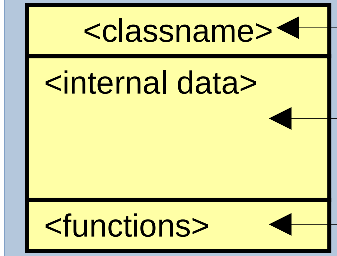
Software Design Patterns Overview in UML

Quick recap of class diagrams in UML

UML (Unified Modelling Language) besides other things, permits for a graphical representation of classes & their relationship → This shall aid in understanding a software system's class structure & thus facilitate the discussion of software design patterns ...

Class Diagram



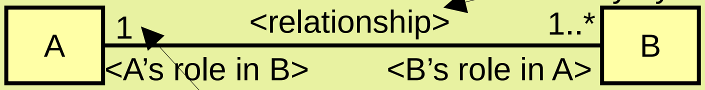
The **name** of the class

Internal **data („attributes“ in UML lingo)**, divided here in two sections: private (leading ‚-‘) & public (leading ‚+‘)
Note: There are other indicators, but these will not be used here!

Functions („operations“ in UML):
Functions may be private (leading ‚-‘) & public (leading ‚+‘) as well ...
Note: There are other indicators, but these will not be used here!

Class Association

An **association** describes the **relationship** of classes which may by directed by ‚<‘ or ‚>‘ arrows at line ends



The **cardinality** (UML lingo) describes the range # of possible instances (where ‚*‘ means zero or more)

- Reference to base (or ‚super‘) class from derived class
- Dependency/navigation (dashed: Interface)
- Composition
- Implementation
- Aggregation

Ref: https://holub.com/uml/

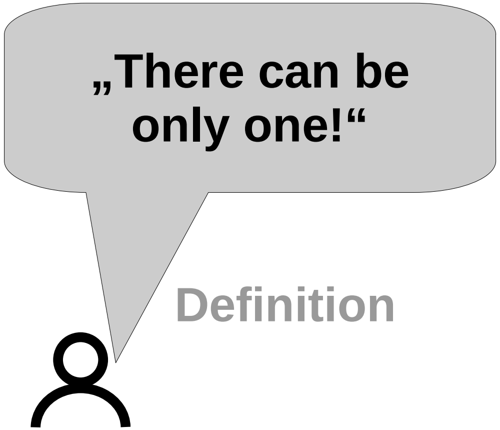
What’s next?

- Creational: Singleton
- Creational: Decoupling
- Creational: Factory Method
- Creational: Abstract Factory
- Creational ignored: Prototype, Builder ...

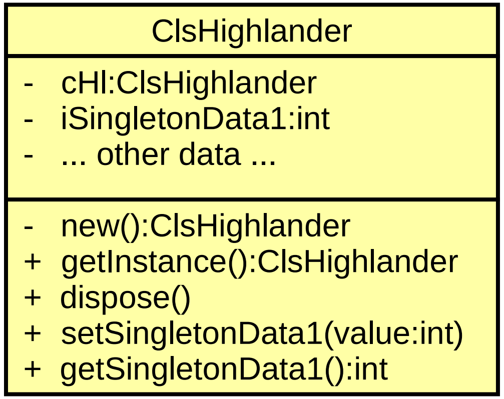
- Structural: Adapter
- Structural: Bridge
- Structural: Composite
- Structural: Decorator
- Structural: Facade
- ...

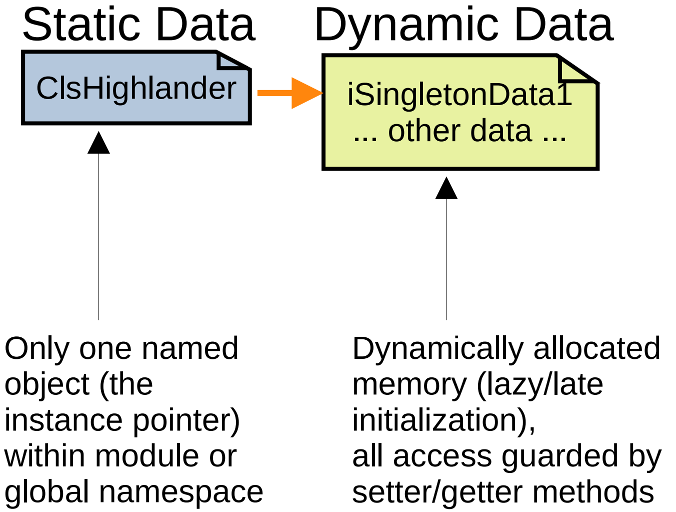
- Behavioural: Observer
- Behavioural: Strategy
- Behavioural: Chain of Responsibility
- Behavioural: Command
- ...

Design Patterns: Singleton



Class Diagram Example





Pros:
1. Declutters namespace
2. Minimizes memory footprint (as a singular object): Use for costly objects (like database connections etc.)
3. Simple (to understand & to control …)

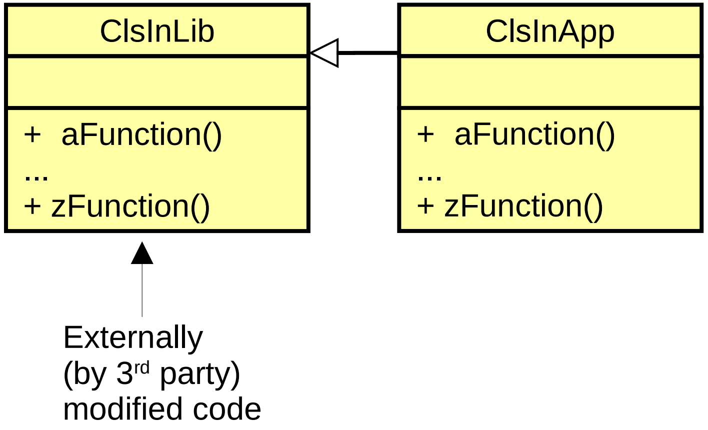
Cons:
1. Simpler/lighter implementations for a similar outcome usually possible/more efficient

Design Patterns: Decoupling

„Reduce or eliminate dependencies between different components by isolating changes.“

Definition

Class Diagram Example



Pros:
1. Permits the use of generators for parts of the application with minimal impact on existing code.
2. Permits for partial extensions & the implementation of wrappers for 3rd party code.

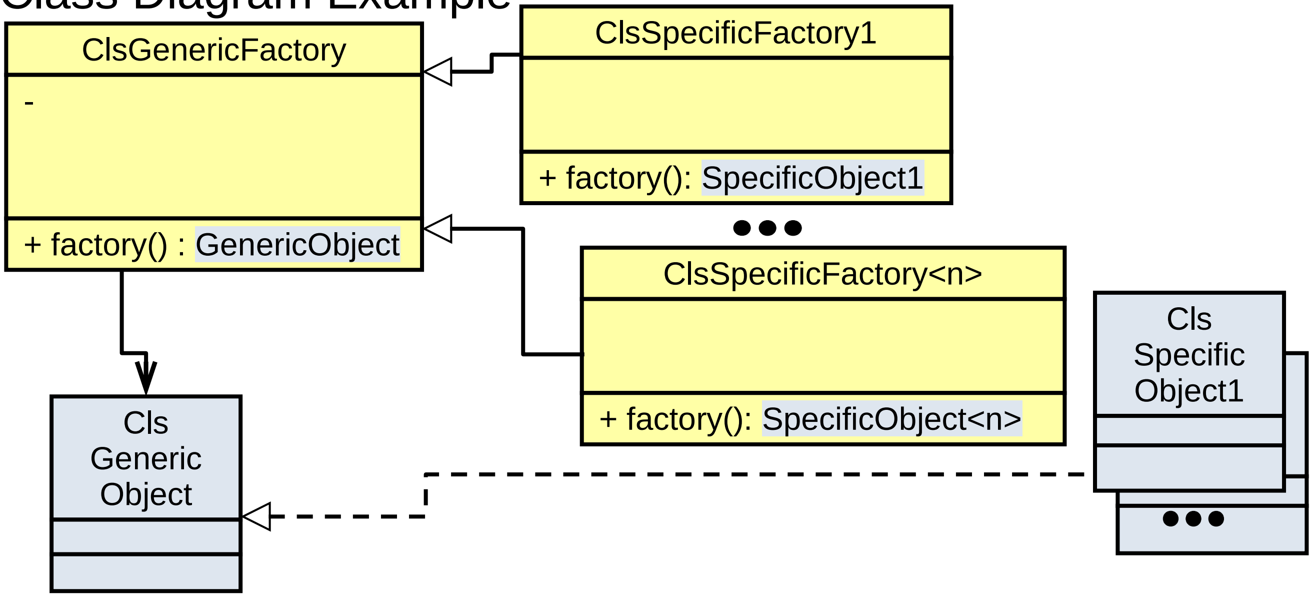
Cons:
1. Performance overhead
2. Debugging complexity
3. Potential ‚over-engineering‘ (cost)

Design Patterns: Factory Method

„Provide an interface for creating objects in a superclass, but allow subclasses to alter the type of objects that will be created“
(s.b. *IoC & +dependency injection).

Definition

Class Diagram Example



Pros:
- Allows for a general treatment via generic objects in a superclass (ex.: apply complex computations to custom user objects).

Cons:
- Introduces additional derivatives/interfaces & thus complexity

* „Inversion-of-control“: Control flow of a program is inverted, a framework or external code takes control and calls into the application code at appropriate spots.
\+ „Dependency Injection“: External code (container/framework) is responsible for providing an object's dependencies rather than the object creating them directly.

Design Patterns: Abstract Factory

„Encapsulate a group of individual factories that have a common theme, allowing clients to create families of related objects without specifying their concrete classes.“

Definition

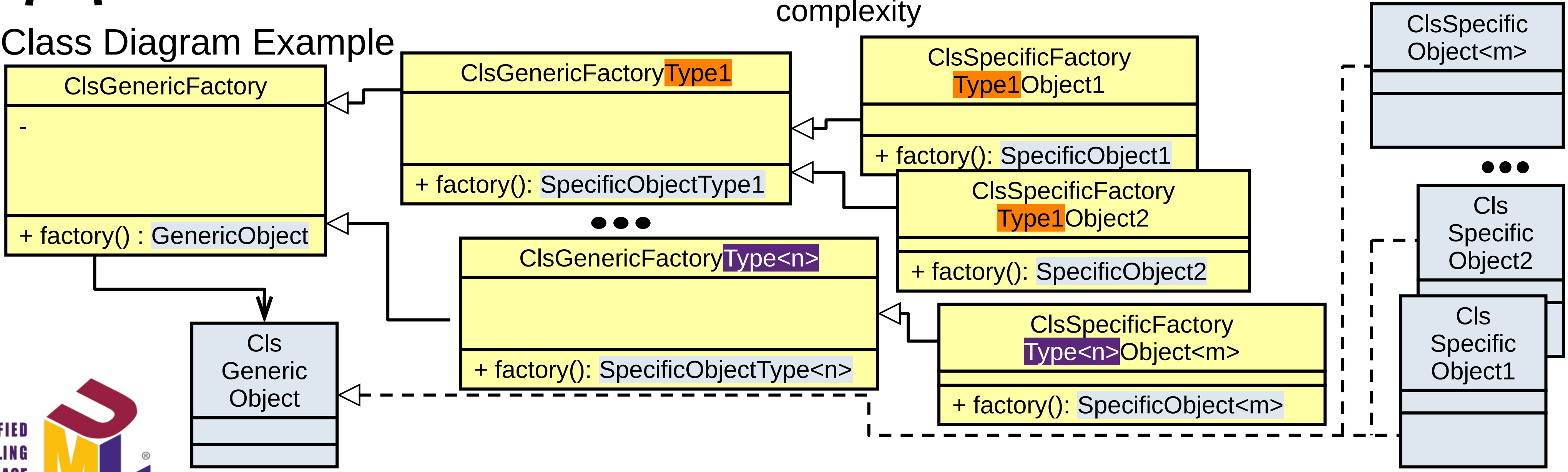
Pros:

1. Allows for a general treatment via generic objects in a superclass & still permitting type specific attributes/methods handling.
2. Enables decoupling of object creation from – say – an existing library, thus permitting custom types for generic frameworks.

Cons:

- Introduces additional derivatives/interfaces & thus complexity

Class Diagram Example

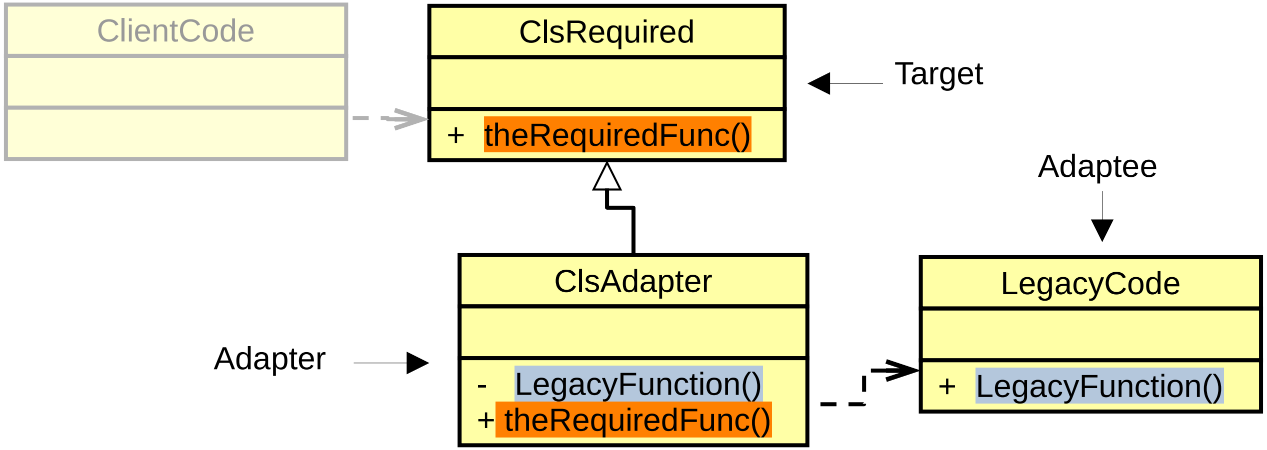


Design Patterns: Adapter

„Two incompatible interfaces, libs or systems are enabled to cooperate by a ‚translator‘.“

Definition

Class Diagram Example



Pros:
1. Permits for the elegant integration of legacy code (or systems).
2. No legacy code modification & no change in own code base necessary (ideally!) → code reuse

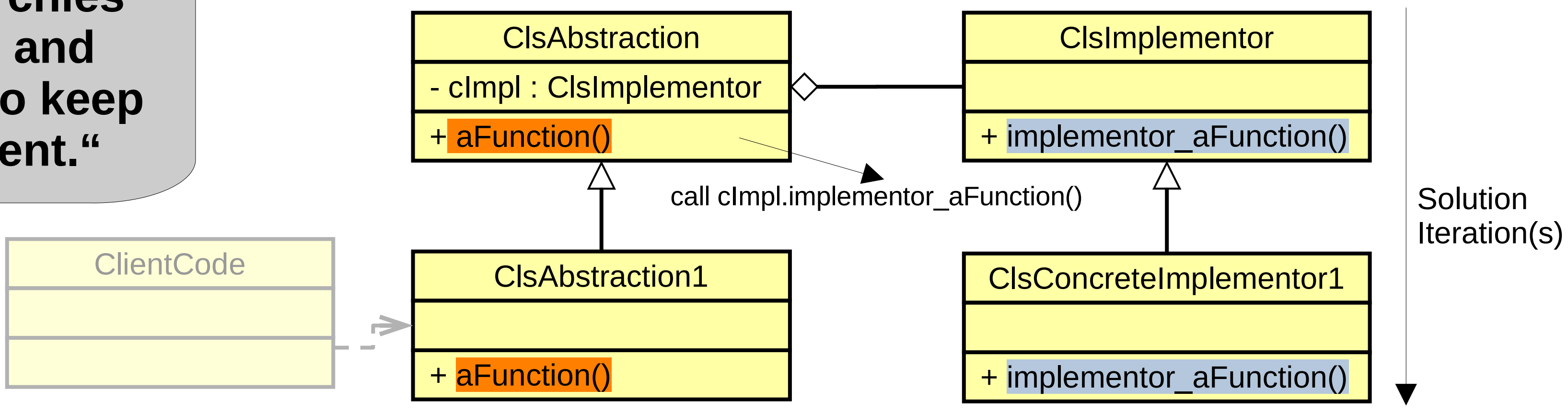
Cons:
1. Performance issues when used w/ heavy transformation (or transfer!) loads.
2. Many adapters may pollute code → complexity

Design Patterns: Bridge

„Separate hierarchies for abstraction and implementation to keep code independent.“

Definition

Class Diagram Example



Pros:
1. Implementation possible in parallel with design phase
2. Compile time independence of code bases, maybe for different target platforms etc.
3. „Prefer composition over inheritance!“

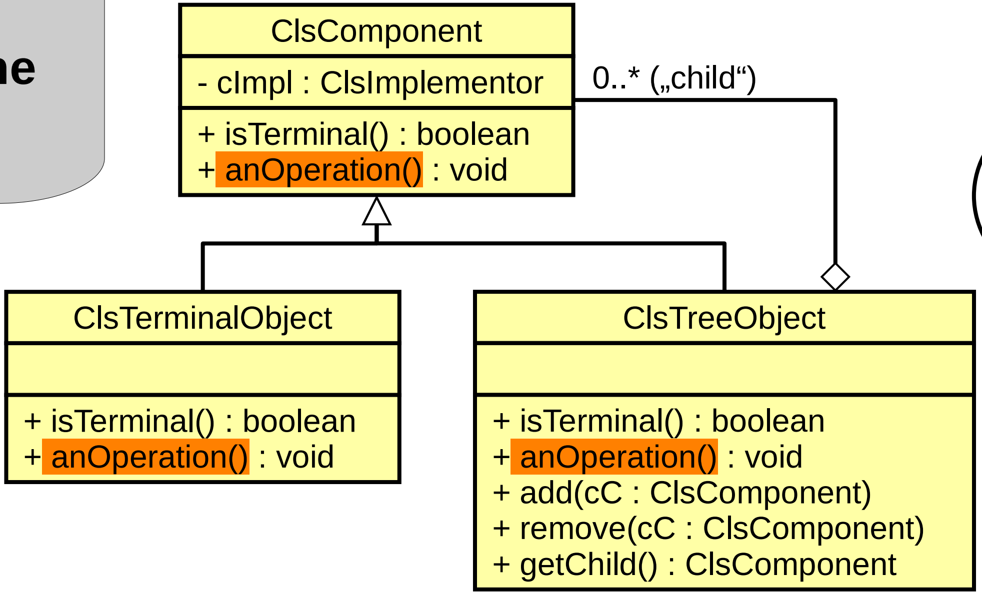
Cons:
- Any?

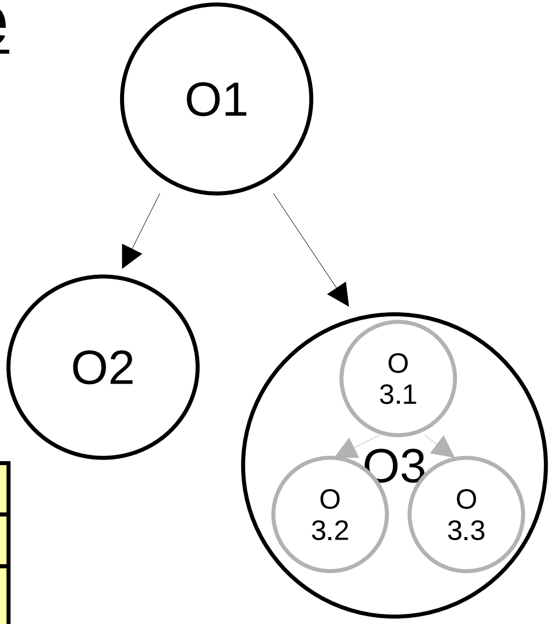
Design Patterns: Composite

„Treat individual & composite objects the same – codewise.“

Definition

Class Diagram Tree Example





Pros:

1. Permits working with leafs, lists, trees & similar structures of varying depth & sizes using all the same interface → ease of handling & flexibility
2. Identical code for different objects → simple usage

Cons:

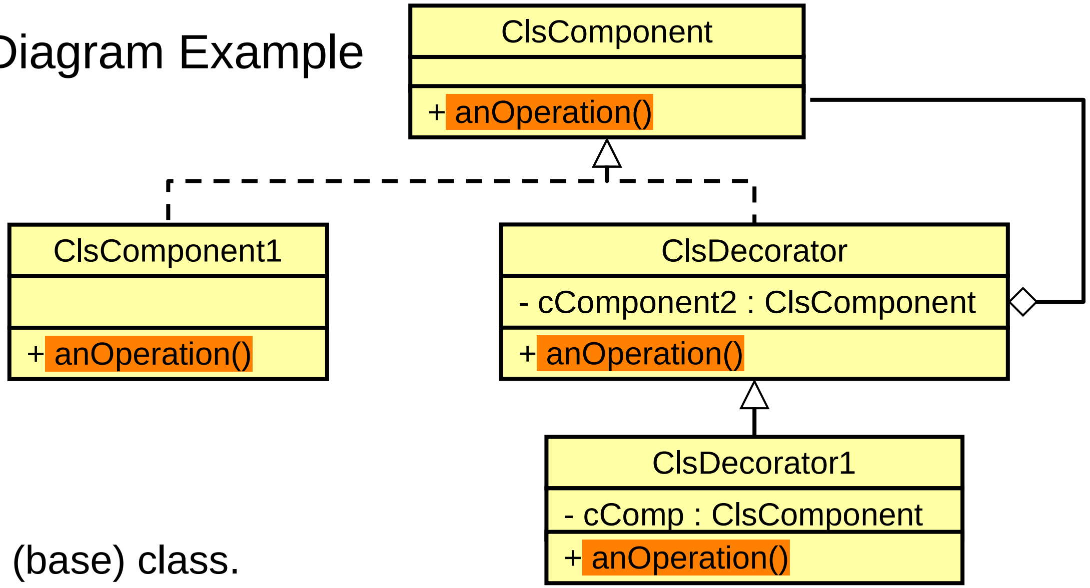
1. Component functions may grow → function creep/cancer
2. Restrictions harder to enforce

Design Patterns: Decorator

„Adds behaviour dynamically to individual objects without affecting class – at runtime!“

Definition

Class Diagram Example



Pros:

1. Add functionality without affecting (base) class.
2. Permits new object compositions out of existing parts (component & decorators → new one).
3. „Open/Closed Principle“: Open to additions/extensions/behavioural mods, closed to changes
4. Favours „composition-over-inheritance“ as usually recommended
5. Good for implementing optional features ...

Cons:

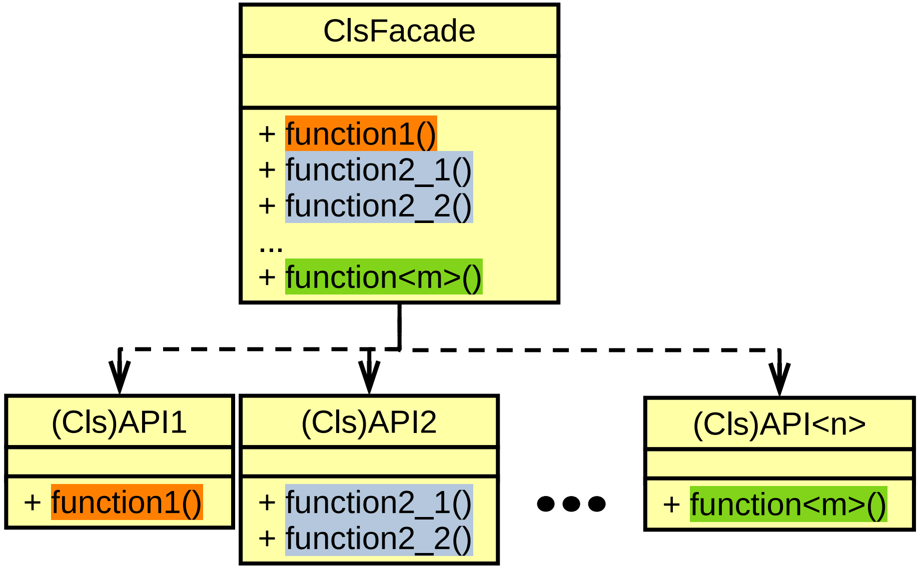
1. Be careful with decorator order – if nec. that is!
2. Complexity increase (as usual!)
3. May lead to „over-engineering“ ...

Design Patterns: Facade

„Provide a simple default view of a subsystem that is good enough for most clients.“

Definition

Class Diagram Example



Pros:

1. Simplifies programming by hiding complexities of – possibly several - APIs.
2. Promotes modularity, reduces dependencies
3. May aggregate multiple APIs to a single (simplified) one

Cons:

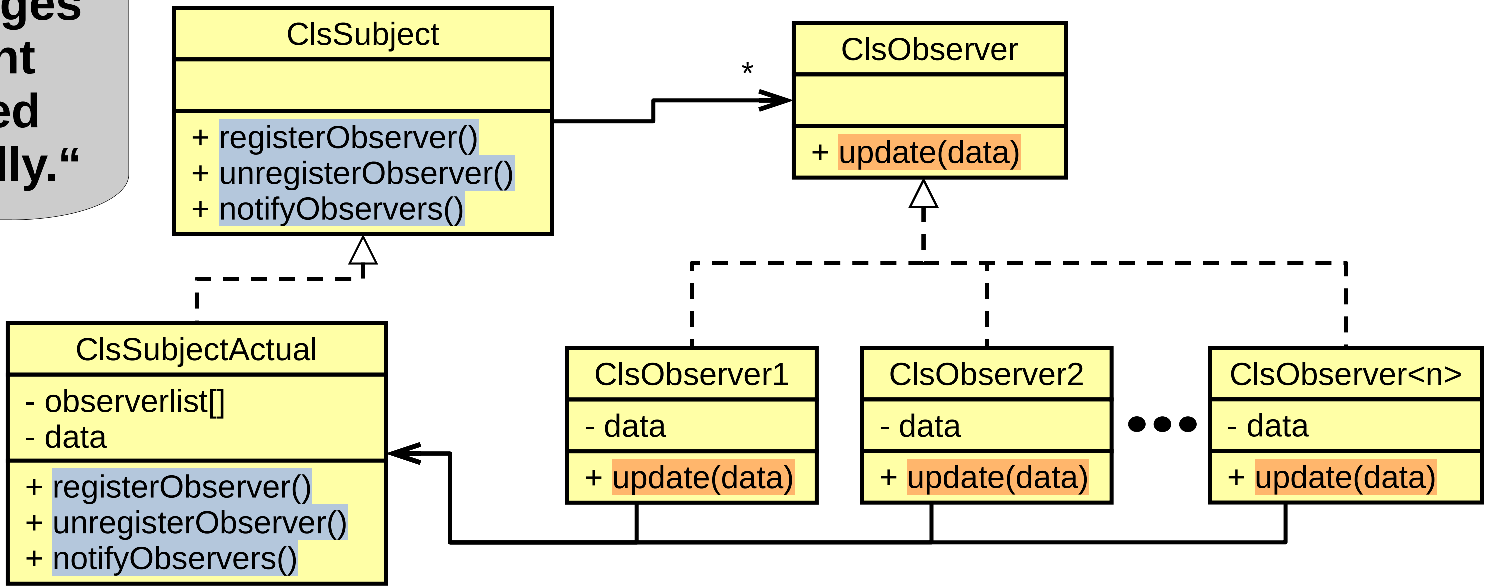
The usual: Performance, complexity, potential over-engineering

Design Patterns: Observer

„When a ‚subject‘ changes state, all its dependent ‚observers‘ are notified and update automatically.“

Definition

Class Diagram Example



Pros:
Permits ‚event-driven‘ processing of observers by still maintaining an only loose coupling to the subject.

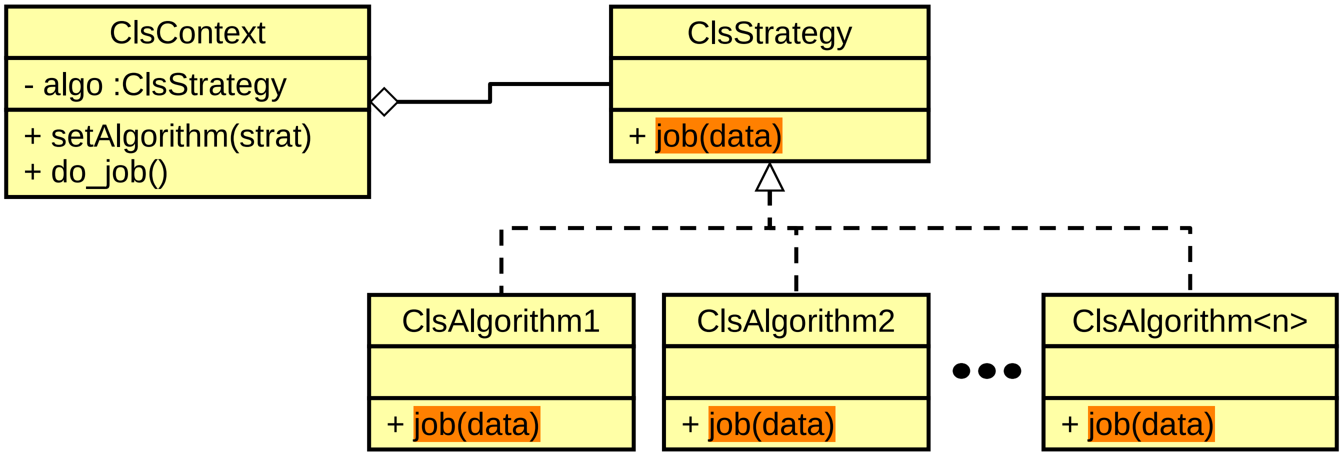
Cons:
Performance may become an issue. Potential over-engineering: With a fixed # of observers or too simple objects. Also a fixed processing order may render this solution unuseful.

Design Patterns: Strategy

„Define a family of algorithms in separate classes to be swapped at runtime.“

Definition

Class Diagram Example



Pros:
Clean de-coupled processing, the context doesn’t need to know algorithm specifics.

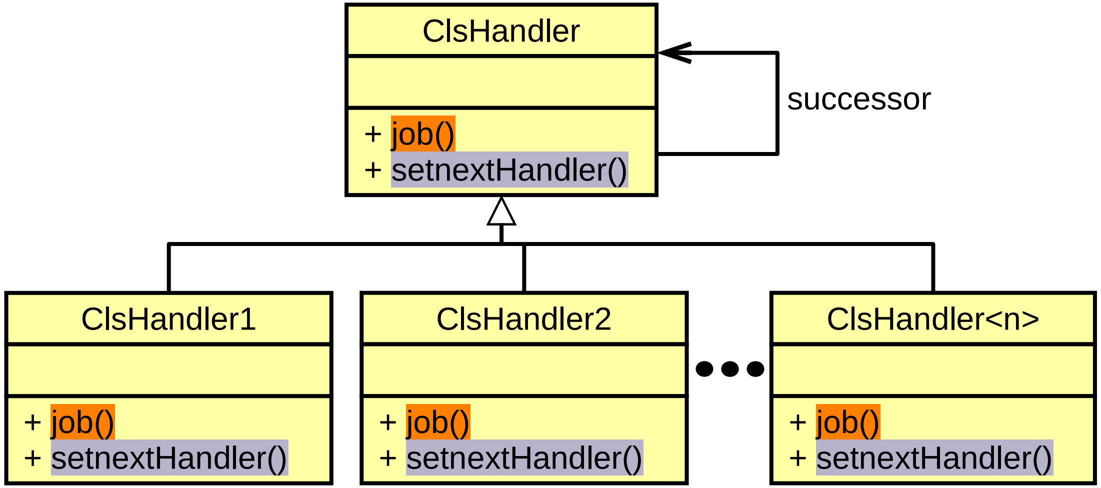
Cons:
1. I’d rather call it (jump table) function pointers …
2. Application must be aware of the different strategies.
3. Application requires a context and a separate strategy instance.

Design Patterns: Chain-of-responsibility

„A chain of receiver objects to either handle a request and/or forward it to a successor."

Definition

Class Diagram Example



Pros:
1. Flexible scalable solution – even at runtime!
2. Sender doesn't need to know anything about receiver processing.

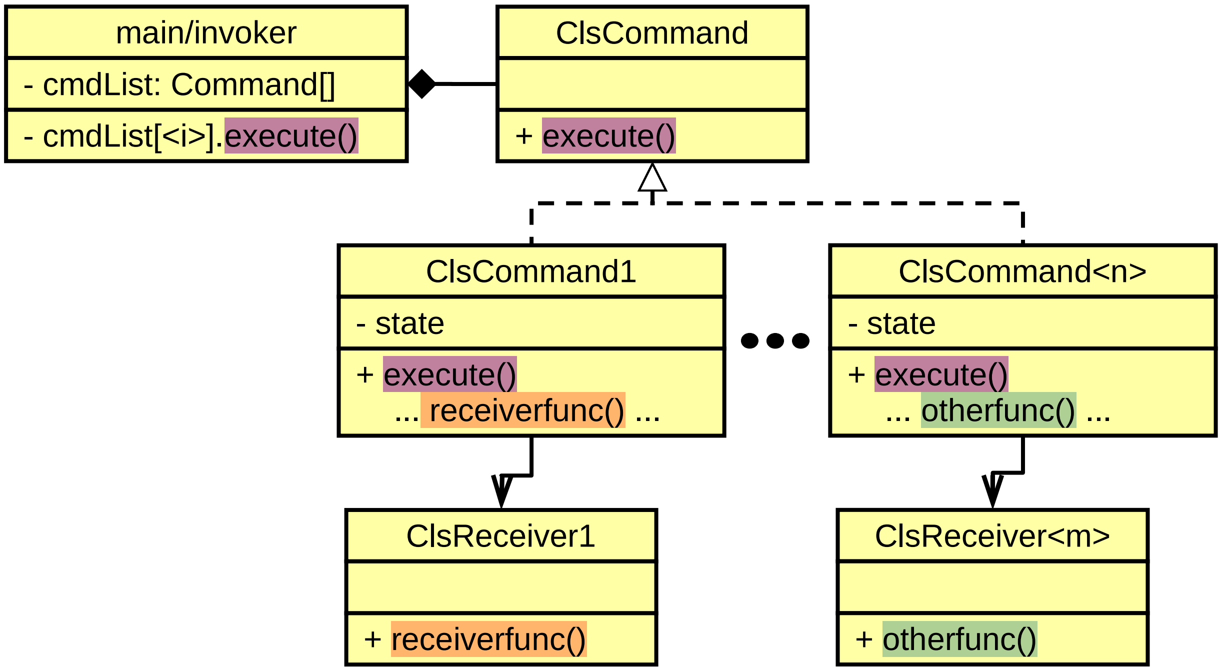
Cons:
1. Performance degradation possible.
2. Debugging may be difficult.

Design Patterns: Command

„Turns a request into a stand-alone object called a command.“

Definition

Class Diagram Example



Pros:

1. Flexible, extendable solution, runtime re-configurable
2. Permits for easy undo/redo integration

Cons:

Not for simple, tightly coupled applications (over-engineering!)