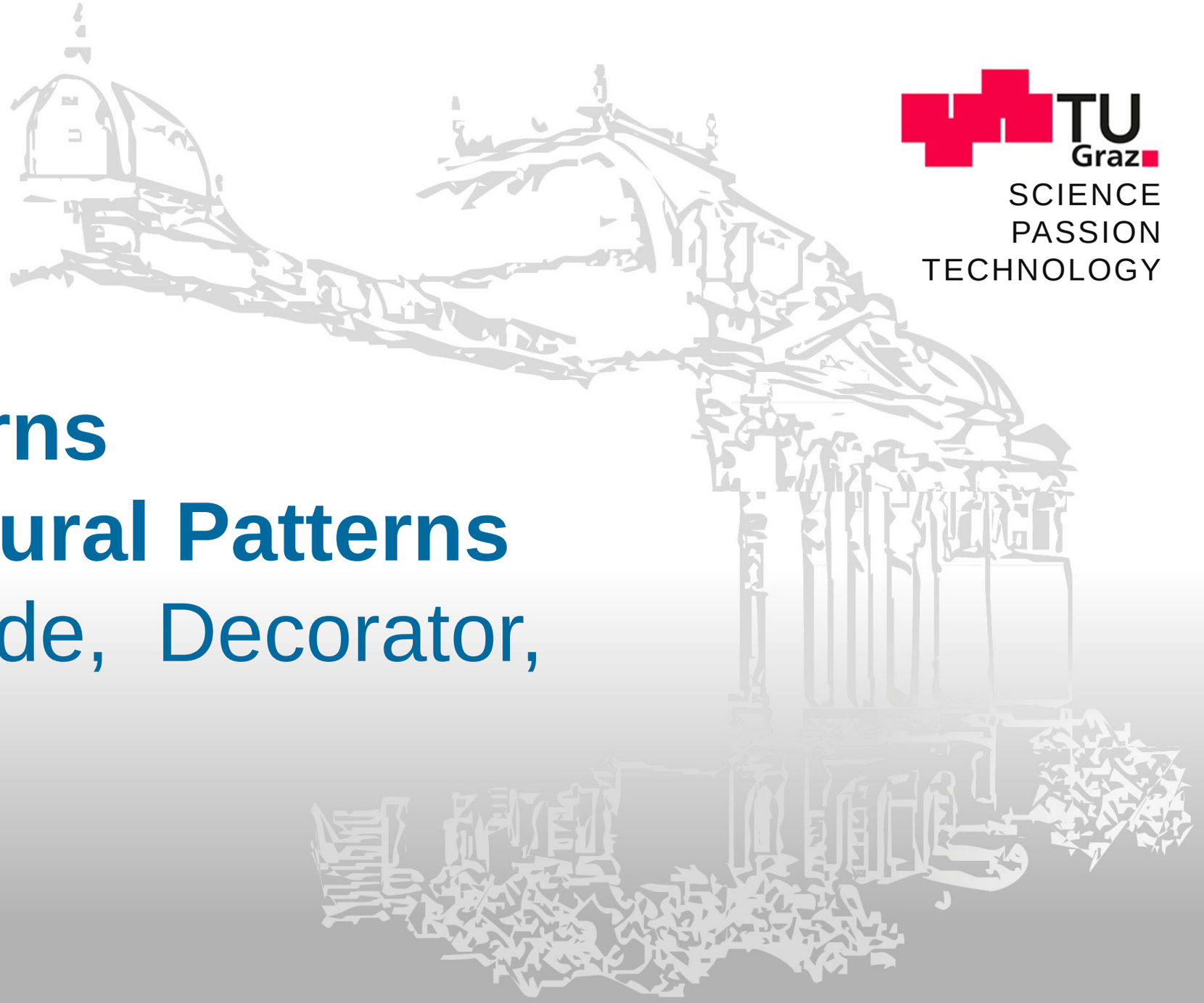
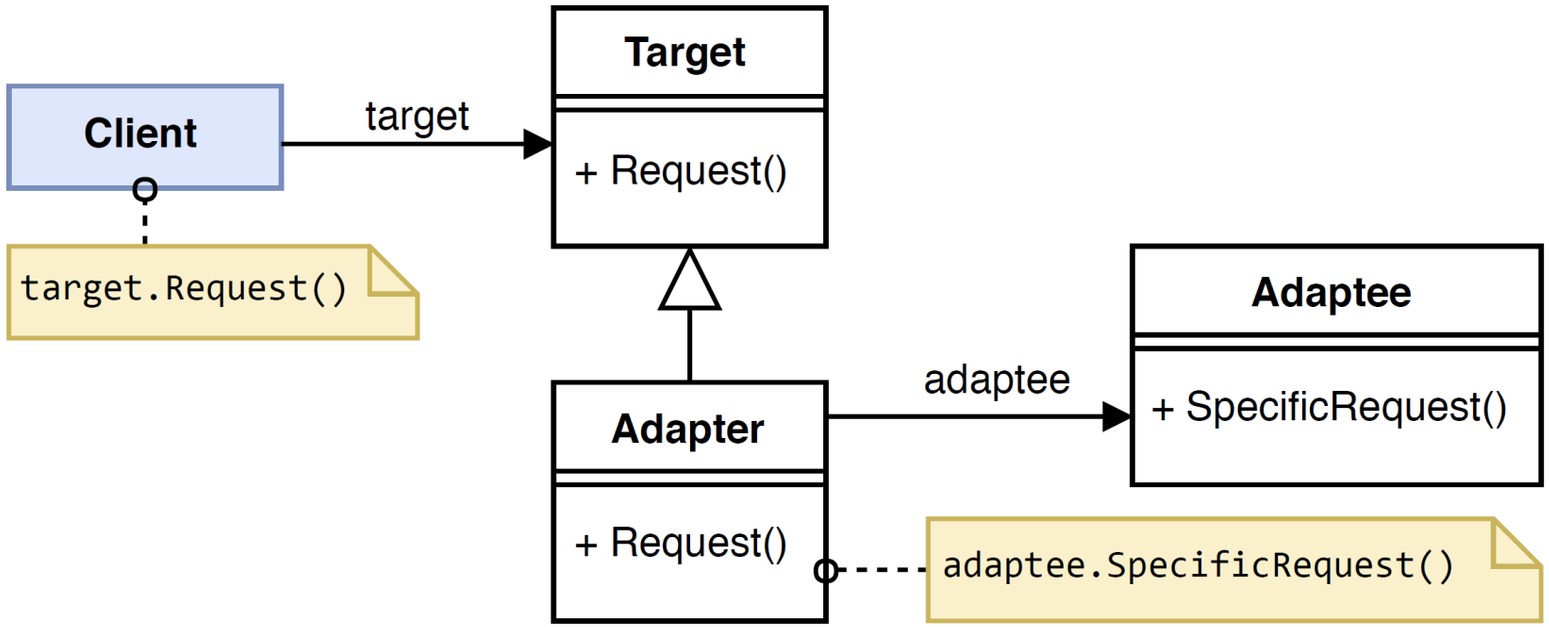


TU Graz
SCIENCE
PASSION
TECHNOLOGY

Design Patterns
Part 3: Structural Patterns
Adapter, Facade, Decorator, Proxy

Adapter

Wrap around a class to make it compatible to another interface.



Adapter

Context: Working with multiple different frameworks or libraries.

Problem: How to make incompatible classes work together?

Forces:

- Existing class interface does not match the one you need.
- You want to reuse the functionality (not just copy it).
- Source code of used class may not be available (copying or changing it is not possible)
- Class may be sealed (inheritance is not possible)

Solution:

- Create an Adapter class which wraps around the Adaptee.
  Variant: **Class Adapter** (inherits from Adaptee)
  Variant: **Object Adapter** (contains Adaptee member)
- Implement the desired new interface using the methods of the Adaptee as underlying basis.

Consequences: (Class Adapter)

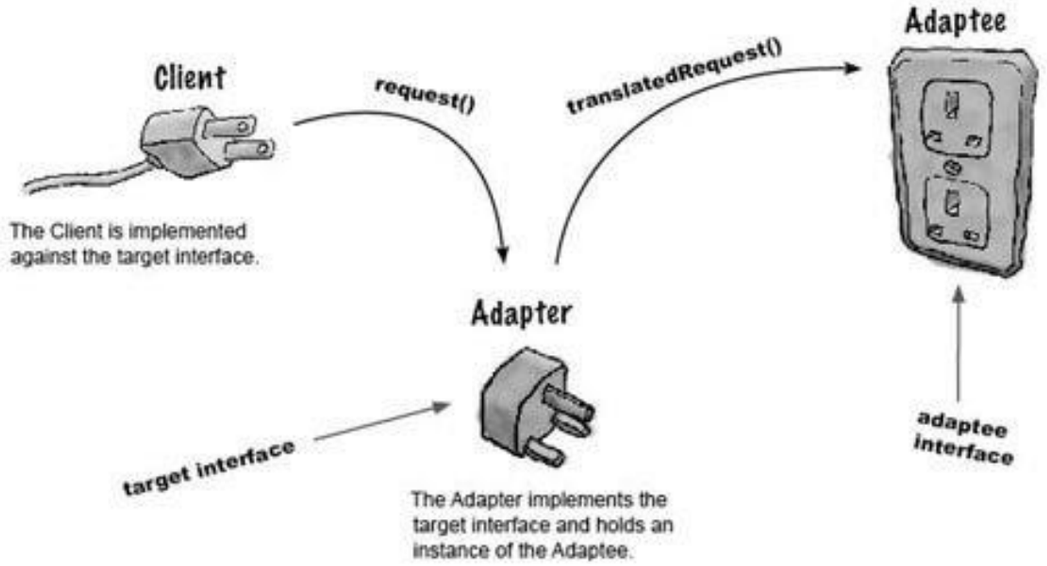
\+ Allows to use override mechanisms (e.g. protected methods, V-table, access to protected members).
\+ No additional indirection.
~ Inheritance approach (all methods of adaptee are inherited automatically, only changes have to be implemented)
\- Won't work when we want to adapt a class and all its subclasses (liskov substitution!), because it is on a different branch of subclasses.

Consequences: (Object Adapter)

\+ Works with base Adaptees and all subclasses (allows liskov substitution).
\+ Adapter hides underlying type of Adaptee (breaks inheritance hierarchy, composition over inheritance!).
~ Explicit implementation approach (no methods inherited automatically; all needed methods have to be implemented explicitly)
\- Adds additional layer of indirection.

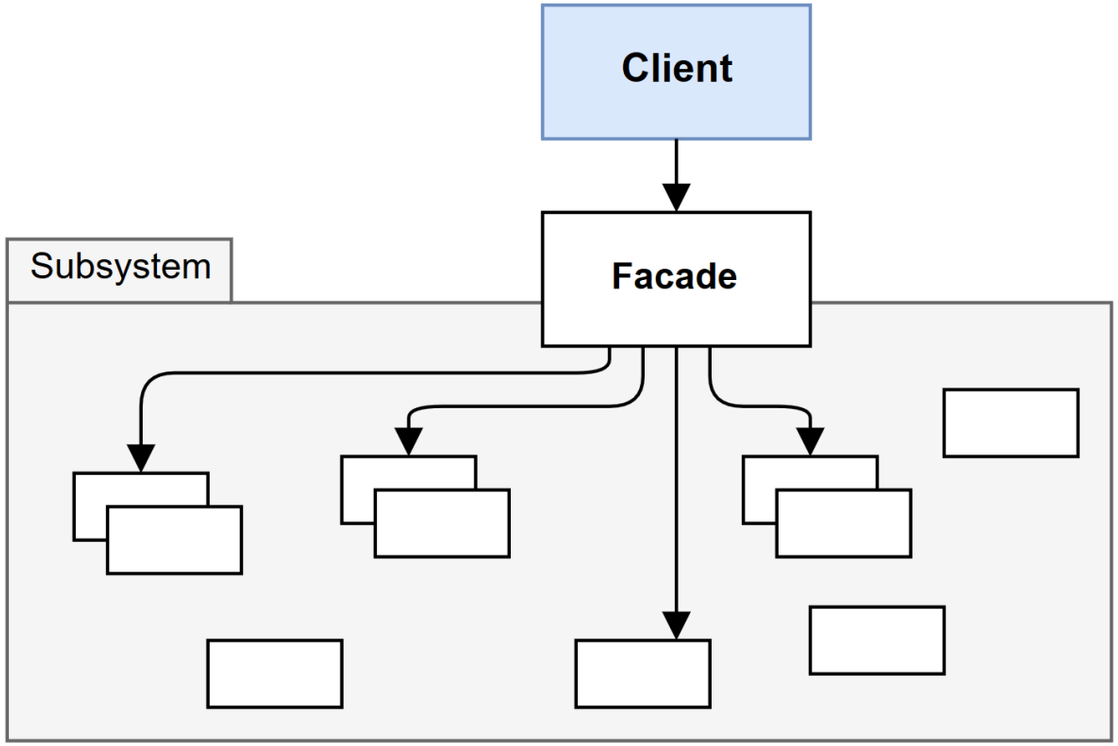
Adapter

Wrap around a class to make it compatible to another interface.



Façade

Provider a higher-level interface to a system.



Façade

Context: Working with a complex structure having many functions, maybe even with different programming paradigms (e.g. object-oriented vs. structured).

Problem:

- How to make it easier to use a complex system of functions, or to use functions of different programming paradigms in a more intuitive way?

Forces:

- Different programming paradigms from different platforms.
- Developers are used to their own environments and conventions.
- Developing heterogenous paradigms makes programs more difficult to maintain.
- Changing the source is seldom possible.
- Details should be hidden away / abstracted away.

Solution:

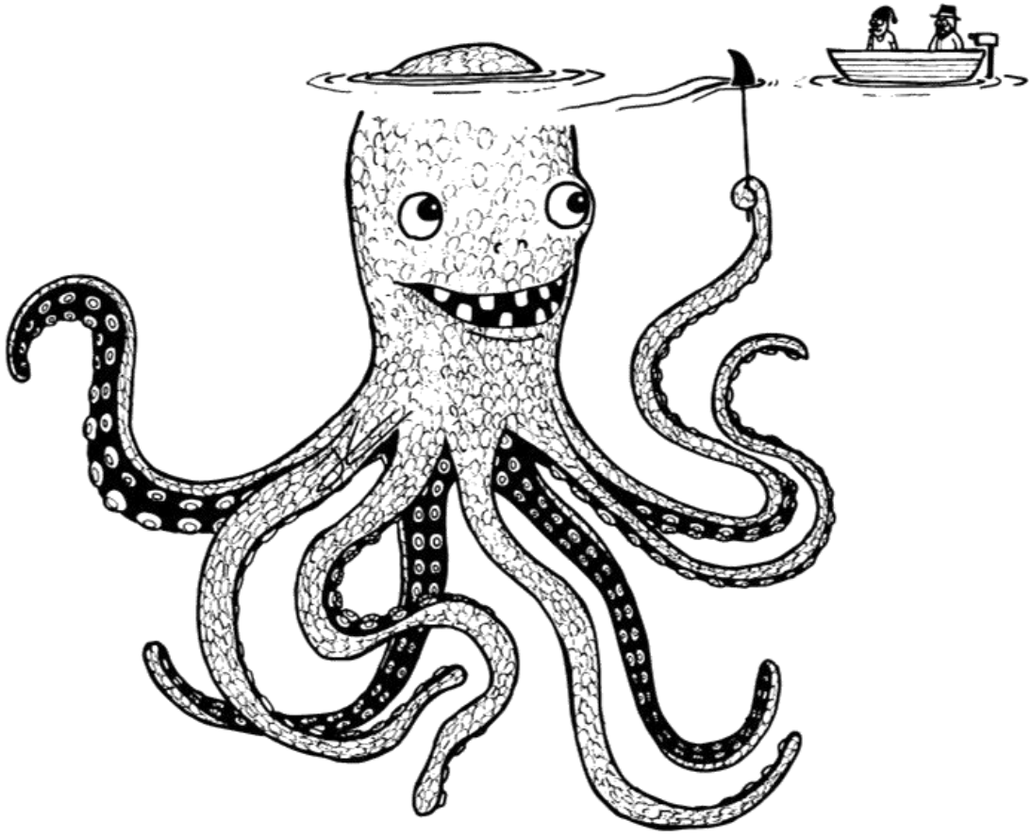
- Implement a simpler, more high-level interface to be used by the client.
- Hide the complexities (implementation details) of the larger system.
- Encapsulate non-OO API data & functions within concise, robust, portable, maintainable, cohesive OO class interface.

Consequences:

+ Provides concise, cohesive and robust higher-level object-oriented programming interfaces.
+ Easier usability and maintainability.
+ Code is more robust, easier to learn and maintain.

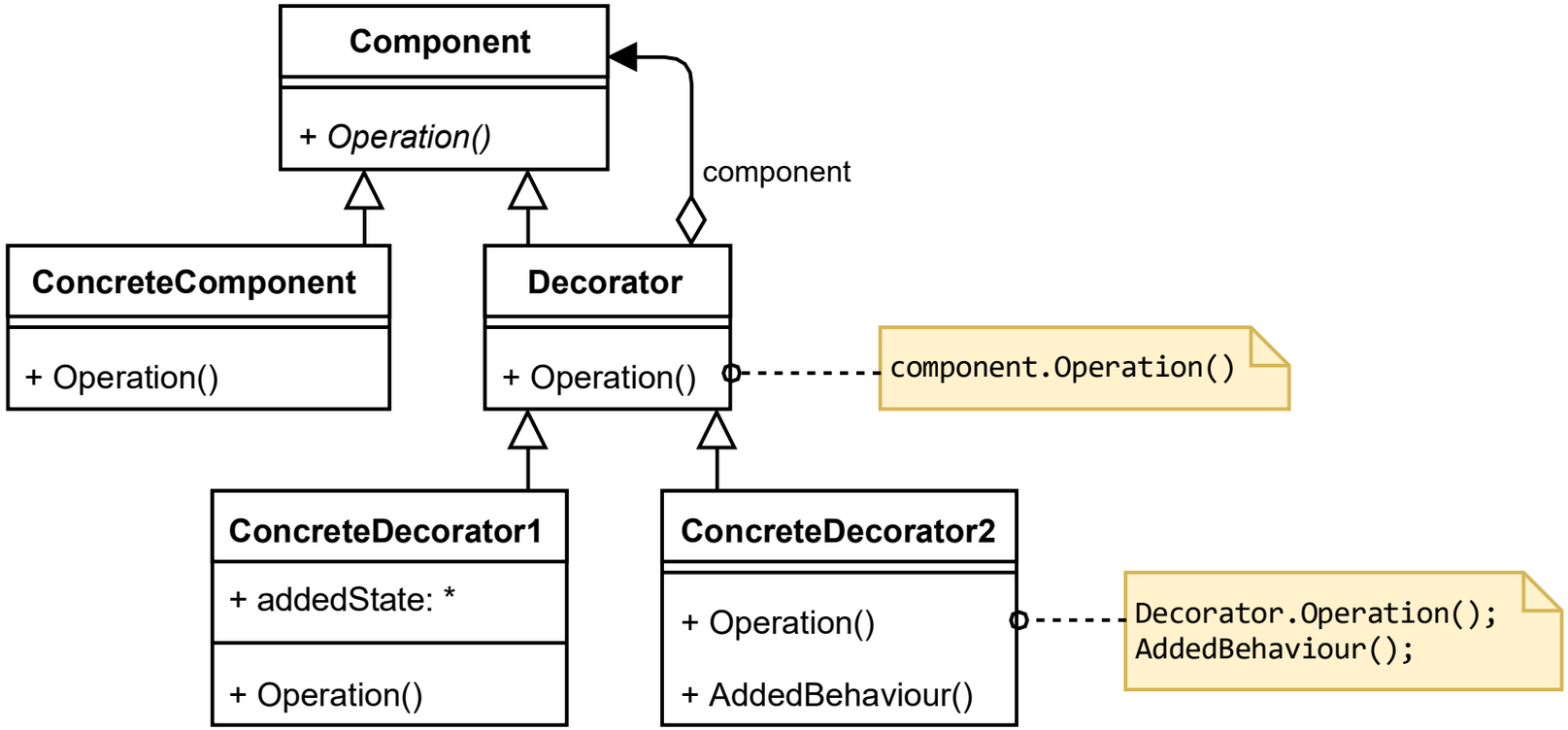
- May diminish functionality and lose benefits of underlying paradigm
- Performance degradation by adding an additional layer of abstraction

Façade



Decorator

Extend the functionality of an object, while maintaining the same interface.



Decorator

Context: Functional extension of objects.

Problem: How to add or extend functionalities without changing the objects.

Forces:

- We want to add responsibilities to individual objects dynamically and transparently, without affecting other objects.
- We want to reuse functionality.
- We want to assemble functionalities.
- We want to be able to withdraw responsibilities.
- The extension by subclassing is impractical:
  - large number of independent possible extensions.
  - hidden class definition or otherwise unavailable for subclassing

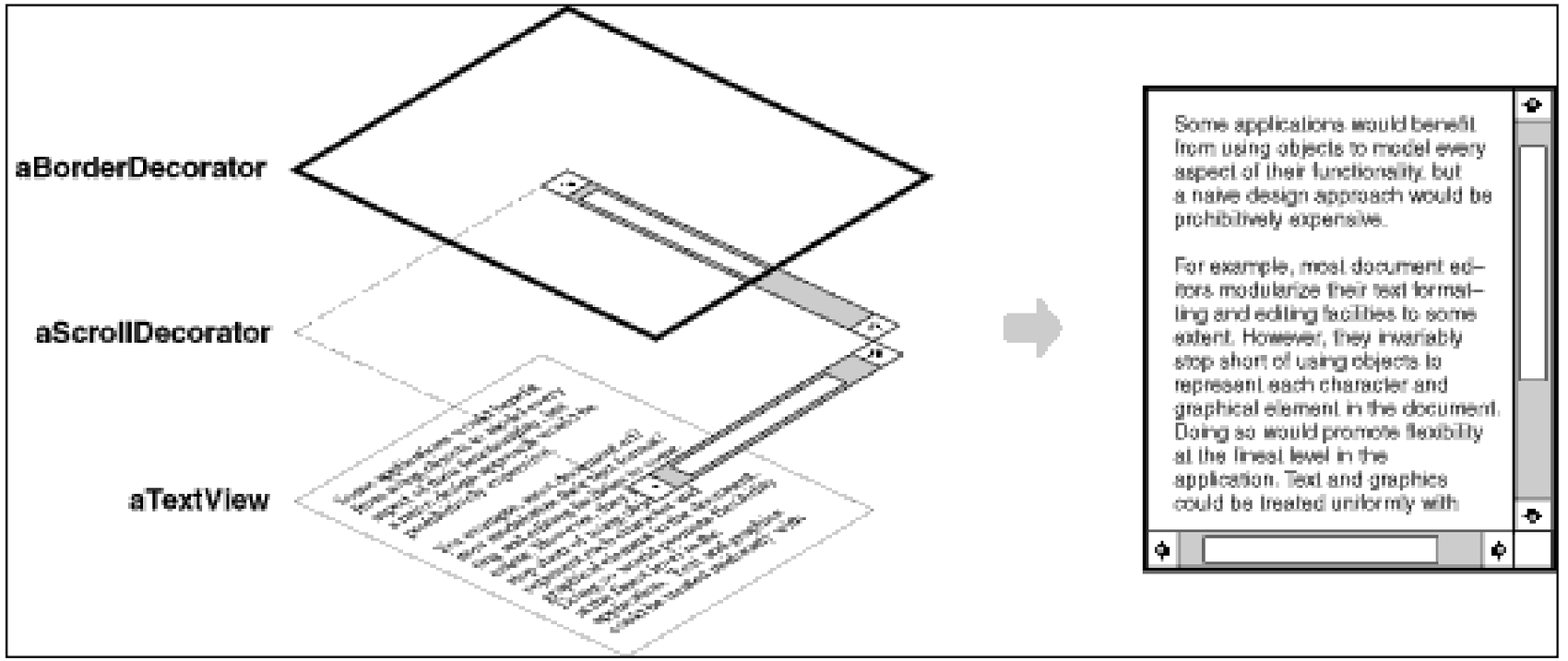
Solution:

- Define a Decorator which forwards requests to its Component object.
- The decorator may optionally perform additional operations before and after forwarding the request.

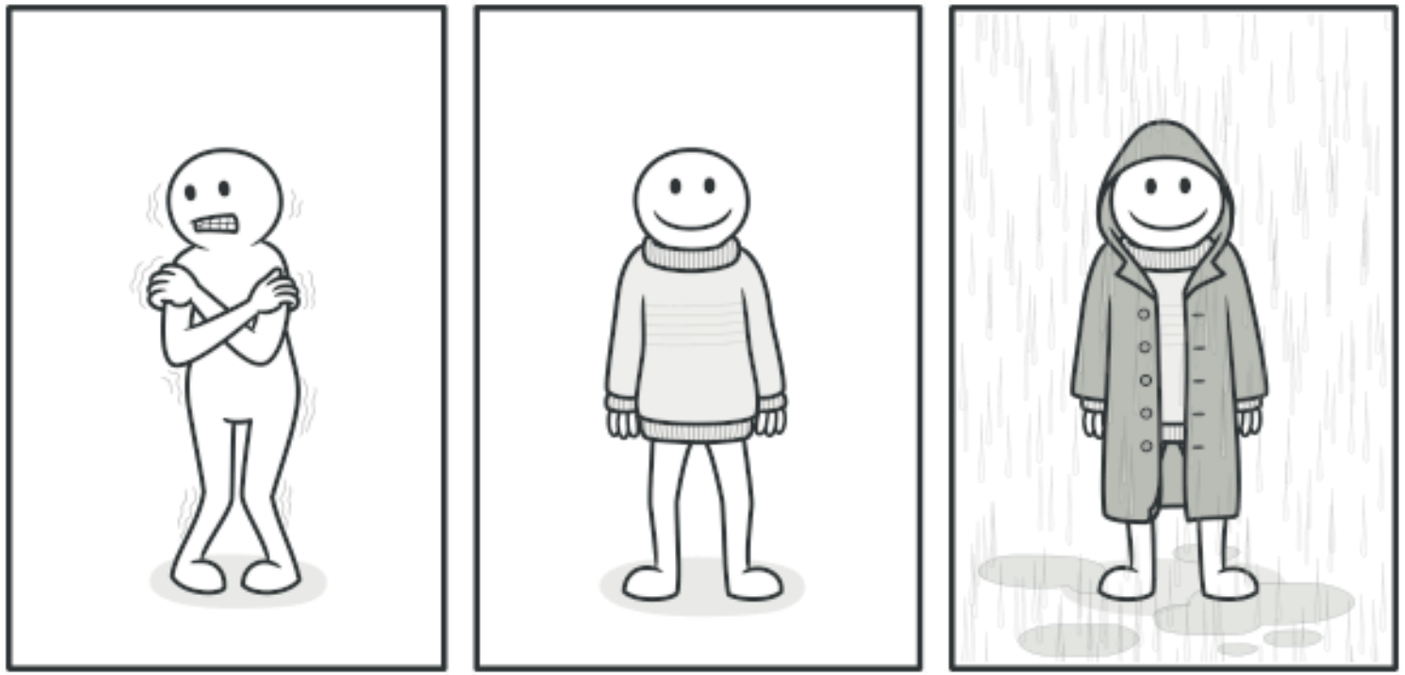
Consequences:

+ More flexibility by adding responsibilities
+ Flexibility responsibilities can be added and removed also at runtime
+ Decorators also make it easy to add a property twice
+ Avoids feature-laden classes high up in the hierarchy
+ Avoids the class explosion issue
- Decorator and its component are not identically
- Can be hard to learn and debug (lots of little objects only different in the way of their interconnection)

Decorator - Example

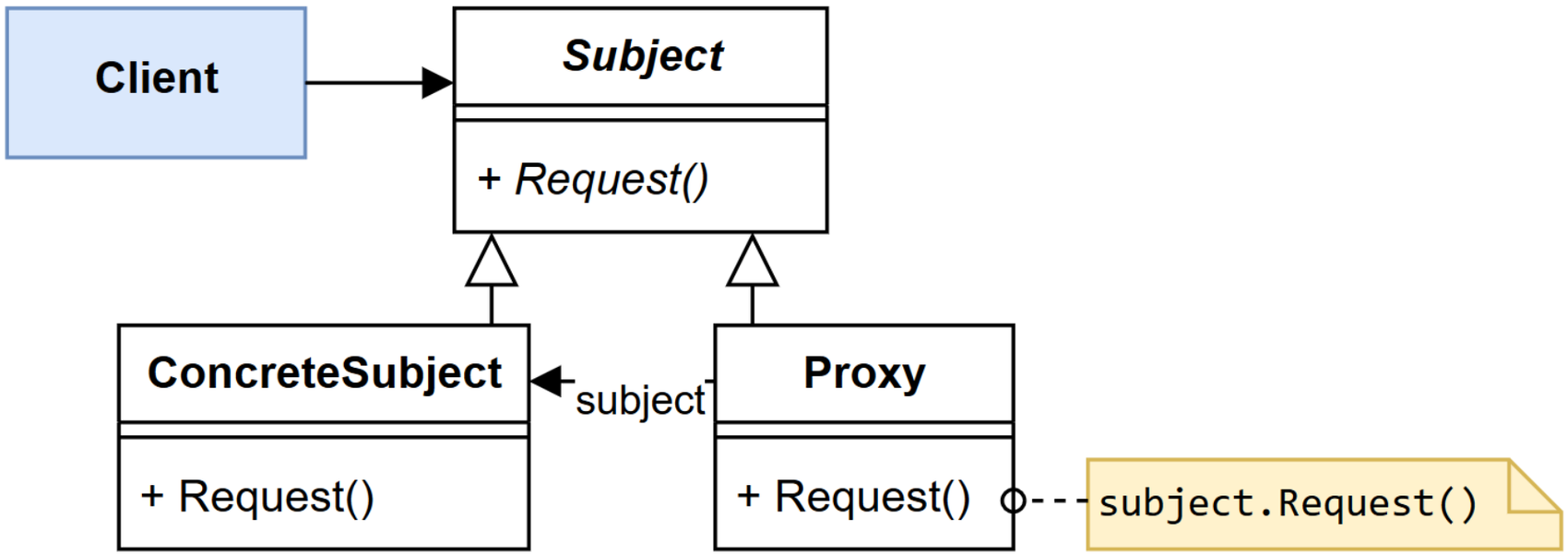


Decorator



Proxy

Provide a placeholder for another object to control it.



Proxy

Context: Need for versatile references to objects.

Problem: How to handle objects which are not directly accessible?

Forces:

- Objects could be in different address space (remote proxy).
- An expensive object needs to be created on demand (virtual proxy).
- The access to the original object must be supervised (access rights! – protection proxy).
- A smart reference is needed as a replacement for a bare pointer that performs additional actions when an object is accessed.

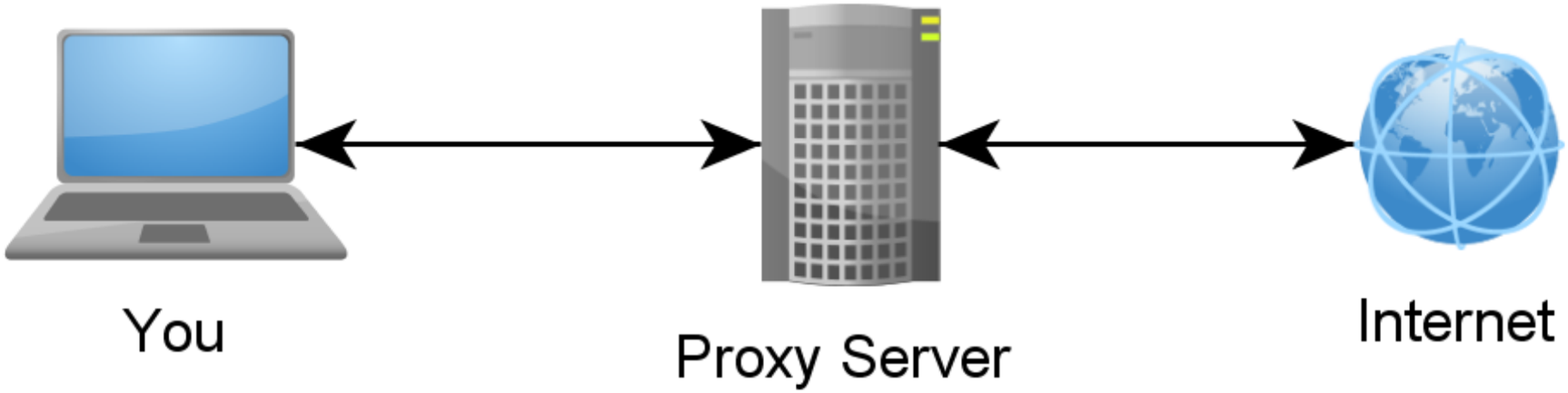
Solution:

- Maintain **a reference** that lets the proxy **access the real subject and provide interface identical** to Subject
- **Control access to the real subject** (may also include creating and deleting) and **act like the real subject**.

Consequences:

+ Introduces a level of indirection when accessing an object (separation of housekeeping and functionality)
+ Remote Proxy decouples client and server
+ Virtual Proxy can perform hidden optimizations
+ Caching Proxy could reuse subjects
+ Security Proxy can control access
- Overkill via sophisticated strategies
- Less efficiency due to indirection

Proxy



Summary

- SOLID Principles:
  Single Responsibility, Open-Closed, Liskov-Substitution, Interface Segregation, Dependency Inversion

- Principles of Good Programming:
  Decomposition, Abstraction, Decoupling, Simplicity & Usability

Patterns:

- **Iterator**: get next item until collection is exhausted.
- **Adapter**: wrap around object to implement another interface.
- **Facade**: Provide a higher-level interface to the customers.
- **Decorator**: wrap around an object to give it more functionality with same interface.
- **Proxy**: wrap around an object with same interface, as a transparent placeholder.