Design Patterns

448.058 (VO

TU Graz
SCIENCE
PASSION
TECHNOLOGY

Team

[Michael Krisper](#)
michael.krisper@tugraz.at
Uncertainty and Risk Propagation
Expert Judgment for Cyber-Security

[Georg Macher](#)
georg.macher@tugraz.at
Safety & Security
in Automotive & Autonomous Driving

ITI - Institute for Technical Informatics

When will you need Design Patterns?

- Every time you develop and design software!

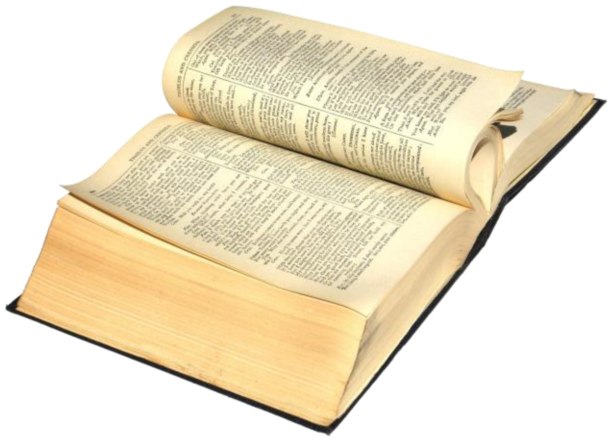
Examples:

- You are a Software Developer and need to implement specific tasks in your product.
- You are a Senior Software Architect in a company and have to manage complex software requirements and design flexible software architectures.
- You are a startup founder and want to write software for a product which is extensible, and flexible.
- You are a student and have to solve a software problem for an exercise at the university.

Learning Goals

Design Patterns Theory

- What is a design pattern? Why do we need them?
- What are the core principles behind design patterns?
- How to describe design patterns?
- What is a pattern language?



Design Patterns in Detail

- Know core ideas and application of important design patterns! (~50)

Application of Design Patterns

- When to use what?

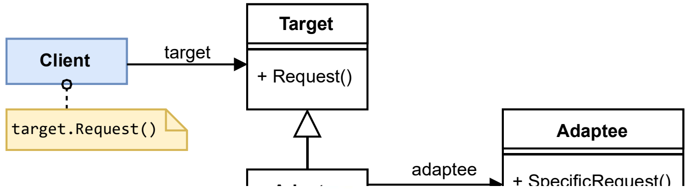
Learning Goals

- You know common design patterns and their core idea (approx. 50 patterns).
- You can apply them in software development regardless of the programming language or development environment.
- You can derive the consequences of design patterns and see the design decisions.
- You decide if the consequences of a pattern are acceptable or not.
- You avoid overengineering and misuse of patterns.
- You can make reasonable design decisions by balancing out the forces, consequences, and requirements for arbitrary problems and contexts.

Course Schedule

Date	from	to	Content
07.10.2020	13:00	16:00	Introduction, Organisation
14.10.2020	13:00	16:00	Theory, Principles, and Guidelines, Iterator
21.10.2020	13:00	16:00	Adapter, Facade, Decorator, Proxy
28.10.2020	13:00	16:00	Layers, Broker, Pipes & Filters, Master/Slave, Client/Server, Broker
04.11.2020	13:00	16:00	Factory Method, Abstract Factory, Builder, Singleton, Prototype, Memento, State, Flyweight
11.11.2020	13:00	16:00	Visitor, Strategy, Command, Composite, Template Method, Fluent Interface
18.11.2020	13:00	16:00	Mediator, Bridge, Blackboard, Microkernel, Messages (Message, Endpoint, Translator, Router), Observer
25.11.2020	13:00	16:00	Locks (Mutex, Semaphor, Condition Variable), Scoped Locking, Double Checked Locking, Monitor, Future/Asynchronous Completion Token, Active Object, Thread Specific Storage, Async-Await
02.12.2020	13:00	16:00	Lazy Acquisition, Eager Acquisition, Partial Acquisition, Caching, Pooling, Leasing, Garbage Collector, Scoped Resource, Active Record
09.12.2020	13:00	16:00	Chain of Responsibility, Counted Pointer, Interpreter/Abstract Syntax Tree, Proactor, Reactor
13.01.2021	12:00	15:00	Model-View-Controller, Model-View-Viewmodel, Model-View-Presenter, Presentation-Abstraction-Control
20.01.2021	13:00	16:00	Summary and Exam Preparation
27.01.2021	**13:00**	**15:00**	**Exam**

What is a Design Pattern?



What is a pattern?

“A proven solution for a (recurring) problem.”

A solution idea, scheme, or template.

Patterns are a universal principle:

- Economics (Etzioni, 1964)
- Social Interaction (Newell,Simon, 1972)
- Architecture (Alexander et. al., 1975)
- Software (General awareness from 1990’s on)

Purpose of Design Patterns

- Easier knowledge transfer
- Efficient problem solving by reusing existing ideas
  “Don’t reinvent the wheel”
- Establishes a common vocabulary, terminology, or language
- Increases usefulness of an idea by generalizing the solution

Types of Design Patterns

Architectural Patterns

- Fundamental structural patterns
- Stencils for whole architectures
- Examples: Layers, Pipes-And-Filters, Broker, Model-View-Controller, Microkernel, Async-Await

Design Patterns

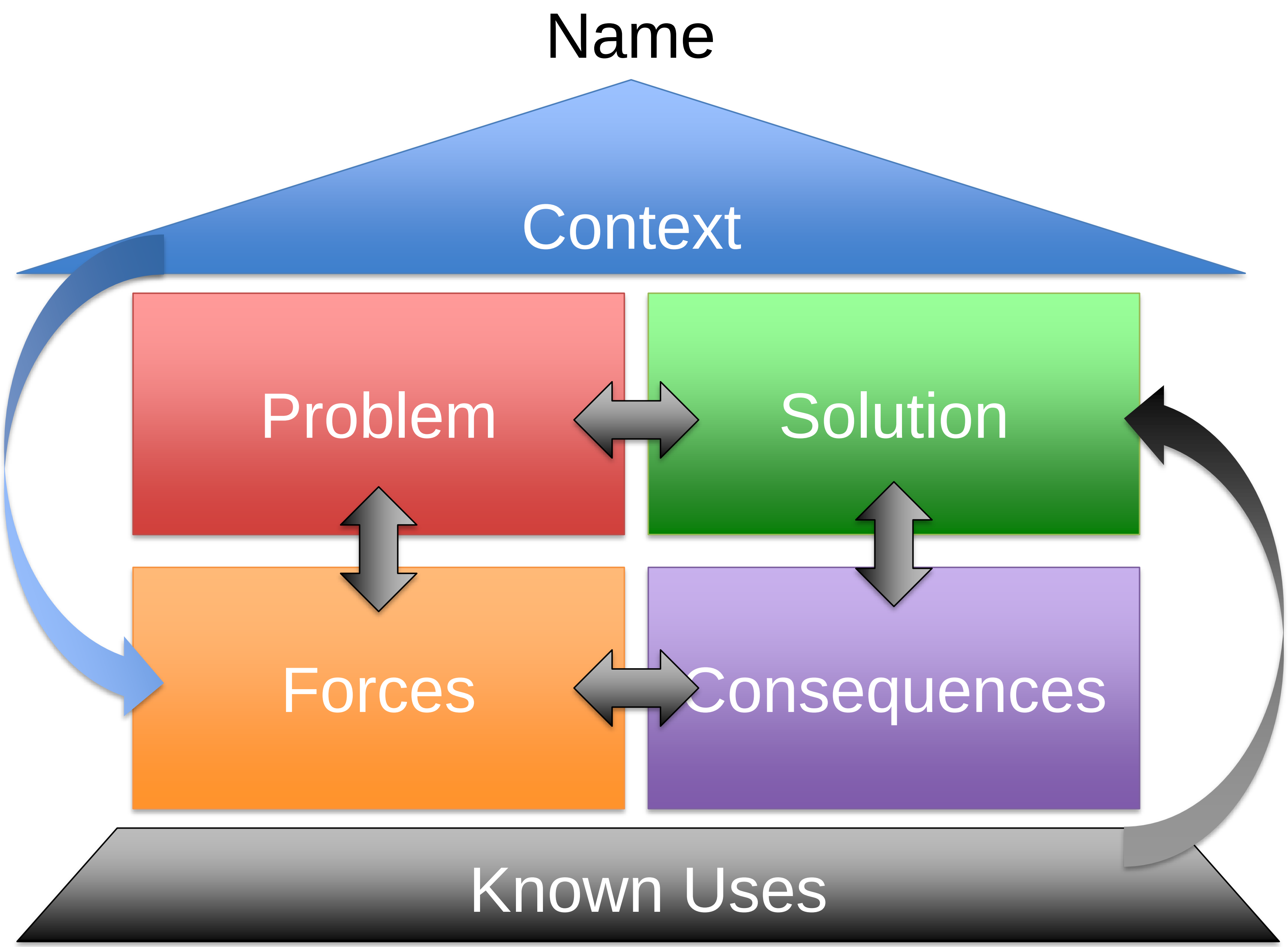
- Solution templates for more isolated problems
- Examples: Composite, Adapter, Proxy, Factory

Idioms

- Fine-Grained Patterns for problems in specific programming languages or environments
- Examples: Counted Pointer, Scoped Locking, Variadic Macros

Pattern format

- **Name**: A catchy name for the pattern
- **Context**: The situation where the problem occurs
- **Problem**: General Problem Description
- **Forces**: Requirements and Constraints - Why does the problem hurt in this context?
- **Solution**: Generic Description of a proven solution.
  Static Structures, Dynamic Behaviour, Actionable Steps
- **Consequences (Rationale, Resulting Context):**
  - What are the benefits and drawbacks? Pro and Contra?
  - What are the liabilities, limitations and tradeoffs?
  - How are the forces resolved?
- **Known-Uses**: Real Life Examples


Name
Context
Problem
Solution
Forces
Consequences
Known Uses

Alexandrian Pattern Format

Name

92 BUS STOP*

Picture

Context

. . . within a town whose public transportation is based on MINI-BUSES (20), genuinely able to serve people, almost door to door, for a low price, and very [illegible] to be bus stops within a few hundred feet of every house and workplace. This pattern gives the form of the bus stops.

✣ ✣ ✣

Problem and Forces

Bus stops must be easy to recognize, and pleasant, with enough activity around them to make people comfortable and safe.

Bus stops are often dreary because they are set down independently, with very little thought given to the experience of waiting there, to the relationship between the bus stop and its surroundings. They are places to stand idly, perhaps anxiously, waiting for the bus, always watching for the bus. It is a shabby experience; nothing that would encourage people to use public transportation.

The secret lies in the web of relationships that are present in the tiny system [illegible] If they knit together, and reinforce each other [illegible] the experience, the system is a good one: but the relationships that make up such a system are extremely subtle. For example, a system as simple as a traffic light, [illegible] corner can be enhanced by viewing it as a distinct node of public life: people wait for the light to change, their eyes wander, perhaps they are not in such a hurry. Place a newsstand and a flower wagon at the corner and the experience becomes more coherent.

The curb and the light, the paperstand and the flowers, the awning over the shop on the corner, the change in people's pockets—all this forms a web of mutually sustaining relationships.

The possibilities for each bus stop to become part of such a web are different—in some cases it will be right to make a system that will draw people into a private reverie—an old tree; another time one that will do the opposite—give shape to the social possibilities—a coffee stand, a canvas roof, a decent place to sit for people who are not waiting for the bus.

Solution and Consequences

Two bus stops.

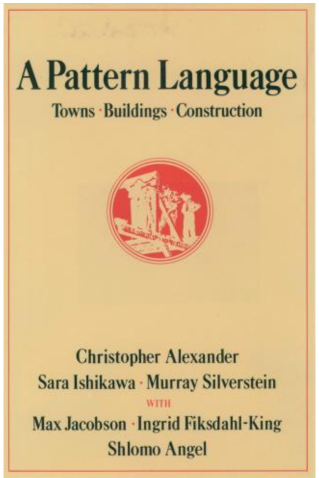
Therefore:

Build bus stops so that they form tiny centers of public life. Build them as part of the gateways into the neighborhoods, work communities, parts of town. Locate them so that they work together with several other activities, at least a [illegible] various combinations, corner groceries, smoke shops, coffee bar, tree places, special road crossings, public bathrooms, squares. . . .

✣ ✣ ✣

Related Patterns, Epilogue

Make a full gateway to the neighborhood next to the bus stop, or place the bus stop where the [illegible] gateway is already—MAIN GATEWAYS [illegible] patterns for PUBLIC OUTDOOR ROOM (69), PATH SHAPE (121), and A PLACE TO WAIT (150); provide a FOOD STAND (93): place the seats according to [illegible]—SEAT SPOTS (241). . . .



How Design Patterns emerge?

Design Patterns are found - not invented!

They emerge out of real use-cases/known-uses

1. Find patterns in real solutions
   ➔ **At least three Known-Uses**, Real Projects!
2. Write down the core idea and experiences
   ➔ Name, Context, Problem, Forces, Solution, Consequences, Known Uses
3. Discuss with others (often & repeatedly)
4. Improve Pattern (and repeat discussions)
5. Publish! (Conferences, Books, Blogs)
6. Continue to improve, apply and discuss pattern

Pattern Languages

… are coherent systems of patterns.

Consisting of:

- Patterns
- Relations
- Principles (Guidelines for design and evolution):
  - How to create / implement
  - Beneficial combination of patterns
  - How to change/evolve

Daily Life Examples: Cooking, Sports, Crafts, Sailing, Architecture, Programming

SOLID Principles (in OOP)

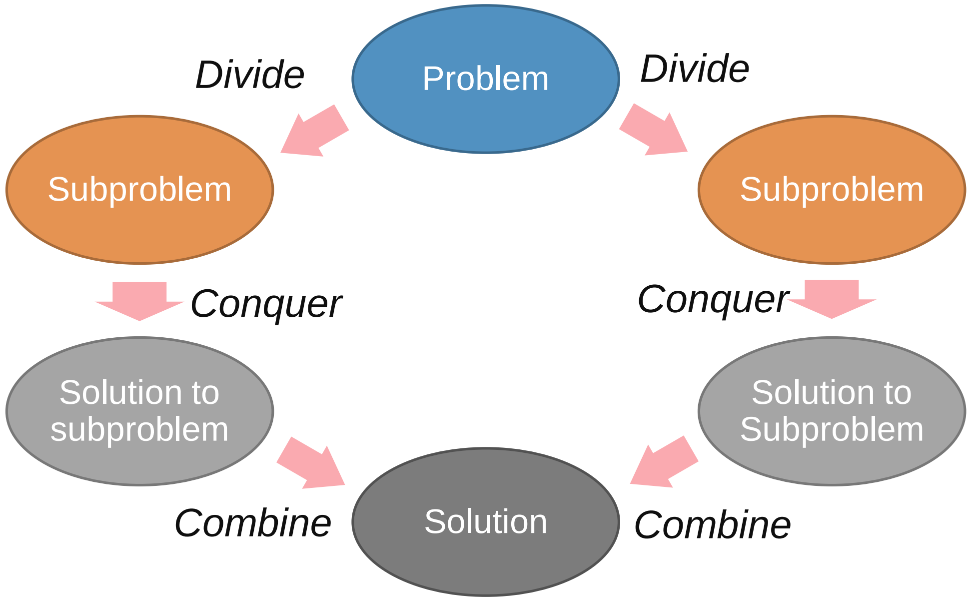
- ***S*ingle Responsibility**: A class should have one, and only one, reason to change.
- ***O*pen Closed**: You should be able to extend a class's behavior, without modifying it.
- ***L*iskov Substitution**: Derived classes must be substitutable for their base classes.
- ***I*nterface Segregation**: Make fine grained interfaces that are client specific.
- ***D*ependency Inversion**: Depend on abstractions, not on concrete implementations.

Principles of Good Programming

- **Decomposition**
  make a problem manageable
  decompose it into sub-problems
- **Abstraction**
  wrap around a problem
  abstract away the details
- **Decoupling**
  reduce dependencies, late binding
  shift binding time to “later”
- **Usability & Simplicity**
  make things easy to use right, hard to use wrong
  adhere to expectations, make usage intuitive

Decomposition

- Split up a problem until it gets manageable
- Divide and Conquer
- Separation of Concerns
- Orthogonality (Separation of Concepts)
- Single responsibility
- Curly’s Law (do just one thing and stick to that)



[Movie: City Slickers (1991)]

Abstraction

- Hide implementation details
- Wrap another layer around a problem.
- Liskov substitution
  Substitute Parent-Classes by Sub-Classes
- Fundamental theorem of software engineering:
  "We can solve any problem by introducing an extra level of indirection.“
  (David Wheeler)

Decoupling

- Minimise coupling / Maximize cohesion
- Separation of Concerns
- Shift Binding time to “later”
- Composition over inheritance
- Inversion of control
  - Hollywood principle:
    “*Don’t call us, we call you!*”

- Open close - encapsulate what changes
- Embrace change
- Law of Demeter: Only use *“direct”* dependencies

Usability & Simplicity

- YAGNI – You ain’t gonna need it!
- DRY – Don’t repeat yourself!
- Principle of least astonishment
  - Don’t make me think
  - Easy to use right, hard to use wrong
  - Code for the Maintainer
  - Command / Query Separation
  - Interface segregation
- Ockham’s razor
  - Do the simplest thing possible
  - KISS – Keep it simple, stupid!
- Avoid premature optimization (Knuth, 1974)

List of Design Patterns

Iterator - Problem & Forces

How can we access all elements of arbitrary collections in the same way?

→ There are many different types of data structures
Trees, Arrays, Lists, Sets, Queues, Dictionaries, Generators, …

→ We want to use a uniform way to access all of them
Get the next element! Are we finished yet?

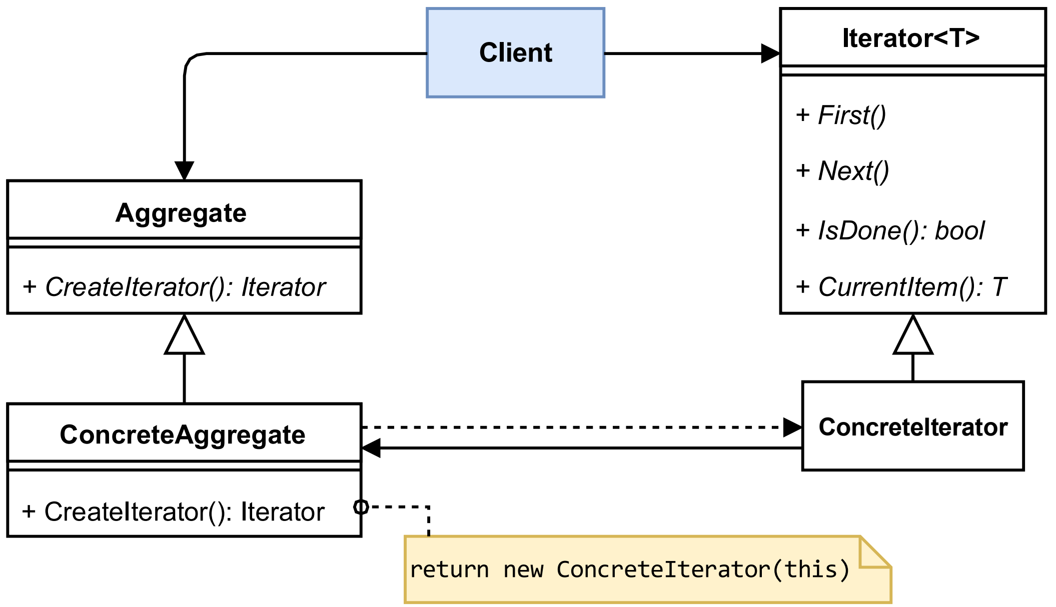
→ We want to define the order dynamically.
from start to end, in reverse order, depth-first, breadth-first, first-in-last-out

Iterator - Solution

Core Idea: Get the next element until a collection is exhausted.

1. Define an Iterator-interface for the following functionalities:
   - Get Next Item!
   - Are we done?

2. Implement a concrete iterator for each needed type of collection



Iterator - Consequences

+Every collection can be accessed in a uniform way.

+Multiple iterations are possible at the same time.

+Traversal algorithm can vary

- Lower efficiency

- Robustness is not guaranteed (insertions, deletions)

- “Hides” underlying data structure

Iterator - Known Uses

- Many programming languages use iterators for looping over collections (C++, C#, Java, Python, …)
- Foreach-Loop also uses Iterator (implicitly)
- Enumerators & Generators are also variants of iterators

```
vector<int> ar = { 1, 2, 3, 4, 5 };
vector<int>::iterator ptr;
for (ptr = ar.begin(); ptr < ar.end(); ptr++)
  cout << *ptr << std::endl;
```

C++ uses "end"-pointer to test the end of iteration

```
int myint[] = {1, 2, 3, 4, 5};
for (int i : myint)
   std::cout << i << std::endl;
```

foreach uses implicit iteration

Iterator - Known Uses

Python uses Exception instead of “IsDone()” or “End”:

Definition:

```
class Counter:
    def __init__(self, limit):
        self.current = 0
        self.limit = limit

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.limit:
            raise StopIteration
        else:
            self.current += 1
            return self.current-1
```

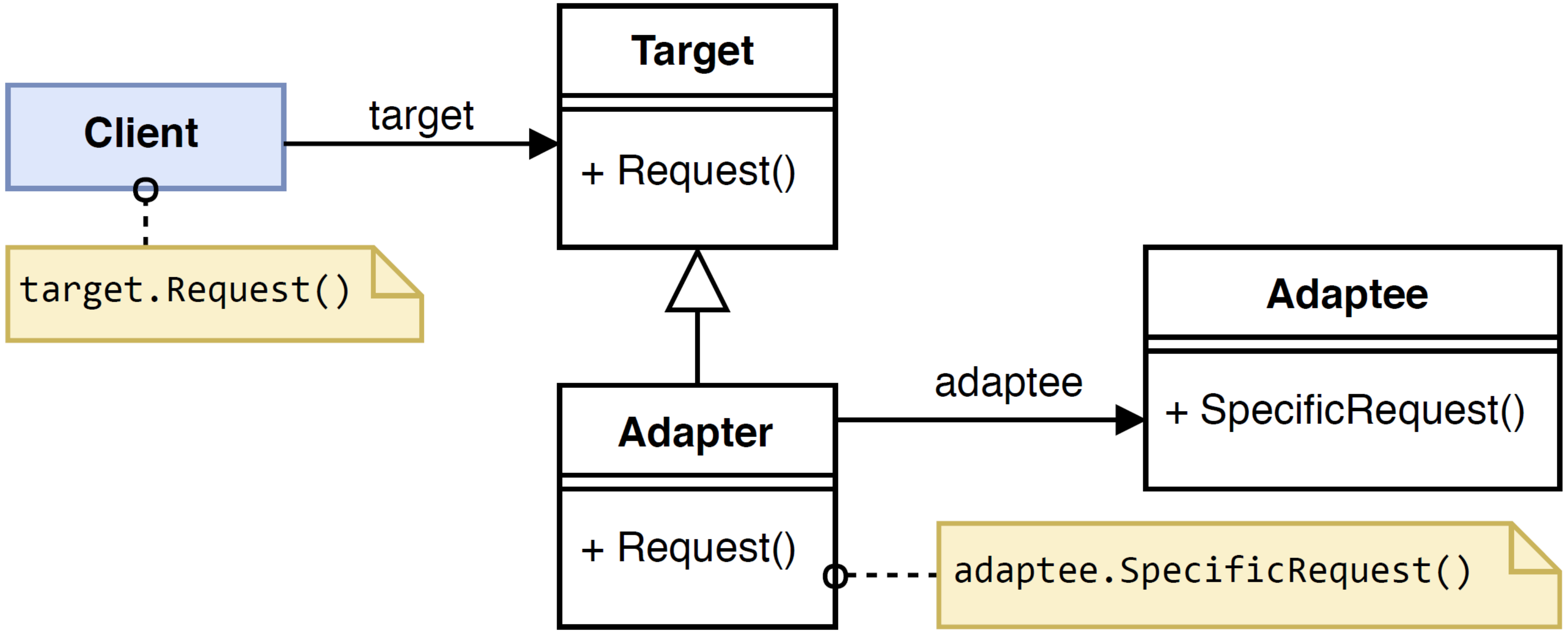
Usage:

```
for c in Counter(2):
    print c


it = Counter(2)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

Python uses StopIteration-Exception to signify the end of iteration

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.

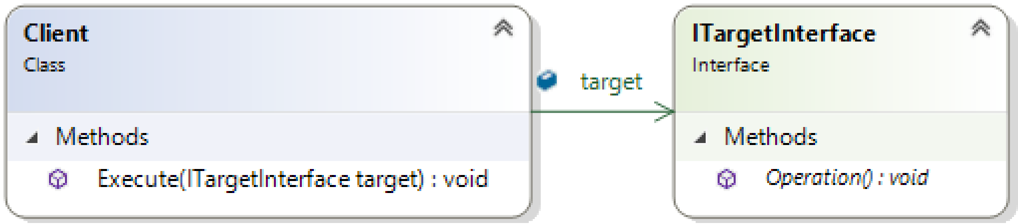
Consequences: (Class Adapter)

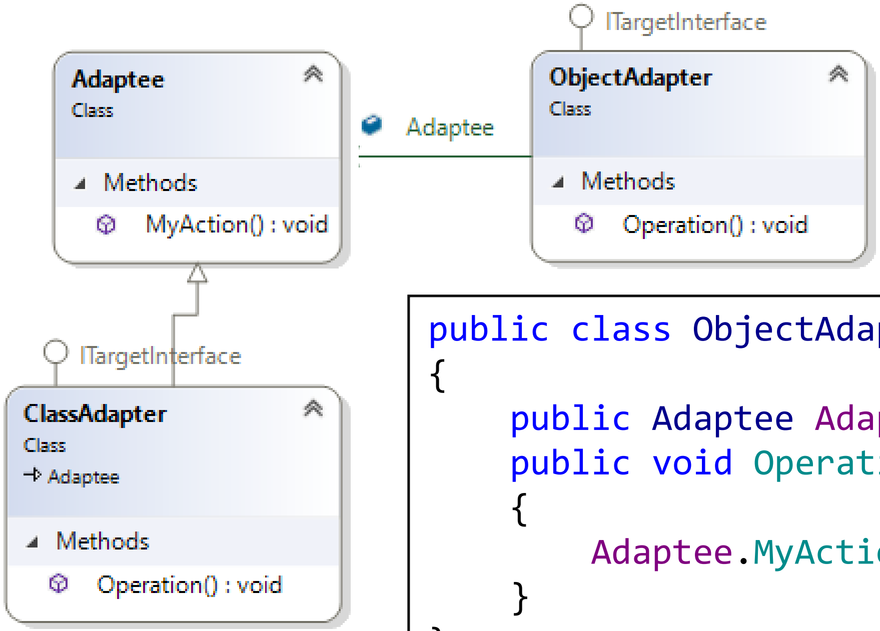
\+ Allows to use override mechanisms (e.g. protected methods, V-table, access to protected members).
\+ No additional indirection.
~ Inheritance approach (all methods of adaptee are inherited automatically, only changes have to be implemented)
\- Won't work when we want to adapt a class and all its subclasses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

Object Adapter vs. Class Adapter






```
class Client
{
    public static void Execute(ITargetInterface target)
    {
        target.Operation();
    }
}
```

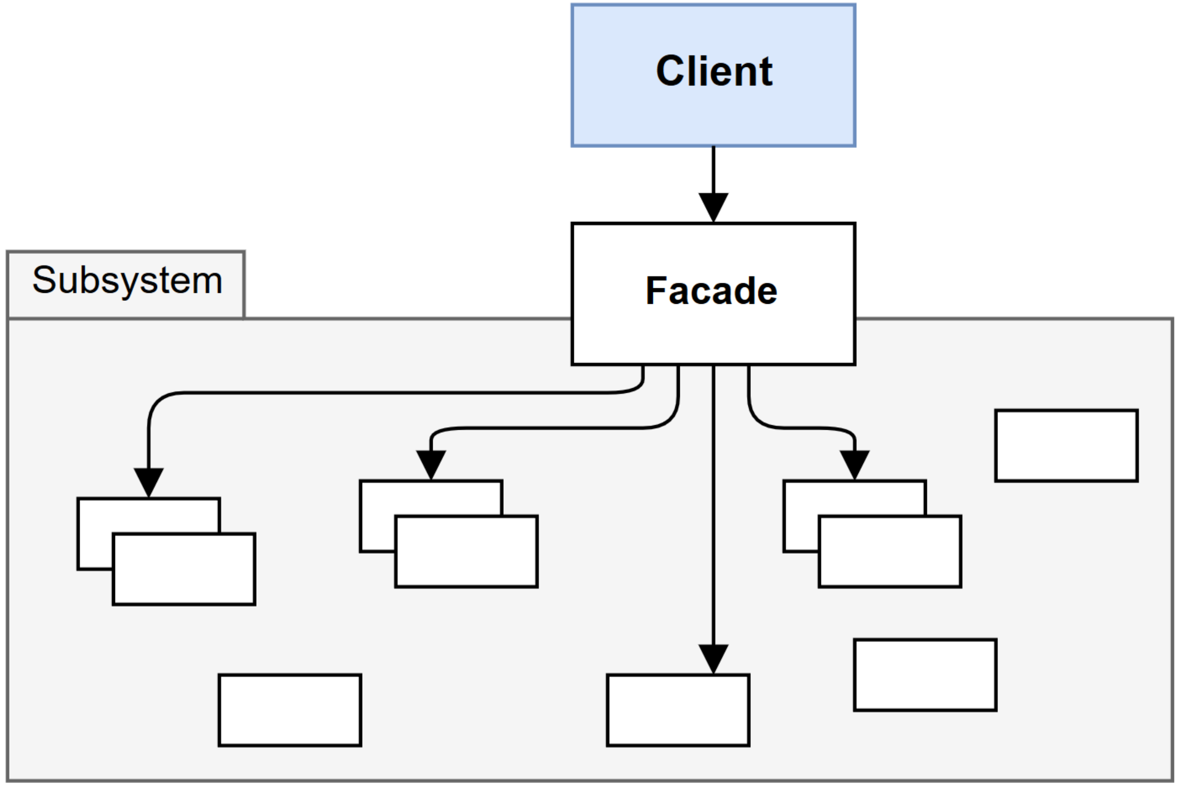
```
public class ObjectAdapter : ITargetInterface
{
    public Adaptee Adaptee;
    public void Operation()
    {
        Adaptee.MyAction();
    }
}
```

```
private static Adaptee adaptee = new Adaptee();

static void Main(string[] args)
{
    ITargetInterface target = new ObjectAdapter(adaptee);
    Client.Execute(target);

    ITargetInterface target2 = (ClassAdapter)adaptee;
    Client.Execute(target2);
}
```

```
public class ClassAdapter : Adaptee, ITargetInterface
{
    public void Operation()
    {
        MyAction();
    }
}
```

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 the larger system.
- Encapsulate non-OO API data & functions within concise, robust, portable, maintainable, cohesive OO class interface.

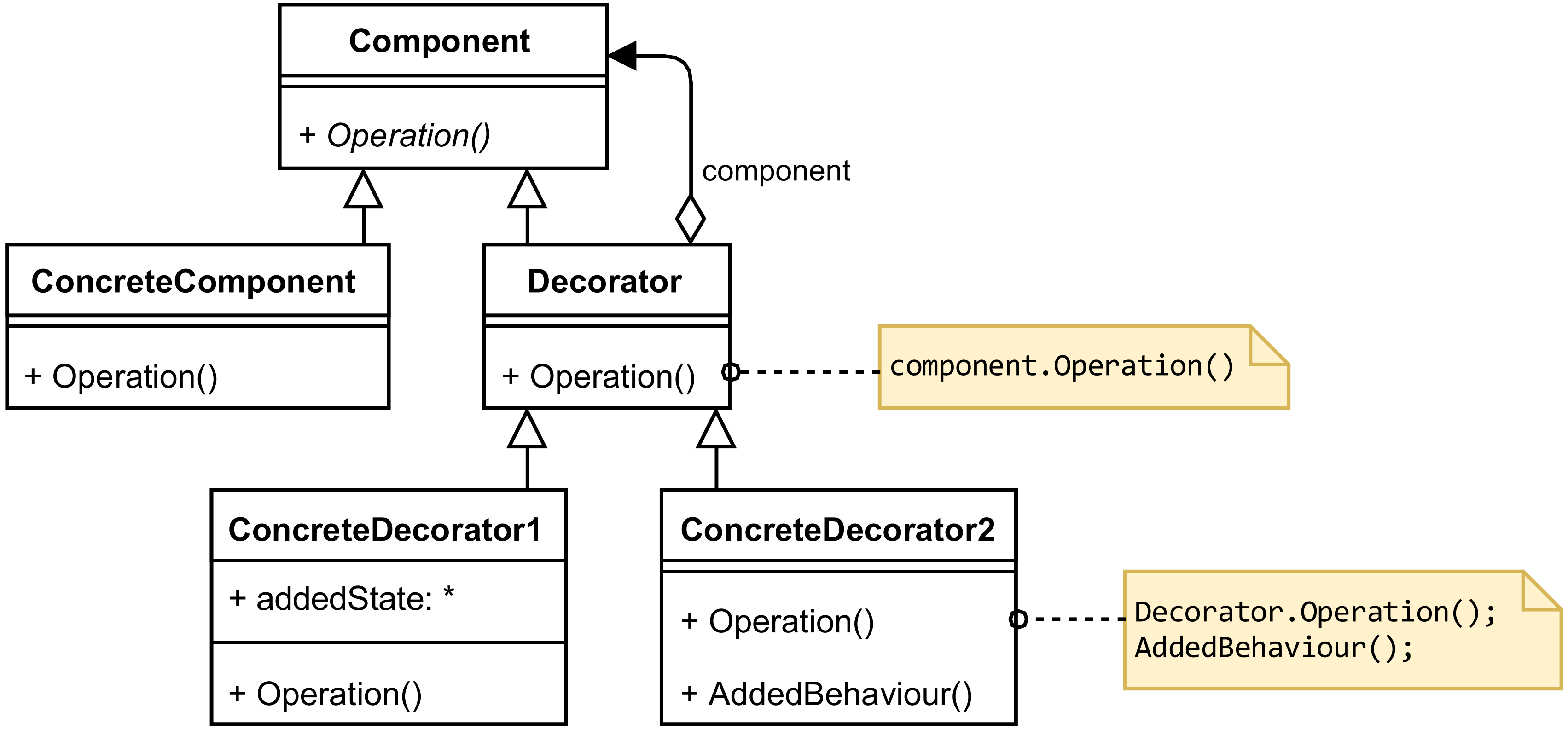
Consequences:

+ 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

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:

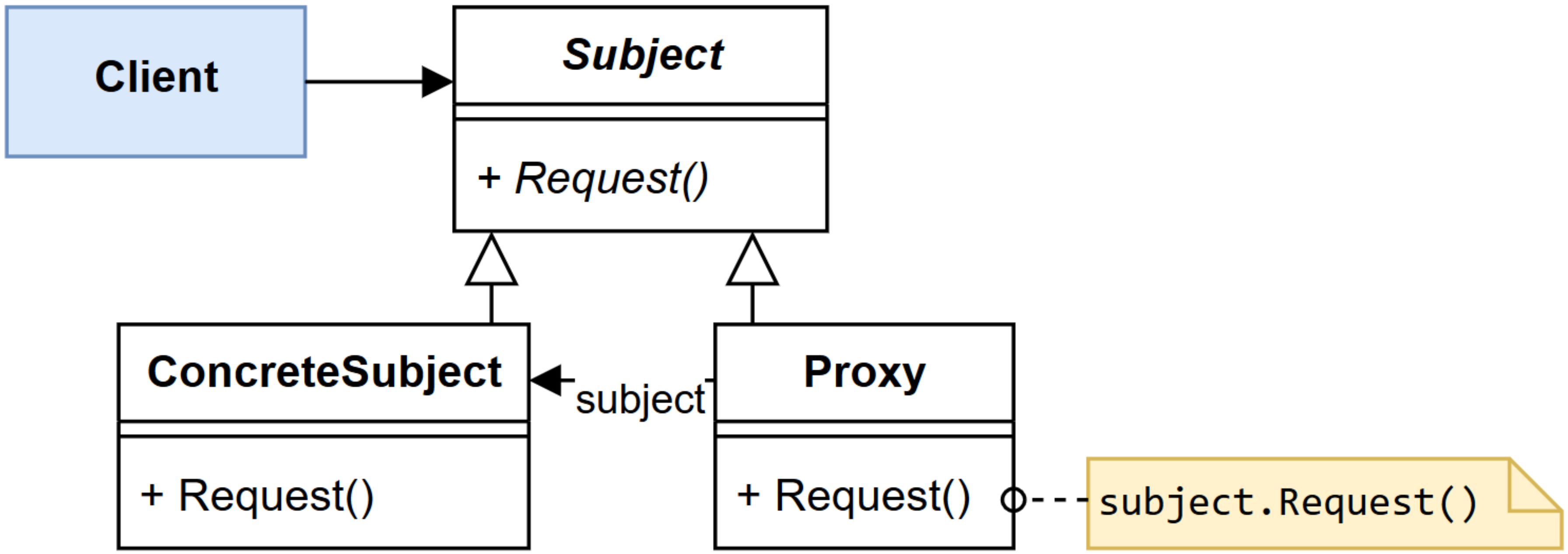
- Define a Decorator which forwards requests to its Component object.
- The decorator may optionally perform additional operations before and after forwarding the request.

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)

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:

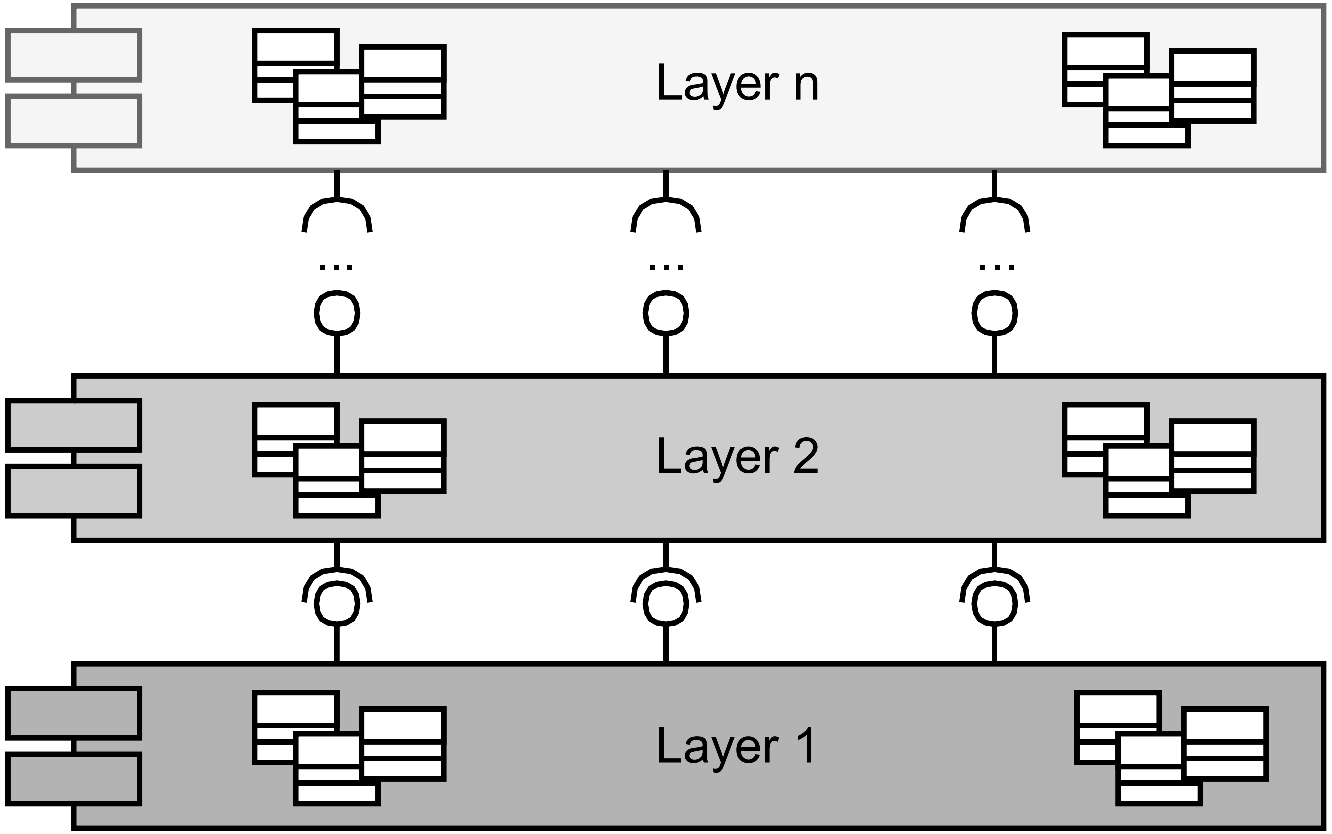
- Maintain **a reference** that lets the proxy **access the real subject and provide interface identical** to Subject
- **Control access to the real subject** (may also include creating and deleting) and **act like the real subject**.

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

Layers

Split your system into layers based on abstraction levels



Layers

Context: Large systems that require decomposition

Problem:
- Many functions and responsibilities
- Hard to understand structure, many dependencies

Forces:
- Changes should be limited to one component
- Clear boundaries of responsibility
- Interfaces should be stable
- Parts should be exchangeable
- Parts should be reusable
- Smaller groups for easier understandability, maintainability

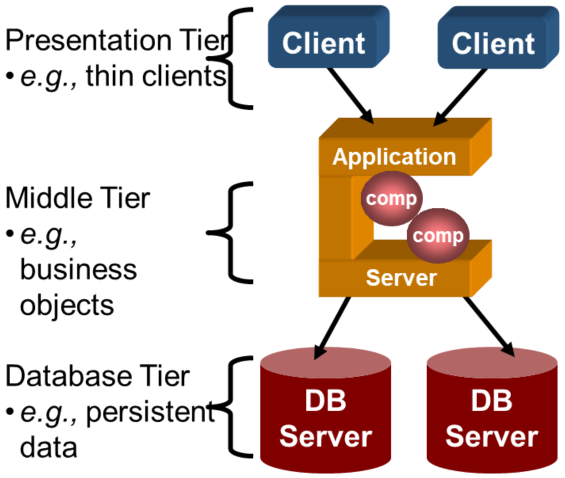
Solution:
- Structure the function into appropriate number of layers, based on their abstraction levels
- Every layer uses defined services of sublayer
- Every layer provides defined services to upper layer

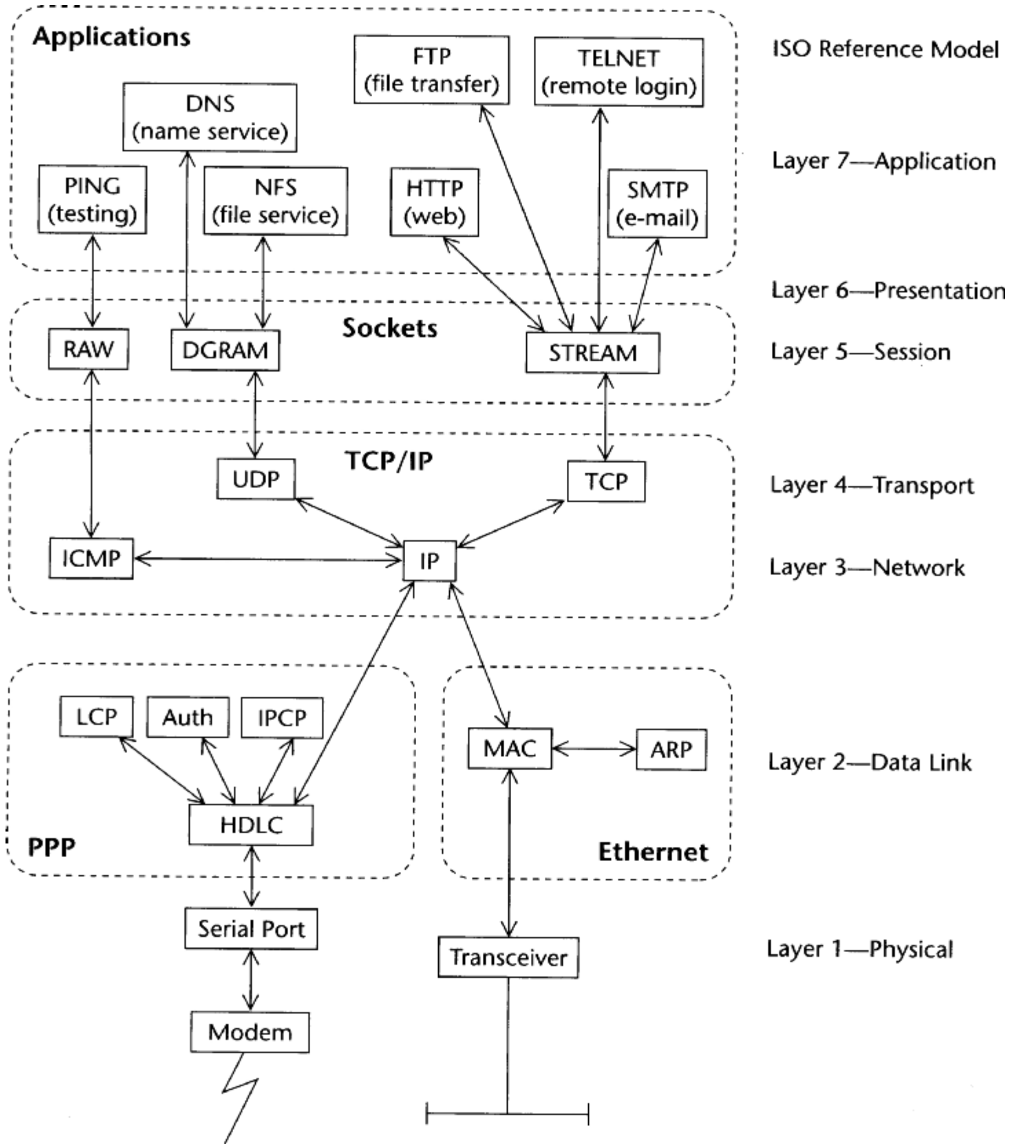
Consequences:

\+ Dependencies/Changes are kept local
\+ Defined Interfaces between Layers
\+ Layers are exchangeable & reusable
\- Lower efficiency
\- No fine grained control of sublayers
\- Changes cascade and are costly
\- Right granularity is difficult to find

Layers – Known Uses

- Network Stack
- Virtual Machines
- API's
- Operating Systems
- Companies
- Cities
- …



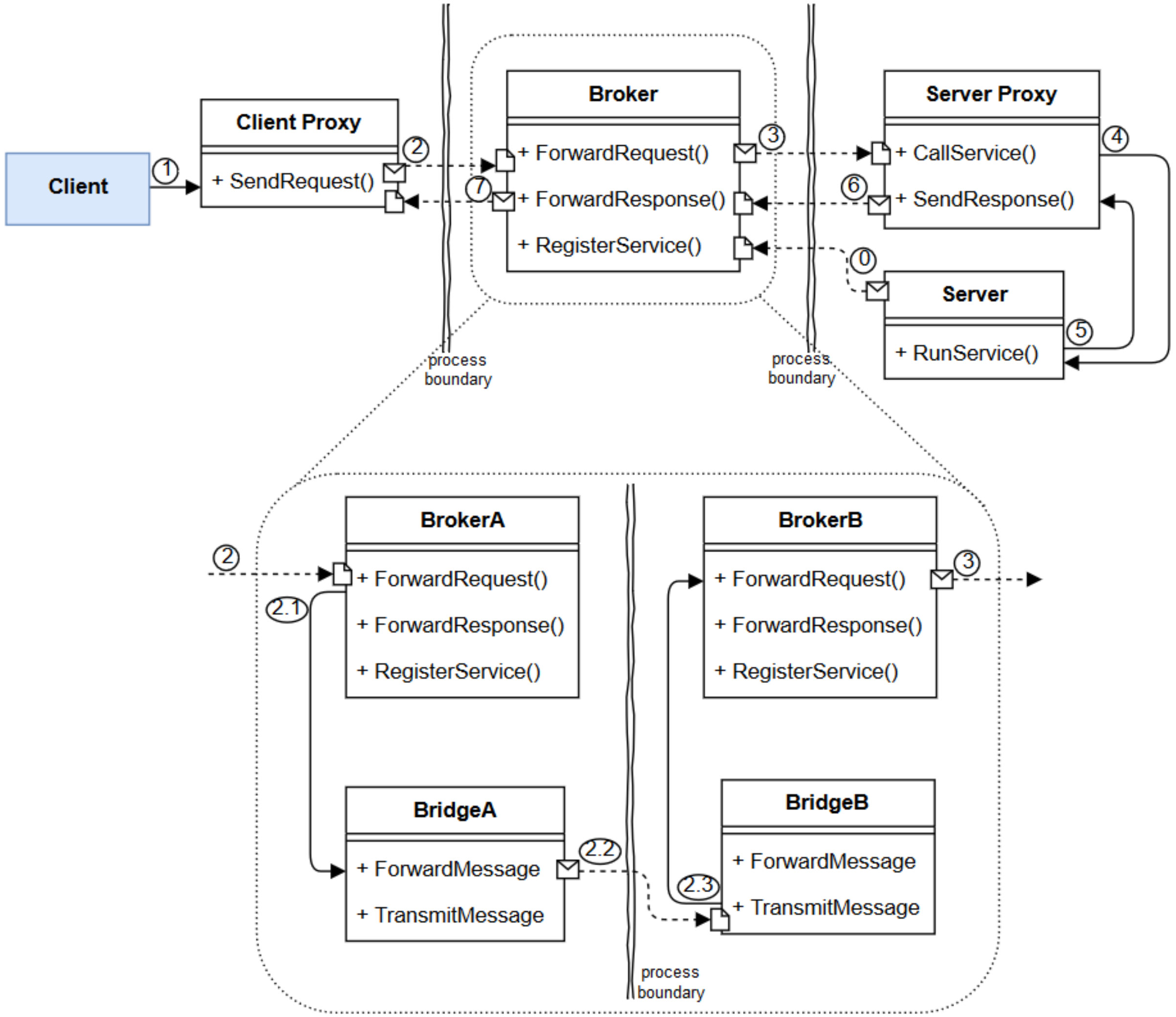


Layers – Implementation Issues

- Who composes the layers at runtime?
- How are Interfaces defined?
- Workarounds / Skip layers?
- Stateless / Stateful Implementations?
- Layers are Black Boxes

Broker

Manage dynamic communication between clients and servers in distributed systems.



Broker

Context: Working in distributed or heterogeneous systems with independent cooperating components.

Problem: You want to build complex systems as a set of decoupled and interoperating components

Forces:

- The addition, exchange, or removal of services shall be supported dynamically
- System details shall be omitted for developer
- The architecture shall support location transparency
- Remote method invocation shall be supported

Solution:

- **Specify broker API** (client side and server side)
- **Define an object model**, or use an existing model (use e.g. CORBA, OLE/COM/.NET, gRPC…)
- Use proxy objects to hide implementation details

Consequences:

\+ Broker is **responsible for locating a server** (location transparency)

\+ **Changeability** & extensibility of components (due proxies & bridges)

\+ Broker hides OS & network details (**portability**)

\+ **Interoperability between different broker**

\+ Reusability of components

\+ Server fault tolerance (servers can fail independently)

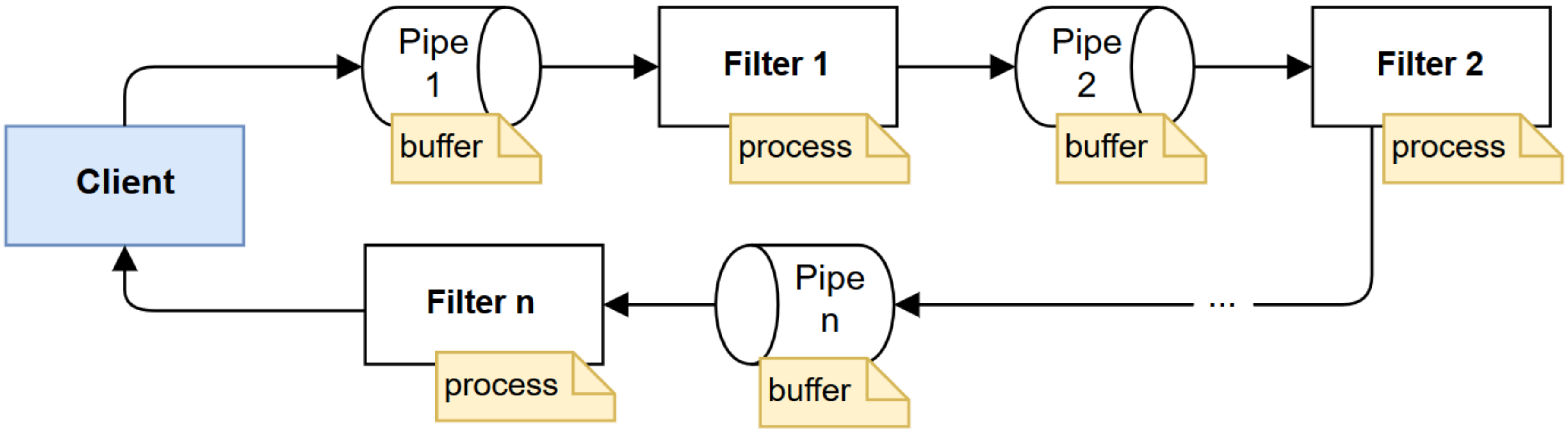
\- Restricted efficiency (communication overhead, communication through broker)

\- **Broker is single point of failure**

\- **Hard to test & debug** (many components involved)

Pipes & Filters

Form a sequence of processing steps using a common interface.



Pipes & Filters

Context: Processing of data streams.

Problem: How to can data streams be decomposed into several processing stages.

Forces:

- **Exchanging or reordering of processing steps** shall be possible (future system enhancements).
- Small processing steps are **easier to reuse** than larger.
- Probably different sources of input data exist (file, network, sensor,..)
- Results shall be storable in different ways.
- Explicit **storage of interim steps** shall be possible.
- **Multiprocessing** shall be enabled.

Solution:

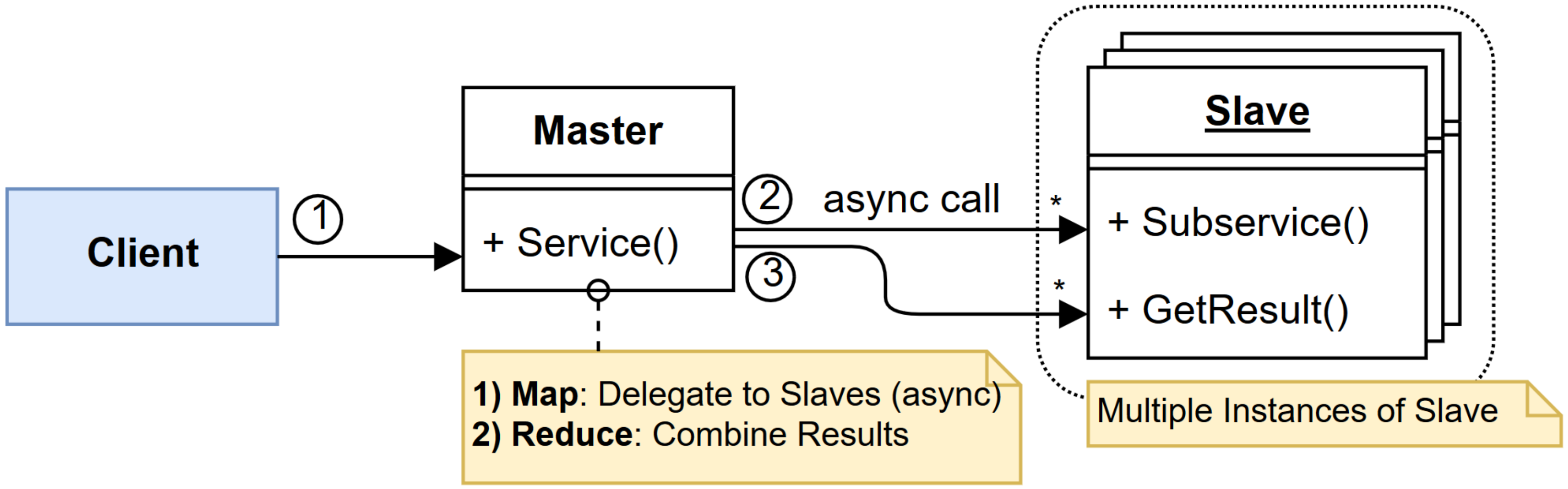
- **Divide** System task **into a sequence** of processing steps (dependent only on output of predecessor and connected by the dataflow)
- **Define a data format** to be passed along each pipe.
- Implement each pipe connection **either push or pull**
- Filter design and implementation
- Design **Error handling**
- Setup processing pipeline

Consequences:

\+ Intermediate files possible
\+ Flexible via filter exchange
\+ Flexible via recombination
\+ Efficient for parallel processing
\- Sharing state infos is expensive
\- Data transformation overhead
\- Error handling is crucial

Master-Slave

Distribute work amongst some helpers.



Master - Slave

Context: Partitioning of work into semantically-identical sub-tasks.

Problem: You want to solve instances of the same problem, **partition identical work** and separate concerns.

Forces:

- Processing of sub-tasks should not depend on algorithms for partitioning work and assembling the result
- Sub-tasks might **need coordination**
- Many **instances of the same problem** must be **solved**
- **Different algorithm implementation** may be required
- **Multi-threaded applications** may be wanted

Solution:

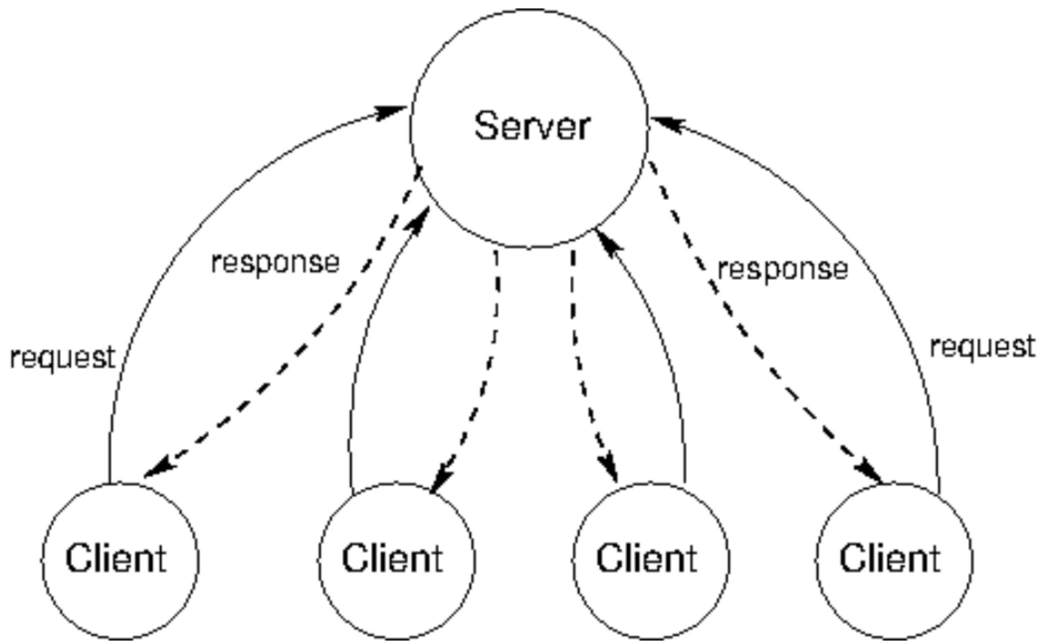
- Introduce a **coordination instance** between clients of the service and the processing of individual sub-tasks
- The master component divides work into equal sub-tasks, distributes these sub-tasks to Slave components & combines results (**maintaining slaves**)
- Provide all slaves with a common interface. The clients will **only communicate with the Master**

Consequences:

\+ **Exchangeability** and extensibility
\+ **Separation of concerns**
\+ Fault tolerance – several replicated implementations can detect and handle failures
\+ **Efficiency** (support of parallel computation)
\- Not always feasible
\- **Partitioning & control can be tricky**

Client-Server

Let clients send requests to servers which answers with responses.



Client-Server

Context: Distributed application.

Problem: You want to cooperate (share resources, content or service function) with multiple distributed clients.

Forces:

- Availability of services (resources, functions,..) is limited, but required by multiple requesters.
- Service might be provided by only one dedicated provider (centralized system).
- Client may not have the processing power.
- Number of possible requests might be unknown.

Solution:

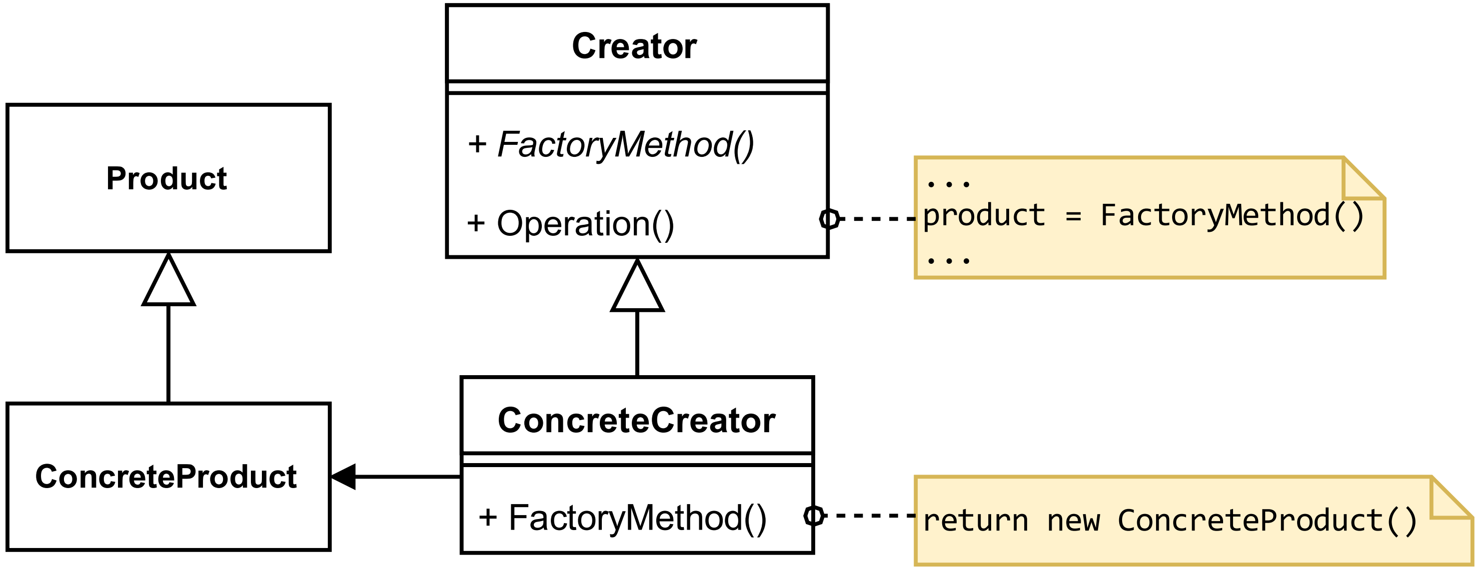
- Service-Interface: Define a protocol for serving a request/response communication.
- Server-Side Implementation: Implement a Listener which waits for requests from potentially multiple clients and individually answers with responses.
- Client-Side Implementation: Implement a Client who sends requests and waits for responses.

Consequences:

+ Encourages Service-Oriented Architectures
+ Centralization of specific services
+ Services get available for many clients
+ Doesn’t need to know exact number of clients
+ Workload gets moved to server. Clients are free to do something else
+ Exchangeability and extensibility
- Server could get overloaded
- Single-Point-Of-Failure, Denial-Of-Service Attacks are possible
- Communication overhead
- Client rely upon network and servers.

Factory Method

Delegate the creation of objects to someone else.



Factory Method

Context: Creation of an object, whose class is not known until runtime.

Problem: How to create an object for which the concrete class is not known.

Forces:

- We **don't care which object** is created, as long as it provides the **same functionality**.
- We **can't anticipate** the class we want to create at coding time.
- We want to **shift the decision** to someone else.

Solution:

- Define an interface of capabilities your objects must implement.
- Define some means (method or own class) to create the actual object somewhere else.
- Let the actual object implement the needed interface.

Consequences:

\+ Isolates Framework and Application code
\+ Flexibility (Compiletime/Runtime)
\+ Lesser Dependencies
\+ Connects parallel class hierarchies
\+ Decoupling of Implementation and Usage
\+ Abstraction of actual instances
~ Hides constructors
\- Needs an interface/abstraction layer!

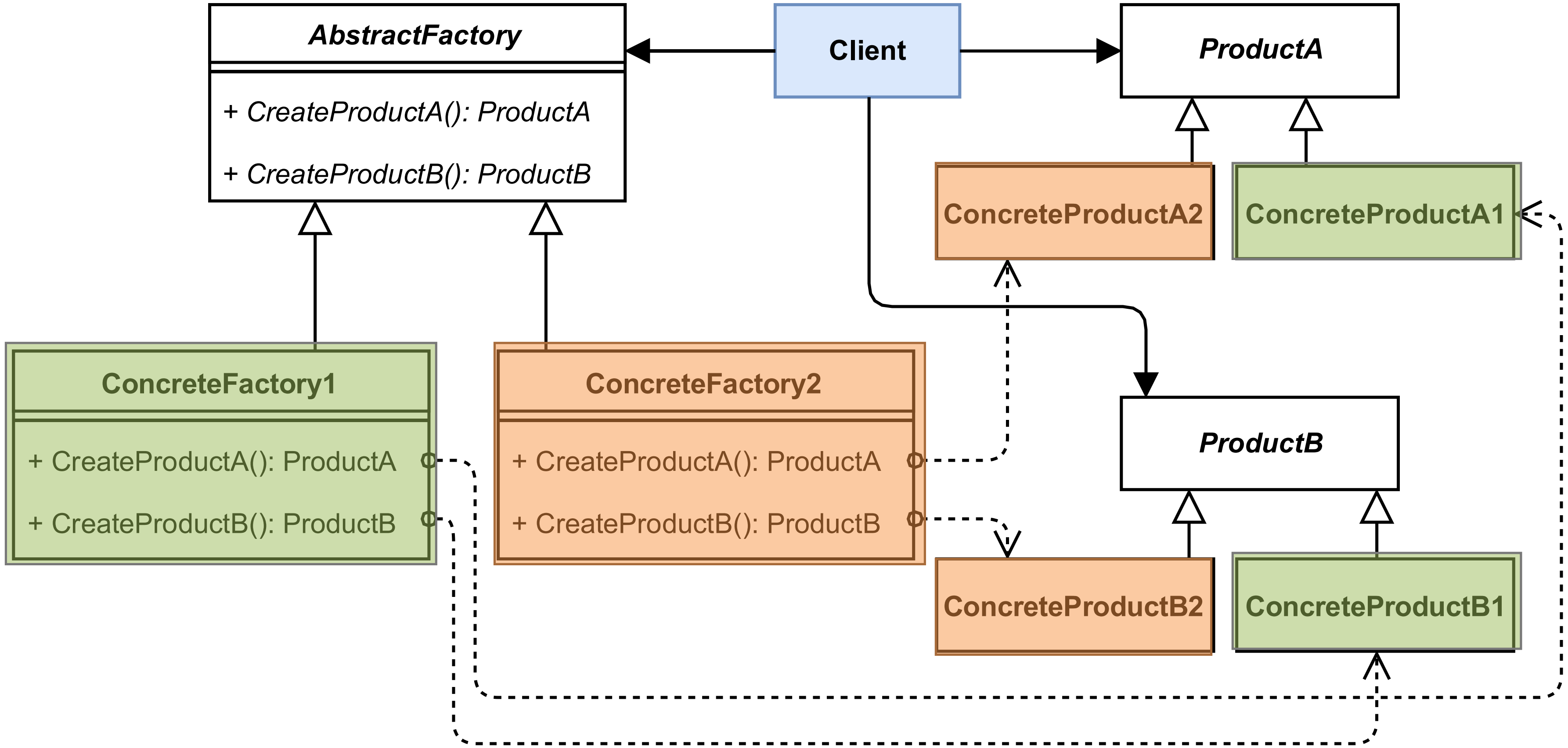
Factory Method – Implementation Issues

- Naming Convention (e.g. `XyzFactory`)
- Constructor Parameters?
- Universal-God-Interface vs. Duck Typing
- How to avoid direct constructions?
  (Private Constructor?)

- Abstract Creator (subclasses must implement)
  vs.
  Concrete Creator (default implementation, but subclasses can override)

Abstract Factory

Create whole families of related objects



Abstract Factory

Context:

Having multiple related families of similar objects

Problem:

How to create only matching objects?

Forces:

- Only create objects which fit together
- Choose object family at runtime
- Reveal just the interfaces, not the implementations

Solution:

- Define **Interface** for **Products**.
- Define **Interface** for **Factories**.
- Implement both accordingly.
- **Select the needed factory** at runtime to create the needed products.

Consequences:

+ Makes exchanging product families easy
+ Promotes consistency among products
+ Isolates concrete classes

~ When is the product family selected? Who selects?

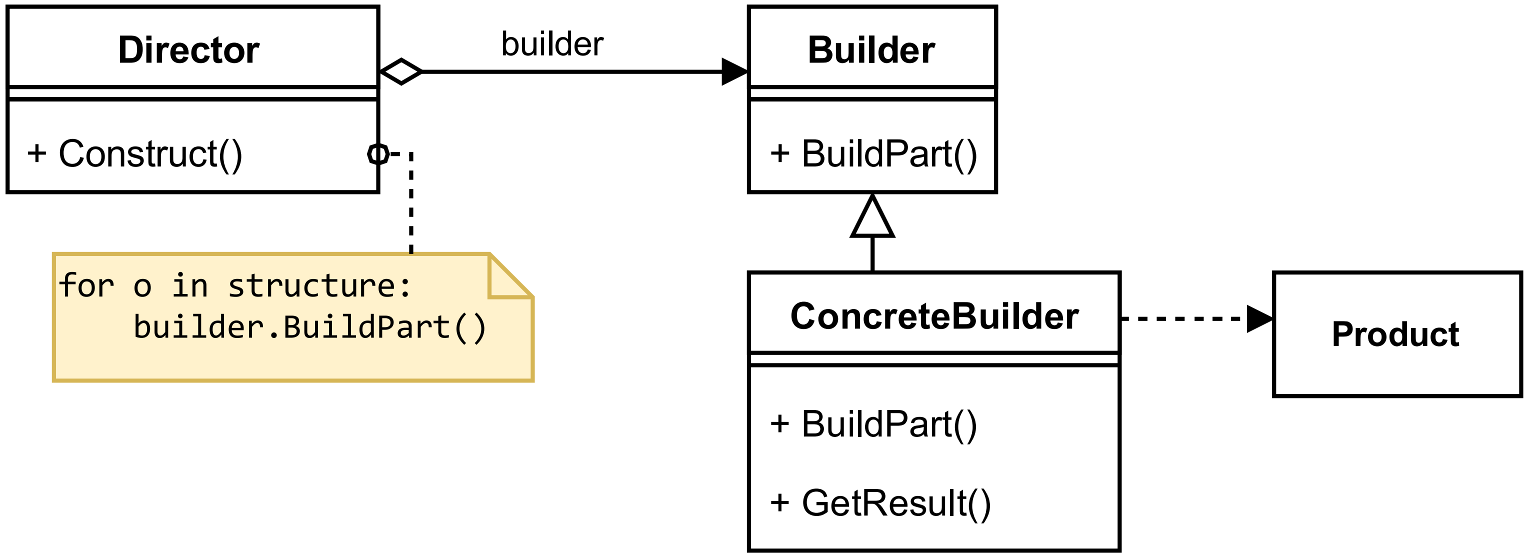
~ Factories as singletons?

~ Use prototypes as templates?

\- Supporting new kinds of products is difficult

Builder

Split up creation into multiple steps



Builder

Context:

Creation of complex objects

Problem:

How to create complex objects in an easy and comfortable way?

Forces:

- Manage many different construction options
- Creation of objects should be independent of assembling

Solution:

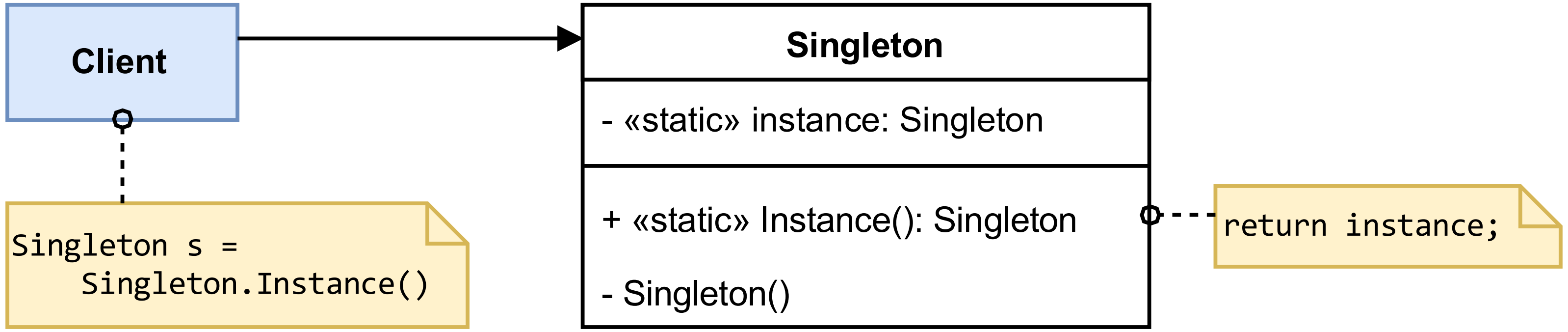
- Split creation from assembling
- Define Interface for creating individual parts & assembling
- Implement methods for parts

Consequences:

+ Allows many combinations of parts
+ Isolates code for construction and representation
+ Allows finer control of construction
- Construction is not a simple “new” anymore
- How to ensure that parts are correctly configured?

Singleton

Allow only one instance of an object



Singleton

Context:

Creation of exactly one instance

Problem:

Ensure a class only has one instance, provide a global point of access

Forces:

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
- When the sole instance should be extensible by subclassing, clients should be able to use and extended instance without modifying their code

Solution:

•Hide the constructor of a class (protected or private)

•Add a static Factory Method to create exactly one instance stored as static member

•Consequent creations only return the already created instance.

•Prohibit deep copying of the object

Consequences:

- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations and representation (subclassing)
- Permits a variable number of instances
- More flexible than static class operations

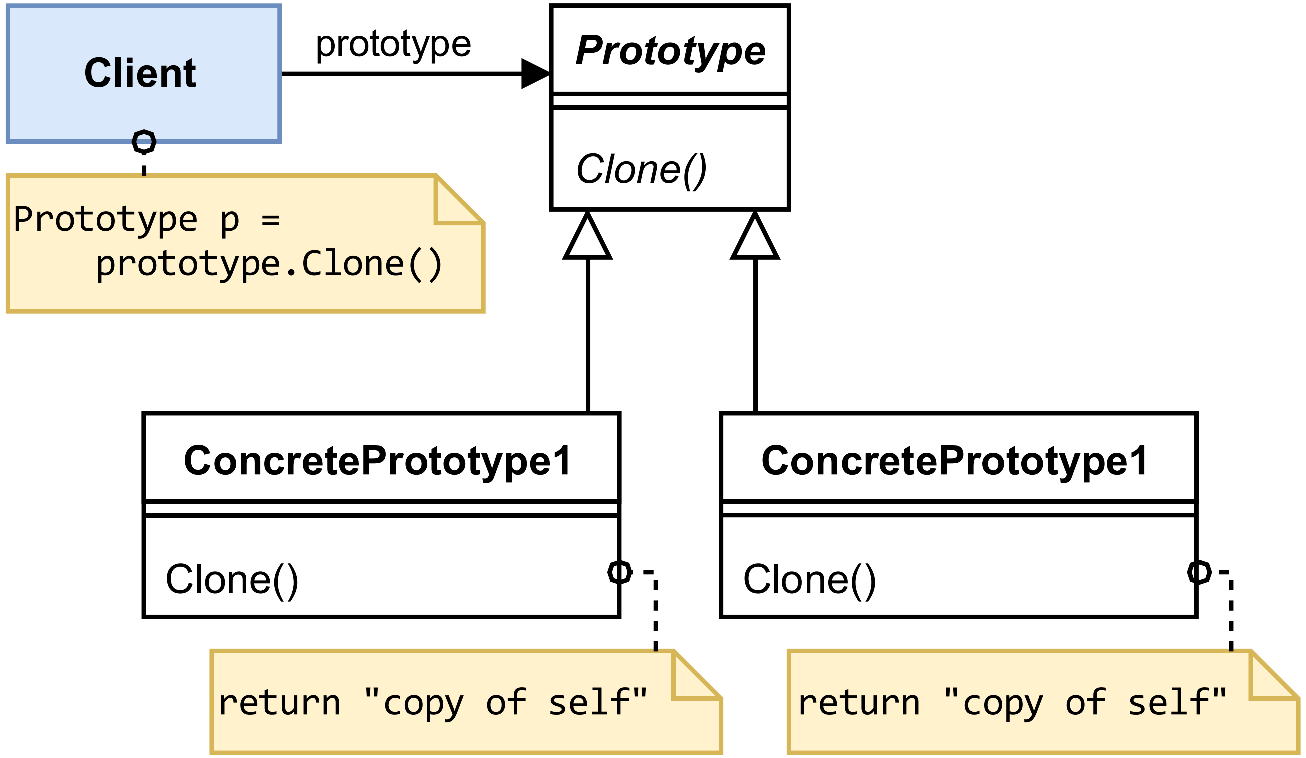
Singleton Example

```
class Singleton
{
    private static readonly Singleton _instance = new Singleton();

    protected Singleton() { }

    public static Singleton Instance()
    {
        return _instance;
    }
}
```

```
void Main()
{
    var s1 = Singleton.Instance();
    var s2 = Singleton.Instance();
    Console.WriteLine($"Singletons are equal: {s1.Equals(s2)}");
}
```

Prototype

Create objects by cloning from templates



Prototype

Context:

Creation of objects whose classes and properties are not known until run-time

Problem:

How to dynamically implement and use objects without knowing its properties?

Forces:

- Object Members are defined at runtime
- Avoid building complex class hierarchies and factories
- Avoid long taking instantiations

Solution:

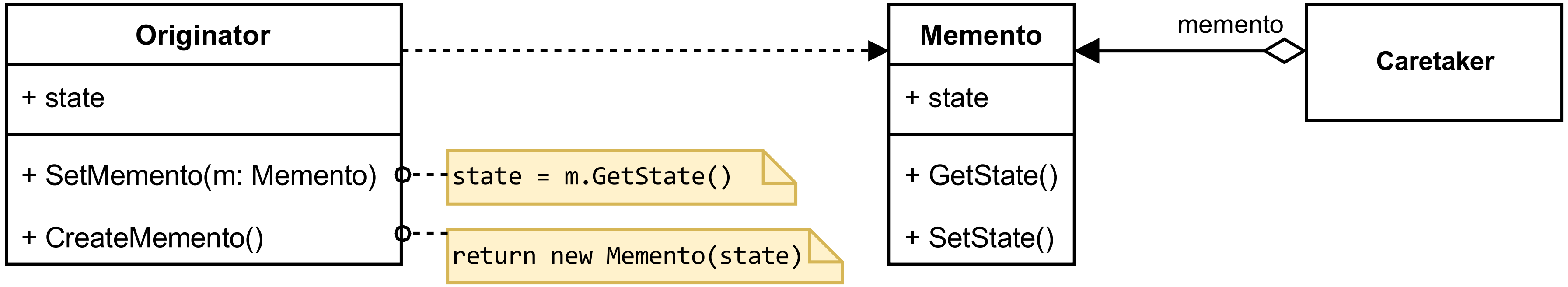
- Declare cloning interface
- Implement cloning interface
- (Add mechanism for dynamically setting/getting members and calling methods → Dictionary!)

Consequences:

+ Dynamic objects can be created at runtime
+ Class system is bypassed
+ No complex inheritance hierarchy
+ Long taking initialisation are done only once
~ Usage of prototype manager? (registry)
~ Shallow vs deep copy?
~ How to access members?
- No type safety!
- No compile-time errors!

Memento

Store & Load the internal state of an object



Problem
How can an object be persisted?

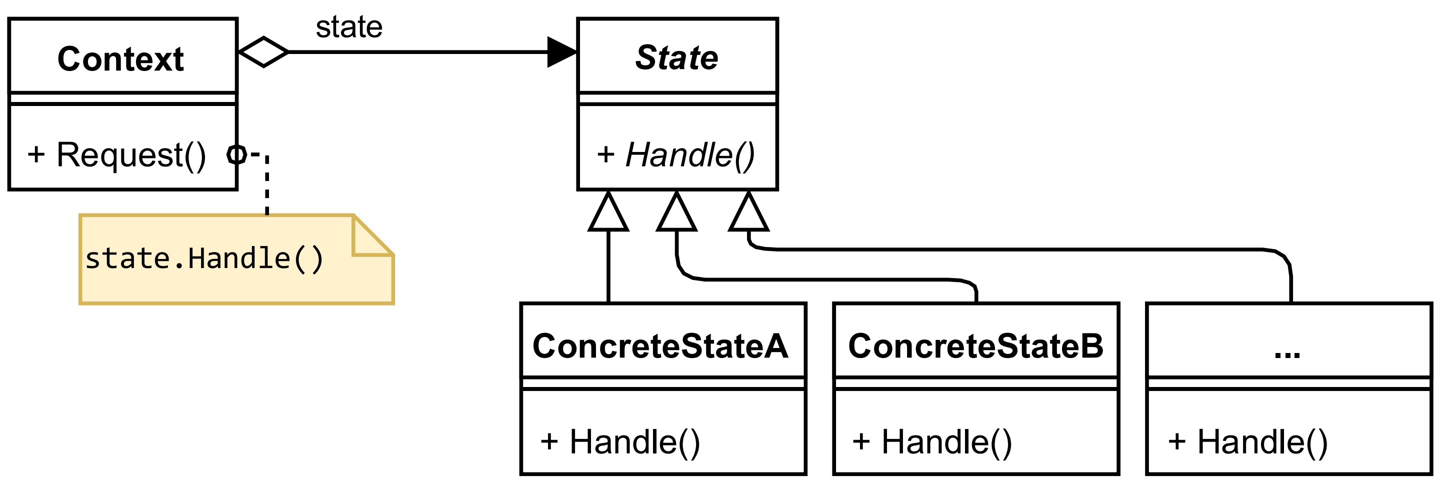
Forces
- State of object should be storable/restorable.
- Do not break encapsulation

Solution:
- Create a Memento-Class: Data class for storing the state.
- Implement method for returning a Memento.
- Implement method for reading a Memento.

Consequences:
+ State can be persisted without exposing all internal members.
+ Persisted state can be used to restore the object.
+ Snapshots are possible.
+ Combines very well with Command Pattern
- If data format is known, data could be manipulated "offline". (make sure to add some checksum or digitally sign the memento)

State

Change object behaviour depending on a situation



State

Context: Objects which change their behaviour according to a situation

Problem: How to switch behaviour of an object without complex implementation?

Forces:

- Behaviour should change with internal state
- Behaviour should change at runtime
- Transition between states should not depend on complex multipart conditional statements (no if-else-if-else-…)
- States should not be mixed.

Solution:

•Define Context(manager) which knows the states and transitions and exposes the client-interface

•Define general Interface for all States

•Implement the different states in individual classes

•Define the transitions between states

Consequences:

\+ State specific behaviour is encapsulated within the state objects.

\+ New States and transitions can easily be defined

\+ Transition logic is partitioned and simple.

\+ Transition are explicit – no mixed states

\+ State Object can be shared (-> Flyweight)

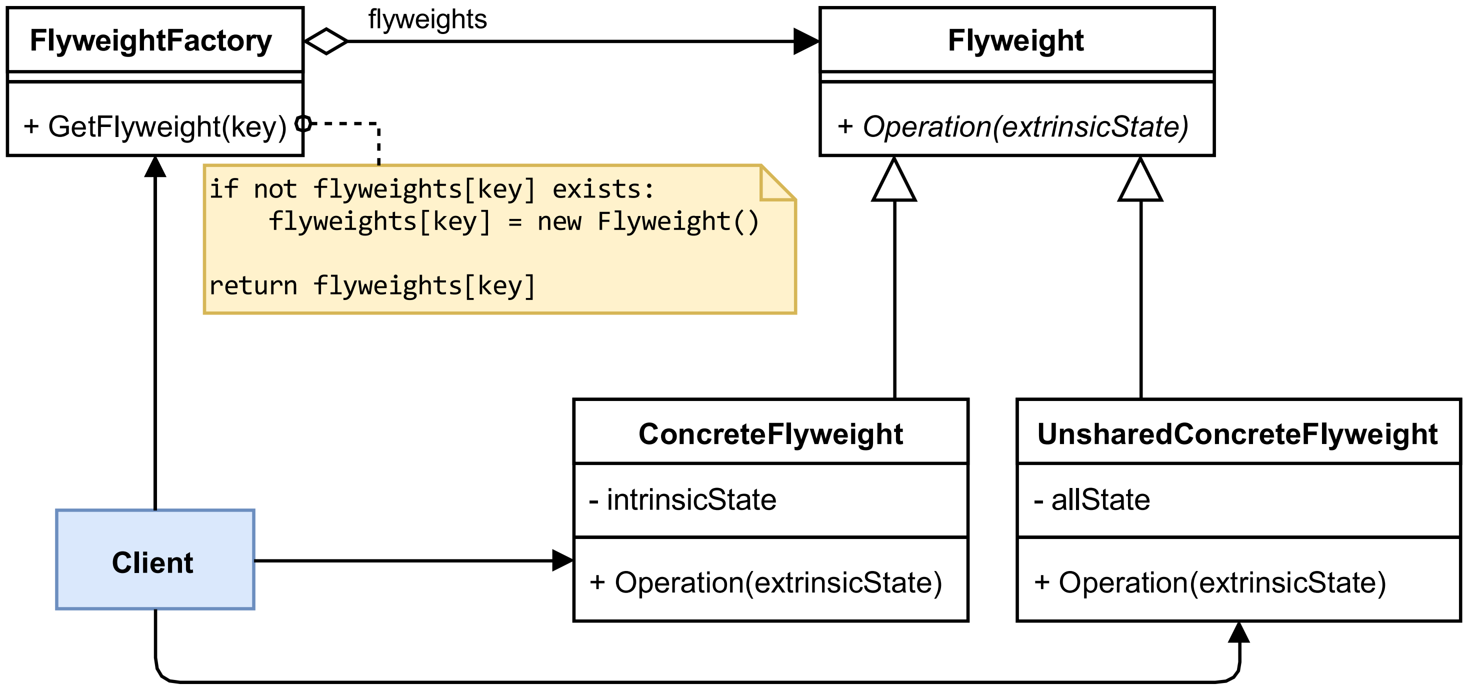
~ Who makes the transitions?

\- More classes

\- Special transitions may be difficult

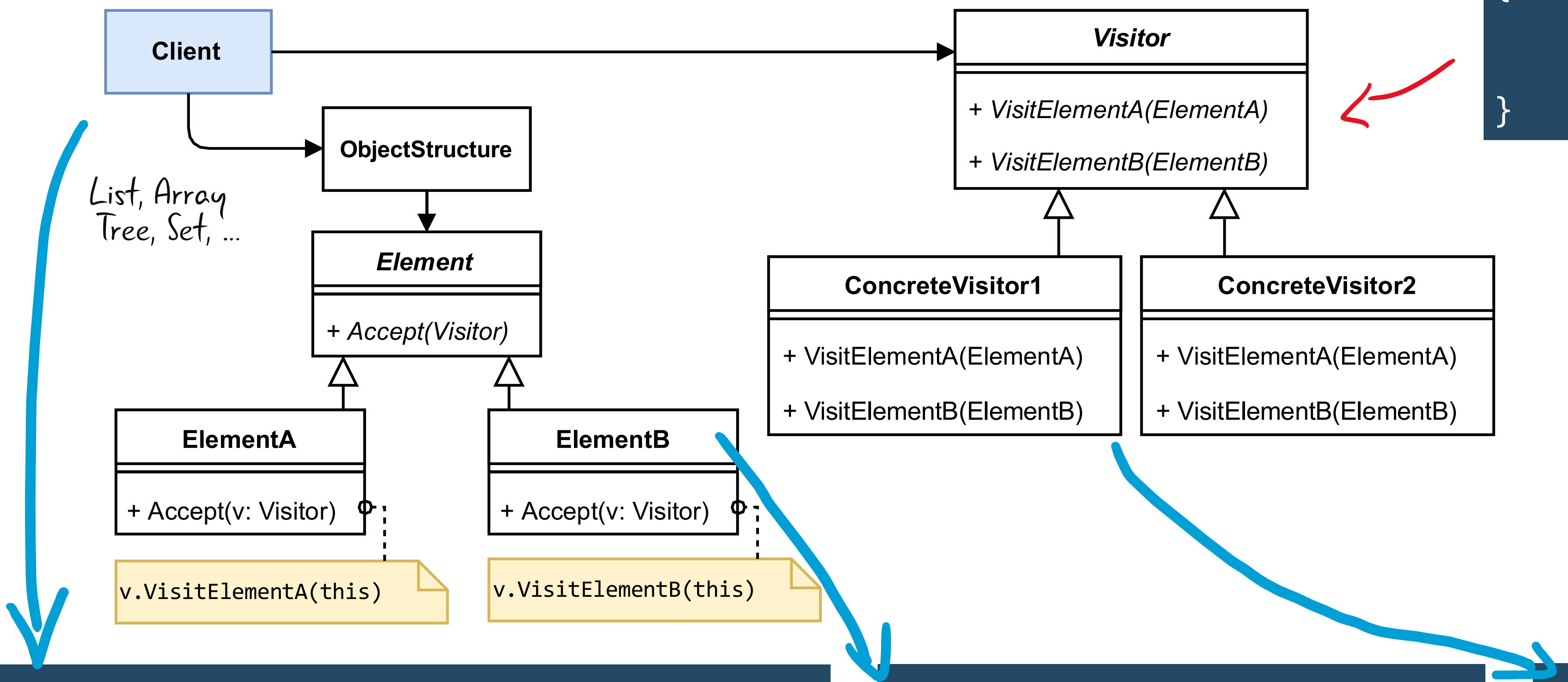
Flyweight

Share global state and vary differences only when needed.



Visitor

Add behaviour on aggregates of different objects




```
interface IVisitor
{
    void Visit(ElementA e);
    void Visit(ElementB e);
}
```

```
void main() {
    IVisitor visitor = new HeightCalculator();
    foreach(e in list)
        e.Accept(visitor);
}
```

```
class Circle : IElement {
    void Accept(IVisitor v) {
        v.Visit(this);
    }
}
```

```
class HeightCalculator: IVisitor {
    void Visit(Circle e){ … }
    void Visit(Triangle e){ … }
    void Visit(Square e){ … }
}
```

Visitor

Context: Performing operations on elements of an aggregate.

Problem: How to execute some behaviour on an aggregate of different objects?

Forces:

- Object aggregate contains different interfaces
- Avoid polluting classes with unrelated operations
- Structure rarely changes

Solution:

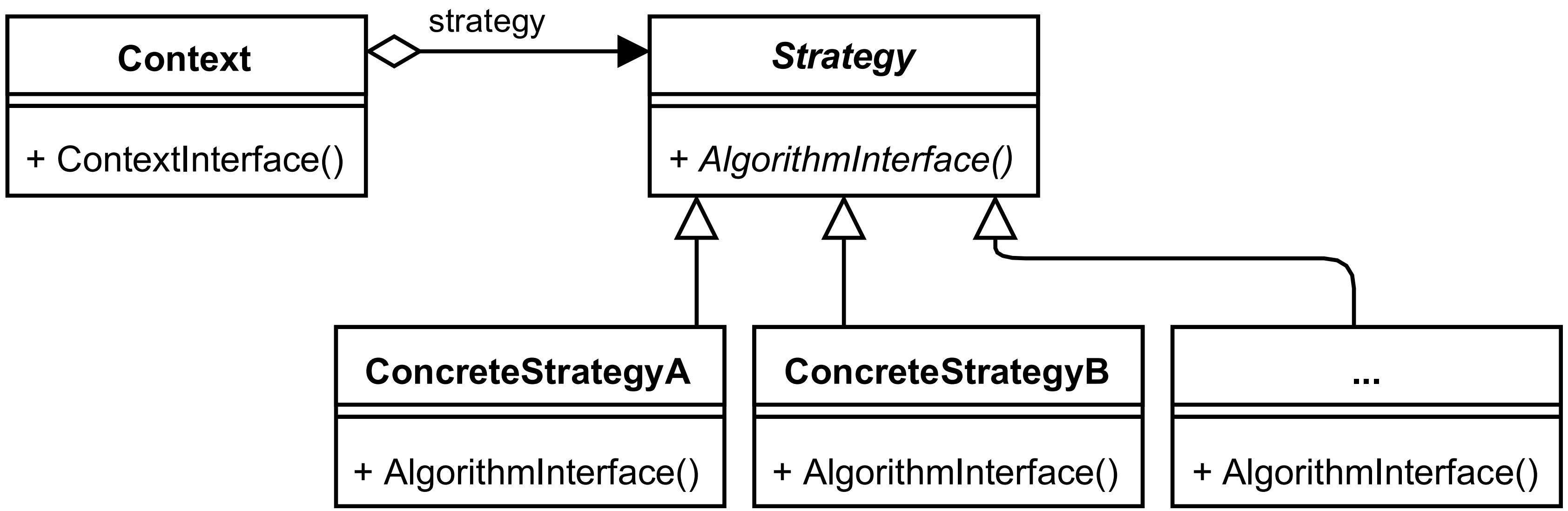
- Implement the functionality for each different object type in an visitor.
- Implement means to apply the visitor to every object.

Consequences:

+ Makes adding new functionality easy
+ Combines related functions
+ Account for different object types
+ Can accumulate state
~ Who traverses the aggregate? How?
~ Double-dispatch or not?
- Adding new class types is expensive
- Visitor may need access to private members (breaks encapsulation)

Strategy

Substitute behaviour later.



Strategy

Context:

•Many related classes which differ only in their behaviour.

•Methods with complex behaviour based on many conditionals.

Problem:

How to manage the different behaviours and simplify the architecture?

Forces:

- You need different variants for an algorithm.
- The behaviour should be exchangeable at runtime.
- You want to split up behaviour of classes to simplify it.

Solution:

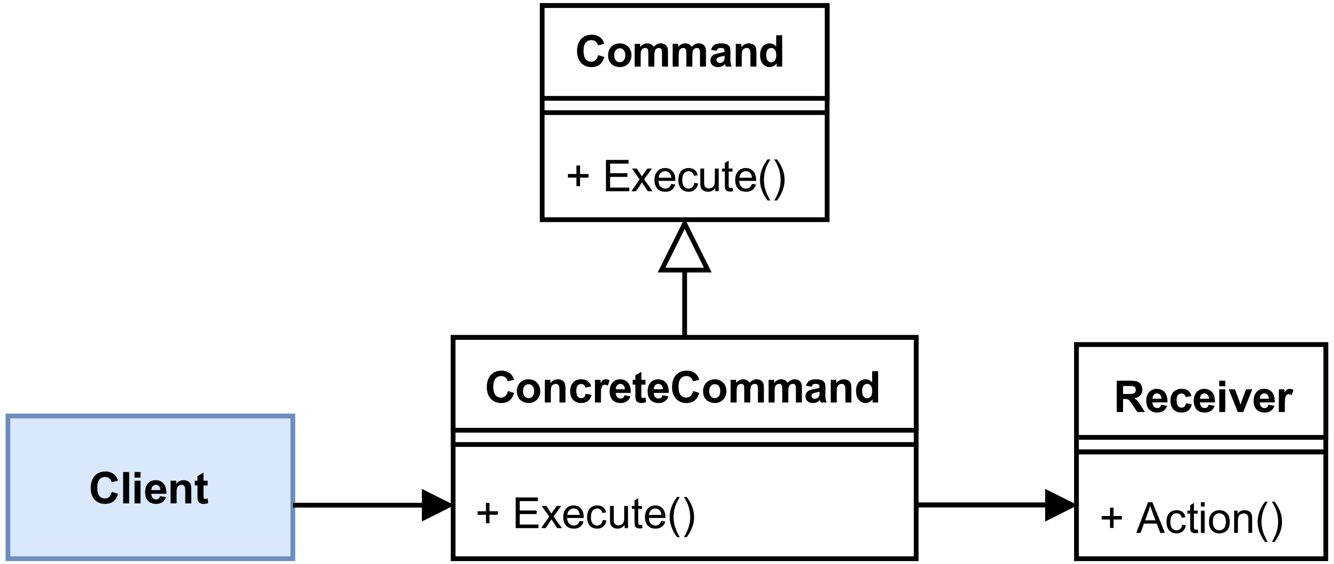
- Define interfaces for algorithms
- Encapsulate the algorithms to make them interchangeable.
- Let the algorithm vary independently from the clients.

Consequences:

+ Split up behaviour and decision logic.
+ Elimination of Subclasses just for different behaviour (composition over inheritance!)
+ Reuse: Behaviour of one class can be reused for others.
- Communication overhead
- Access to private fields?
- Increased number of objects (every behaviour is an own object)
~ Who assembles the concrete strategies at runtime?

Command

Encapsulate a request. Decouple invocation from execution.



Command

Context:

Invoking some behaviour of an object

Problem:

We just want to invoke an operation, regardless of its concrete implementation and executing context.

Forces:

- Avoid coupling of the invoker and the context of the request.
- We do not know the exact implementation of a request
- A request should be undoable

Solution:

•Define an interface for commands with a very simple interface (just Execute()).

•Encapsulate the behaviour in concrete commands implementing this interface and containing all needed parameters as members.

•Implement means to let the client initialise the parameters.

Consequences:

\+ A request does not depend upon the creating class anymore.

\+ A request can be executed in isolation.

\+ Undo/Redo-Operations become possible

\+ Switching the receiver at runtime becomes possible

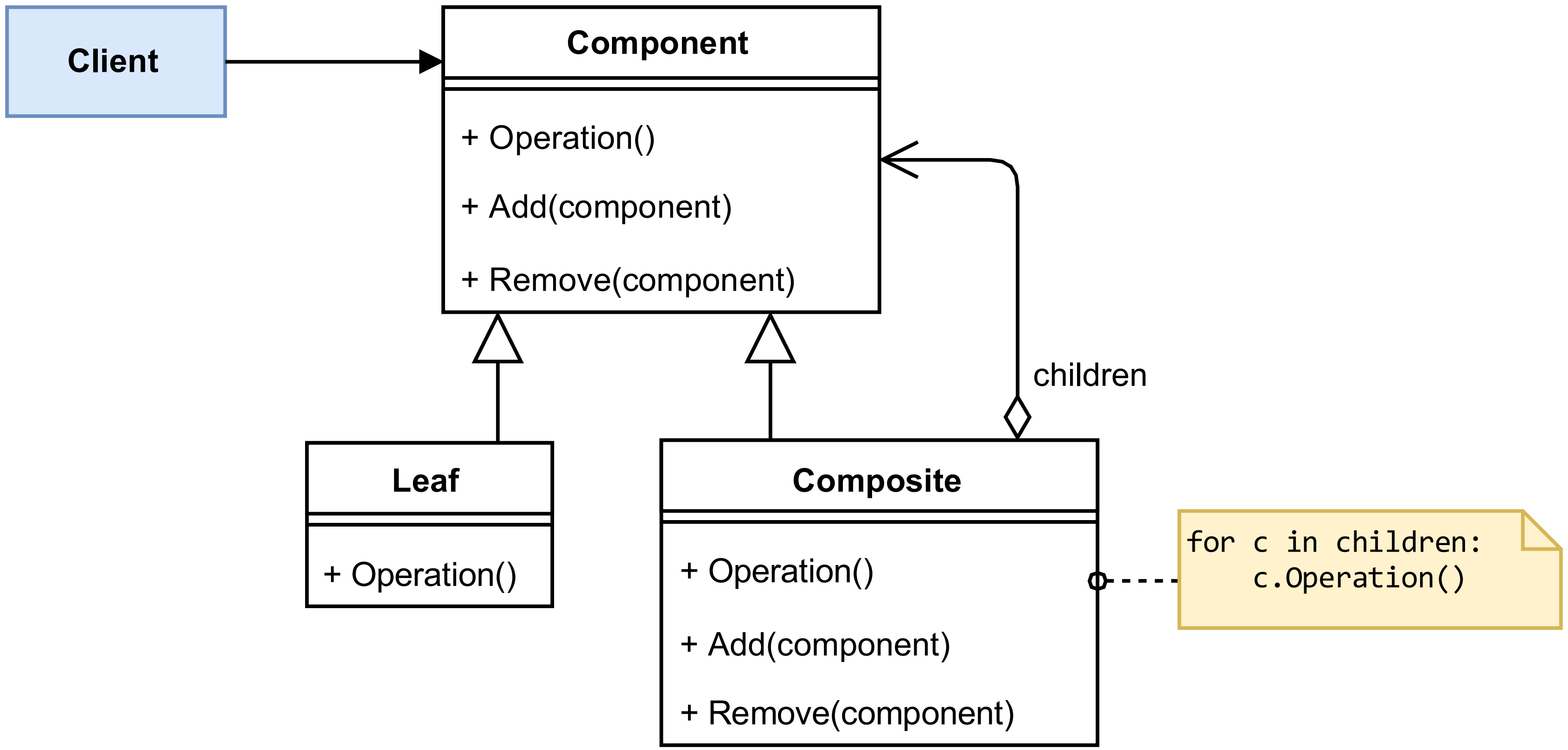
\+ Behaviour can be reused for multiple receivers.

\- Increased number of objects

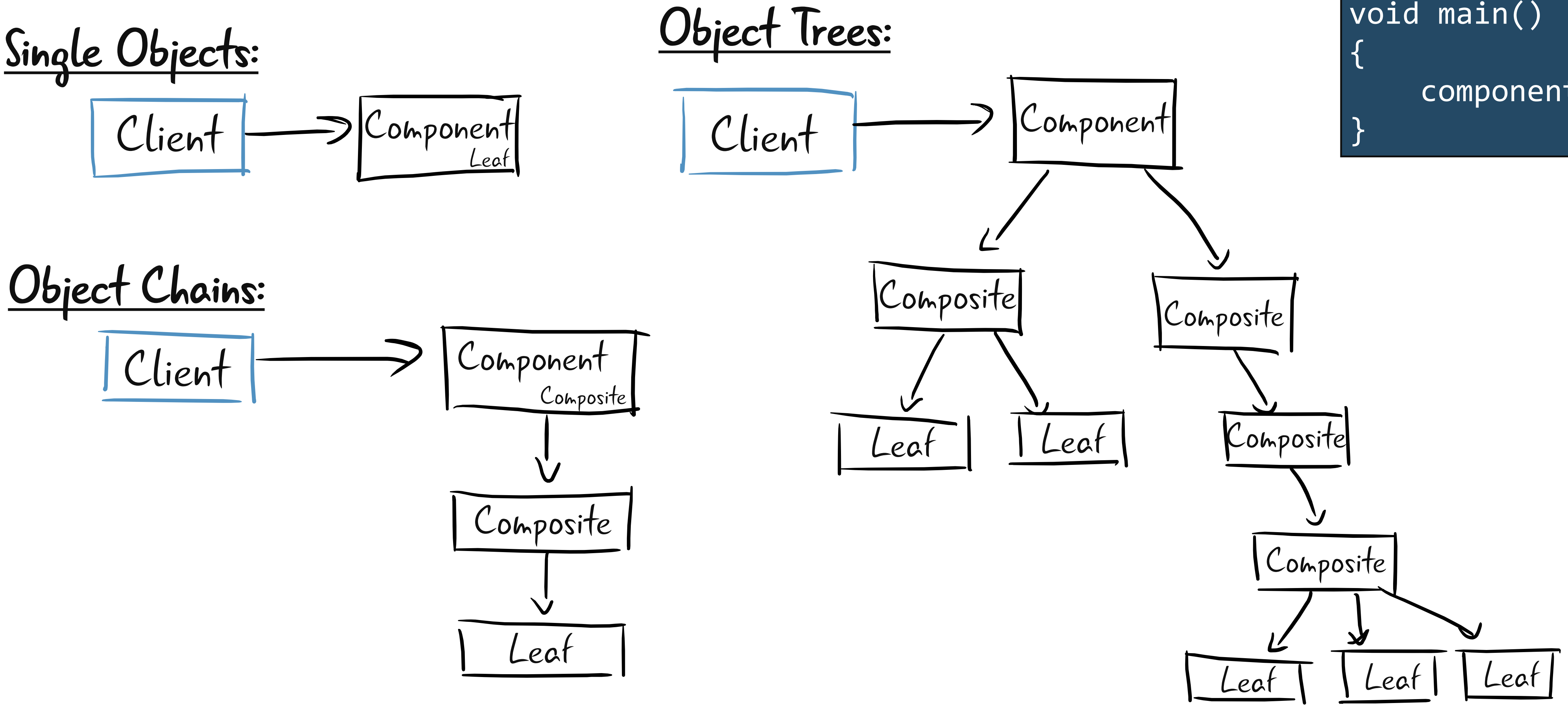
\- References to all needed parameters must be stored

Composite

Handle different granularities of objects uniformly



Composite



Client:

```
void main()
{
    component.Method();
}
```

Composite Examples

jQuery https://jquery.com

```
$(document).ready(function(){
  $("button").click(function(){
    $("p").hide();
  });
});
```

IDisposable (.NET)

```
public class MyControl : IDisposable
{
    private IDisposables _subControls;

    public void Dispose()
    {
        foreach (var c in _subControls)
            c.Dispose();
    }
}
```

Composite

Context:

Hierarchies of objects with different granularities

Problem:

How to uniformly handle different granularities of objects in hierarchies?

Forces:

- Treat all object-granularities uniformly
- Represent arbitrary hierarchies of objects
- Ignore differences behaviour of individual objects and aggregates
- Apply/Reroute a method call to all objects

Solution:

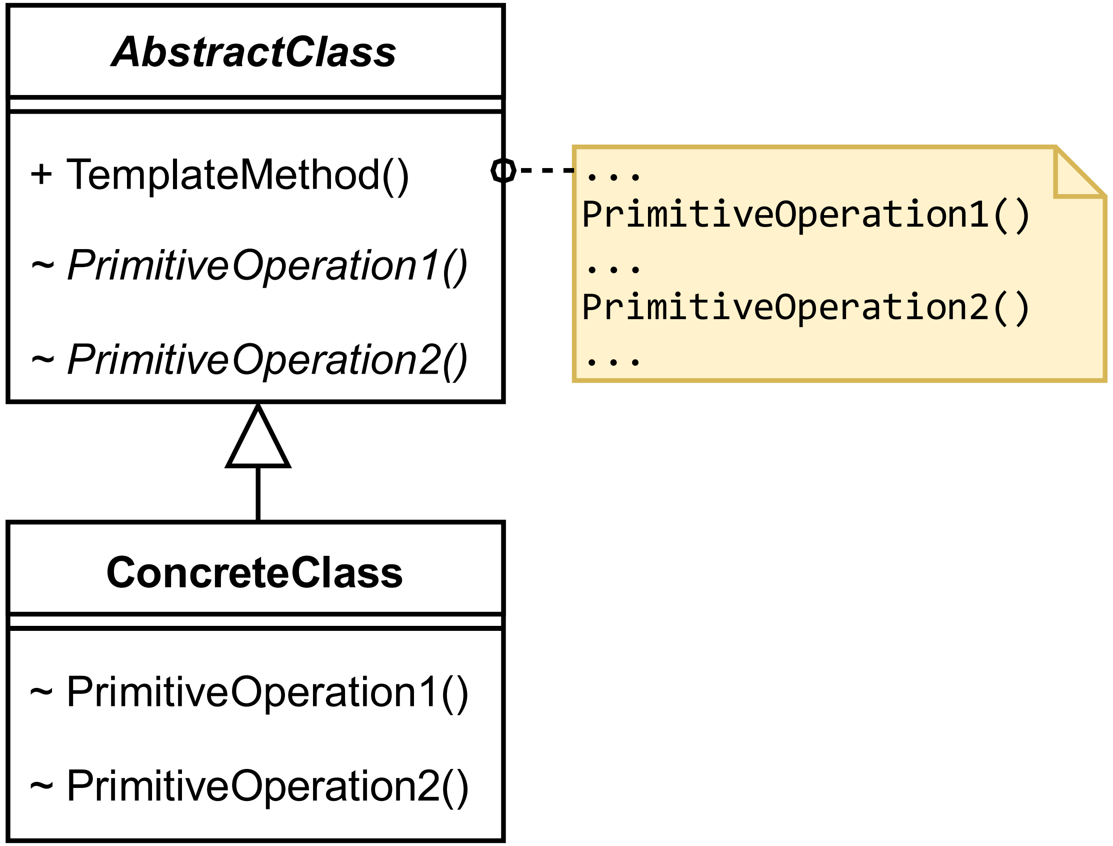
- Define common Interface for all granularities to manage children and call methods.
- Implement Composites: Forward call to children
- Implement Leaves: Execute calls directly

Consequences:

+ Defines hierarchies of primitive and composite objects
+ Simple handling for client
+ Adding new kinds of composites is easy
~ Default implementations?
~ Parent references?
~ Changing roles? Leaf ↔ Composite?
~ Caching?
- Client doesn’t recognize complexity of calls.
- High call hierarchy
- Possibly unrecognized side effects?

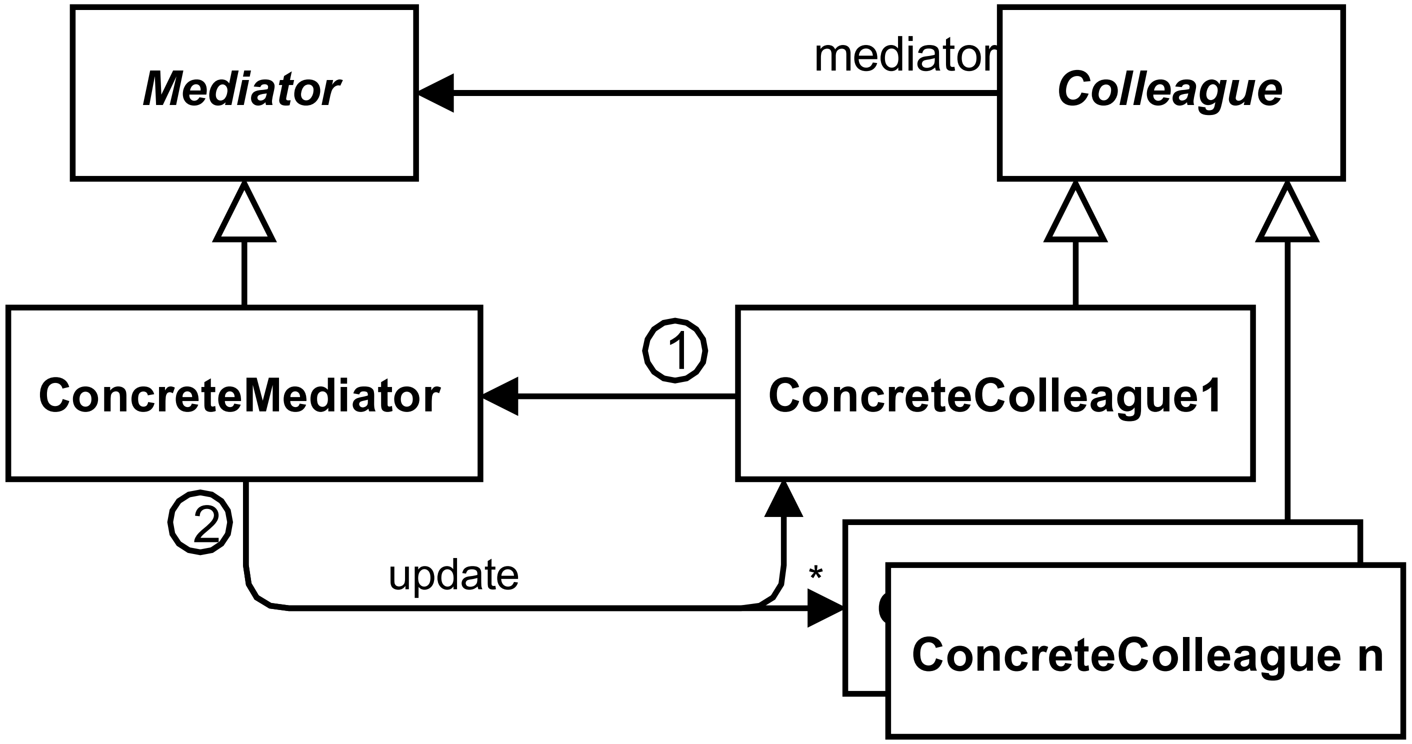
Template-Method

Define methods and let children implement the behaviour.



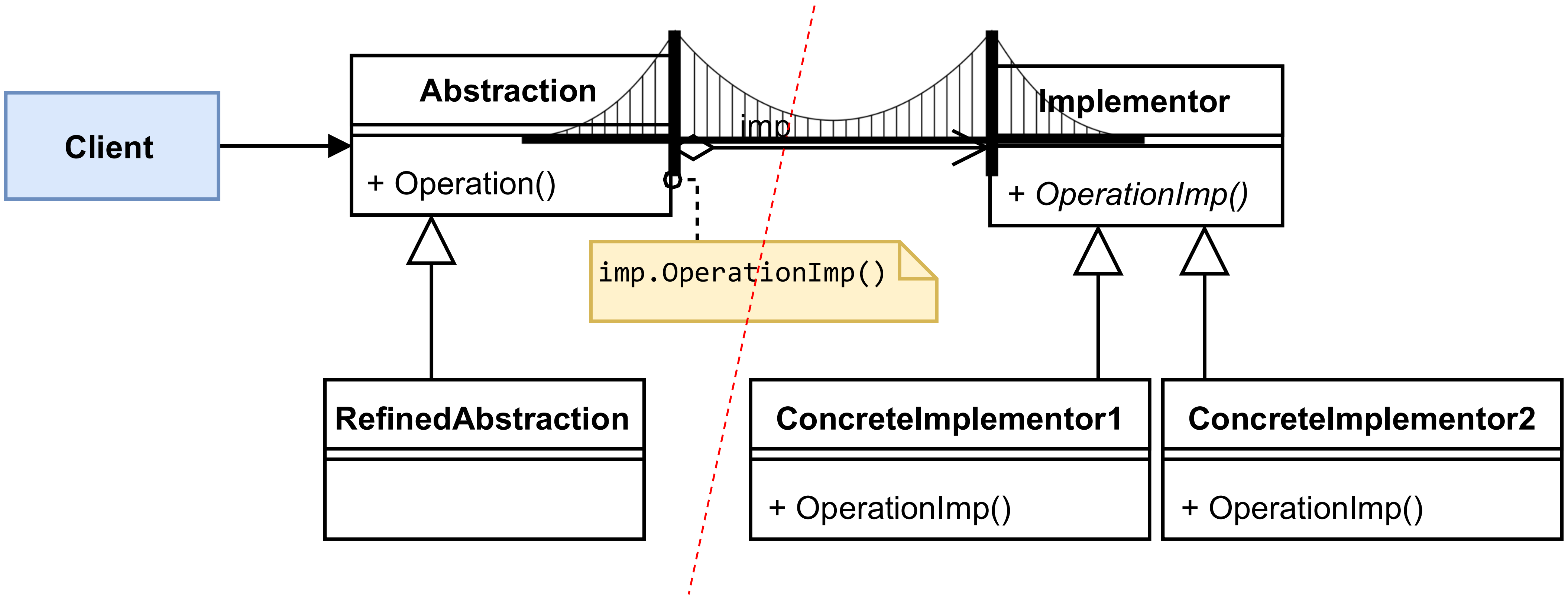
Mediator

Mediate communication between multiple objects



Bridge

Decouple abstractions from implementations



Bridge

Context: Application with Abstraction and Implementation Hierarchies

Problem: How to decouple the development of abstractions from its implementations

Forces:

- Avoid permanent binding between abstraction and implementation
- Both sides should be extensible by subclassing
- Changes should be contained to one side
- You want to hide the implementation side completely
- Implementations should be compatible to multiple abstractions

Solution:

- Create two interfaces:
  1. Implementor-Interface (internal primitives)
  2. Abstraction-Interface (client-requirements)
- Implement those Interfaces with individual classes.
- Only use the implementation-interface in the abstraction

Consequences:

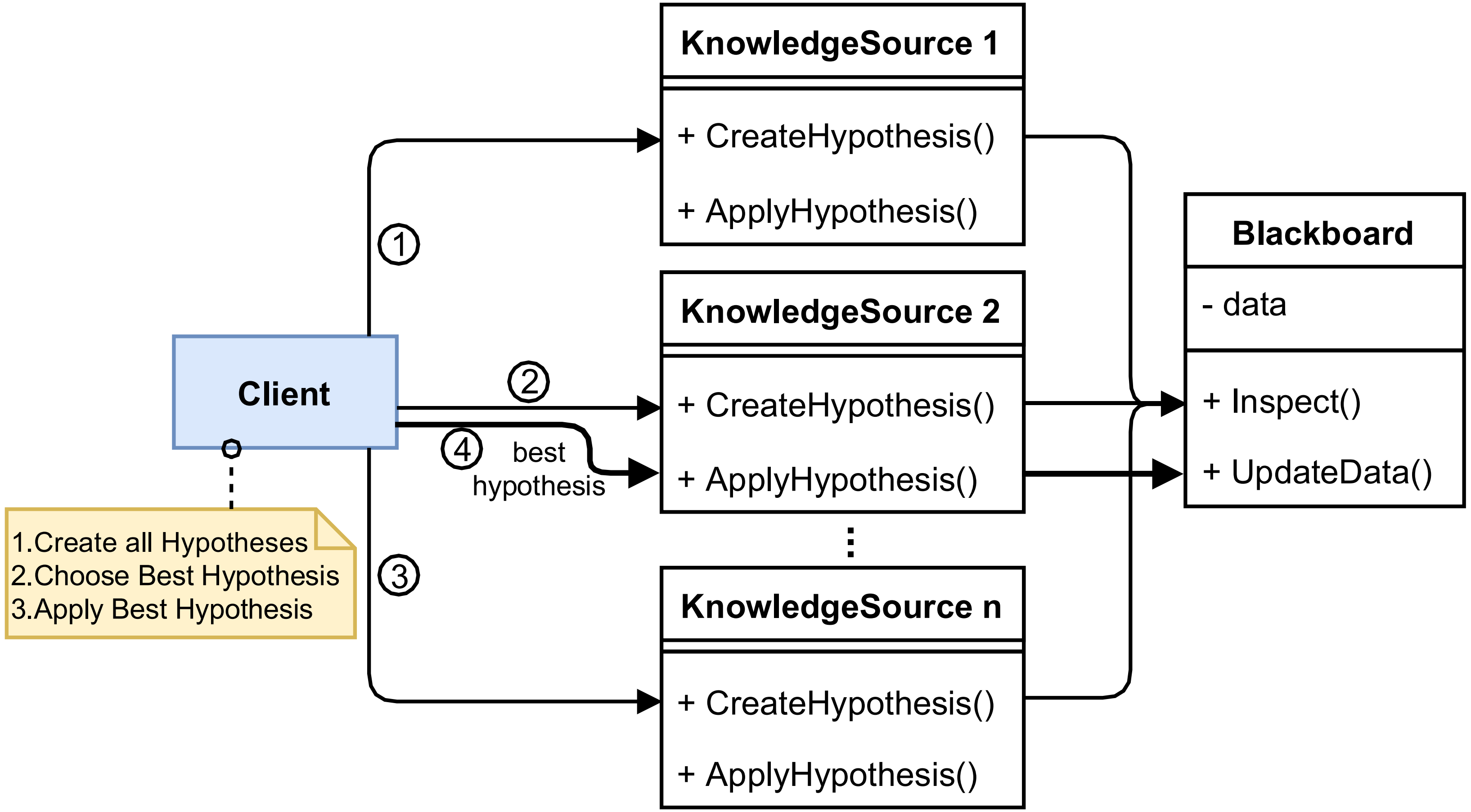
+ Decoupling of abstraction and implementation
+ Improved extensibility: Both sides can grow independently
+ Hiding implementation details from client
+ Implementation can be configured at runtime
+ Elimination of compile-time dependencies
+ Encourages layering

~ Who defines the composition? (Who builds the bridge?)

\- Higher complexity (more classes, more interfaces)

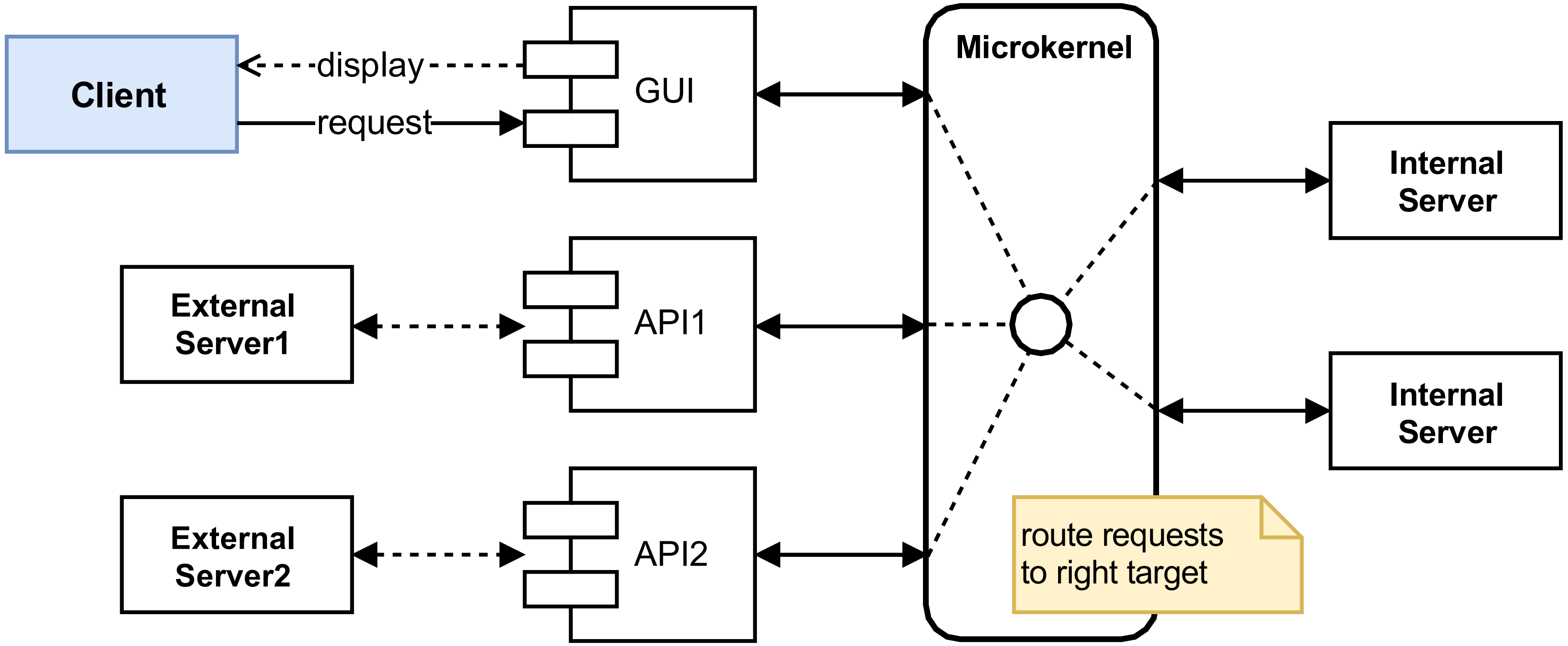
Blackboard

Collaborate on common data to get the best solution.



Microkernel

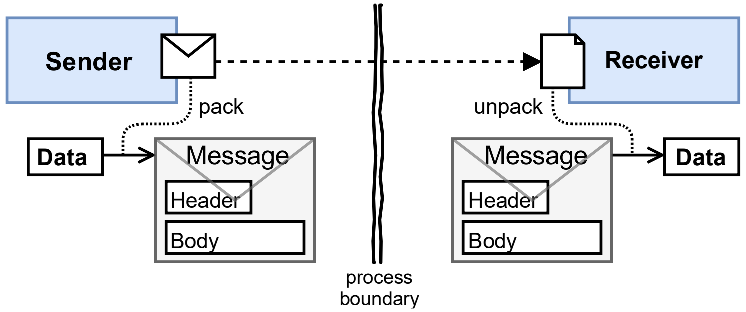
Route requests to the responsible components



Messaging Family

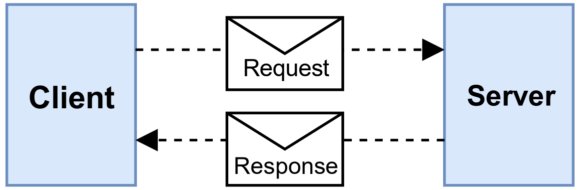
Pack information in messages and send these messages between components.

Messages

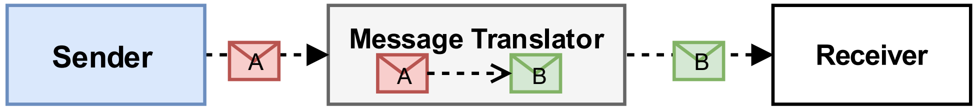


Message-Endpoint

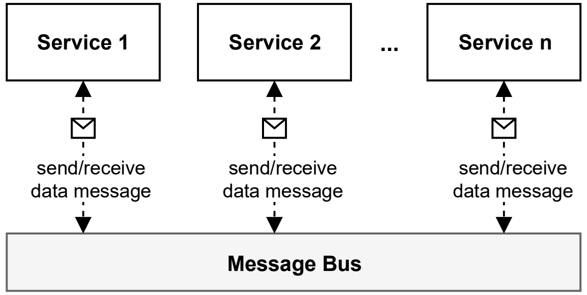
Request-Response



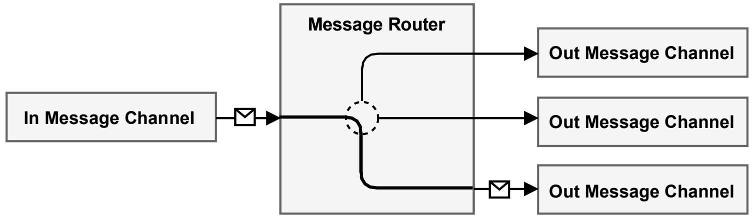
Message Translator



Message Bus

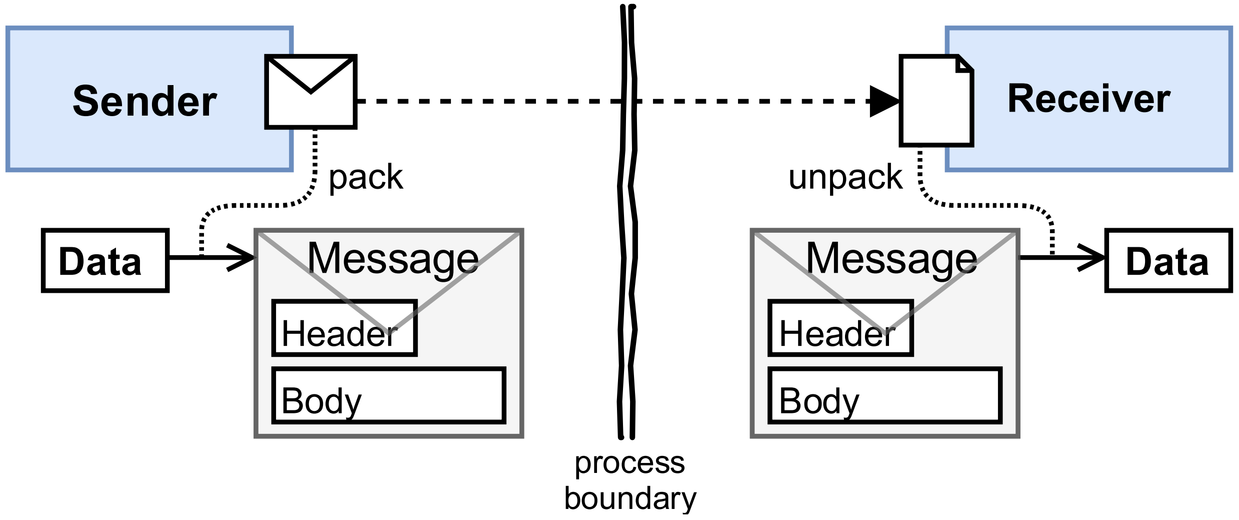


Message Router



Messages

Encapsulate information in a standardized way



Benefits:

+ Message combines Data and Meta-Data
+ Explicitly defined format/protocol
+ Enclosed packet instead of continuous data-stream
+ Meta-Data allows extra functionalities

Drawbacks:

- Computation overhead for serialization and deserialization
- Communication overhead due to protocol
- Version Chaos / Change-Management
- Data-Format must be exactly defined

Message Endpoint

Provide functionality to send and receive messages

Benefits:

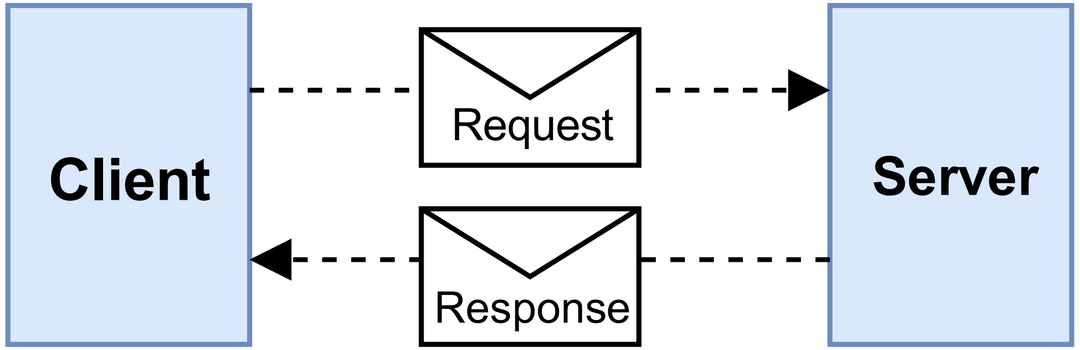
+ Clearly defined responsibility
+ Endpoint converts into/from a message
+ Endpoint can be reused
+ Decoupling of external protocol and internal communication

Drawbacks:

- Changes in message-protocol have to be communicated
- May introduce performance overhead (additional abstraction layer)
- Single Point of Failure? Bottleneck?

Request-Response

Answer every request with a response message



Benefits:

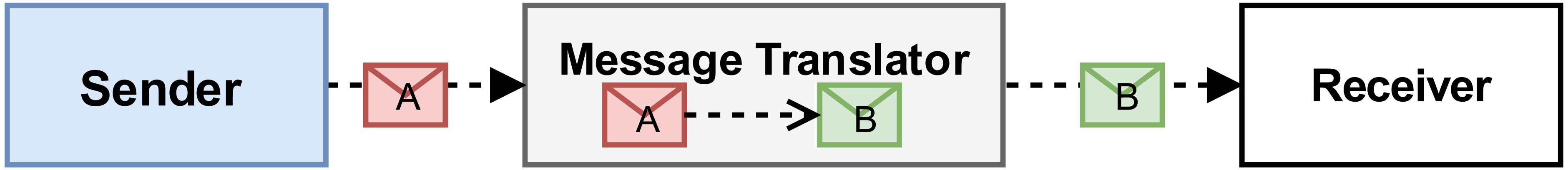
+ Every request gets answered
+ Timeouts can be detected
+ Windowing and Buffered Responses possible
+ Two decoupled communication events: Question and Answer

Drawbacks:

- Continuous Data Stream not possible
- Broadcast/Multicast not possible
- Asynchronous Communication is more difficult to debug
- Error Handling? (Error-Response or no Response?)

Message Translator

Translate between different message formats



Benefits:

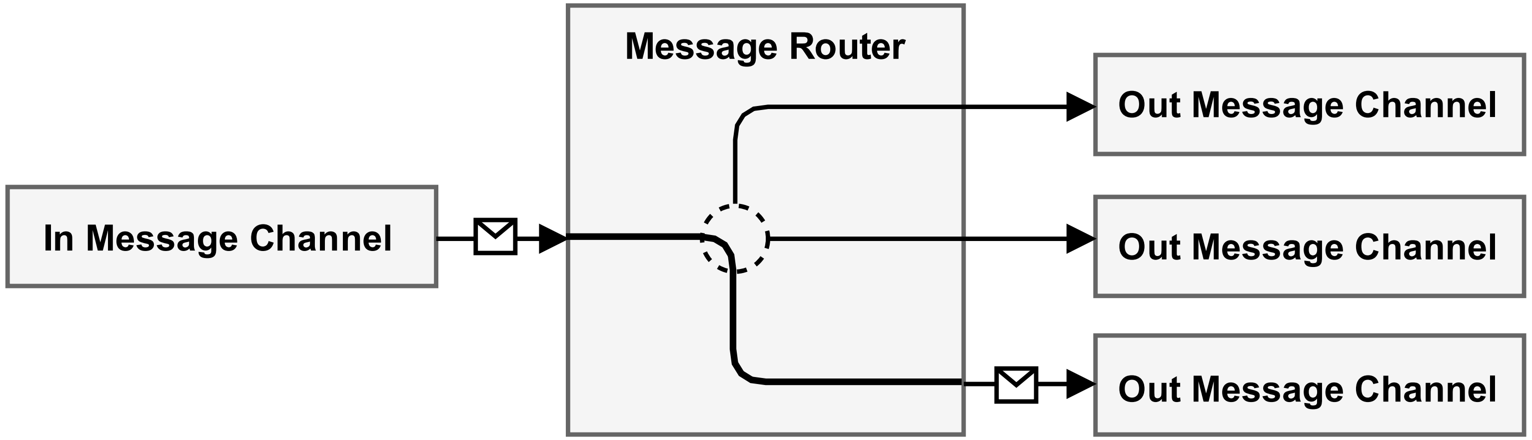
- + Sender and Receiver don‘t have to know the same protocols/message format
- + Translator can be reused
- + Translation can be parallelized

Drawbacks:

- - Translator needs to know both protocols!
- - Performance Overhead for additional translation
- - Protocols may be incompatible / Only degraded basic communication is possible

Message Router / Message Queues

Transmit messages to the right receiver



Benefits:

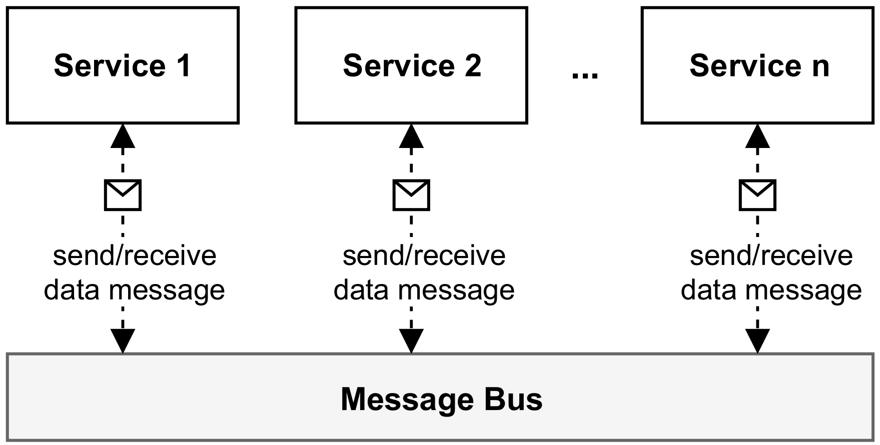
- \+ Sender and Receiver are decoupled
- \+ Dynamic rerouting is possible
- \+ Message Queues allow: Retransmissions, Guaranteed Delivery, Adaptive Transmission-Rate
- \+ Multicast / Broadcast

Drawbacks:

- \- Bottleneck / Single Point of Failure
- \- Man-In-The-Middle Attacks
- \- Loosing Messages on Failure?
- \- Loops and Broadcasting misuse
- \- Configuration Overhead

Message Bus

Provide a common communication platform which can be used to send and receive messages.



Benefits:

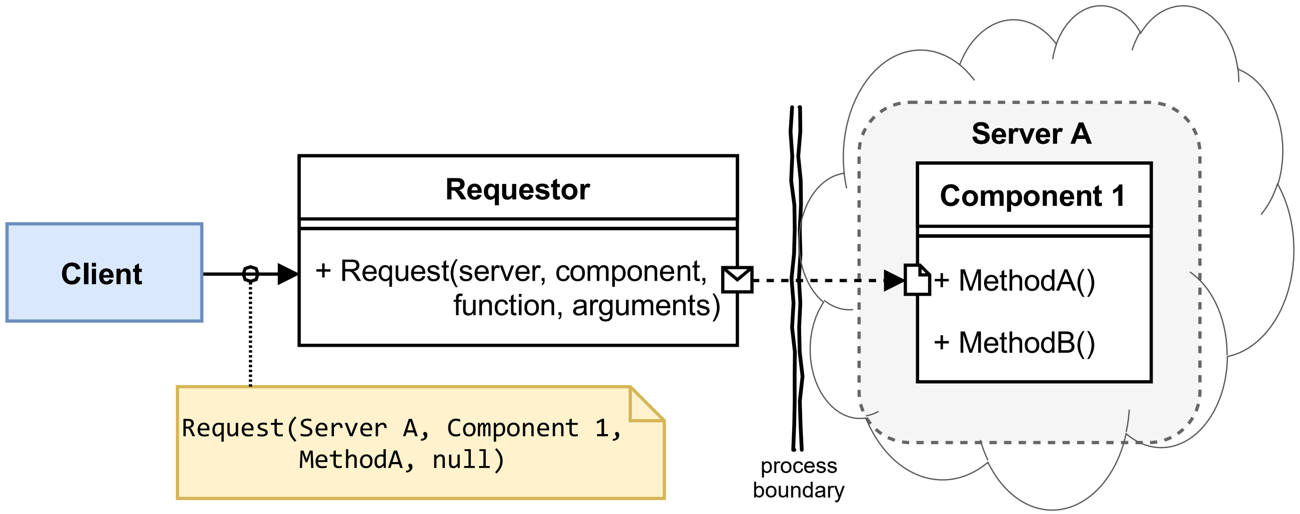
- + Unified communication platform & protocol
- + Communication can be controlled (Congestion Control)
- + Prioritization
- + Single interface for communication

Drawbacks:

- \- Bottleneck / Single Point of Failure
- \- Broadcasting / Babbling Idiot
- \- Security Issues?
- \- Forced communication protocol / maybe inefficient

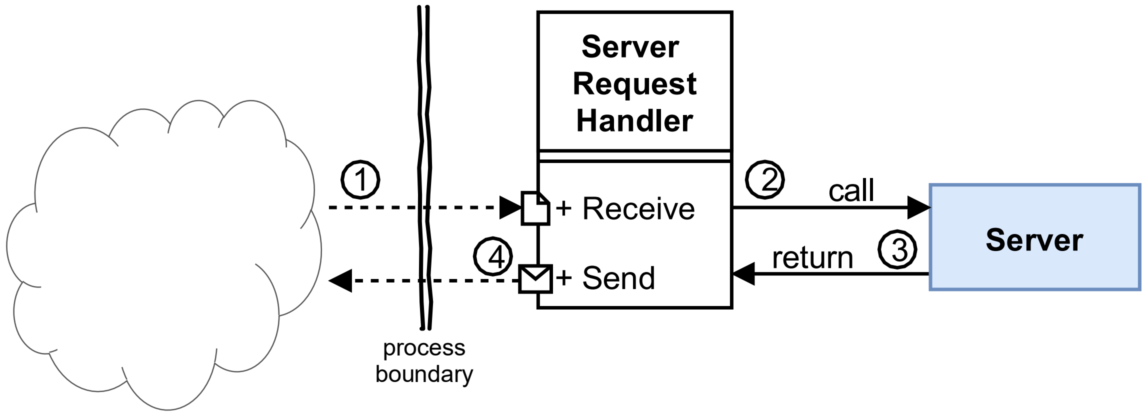
Requestor

Send generic requests and arguments



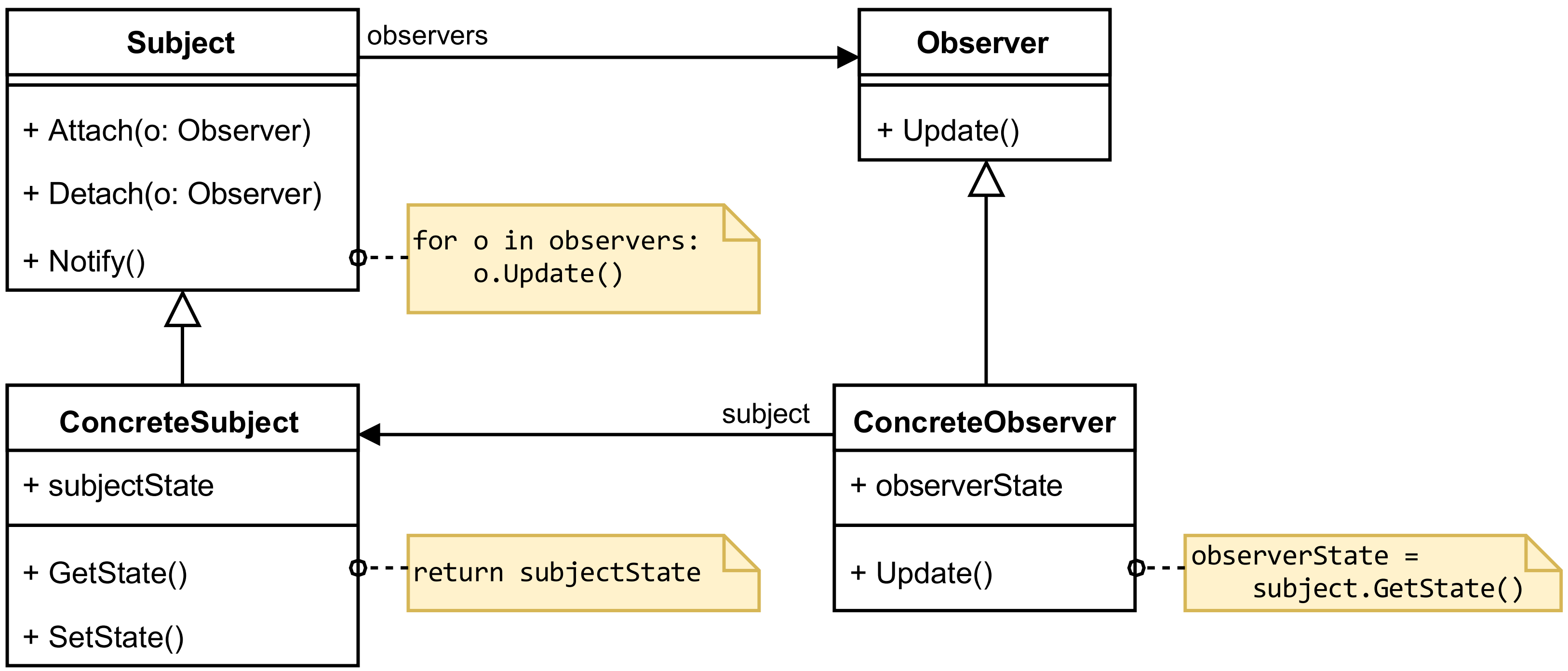
Request Handler

Listening, Receiving, Sending, and Handling of Message-Based Communication



Observer

Inform registered observers about changes.



Observer

Context: data is distributed over multiple related objects.

Problem: Maintain consistency between related objects.

Forces:

- When one object changes, others should be held consistent.
- Polling is very costly or not possible.
- The other objects are not known at compile-time and should not be tightly coupled.
- Reuse even in isolation should be possible.

Solution:

- Define means to manage observers for a subject (register, unregister).
- On changes: notify all observers that a change happened.
- Give the observers the possibility to access the changed data.

Consequences:

+ Decouple subjects and observers.
+ Reuse subjects and observers.
+ Polling is not needed anymore.
+ Support for m:n communication.
- Unexpected updates / Frequent updates / Cascading updates.

~ Synchronous vs. Asynchronous updates!

~ Who initiates the update?

Observer - Example

```
abstract class Subject {
  private readonly List<Observer> _observers = new List<Observer>();
  public void Attach(Observer observer) => _observers.Add(observer);
  public void Detach(Observer observer) => _observers.Remove(observer);
  public void Notify() => _observers.ForEach(o => o.Update());
}
```

```
abstract class Observer {
  public abstract void Update();
}
```

```
class ConcreteSubject : Subject {
  private string _state;
  public string State {
    get { _state; }
    set {
      _state = value;
      Notify();
    }
  }
}
```

```
class ConcreteObserver : Observer {
  public ConcreteSubject Subject;
  private readonly string _name;
  public ConcreteObserver(ConcreteSubject subject, string name) {
    Subject = subject;
    subject.Attach(this);
    _name = name;
  }
  public override void Update() {
    Console.WriteLine($"Observer {_name} was informed
                          that subject changed: {Subject.State}");
  }
}
```

```
void Main() {
  var s = new ConcreteSubject();
  var o1 = new ConcreteObserver(s, "O1");
  s.State = "Change 1";
}
```

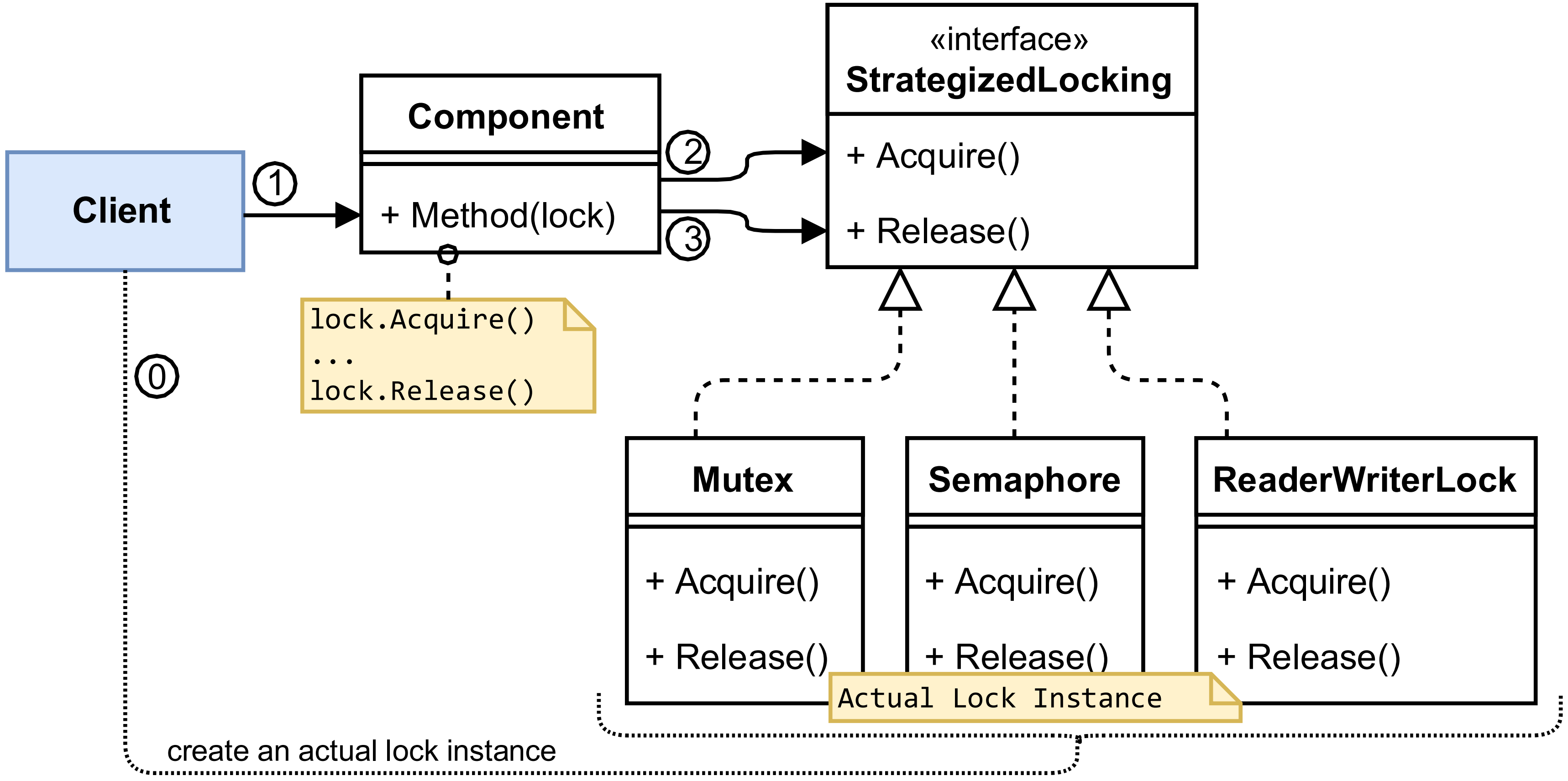
Observer Known Uses

- Events and Signals in many programming languages and operating systems
  e.g. Events like OnClick, OnEnter, OnKeyDown in C#, Java, JavaScript, …

- Message Queue Systems: MQTT, Apache Kafka, RabbitMQ

Locks: Mutex, Semaphore, Read/Write Lock, Condition Vars

Ensure mutual exclusive access to some resource.



Locks

Context: Simultaneous access to resources

Problem: How to avoid conflicts and ensure the same view for all accessors?

Forces:

- Parallel access to shared resources (multiple Threads or Processes)
- Locally on one machine
- Read or Write access
- Avoid conflicts (who writes first)
- Enforce consistency (same view for all accessors)

Solution:

- Acquire lock before accessing a resource or wait until lock is available.
- Release the lock after resource is not needed anymore.
- Use a Lock which is synchronized and atomic to the client

Consequences:

\+ Access to resources is mutually exclusive

\+ Logic order is established

~ Which lock is appropriate?

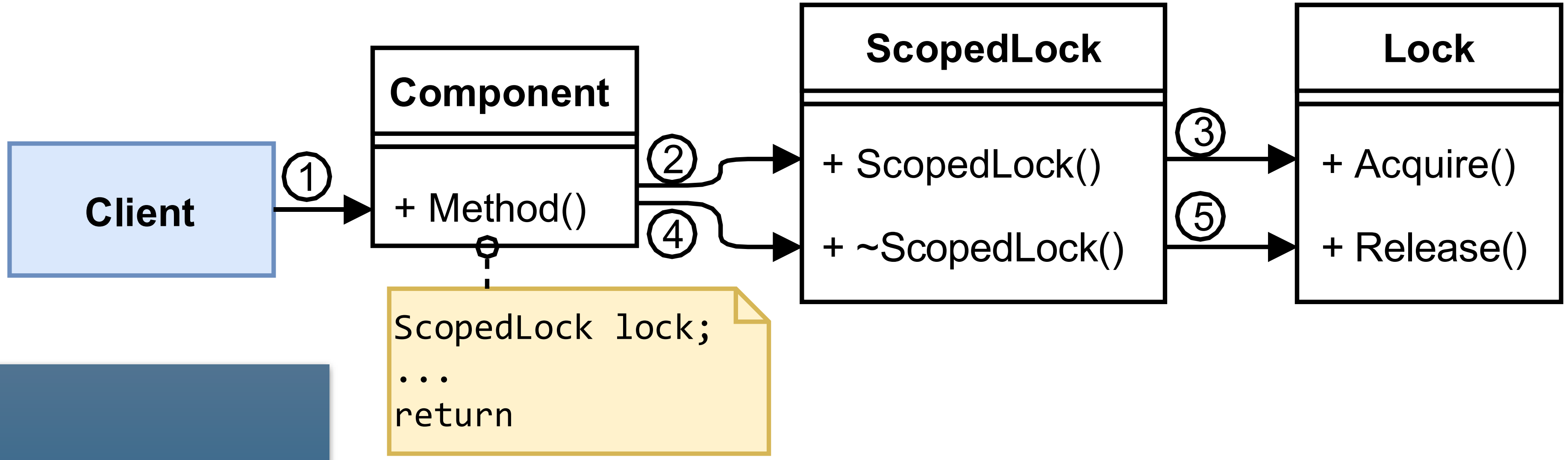
~ Maybe lock is not needed? (Immutable data types? Thread specific storage? Lock-Less Implementations?)

\- Using locks produces overhead & waiting times

\- Race-Conditions & Deadlocks

Scoped Locking

Use language scope semantics for acquiring and releasing locks.




```
class lock_guard
{
    …
    void lock_guard(mutex_type& m)
    {
        _m = m;
        _m.lock()
    }

    void ~lock_guard()
    {
        _m.unlock();
    }
}
```

```
std::mutex _mutex;
int _current = 0;

void increment() {
  std::lock_guard<std::mutex> lock(_mutex);
  ++_current;
}
```

Scoped-Locking

Context: Using locking mechanisms to protect a critical section

Problem: How to avoid forgetting to release the lock?

Forces:

- A critical section of code should be protected for concurrent access with a lock
- The section may have multiple exit points
- Developers tend to forget to release locks on the right places

Solution:

- Implement a class which:
  - Acquires a mutex in constructor
  - Releases the mutex in destructor
- Hide copy constructor and assignment operator
- Use like a normal stack variable and rely on stack-unwinding to call the destructor on leaving a scope

Consequences:

\+ Increased robustness

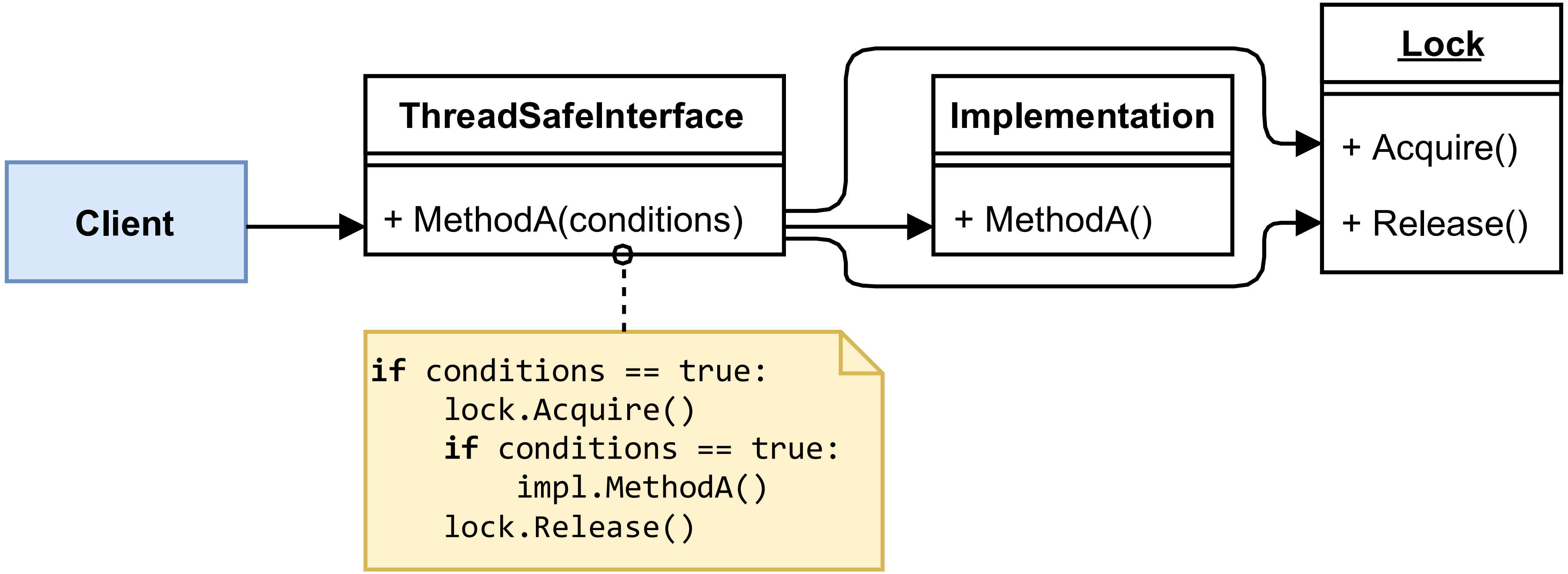
\+ Very simple usage

\- Potential deadlock when used recursively (reacquire lock needed?)

\- Limitations due to language specific semantics (process abort SIGC, longjmp)

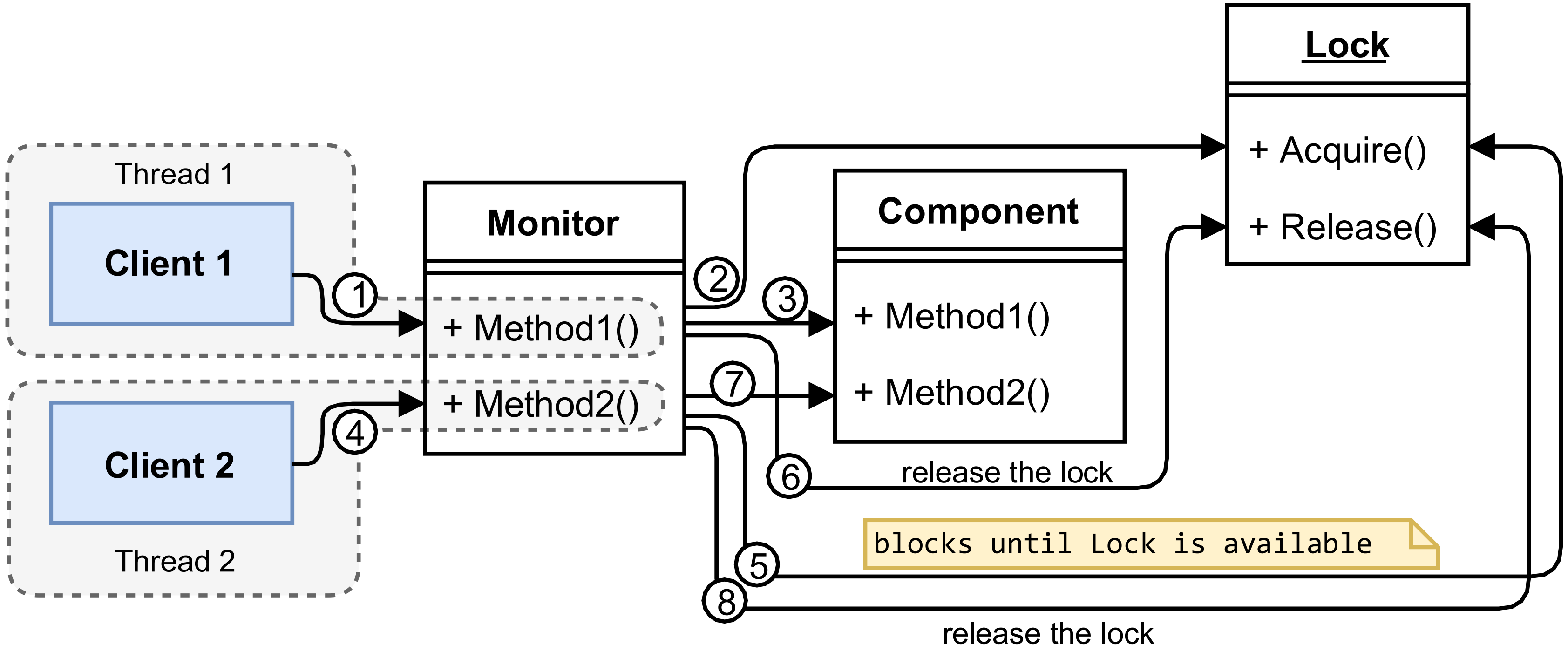
Double Checked Locking

Check twice to ensure conditions



Monitor

Synchronize method calls to an object



Monitor

Context: Multiple threads accessing an object concurrently

Problem: How to concurrently access an object and call the method without manual synchronization?

Forces:

- Concurrent invocation of methods in an object by multiple threads
- Prevent race conditions: only one method should be active
- Method calls should be synchronized
- Object state should stay stable and resumable

Solution:

•Use a general lock for one object instance

•Acquire lock before method call, Release after method is finished.

Consequences:

\+ Simplification of concurrency control

\+ Simplification of scheduling method execution

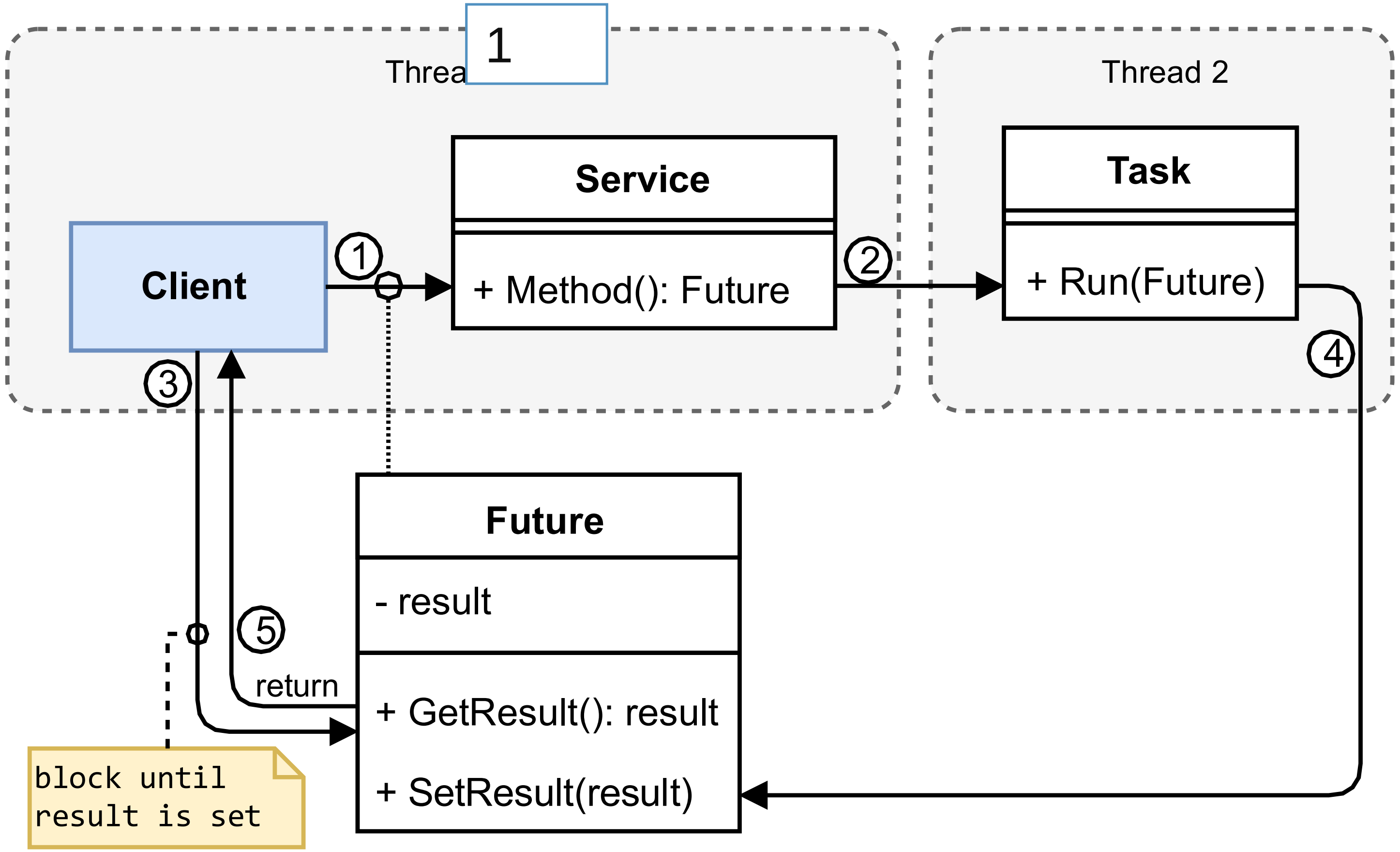
\- Limited Scalability – too coarse lock! (extreme case: GIL in python!)

\- Inheritance/Extension is dangerous

\- Nested monitors – reaquiring locks?

Future

Supply a placeholder for future results



Future

Context: Asynchronous method calls

Problem: How to get the result of an asynchronous method call?

Forces:

- You want to do the call asynchronously.
- You don't know when the call is finished.
- You want to access the result.
- You don't want to busy wait.
- You don't want to expose internal concurrency mechanisms

Solution:

- On calling a method immediately return a handle which will contain the result in the future.
- Execute the task asynchronously.
- As soon as the Task is finished, write the result to the future-handle.
- Give the client a possibility to check if the result is available or wait for it.

Consequences:

\+ User has the possibility to work with "future" results

\+ Asynchronous programming gets easier

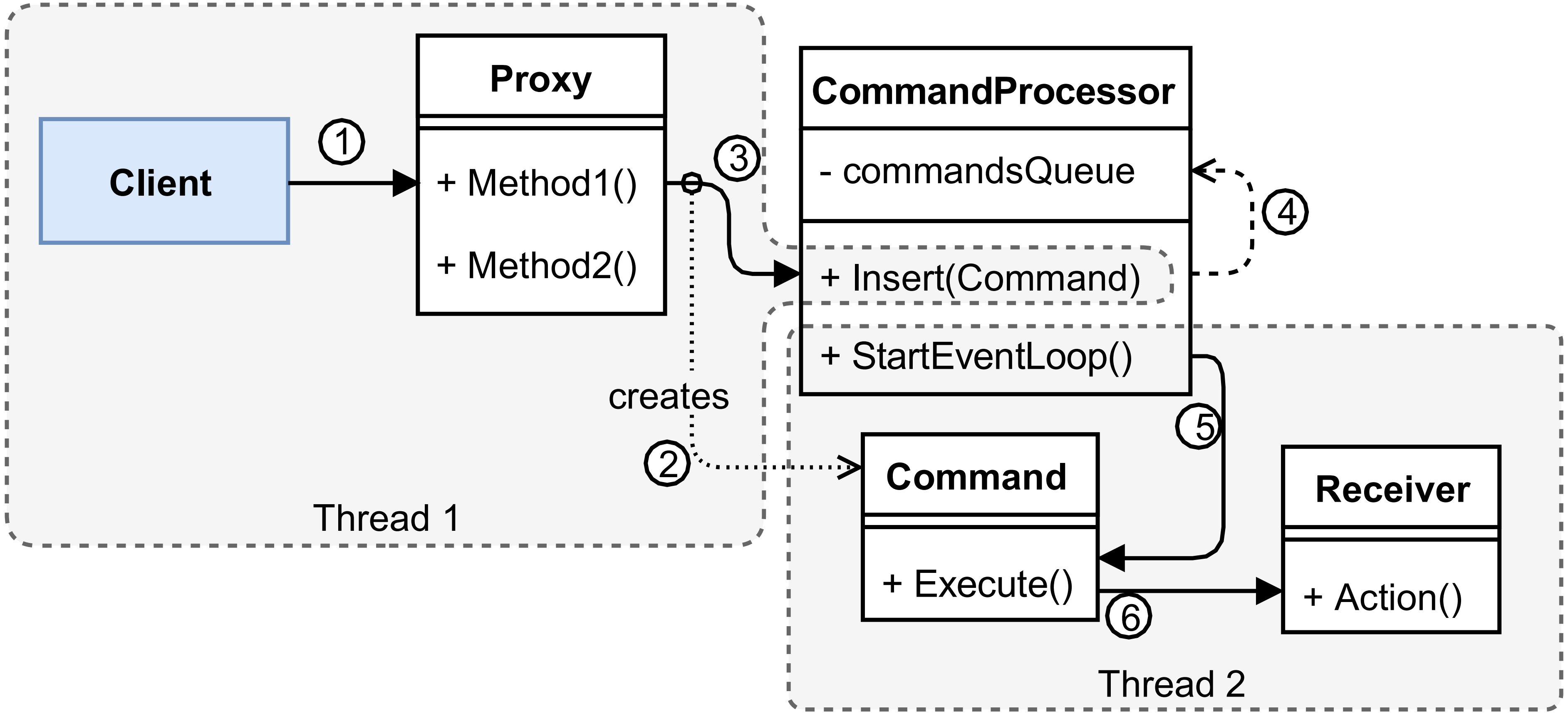
\- No immediate control over the executing thread (no way to cancel, pause)

\- Additional memory is needed, to hold results when thread is finished.

\- When result is not needed the future-handle is useless.

Active-Object

Encapsulate method invocation and execute asynchronously



Active-Object

Context: Multiple clients access objects running in different threads or contexts.

Problem: How to execute commands in a different context than the client.

Forces:

- Clients invoke remote operation and retrieve results later (or wait)
- Synchronized access to worker threads
- Make use of parallelism transparently

Solution:

- Implement a proxy with encapsulates all method calls in commands
- Use a Scheduler/CommandProcessor to execute the commands in a separate thread(pool).
- Give the client the possibility to retrieve or wait on the results (async/sync)

Consequences:

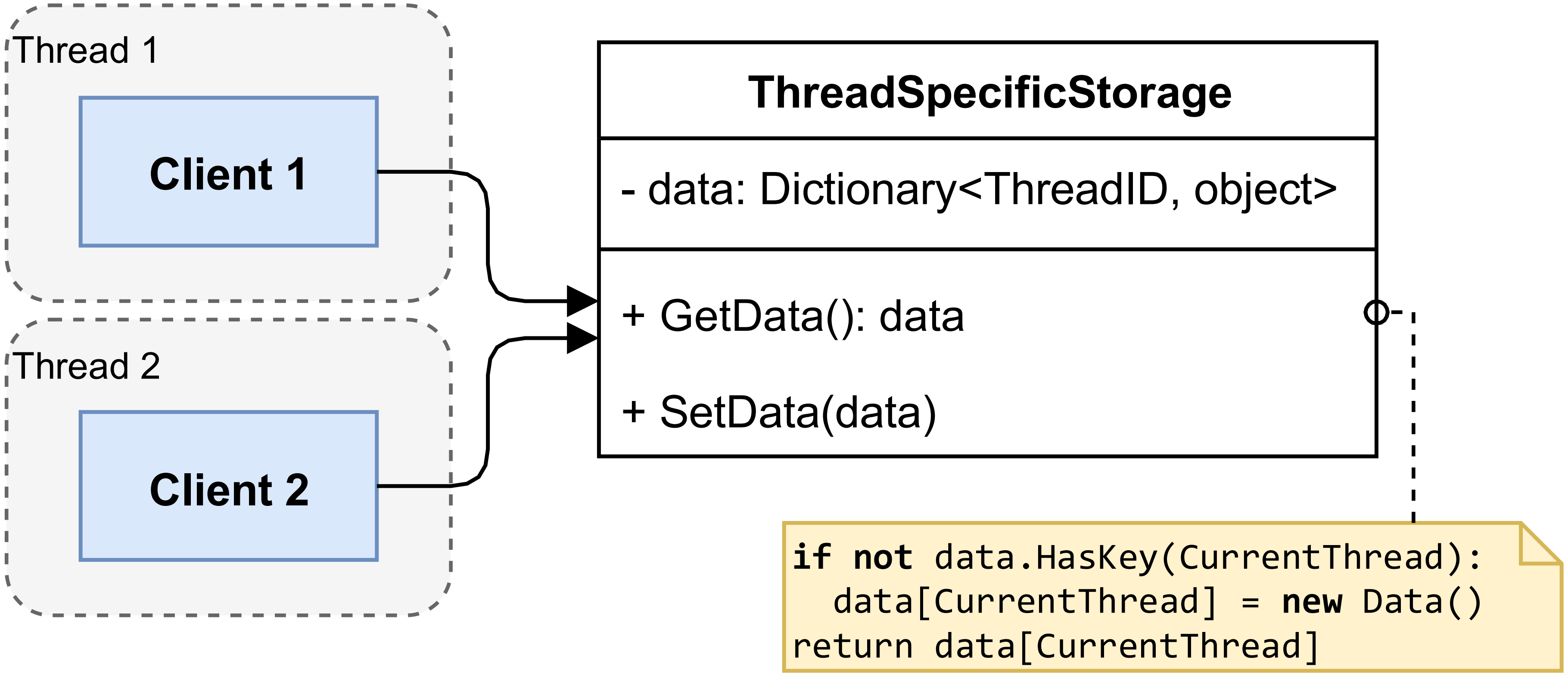
+ Simplifies sychronization complexity
+ Client calls an ordinary method
+ Command is executed in a different thread than the client thread
+ Typesafety compared to message passing (usage of classes/objects)
+ Transparent leveraging of parallelism

~ Order of method execution may differ from invocation

- Performance overhead
- Complicated debugging

Thread-Specific Storage

Store separate data instances for each thread.



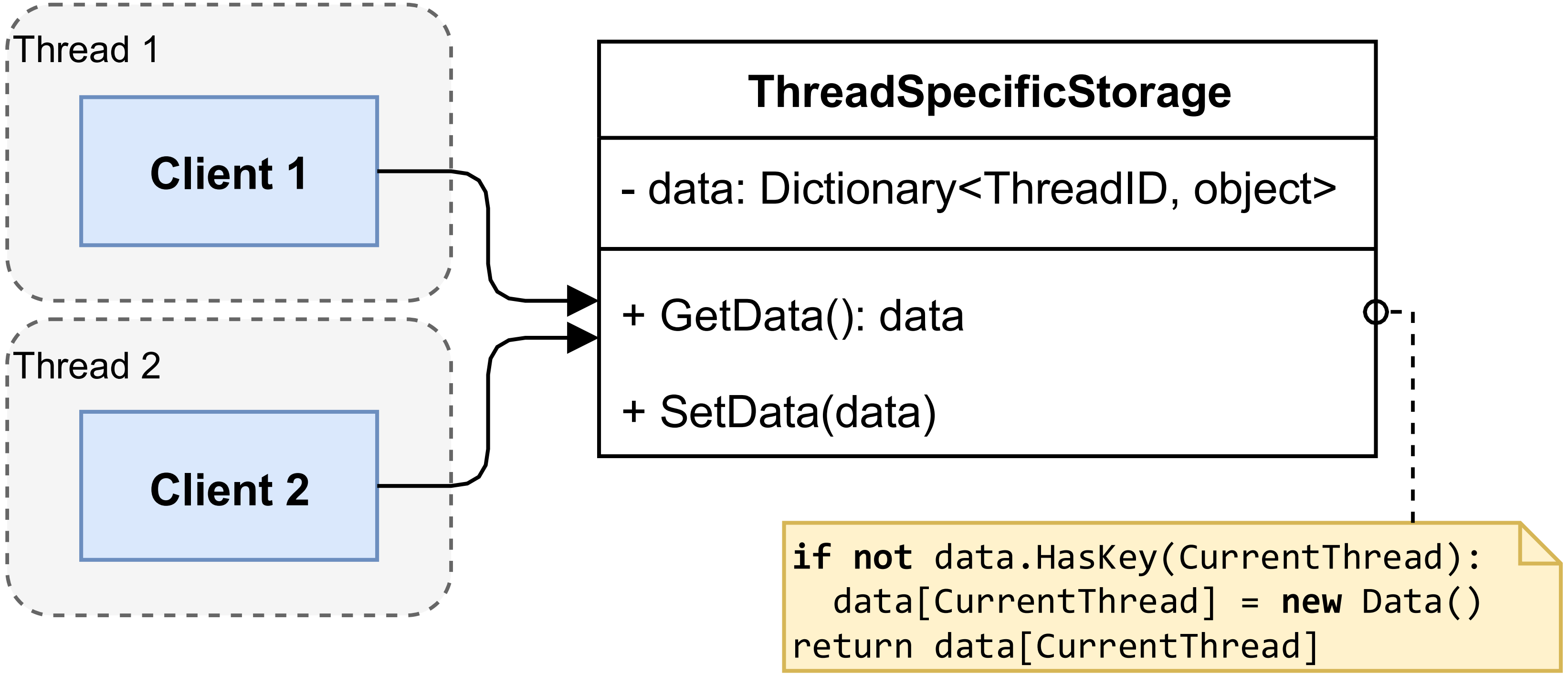
Thread-Specific Storage

Store separate data instances for each thread.

Thread ID	Data
1	{ Value = 5 }
2	{ Value = 38371 }

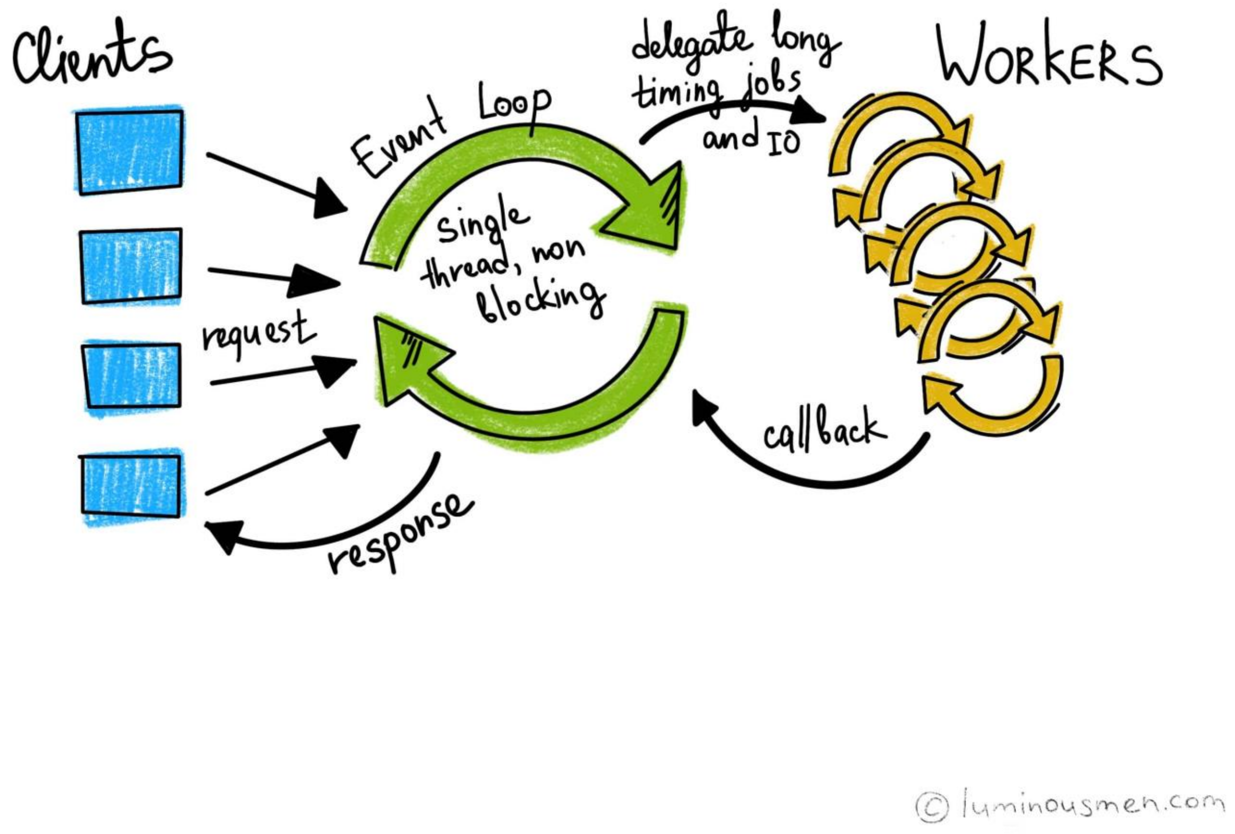
```
void main()
{
    var myObj = storage.GetData();
    myObj.Value = 5;
    data.SetObject(MyObj);
}
```

```
void main()
{
    var myObj = storage.GetData();
    myObj.Value = 38317;
    data.SetObject(MyObj);
}
```



Async / Await

Execute functions cooperatively in an event loop.



Async / Await

Context: Executing multiple functions waiting for I/O resources.

Problem: How to execute I/O-bound functions in parallel without having to use multithreading and synchronisation.

Forces:

- Executing the blocking functions sequentially is slow.
- Executing the functions in own threads may cause synchronisation problems or wasting resources due to context switching.
- Multithreading programming is error-prone.
- Some environments don't have true multithreading (python, javascript)

Solution:

•Compile the functions as state machines, with transitions at the "await" statement

•Execute the state machines in an event loop, advancing them based on a "ready"-condition (or signal).

Consequences:

\+ No need to use multiple threads.

\+ No need to synchronise.

\+ No unnecessary waiting times due to blocking functions.

\+ Simple usage (nearly like single-threaded programming, except for the "await" keyword).

\- Syntax and Compiler support needed.

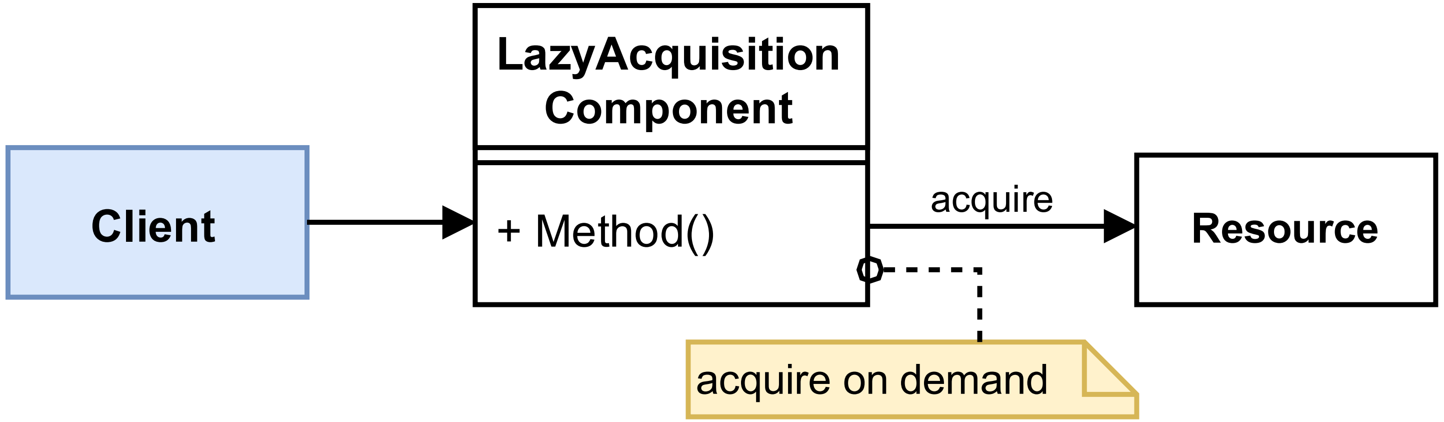
\- Must be supported throughout the whole application (async/await and non-blocking functions virtually everywhere)

\- Relies on cooperativeness!

\- CPU-bound functions still block everything.

Lazy Acquisition

Defer acquisition of resources to time of actual usage



Lazy Acquisition

Context: Using resources in an application

Problem: How to save resources and load an application faster?

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- They take time to load.
- They may be scarce.
- They are not needed from the beginning, but later on.

Solution:

•Implement a proxy which can be used by the client

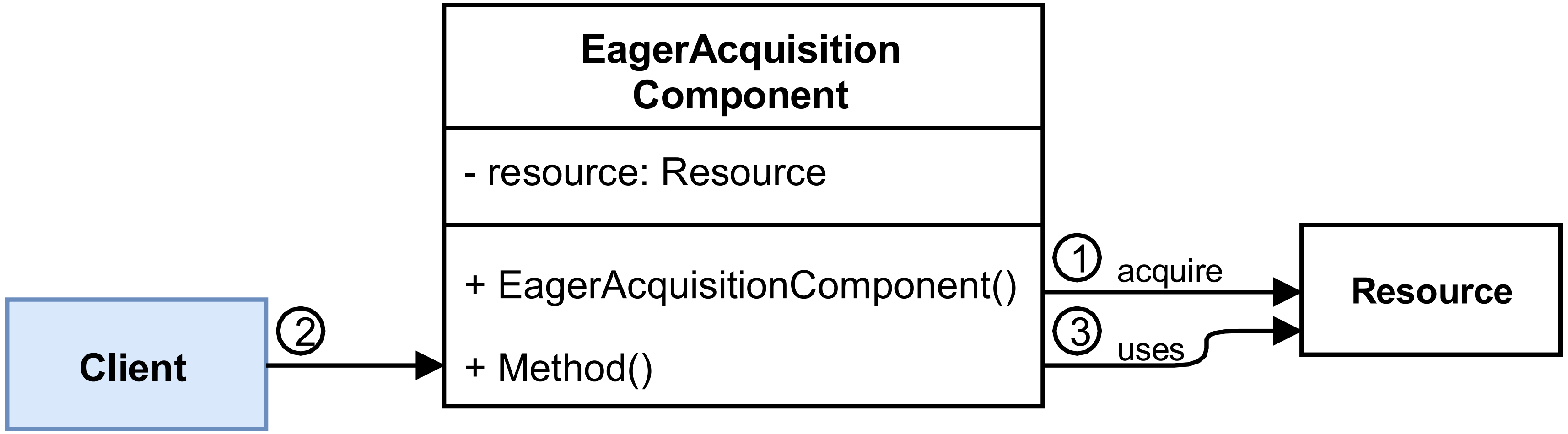
•The proxy should defer acquisition of the resource until the last possible moment.

Consequences:

+ Resources are only acquired when really needed.
+ Client doesn't have to care about using to much resources early on.
+ Application starts faster.
- Waiting times during acquiring the resources (do it async!)
- Additional layer of abstraction
- Avoid acquiring resources to often (caching & pooling!)

Eager Acquisition

Acquire resources in advance.



Eager Acquisition

Context: Using resources in an application

Problem: How to avoid having to wait for resources during runtime.

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- Exclusive access is no problem.
- They are always needed in the application.

Solution:

- Acquire the resources on startup and store them in some cache or vault (singleton).
- Give access to the already loaded resources

Consequences:

\+ Resources must not be loaded later on.

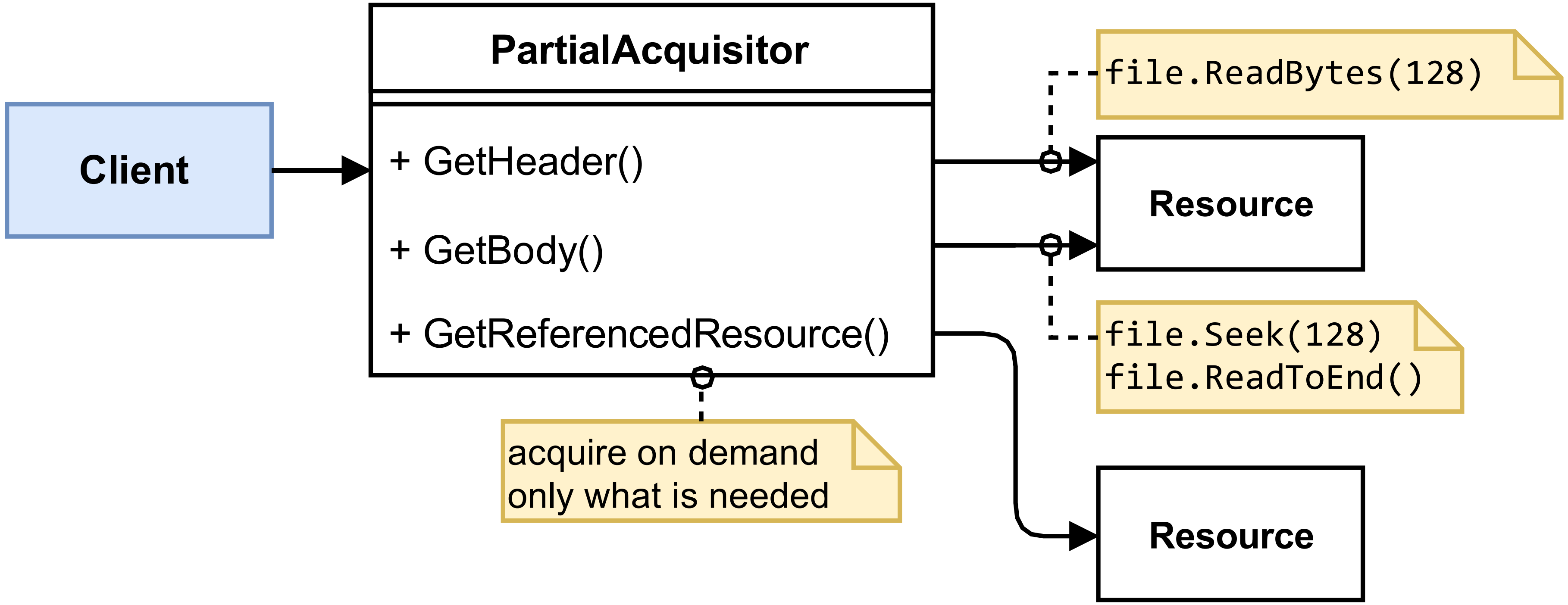
\+ No delay on using the resources.

\- Resources take up memory space.

\- Startup may be slowed down due to loading the resources (do it async!)

Partial Acquisition

Acquire resource in parts. Only use the part which is currently needed.

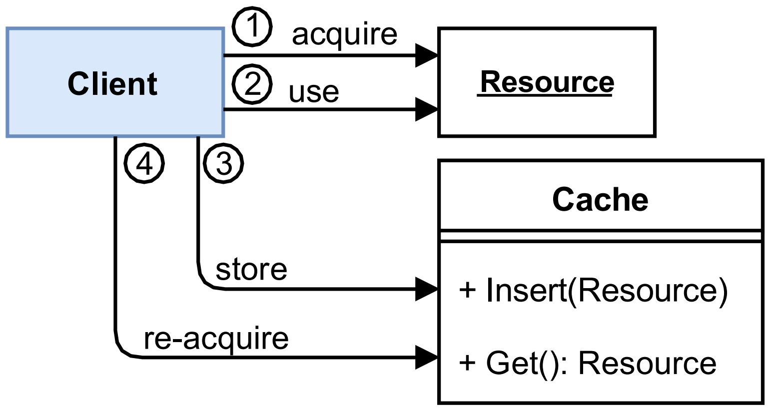


Caching & Pooling

Save resources for later reuse.

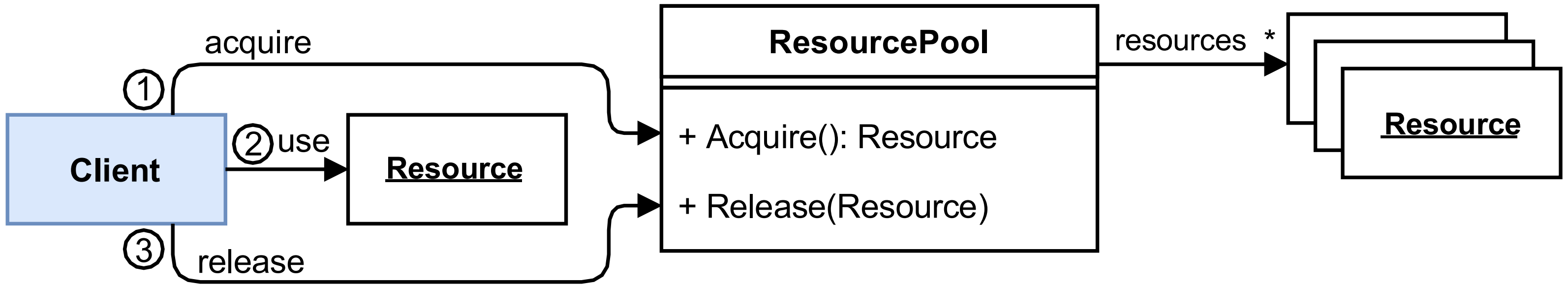
Caching

Let the client decide what to cache



Pooling

Wrap access to the resource in a manager.



Caching & Pooling

Context: Using resources in an application

Problem: How to avoid loading or creating resources over and over?

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- They may take time to load.
- They are needed more than once and in different places.

Solution:

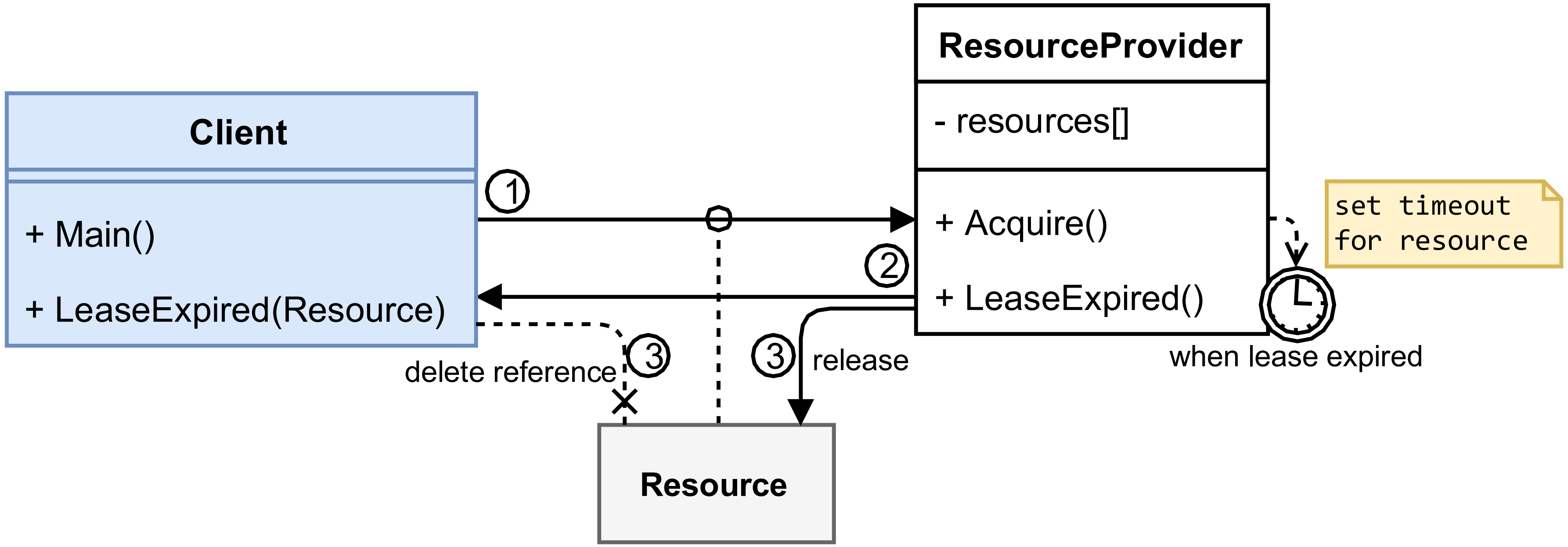
- •Provide a cache to store already loaded resources there (singleton).
- •Supply means to access the cache to the client (factory).
- •Restrict access if needed.

Consequences:

+ Resources are loaded only once and reused afterwards
+ Subsequent usages are much faster
~ Mutual exclusive access for other applications?
- Uses much memory space
- Resources may be outdated

Leasing

Set expire-timeout for resources.



Leasing

Context: Using resources in an application

Problem: How to avoid that resources can be exclusively be used by only one client.

Forces:

- Special resources are needed in an application (Memory, Files, Network).
- The resource may be used by multiple clients.
- It should be avoided that one client can exclusively use a resource forever.

Solution:

- Supply access to the resource via a LeasingProxy which invalidates the resource some time after acquisition.
- Inform the client that the usage time is over.
- Restrict direct access to the resource.

Consequences:

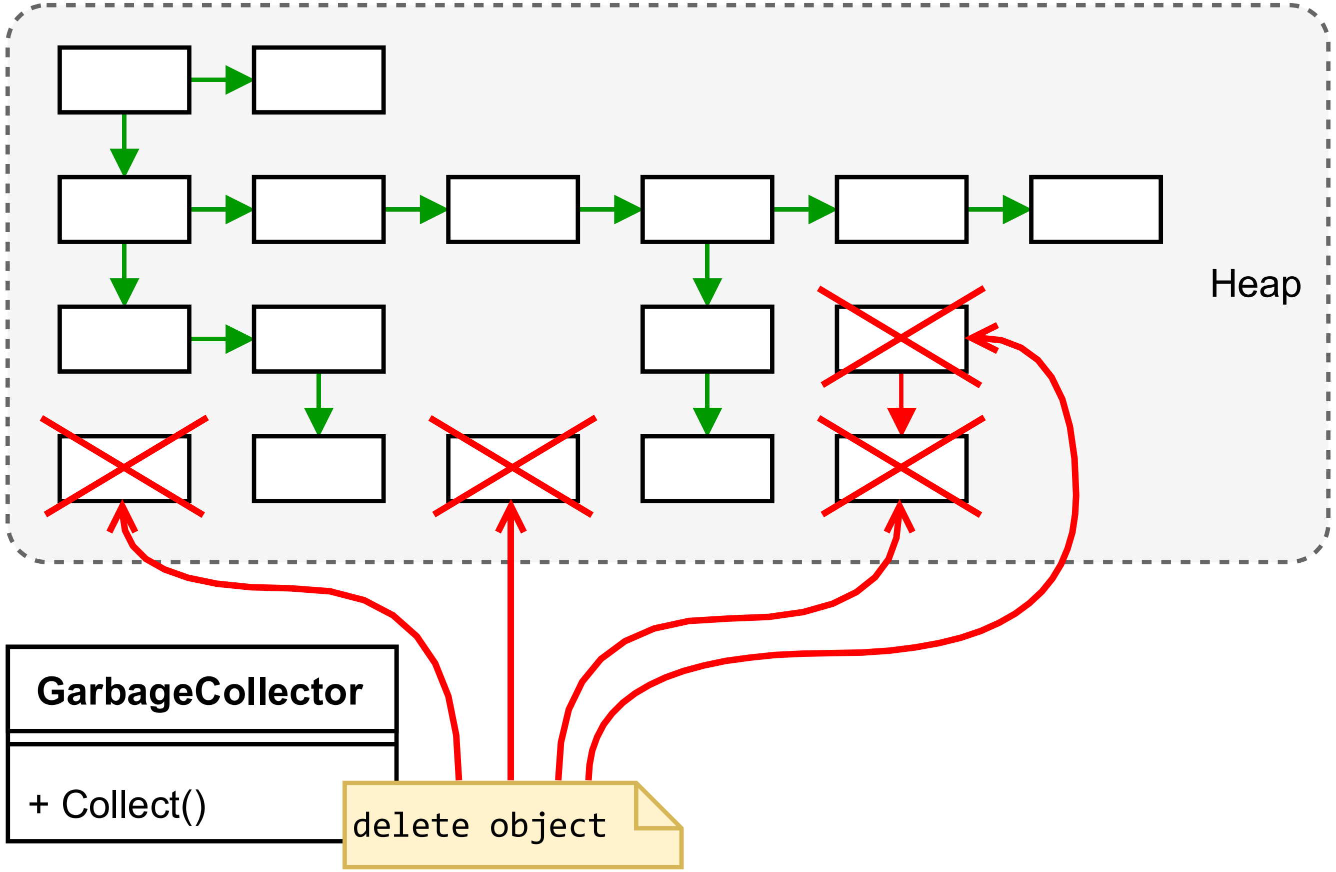
+ Resources cannot be used exclusively anymore
+ If client forgets to release the resource it gets released automatically after some time.

~ What is the right duration?

- To early release could lead to errors.

Garbage Collector

Maintain reference-graph of objects and delete unreachable branches.



Garbage Collector

Context: Application which acquires dynamic memory.

Problem: How to avoid dangling references in an application to avoid memory leaks?

Forces:

- Memory can be dynamically acquired to store objects
- Pointers/References can be freely passed and copied
- Client doesn't want to care about memory allocation.

Solution:

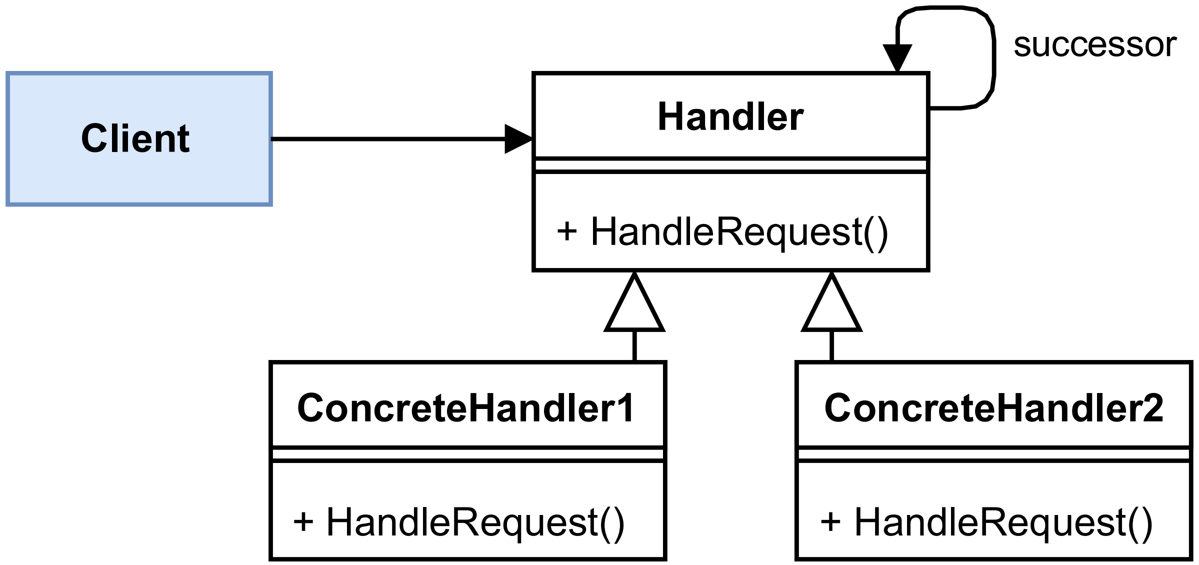
- Maintain reference graph for each and every dynamically created object.
- Periodically search over graph for unreachable branches/subgraphs
- Delete unreachable subgraphs.

Consequences:

+ Client doesn't has to care for manual memory management.
+ No memory leaks
~ How often should collection be done? Performance Optimizations (Generation Concept)?
- Performance overhead during creation and garbage collection (traversal)
- Memory overhead by storing all reference counts

Chain of Responsibility

Forward a call until an object can handle it.



Chain of Responsibility

Context: Having a task or problem which can be handled by several objects.

Problem: How to dynamically resolve which object is responsible for a specific problem/task?

Forces:

- Having different types of tasks which have to be handled.
- Having several objects which can handle different tasks.
- Tasks and the actual Handlers are not known at compile-time.
- There should be multiple escalation levels.

Solution:

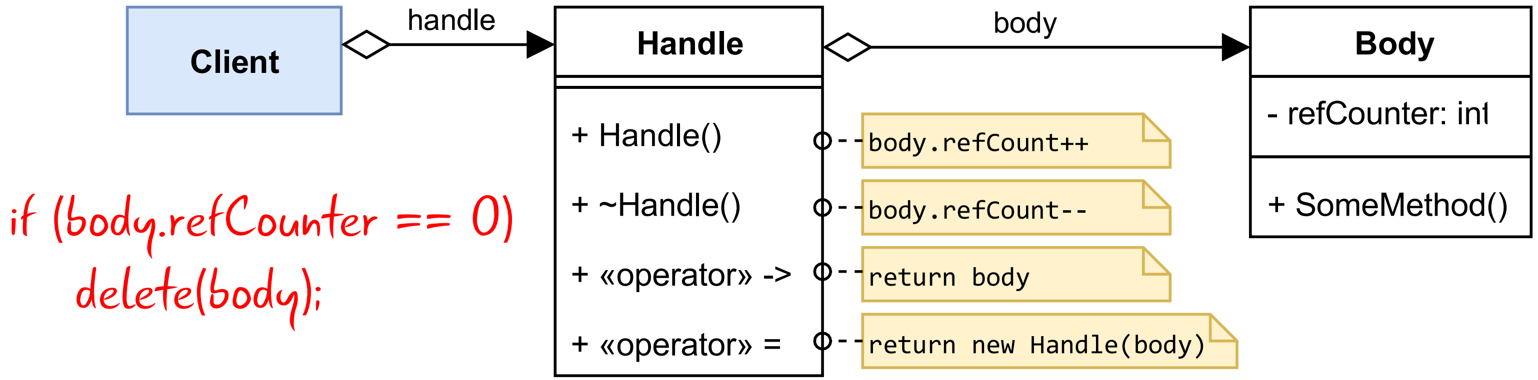
- Implement a chain of handlers.
- Forward the task to the first object which can handle it.
- Add more general handlers in the end of the chain.

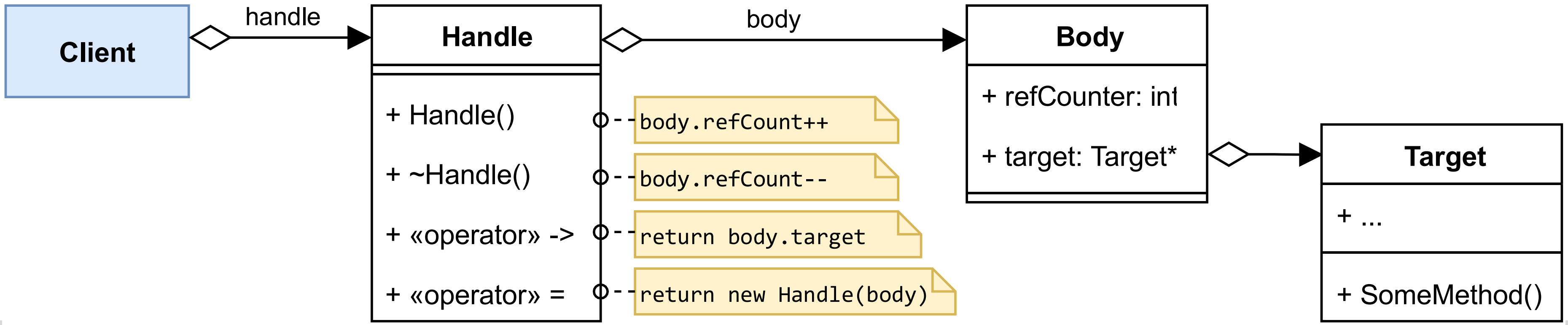
Consequences:

\+ Dynamic handling of events
\+ Loosely coupled responsibility
\+ Can be changed at runtime
~ Who builds the chain?
~ Common standards/conventions?
~ Only one handler or multiple? (decorator-style)
~ Fallbacks?
\- Possible huge call stack
\- Critical path is single point of failure

Counted Pointer / Smart Pointer / Shared Pointer / Auto Pointer

Count references and call destructor when no one is using the object anymore.





(Wrapper Variant)

Counted Pointer

Context: Manual dynamic memory management with pointers.

Problem: When can we safely destroy an object?

Forces:

- If an object is not referenced anymore it should be destroyed (and its memory and resources should be released)
- Several clients may share the same objects
- We don't know exactly who still has a reference to our objects.
- We want avoid dangling references.
- We tend to forget to delete objects (memory leaks).
- It should be "fool-proof" – client should not need to think too much.
- Garbage collectors introduce performance overhead.

Solution:

- Store a counter for the number of references somewhere.
- Implement a proxy which represents a pointer which…
  - … increases the ref-counter in the constructor.
  - … decrease the ref-counter in the destructor, and on reaching 0 it deletes the object.
  - … implements the arrow operator "->" similar to pointers.
  - … returns a new instance on assignment "=" and copy constructor.
- Only allow access to object via the proxy object.

Consequences:

\+ Automatic immediate destruction if object is not referenced anymore

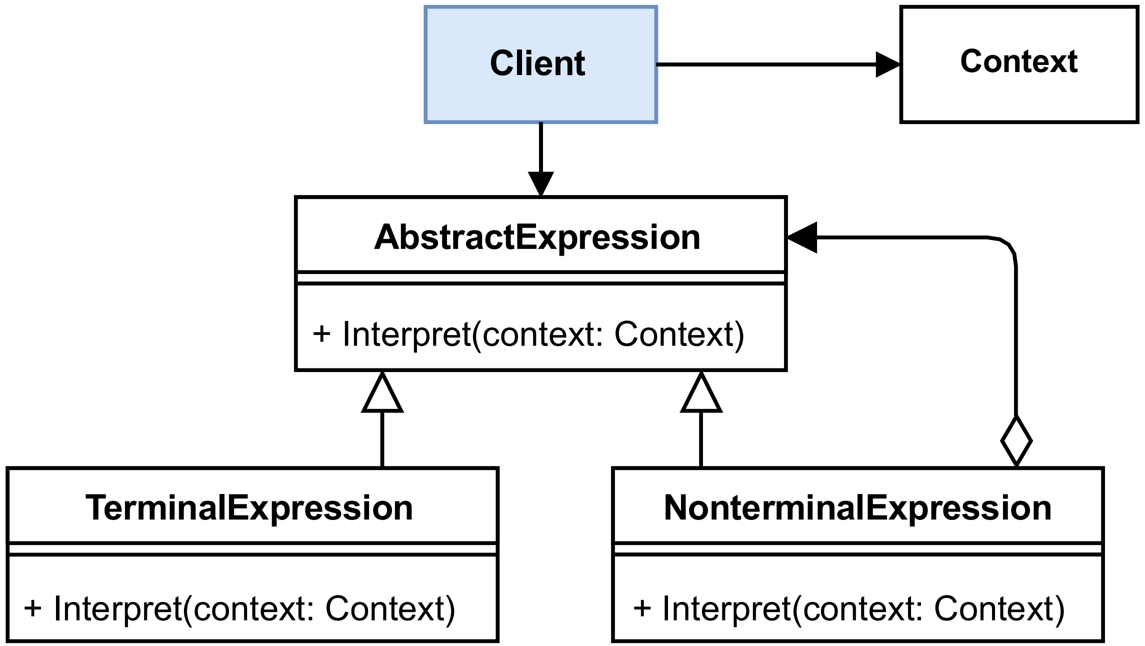
\+ Client does not need to worry about dangling references, or memory leaks.

~ Shared vs. Unique Pointers?

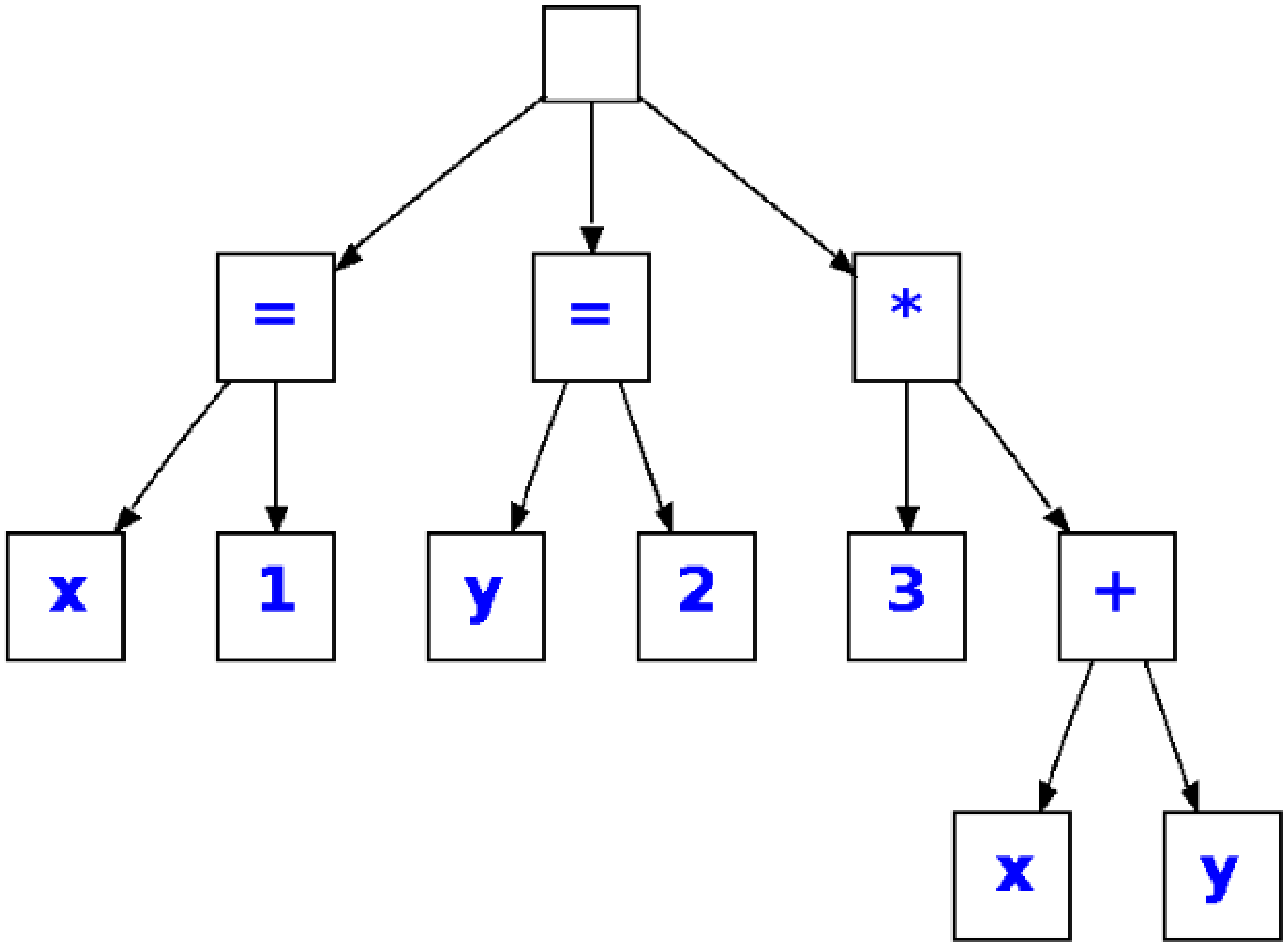
\- Circle references!

Interpreter / Abstract Syntax Tree (AST)

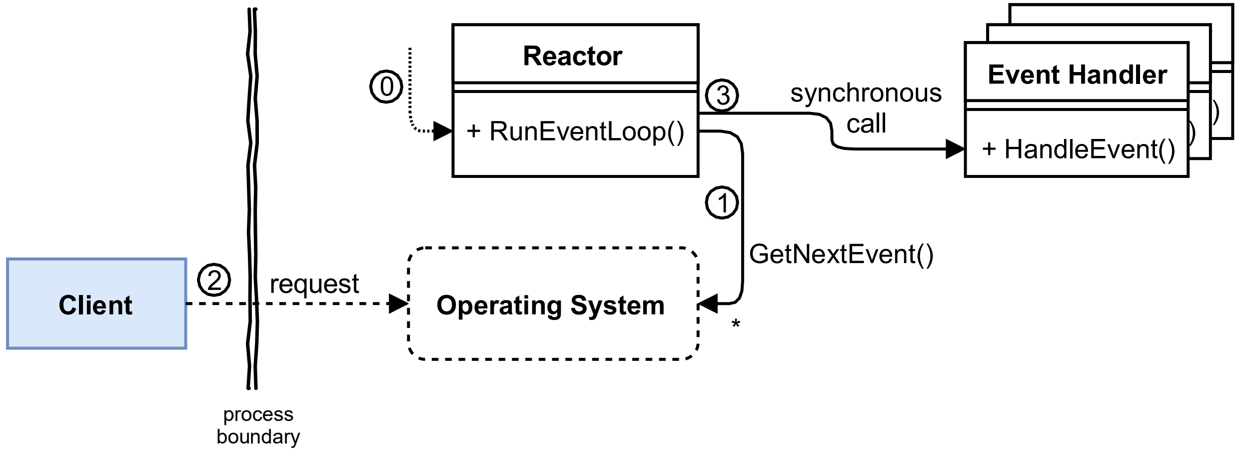
Read expressions one after another and build a tree of expressions.



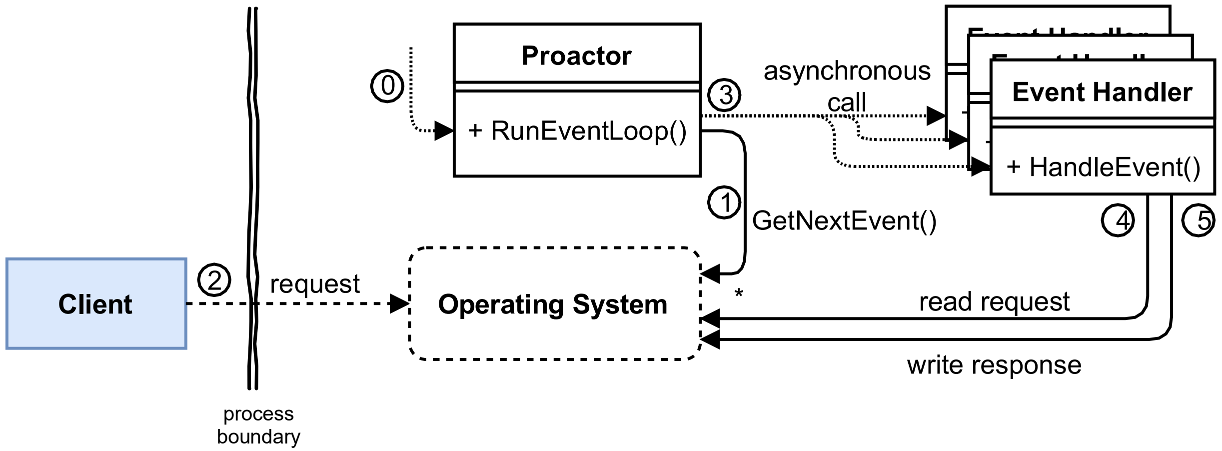
Interpreter / Abstract Syntax Tree (AST)



Reactor (sync)

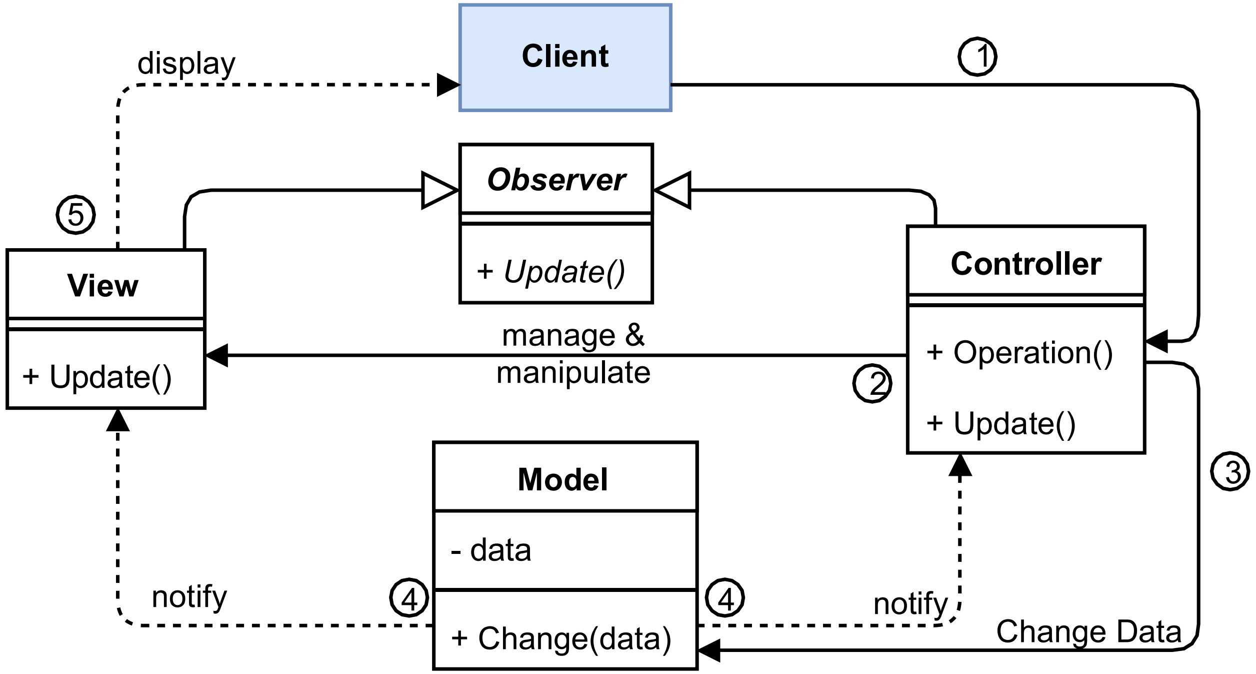


Proactor (async)



Model-View-Controller (MVC)

Separate the responsibilities of visualizing, processing and data management for GUI applications.



MVC / MVP / MVVM

Context: Important dataset that needs to be provided to be processed.

Problem: Tight coupling of data and representation. I want to separate data and representation.

Forces:

- Independent change of data and views
- Separation of concerns
- Different lifecycles / update rates
- Different expertise

Solution:

- Decouple components for data, visualisation, and control
- Dedicated part for representation (view)
- Part for manipulation of data (controller)
- Independent model for storage of data (model)

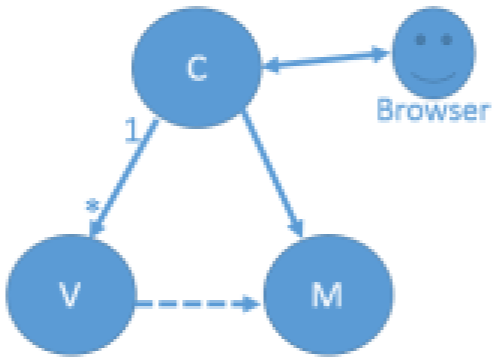
Consequences:

+ Increased reusability of code
+ Separable for different development teams
+ Independence between data and representation (decoupling)
- Complexity increase
- Unit testing more complex

MVC vs. MVP vs. MVVM

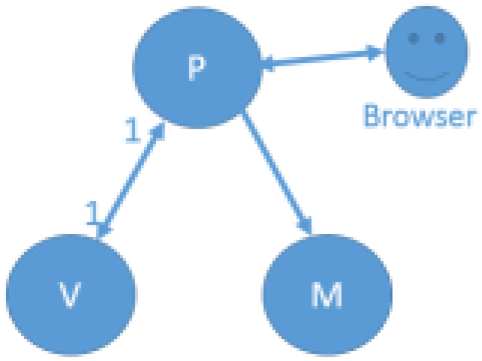
MVC	MVP	MVVM

MVC



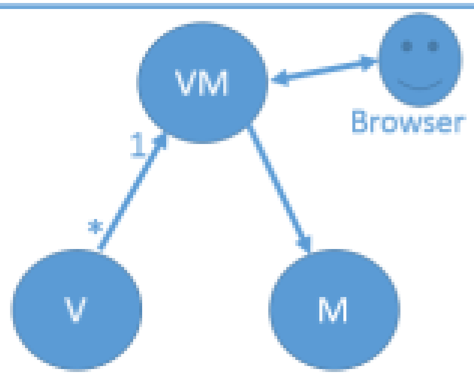
- Controller is the entry point to the application
- One to Many relationship between Controller and View
- View does not have reference to the Controller
- View is very well aware of the Model
- Smalltalk, ASP.Net MVC

MVP



- View is the entry point to the application
- One to One mapping between View and Presenter
- View have the reference to the Presenter
- View is not aware of the Model
- Windows forms

MVVM



- View is the entry point to the application
- One to Many relationship between View and ViewModel
- View have the reference to the View Model
- View is not aware of the Model
- Silverlight, WPF, HTML5 with Knockout/AngularJS

Presentation-Abstraction-Control (PAC)

Decompose GUI generation into smaller agents, each consisting of three parts: presentation, abstraction and control.