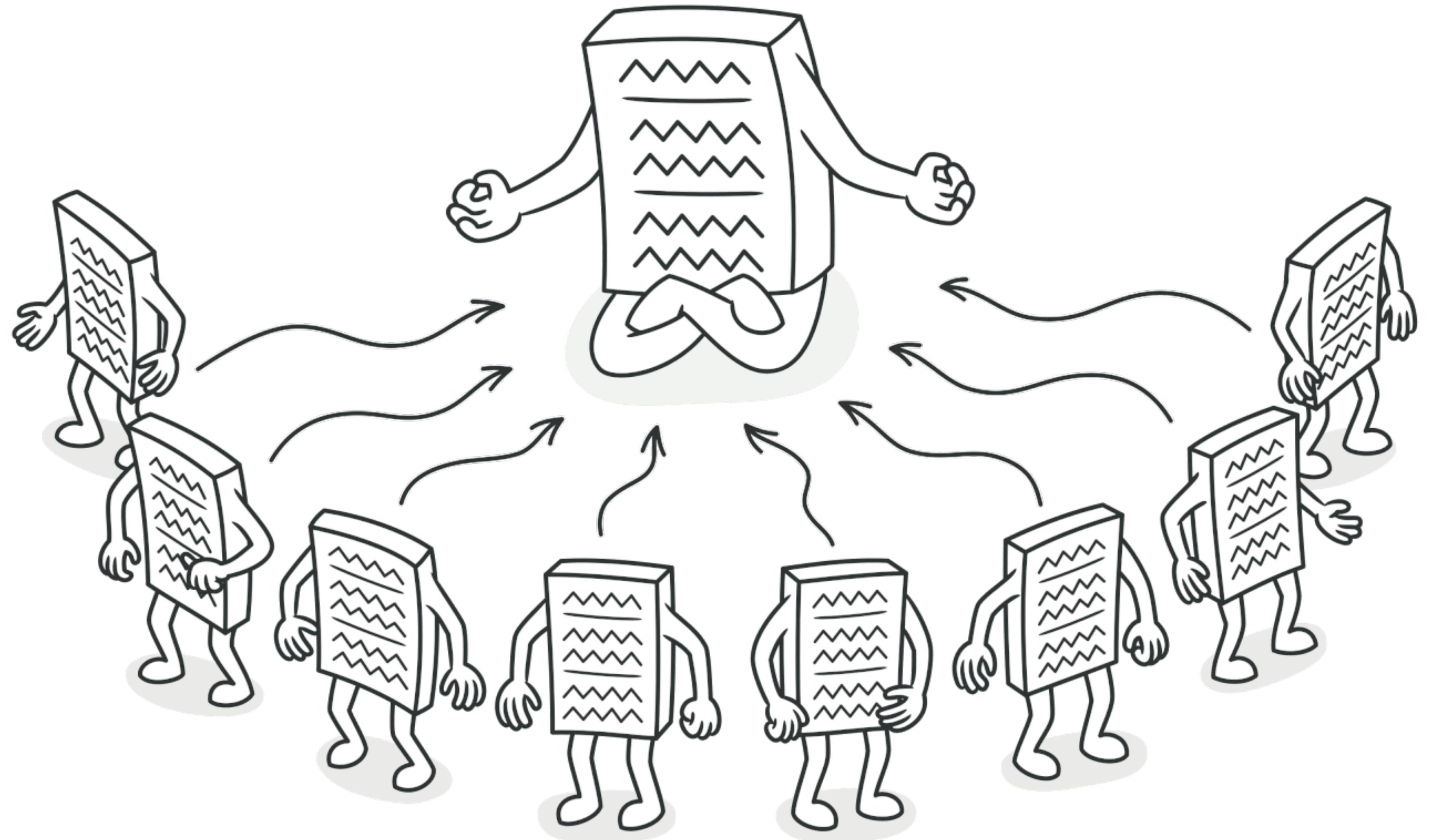
„Software entities ... should be open for extension, but closed for modification.“

Object Composition & Dependency Inversion



Singleton

Singleton



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

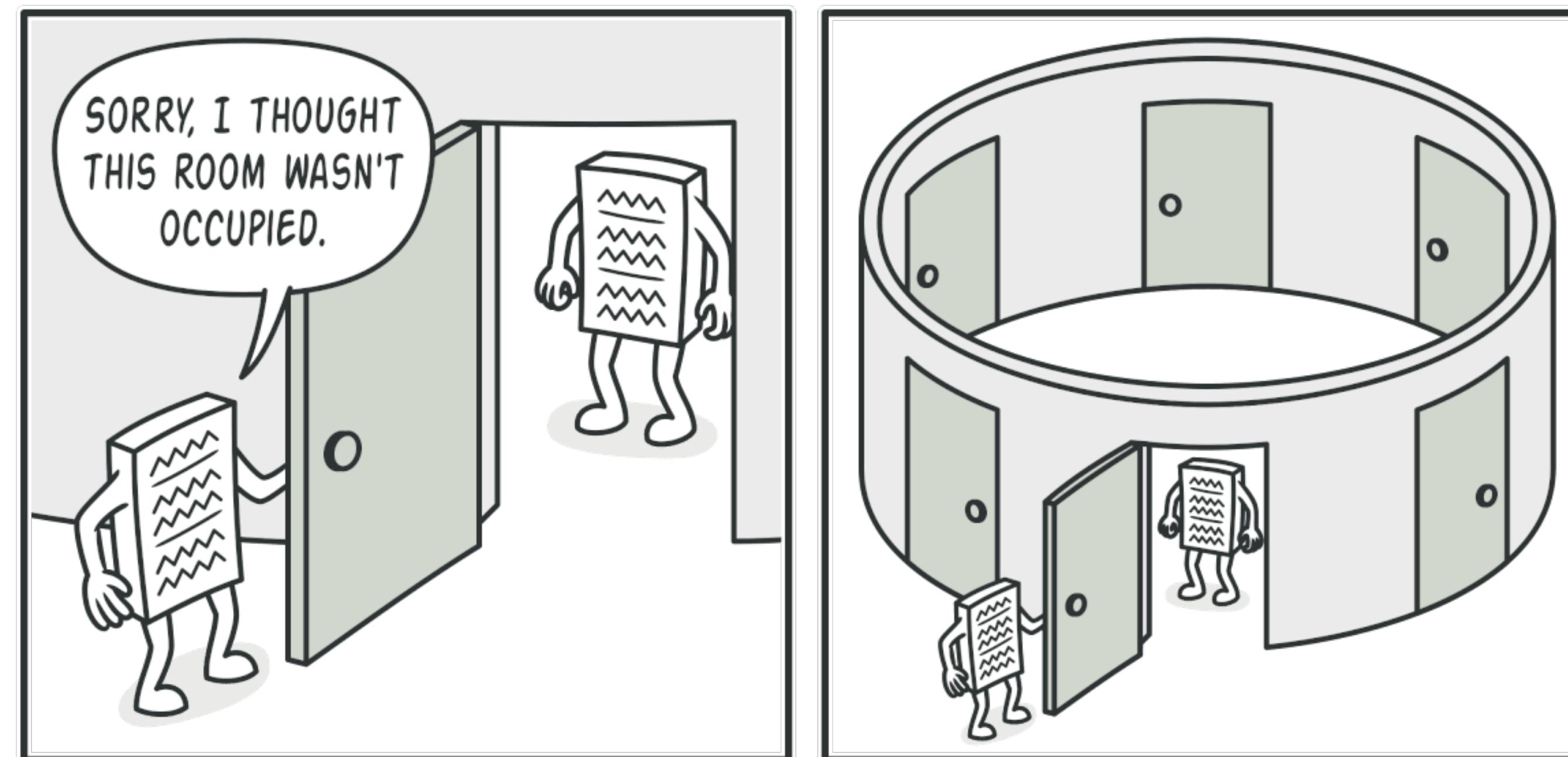
Demo

BlackTea

- static instance: BlackTea
 - id: number
 - type: string = "Black Tea"
-
- BlackTea()
 - + static getInstance(): BlackTea
 - + getType(): string
 - + getId(): number
 - + cook(): string

Zweck

- Existenz nur einer einzigen Klasseninstanz
- Bereitstellung eines globalen Zugriffspunkts für diese Instanz



Quelle: <https://refactoring.guru/images/patterns/content/singleton/singleton-comic-1-en-2x.png>

Anwendbarkeit

- Existenz einer einzigen Instanz einer Klasse und Bereitstellung eines Zugangspunktes zum Client
- Instanz ist durch Unterklassenbildung erweiterbar und Clients können erweiterte Instanz nutzen, ohne bestehenden Code zu ändern

Konsequenzen (Vor- und Nachteile)

- Kontrollierter Zugriff auf einzige Instanz
- Eingeschränkter Namensraum
- Verbesserte Operationen und Darstellung (durch Spezialisierung/Vererbung)

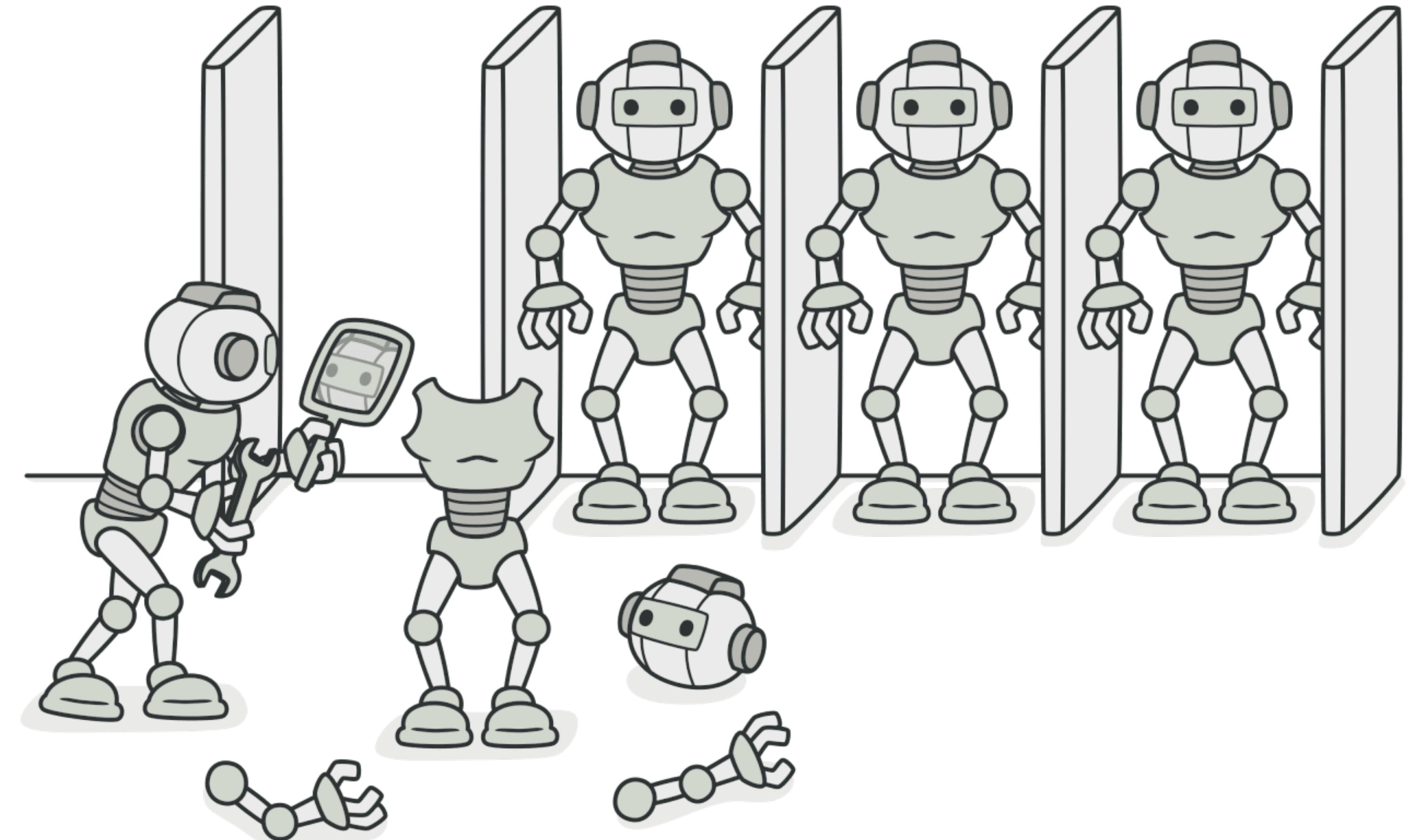
Singleton

- static instance: Singleton
- Singleton()
- + static getInstance(): Singleton

Code-Beispiel

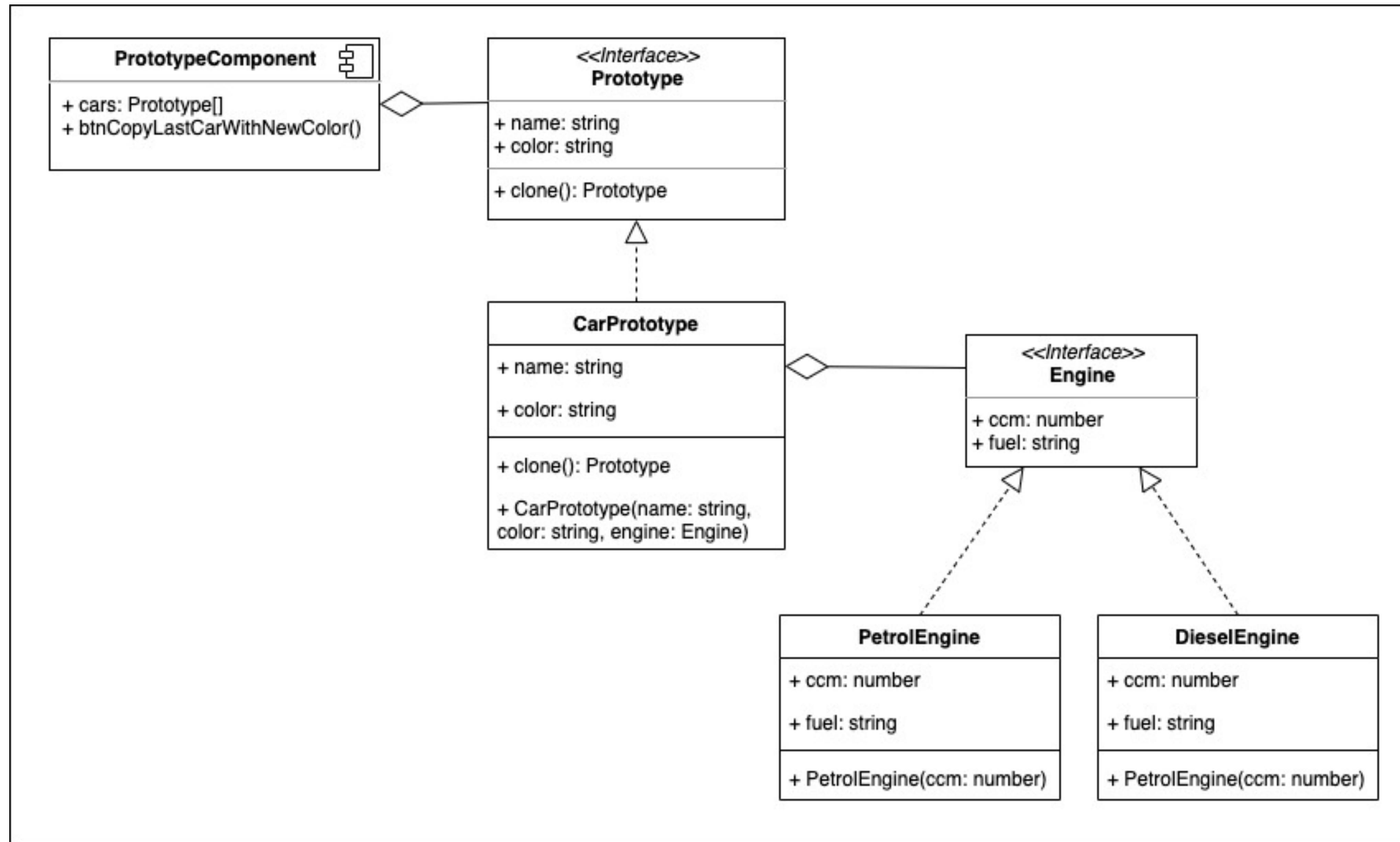
Prototype

Prototyp



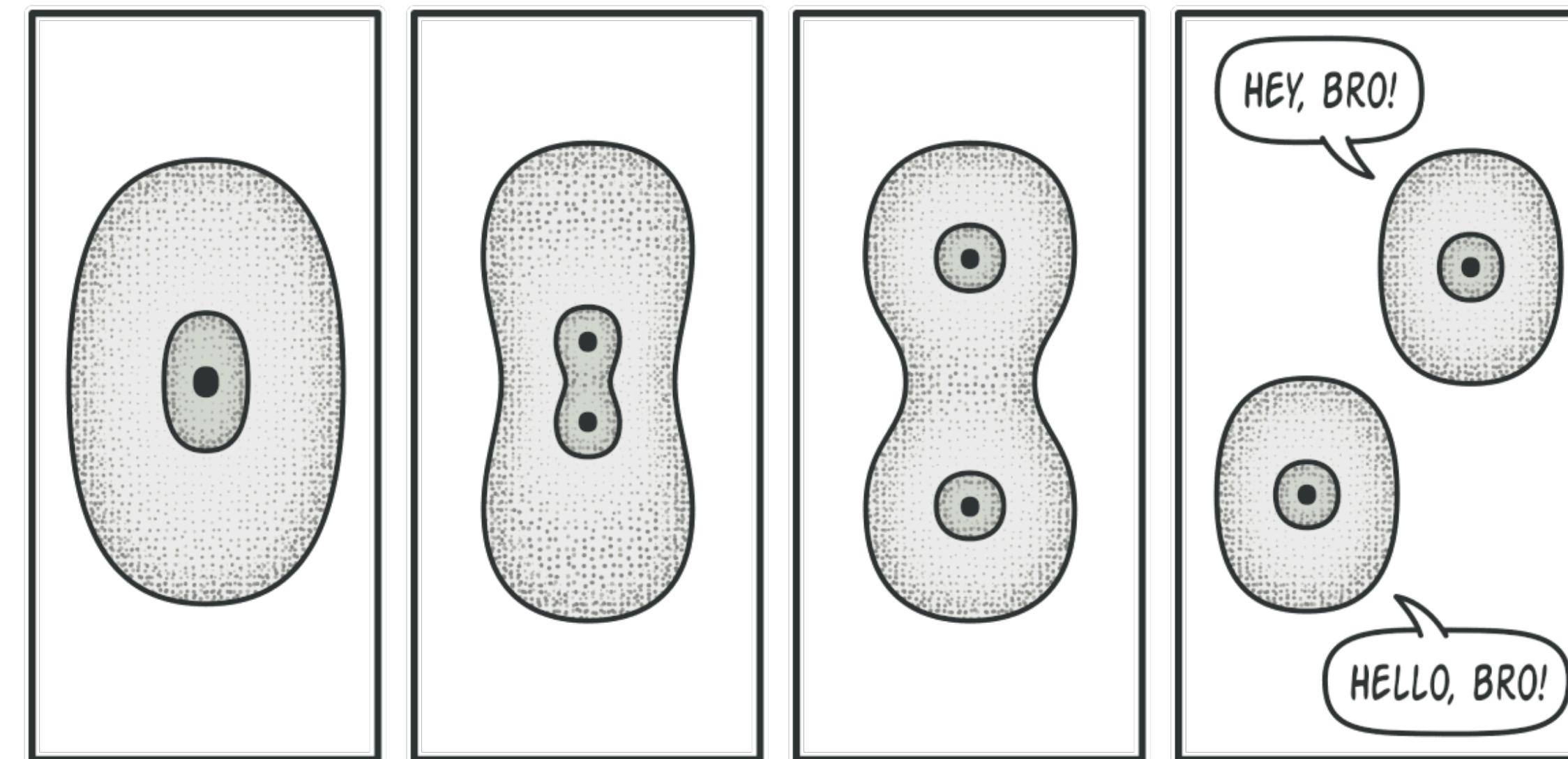
Quelle: <https://refactoring.guru/design-patterns/prototype>

Demo



Zweck

- Spezifikation einer prototypischen Instanz
- Erzeugung neuer Objekte durch Kopieren des Prototyps



Quelle: <https://refactoring.guru/images/patterns/content/prototype/prototype-comic-3-en-2x.png>

Anwendbarkeit

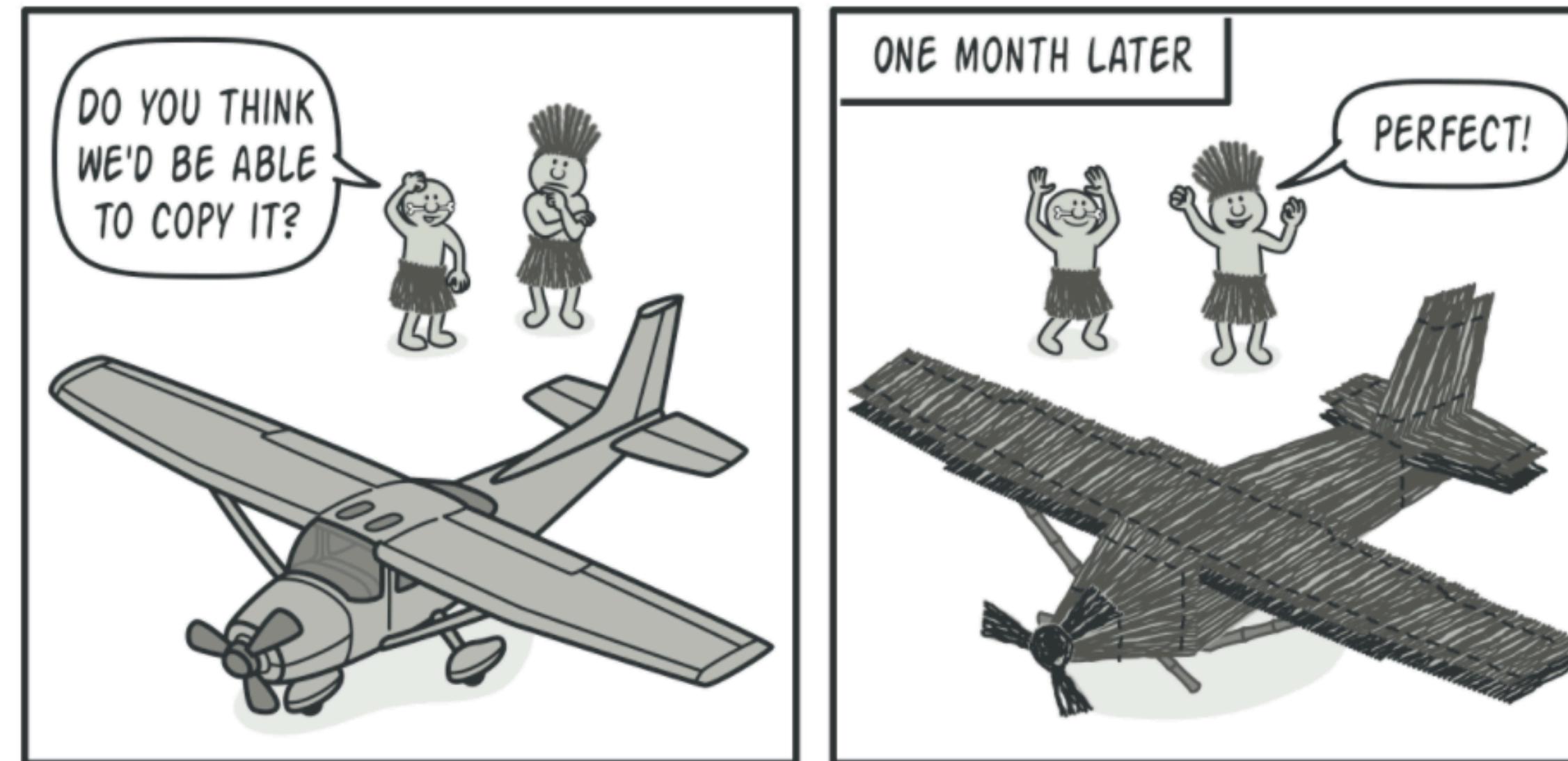
- wenn zu instanzierende Klassen zur Laufzeit spezifiziert werden (dyn. Laden)
- Vermeidung von Fabriken, da sonst zu viele konkrete Klassen existieren
- Instanzen weisen wenige Zustandskombinationen auf



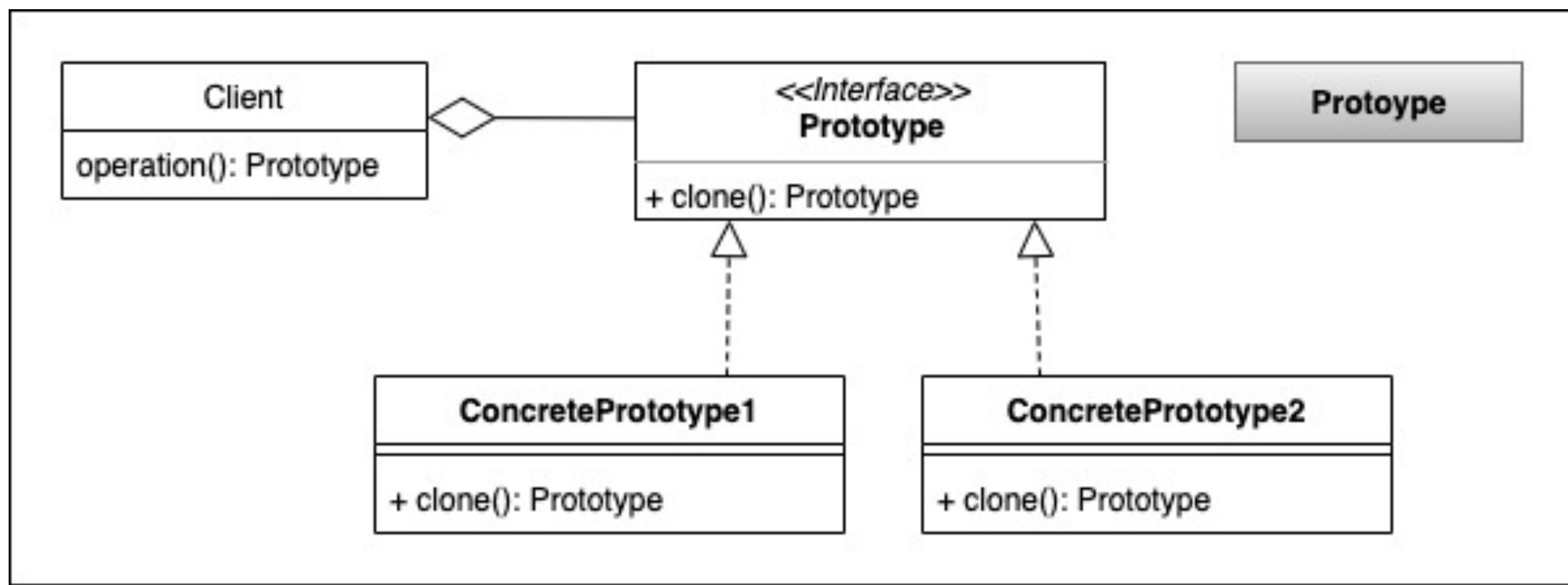
Quelle: <https://refactoring.guru/images/patterns/content/prototype/prototype-comic-2-en-2x.png>

Konsequenzen (Vor- und Nachteile)

- Produktergänzung und -entfernung zur Laufzeit (mehr Flexibilität als and. EM)
- Spezifikation neuer Objekte mittels Wertevariation (und dadurch Minderung konkreter Klassen)
- Reduzierte Unterklassenbildung



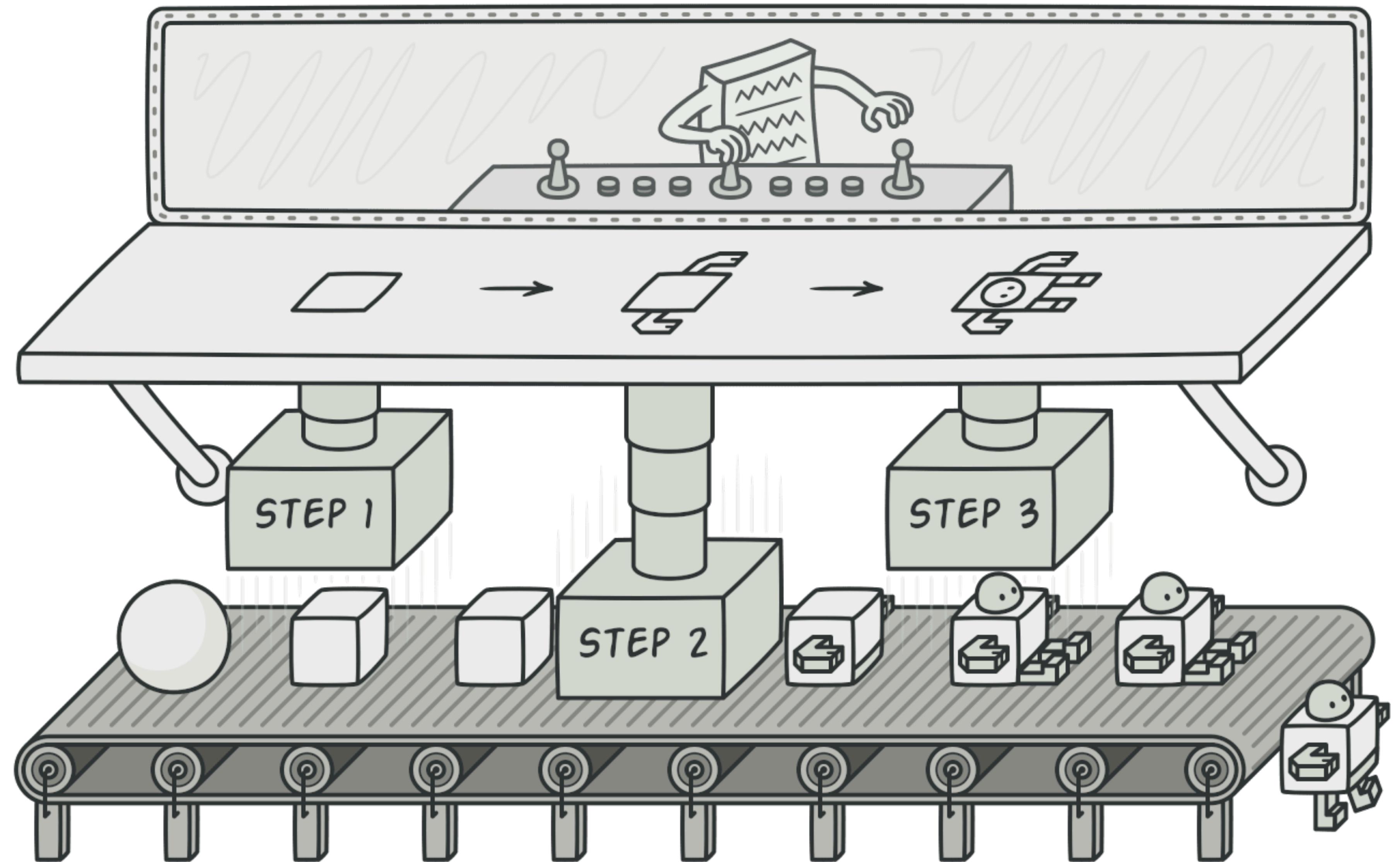
Quelle: <https://refactoring.guru/images/patterns/content/prototype/prototype-comic-1-en-2x.png>



Code-Beispiel

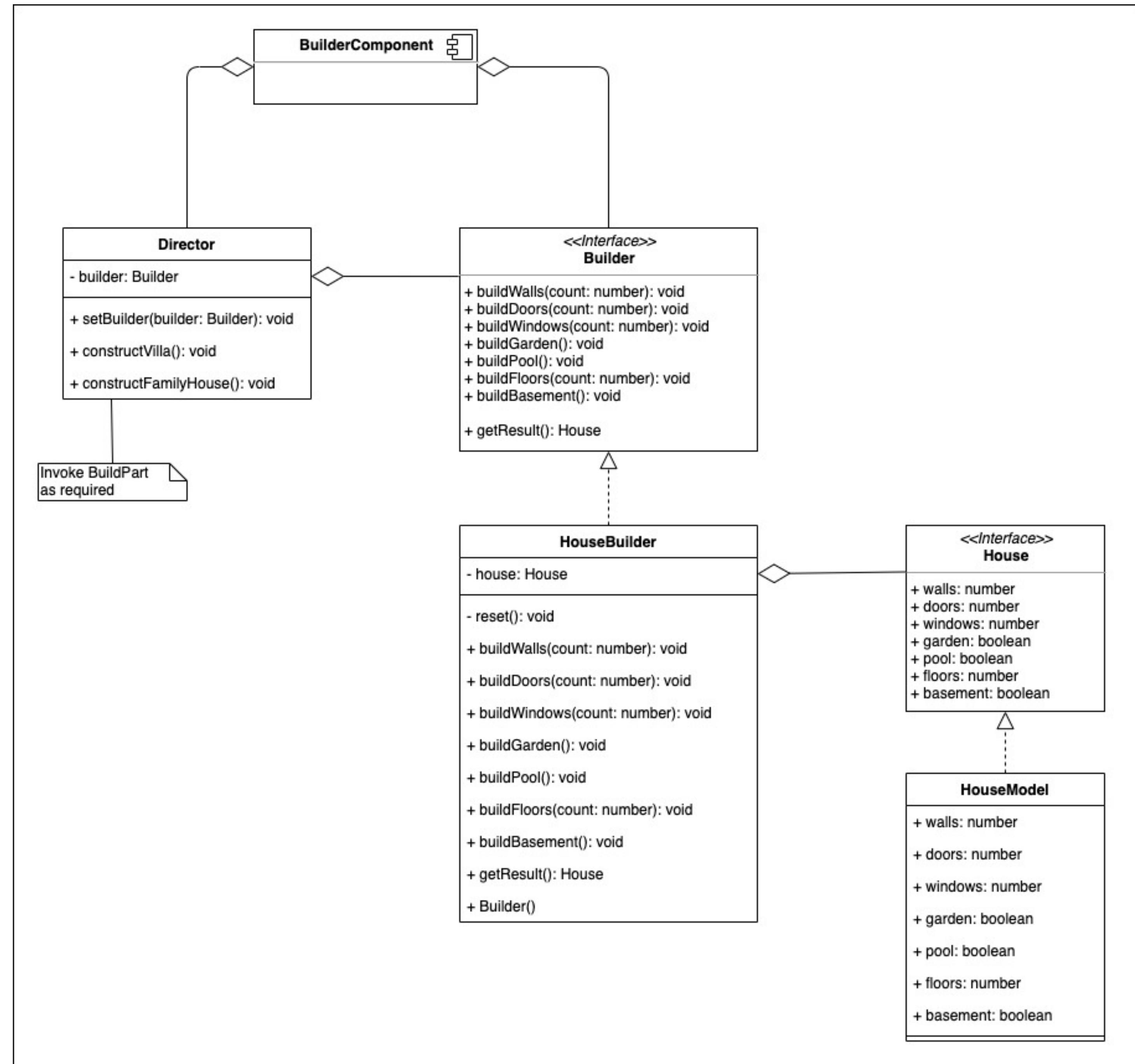
Builder

Erbauer



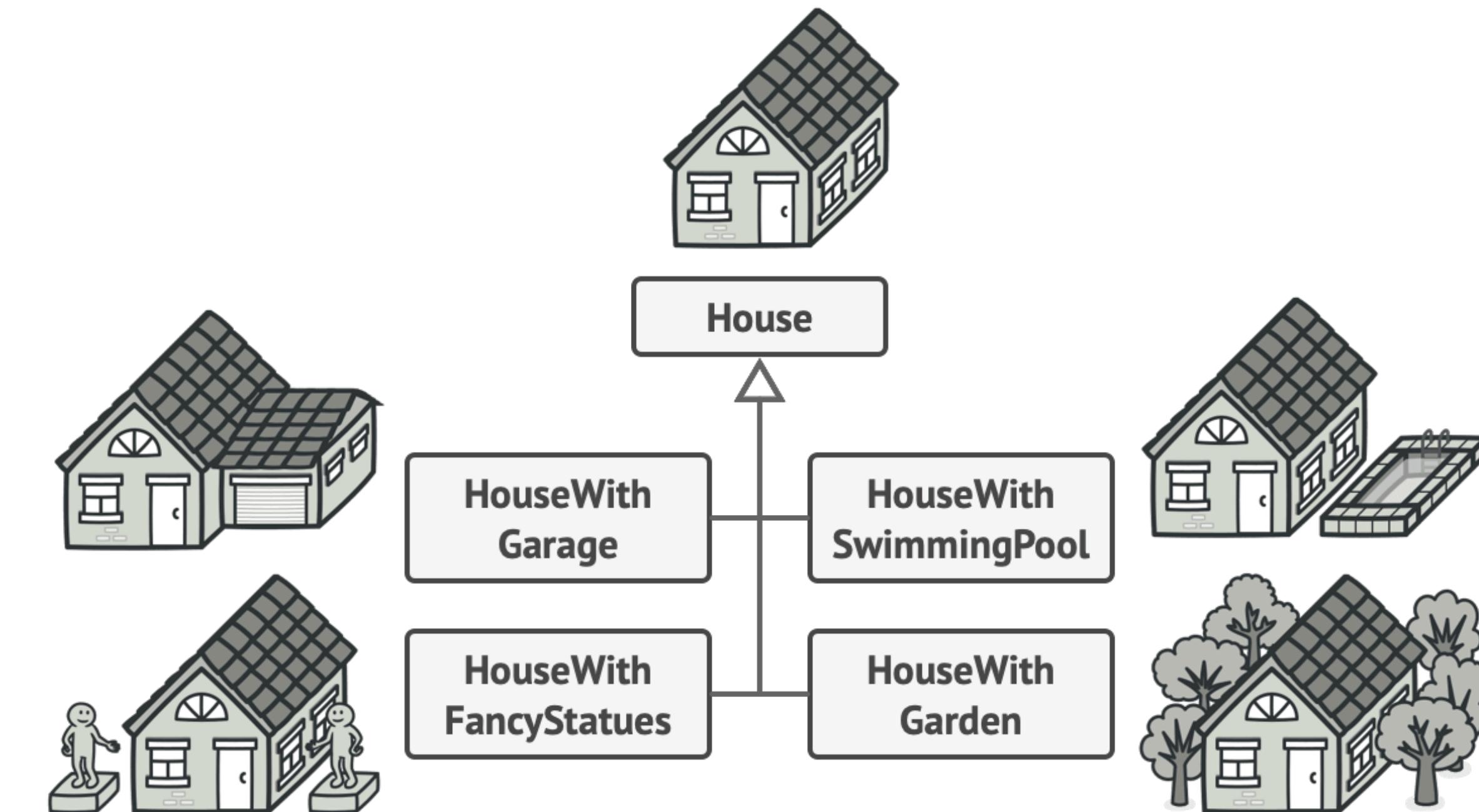
Quelle: <https://refactoring.guru/design-patterns/builder>

Demo



Zweck

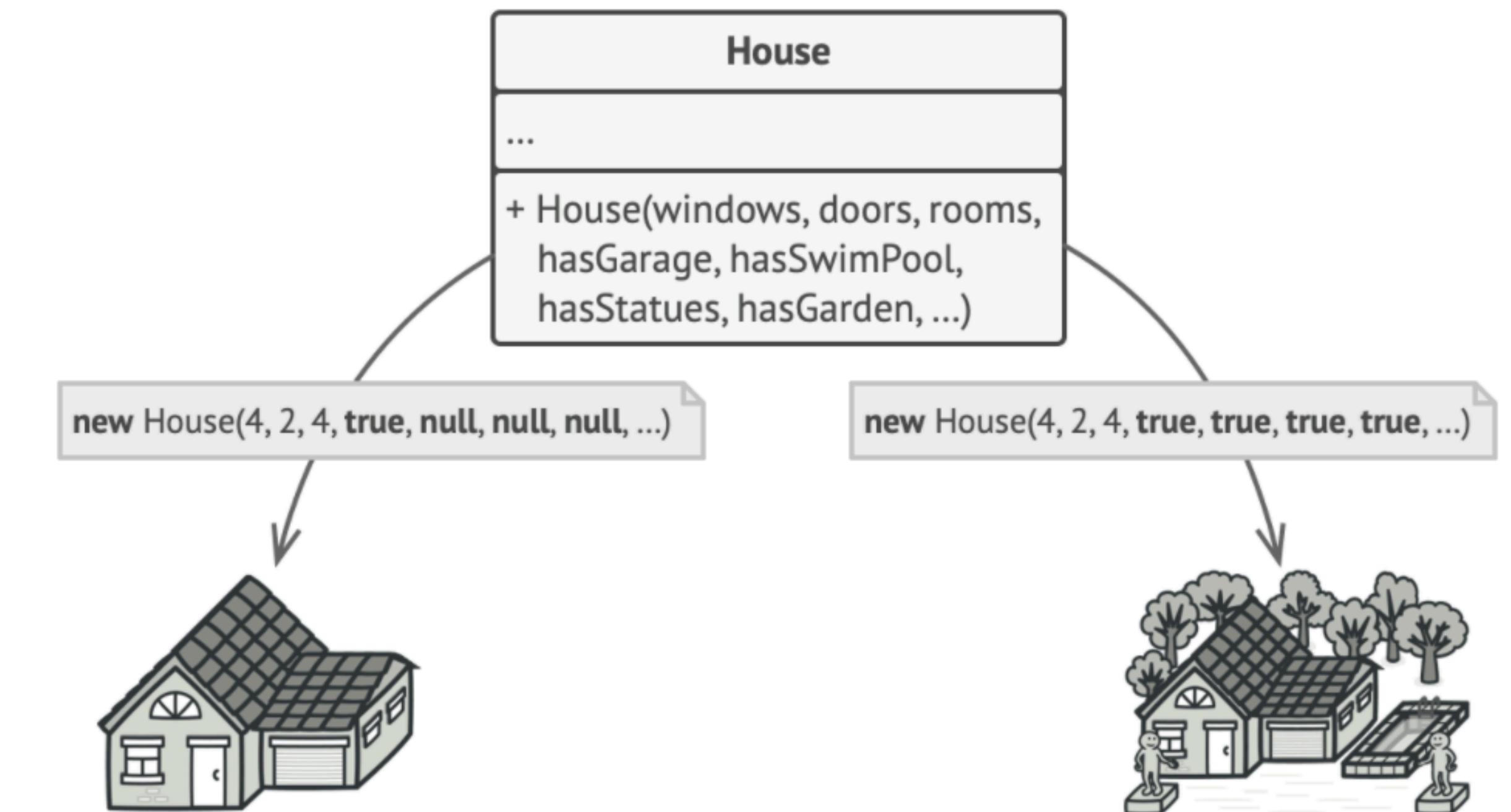
- Erstellung komplexer Objekte nach Bauplan
- Getrennte Handhabung von Erzeugung- und Darstellungsmechanismen in einem einzigen Erzeugungsprozess



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

Anwendbarkeit

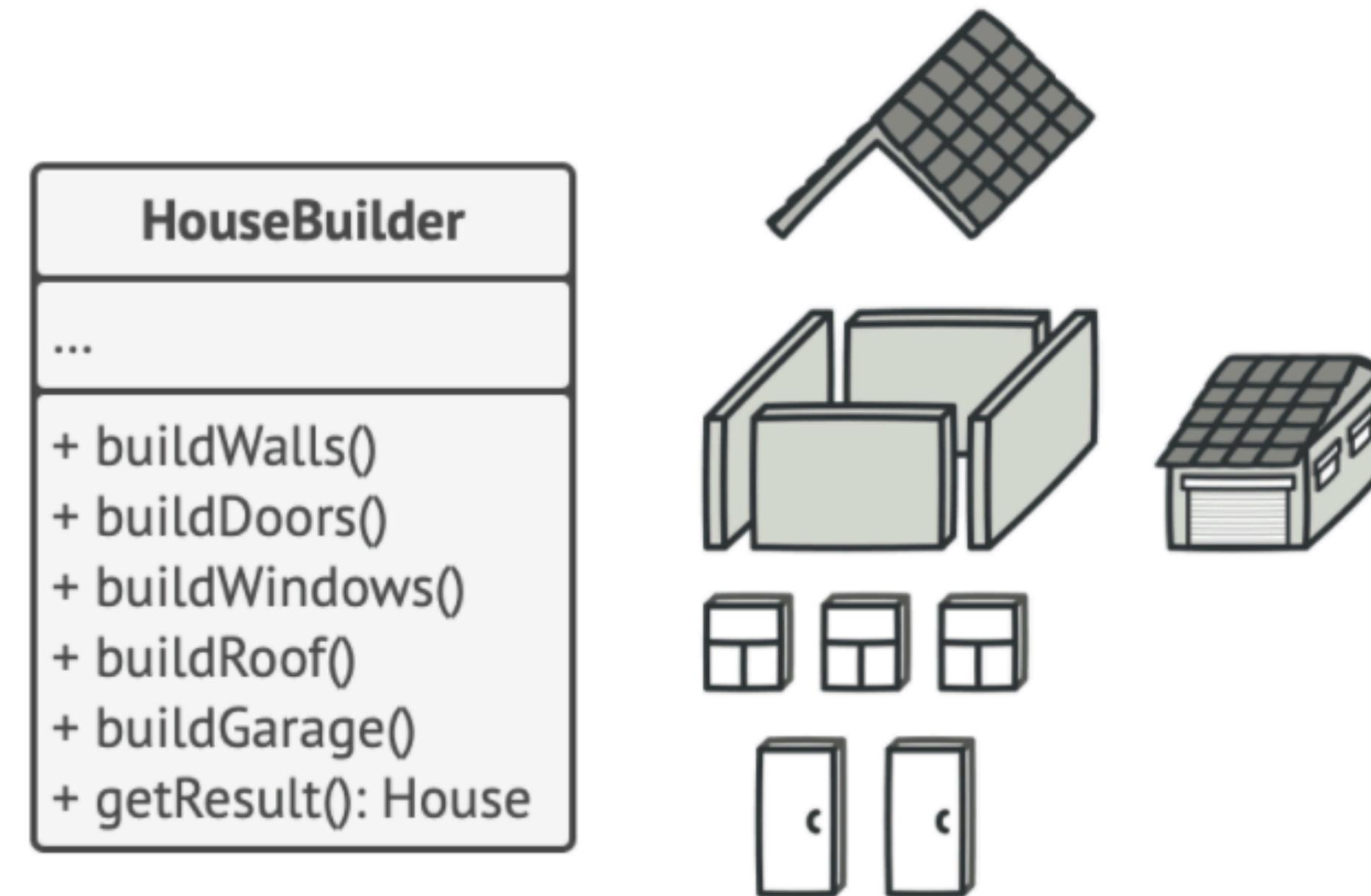
- Gewährleistung der Unabhängigkeit zur Erzeugung komplexer Objekte von deren Bestandteilen und Komposition
- Zulassen von verschiedenen Darstellungsformen des zu generierenden Objekts
- SQL-Builder ist gängiges Beispiel



Quelle: <https://refactoring.guru/images/patterns/diagrams/builder/problem2-2x.png>

Konsequenzen (Vor- und Nachteile)

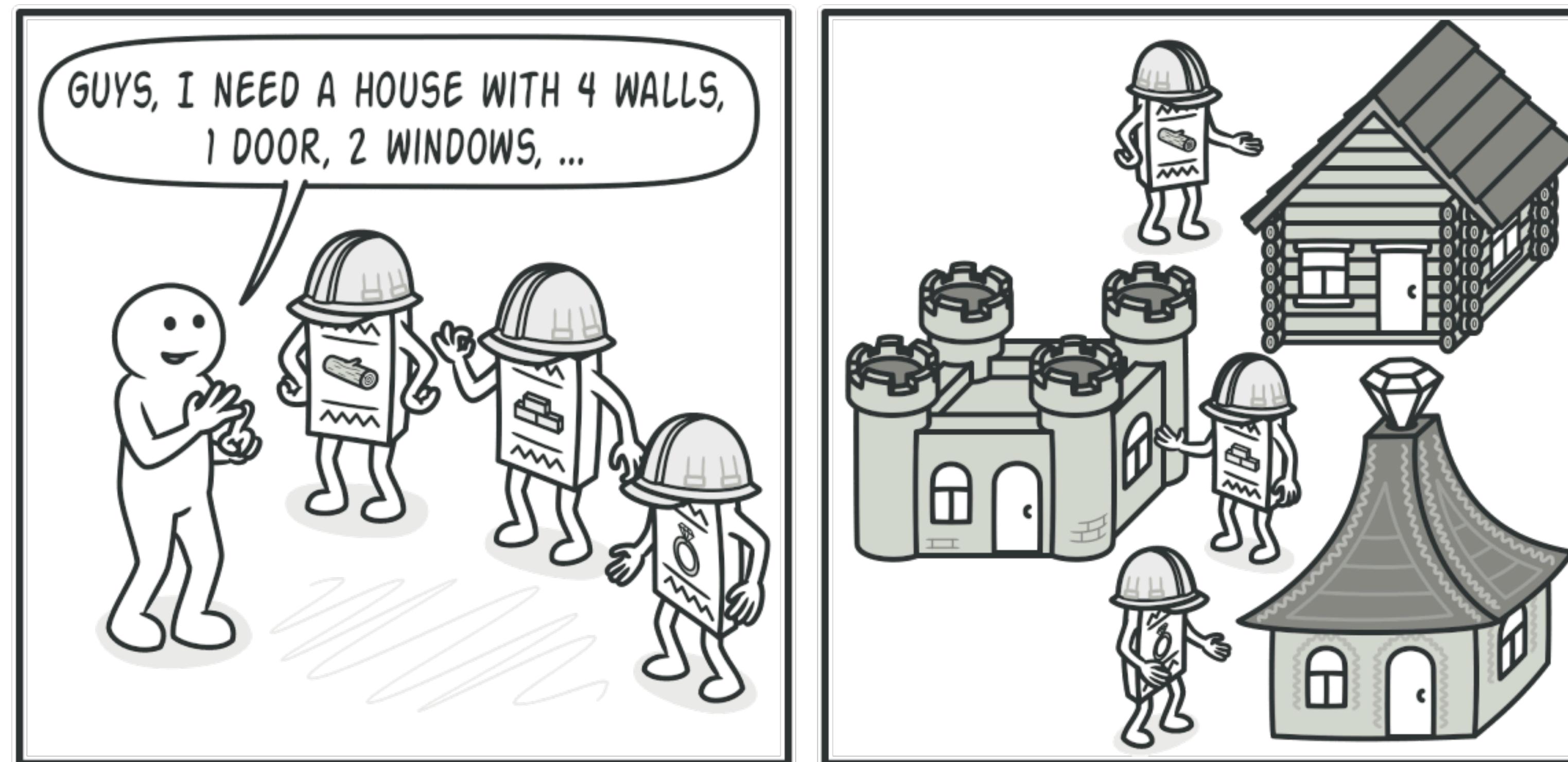
- Variable interne Darstellung eines Produktes
- Isolierung des Code in Bezug auf Erzeugung und Darstellung
- Überwachung des Erzeugungsprozesses (Kontrolle durch Director - Abruf erst wenn Objekt erzeugt wurde)



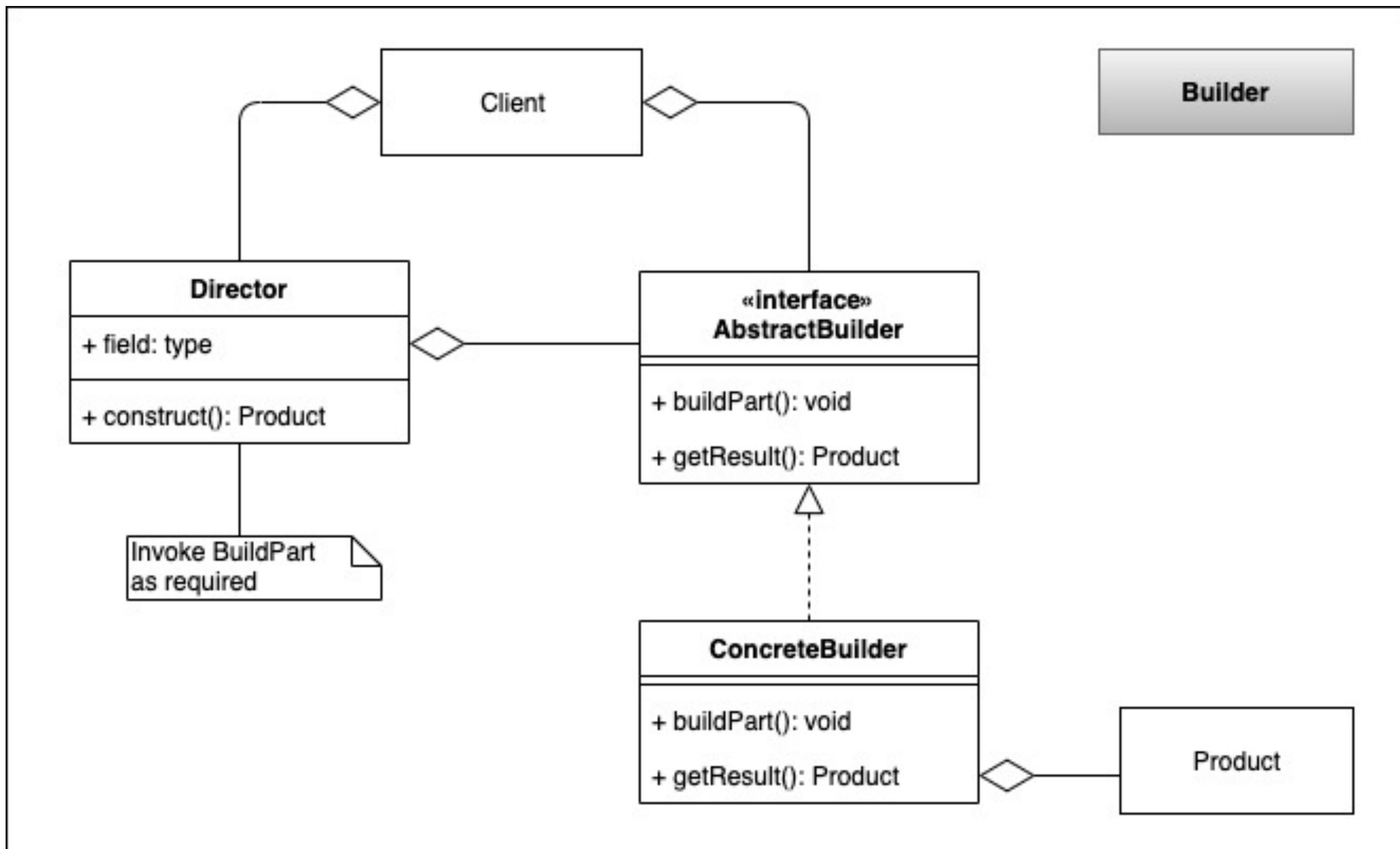
Quelle: <https://refactoring.guru/images/patterns/diagrams/builder/solution1-2x.png>

Erweiterbarkeit

- Produkte benötigen keine abstrakte Klasse, da erzeugten Klassen die starke Varianz aufweisen



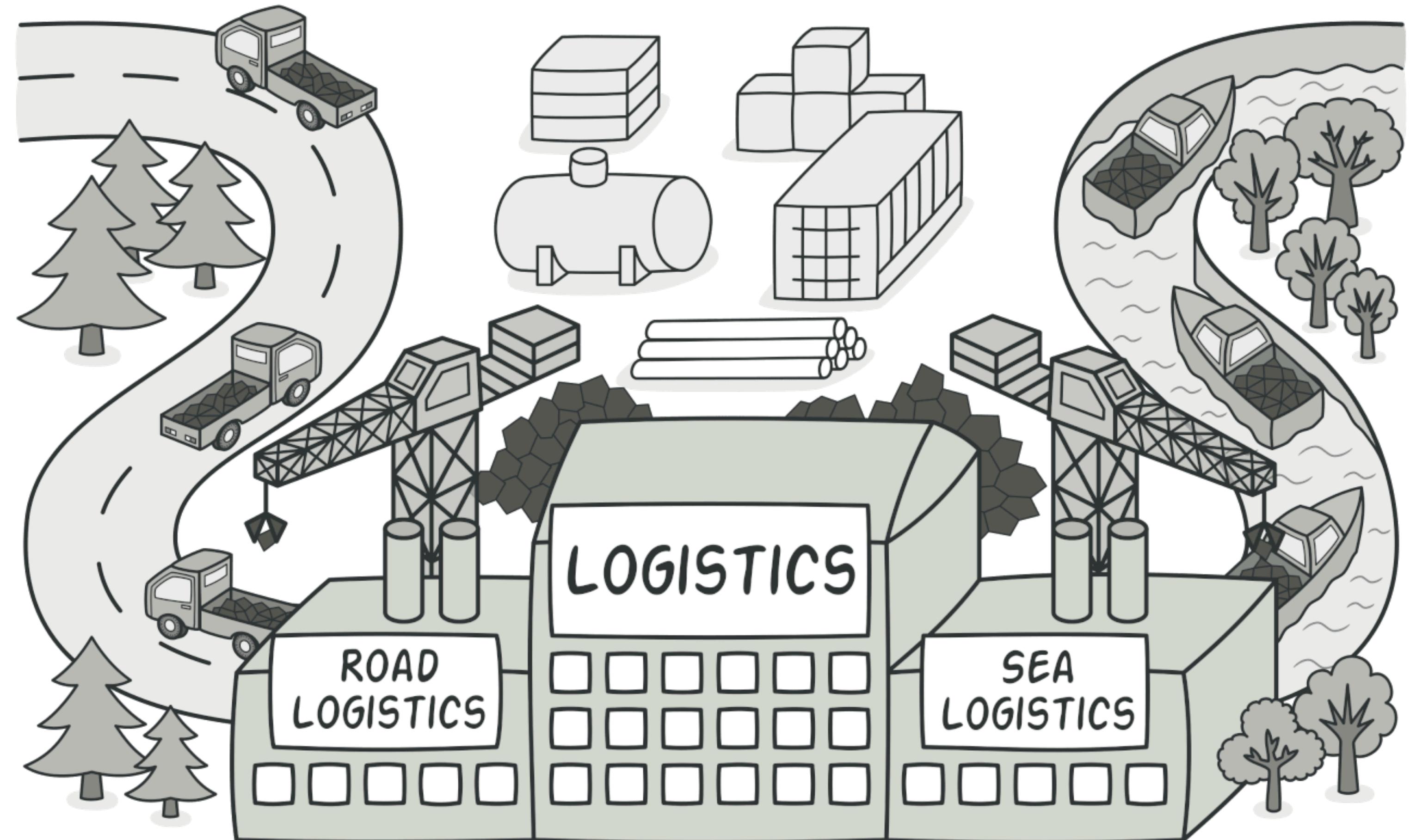
Quelle: <https://refactoring.guru/images/patterns/content/builder/builder-comic-1-en-2x.png>



Code-Beispiel

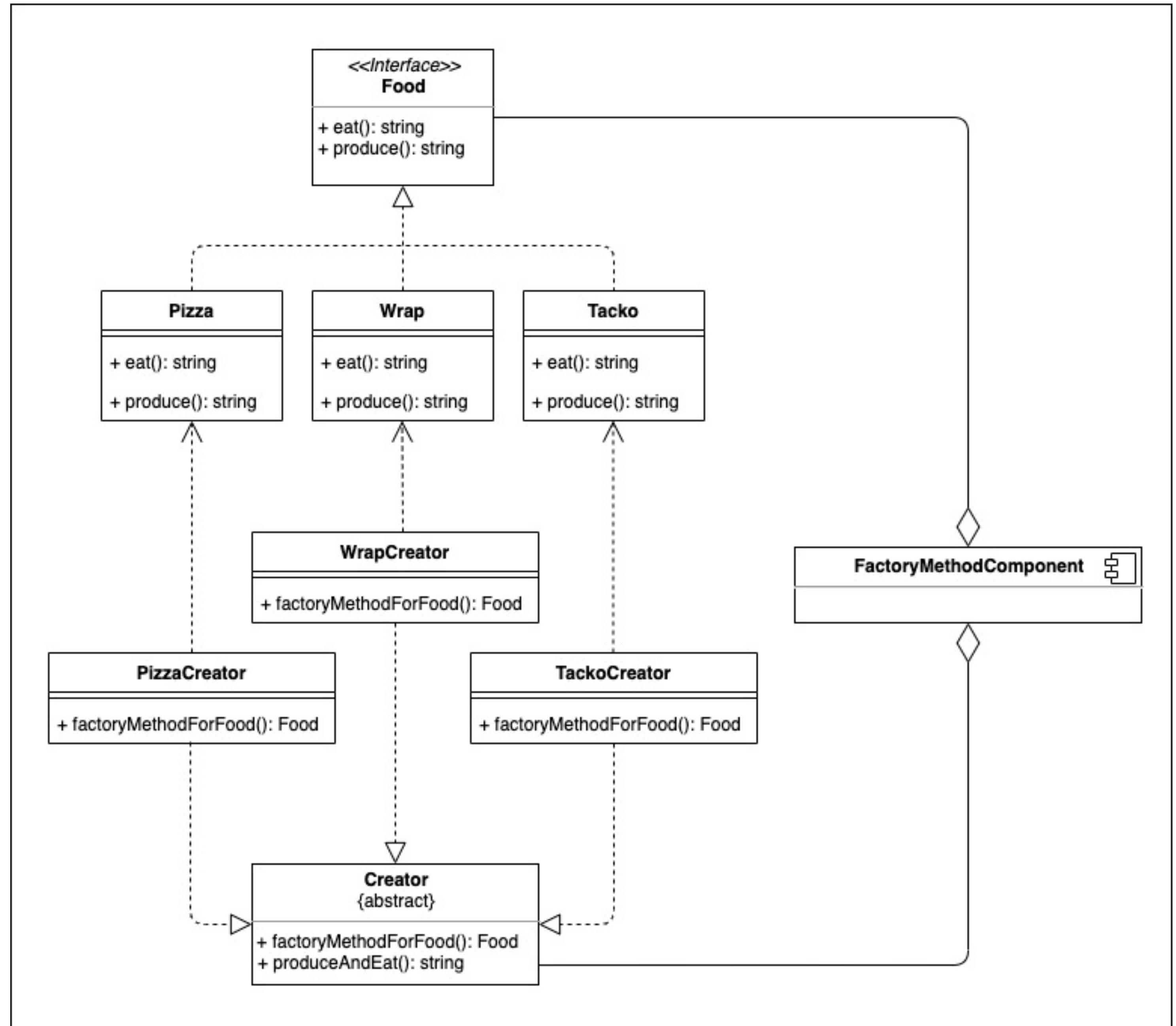
Factory Method

Fabrikmethode



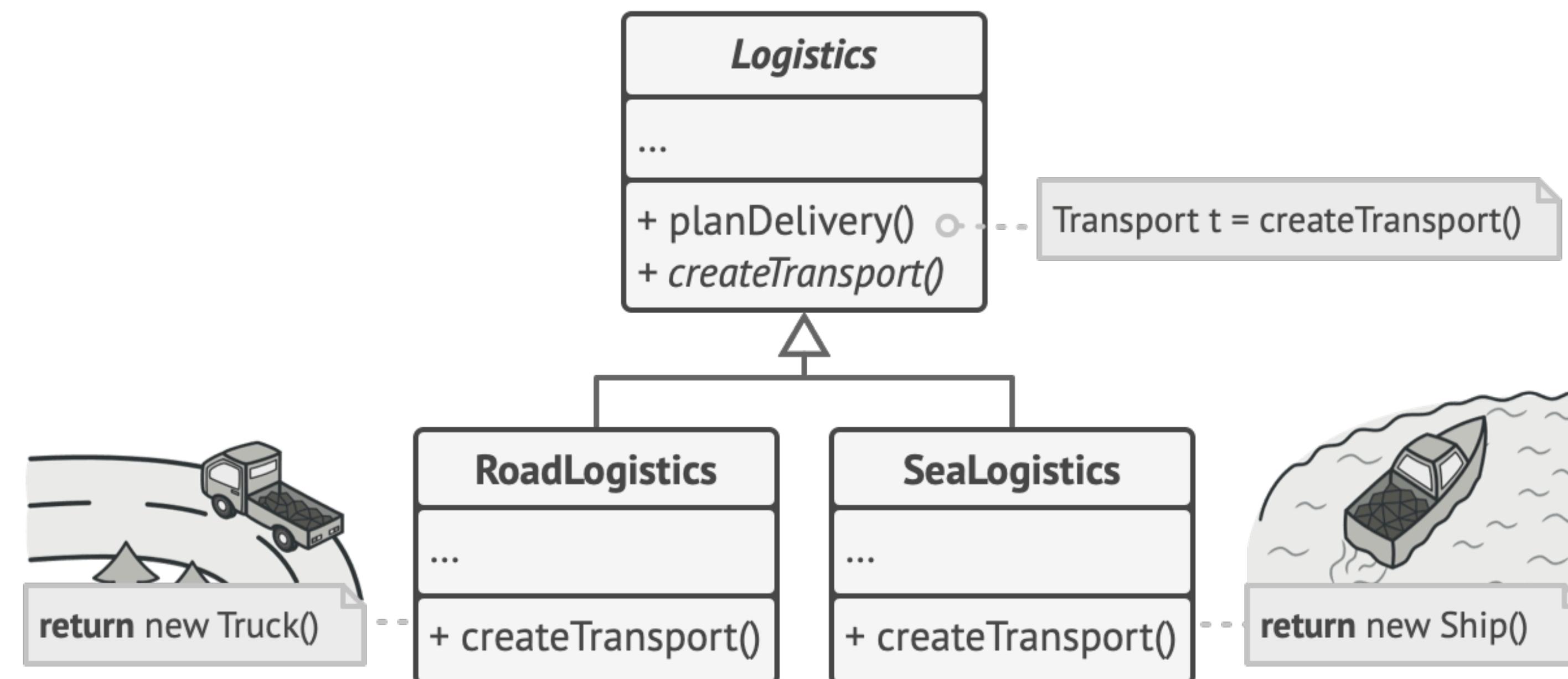
Quelle: <https://refactoring.guru/design-patterns/factory-method>

Demo



Zweck

- Schnittstelle zur Objekterzeugung, bei der die Bestimmung der instanziierenden Klasse der Unterklasse überlassen wird
- Delegierung der Instanziierung an eine Unterklasse



Anwendbarkeit

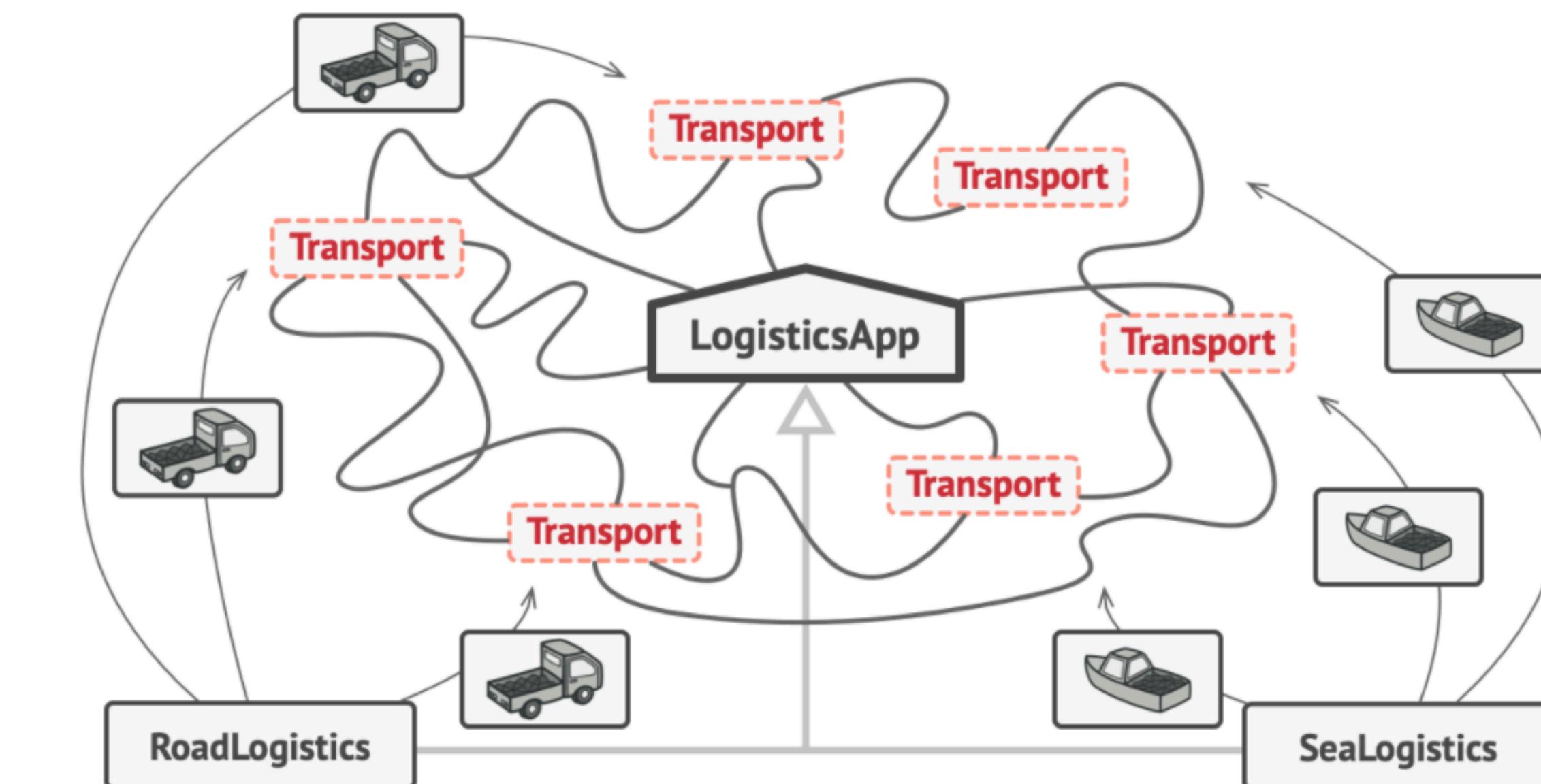
- zu erzeugende Klassen können nicht von Vorhinein bestimmt werden (deswegen abstrakt & Funktion ist bekannt)
- eine Klasse erwartet von ihren Unterklassen eine Spezifizierung der zu erzeugenden Produkte

Konsequenzen (Vor- und Nachteile)

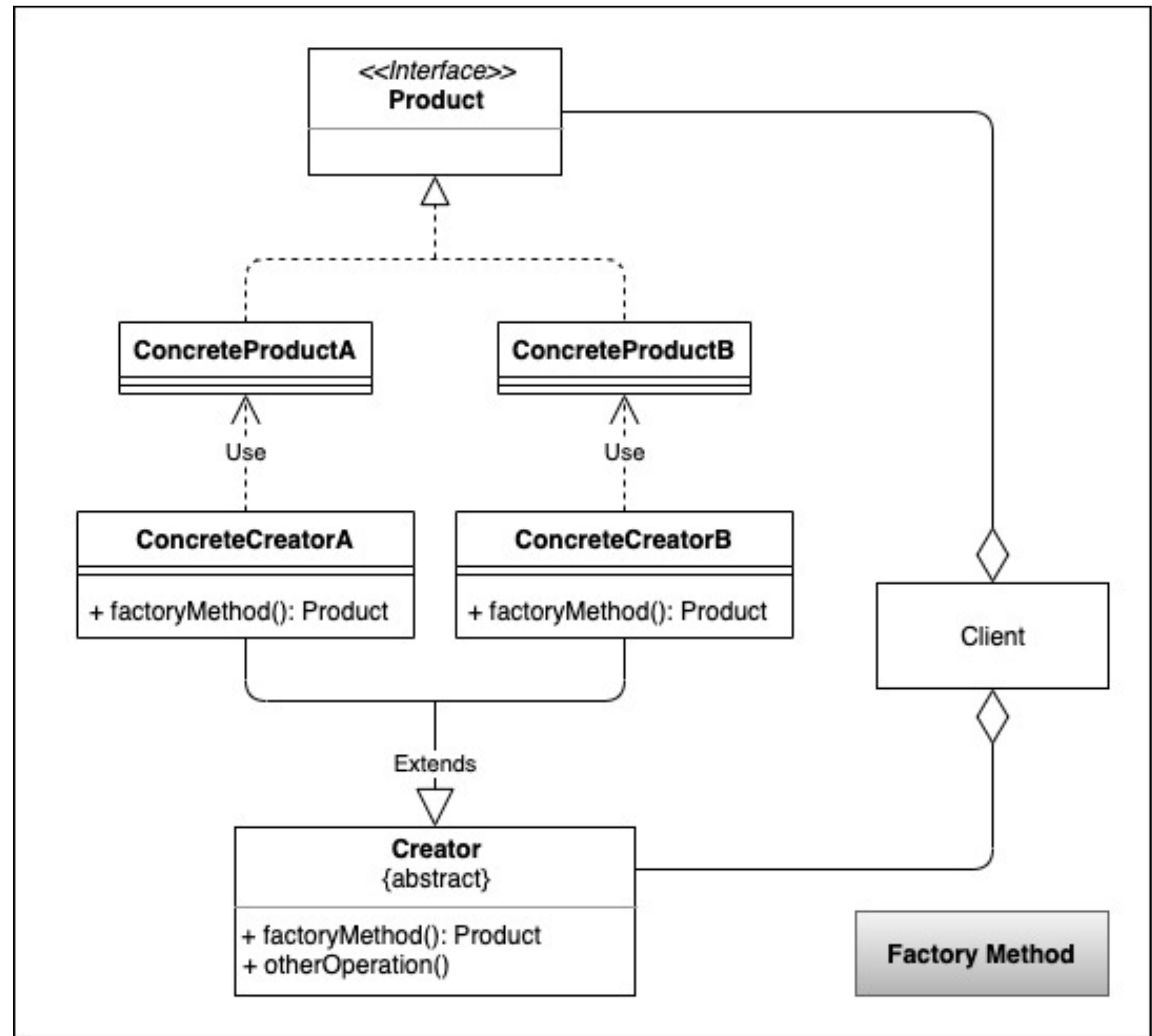
- Spezialisierungsoptionen für Unterklassen, da durch Pattern gegenüber der direkten Erzeugung des Objekte eine erweiterte Version des Objektes bereitgestellt wird

Best Practise

- Wenn man ein Erzeugungsmuster verwenden möchte, sich aber unsicher ist, welches, beginnt man mit der Implementierung der Fabrikmethode
- im späteren Verlauf schwenkt man auf andere Erzeugungsmuster um
- Fabrikmethode am einfachsten ins System zu integrieren
- hohe Flexibilität



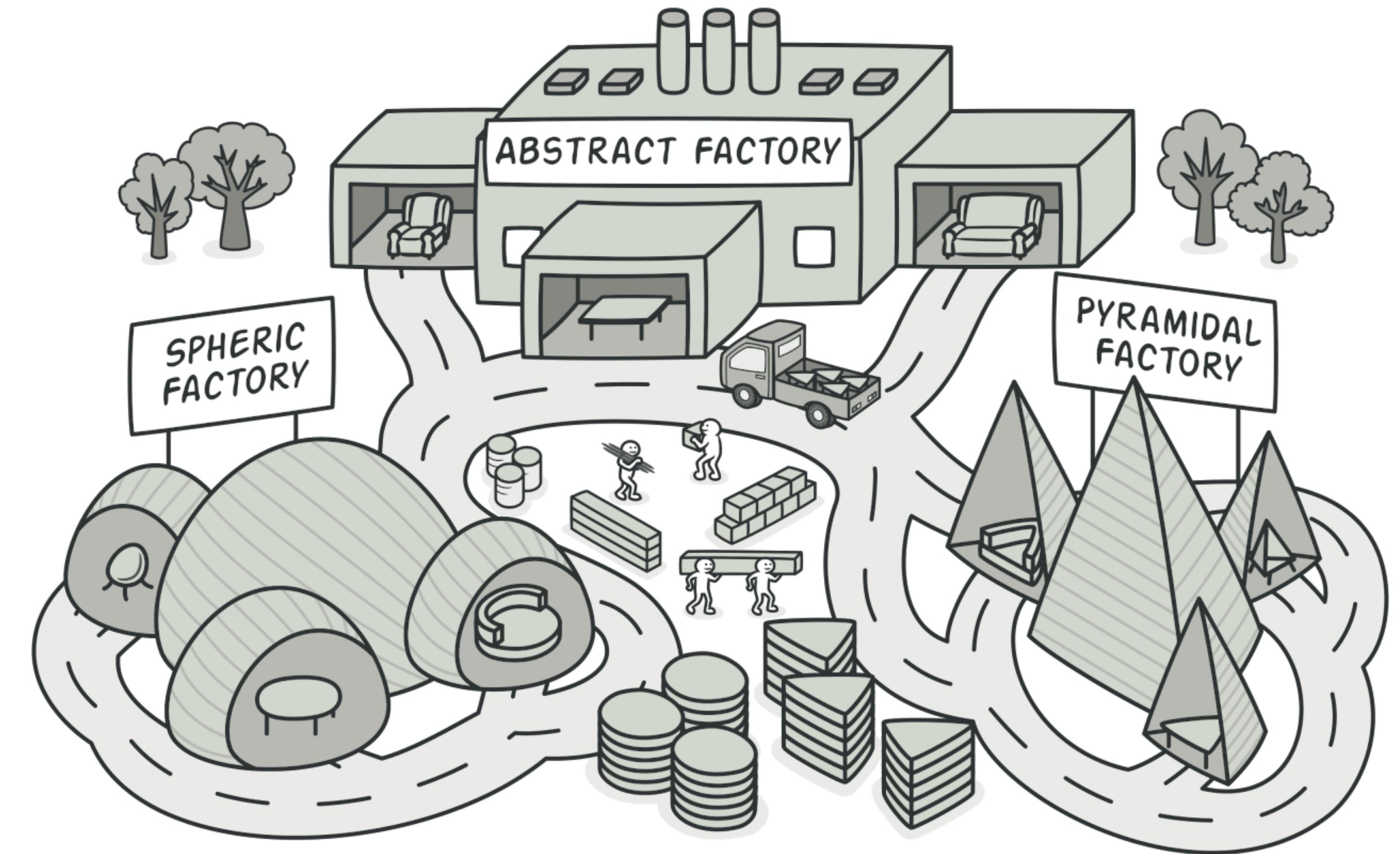
Quelle: <https://refactoring.guru/images/patterns/diagrams/factory-method/solution3-en-2x.png>



Code-Beispiel

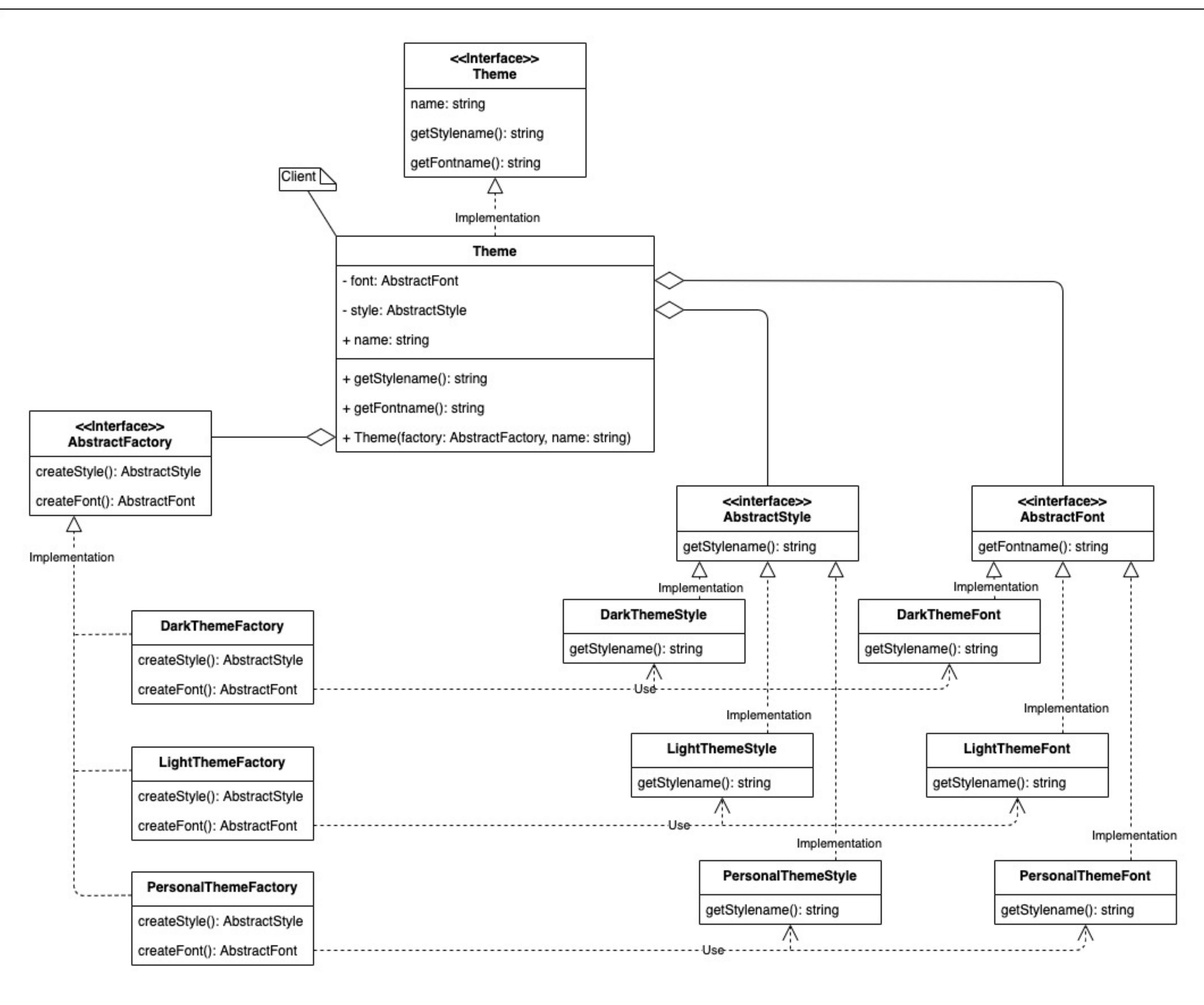
Abstract Factory

Abstrakte Fabrik



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

Demo



Zweck

- Schnittstelle zum Erzeugen verwandter Objektfamilien
- Verbergung der konkreten Klassen



Quelle: <https://refactoring.guru/images/patterns/diagrams/abstract-factory/problem-en-2x.png>

Anwendbarkeit

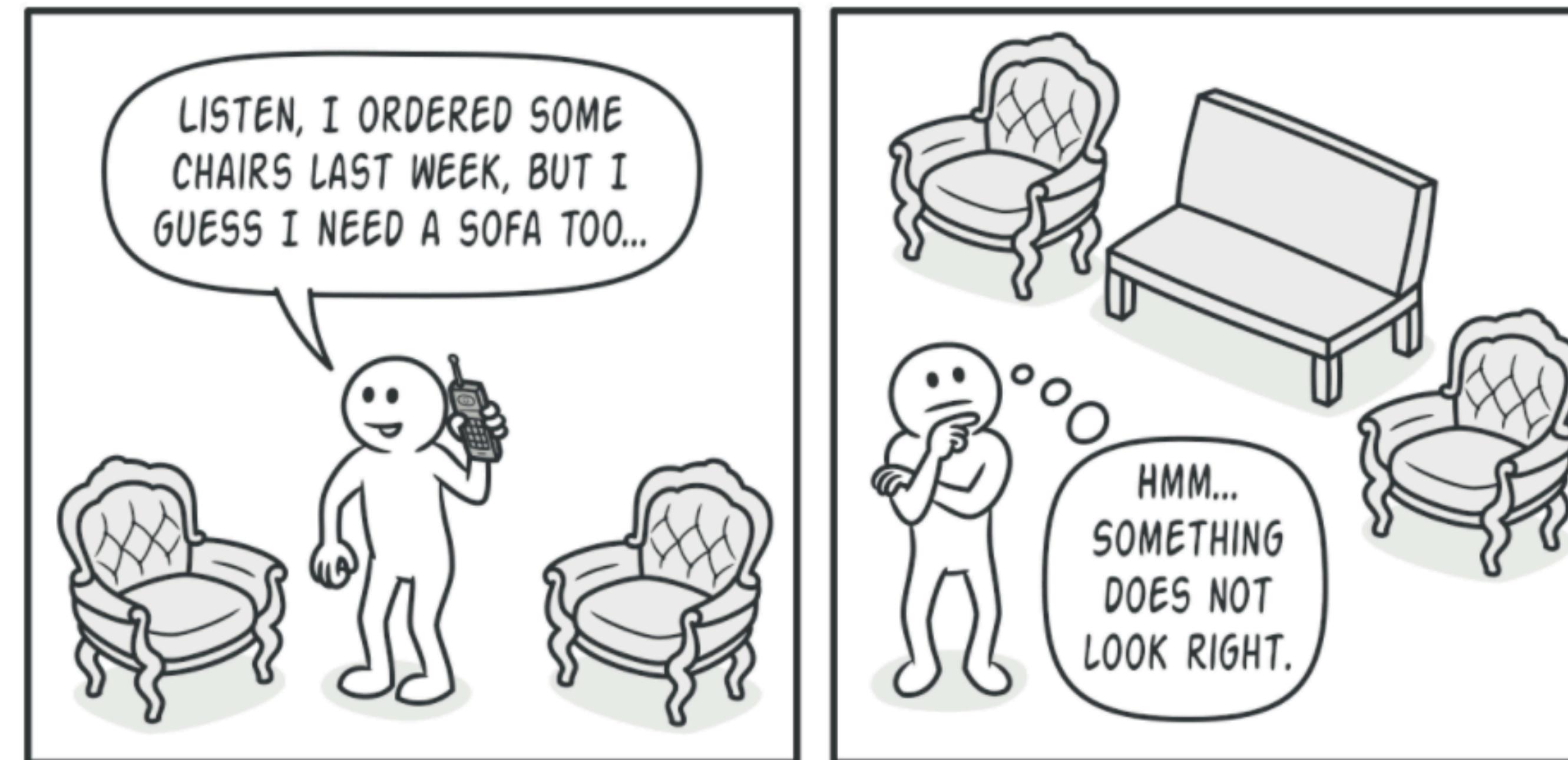
- System soll unabhängig von Generierung, Komposition und Darstellung seiner Objekte arbeiten
- System soll von mehreren Produktfamilien konfiguriert werden
- Familien sollen gemeinsam verwendet werden (Zwang)
- nur Schnittstellen sollen verwendet werden dürfen

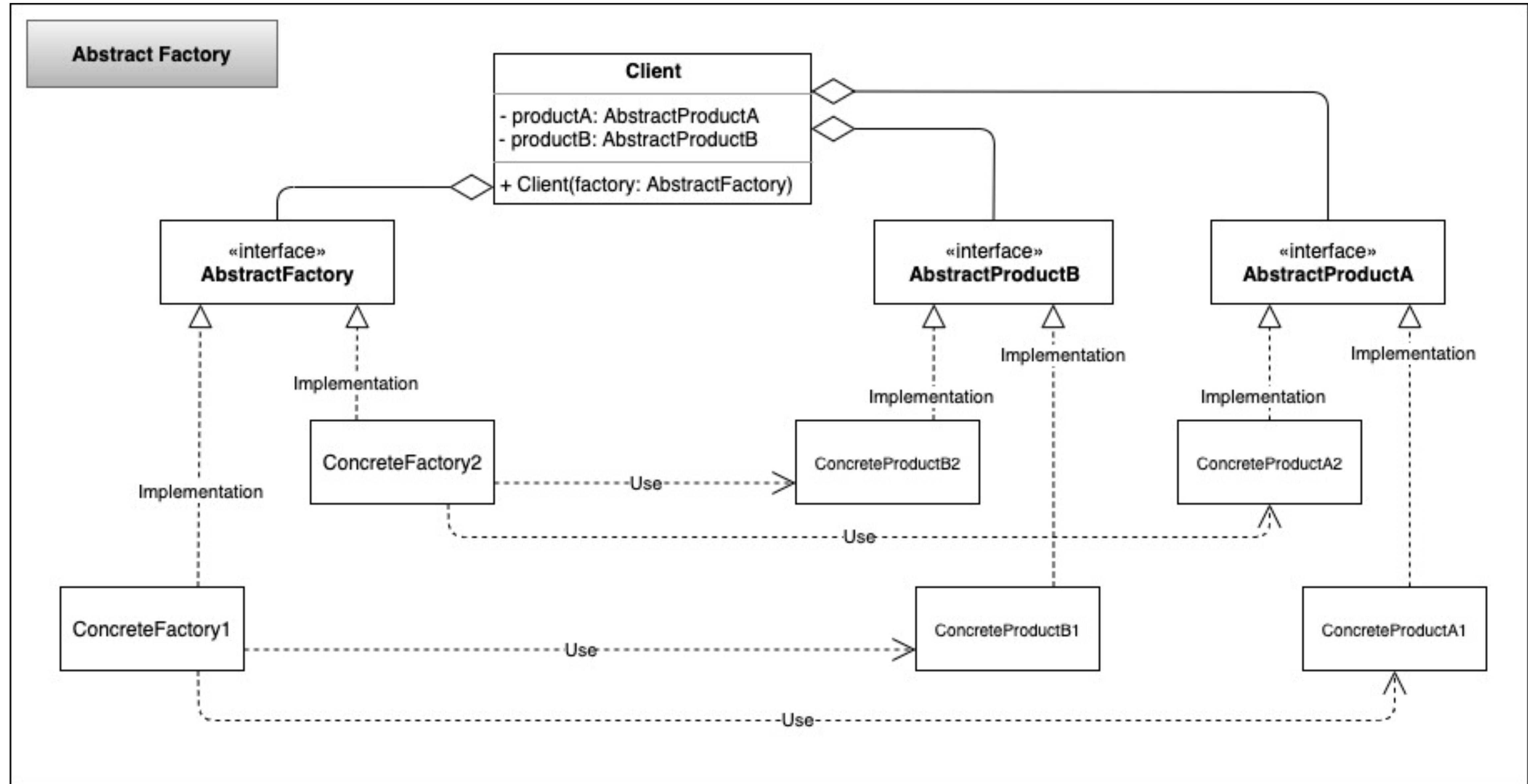
Konsequenzen (Vor- und Nachteile)

- Isolierung konkreter Klassen
- Einfacher Austausch von Produktfamilien
- Produktkonsistenz (Einhaltung der Zusammenarbeit der Produkte)
- Unterstützung neuer Produktarten (Bewahren des OCP)

Erweiterbarkeit

- Fabriken als Singletons
- Variierung der Erzeugung der Produkte in den Fabriken (z.B. durch Fabrikmethode oder Prototype, etc)





Code-Beispiel

Fragen?

Happy Coding! :)