GoF Design Pattens

Kayartaya Vinod
https://vinod.co
vinod@vinod.co

OOP

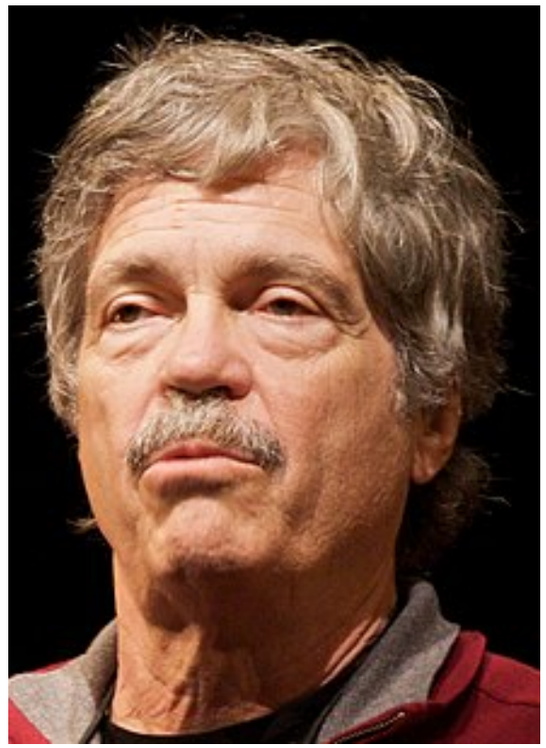
Object Oriented Principles

History of programming

- Low level languages like machine code and assembly languages appeared in the 1940s
- By the end of 1950s, the first popular high level languages appeared.
  - LISP – functional programming language
  - FORTRAN and COBOL – imperative programming languages
- C-family languages have replaced both COBOL and FORTRAN for most applications

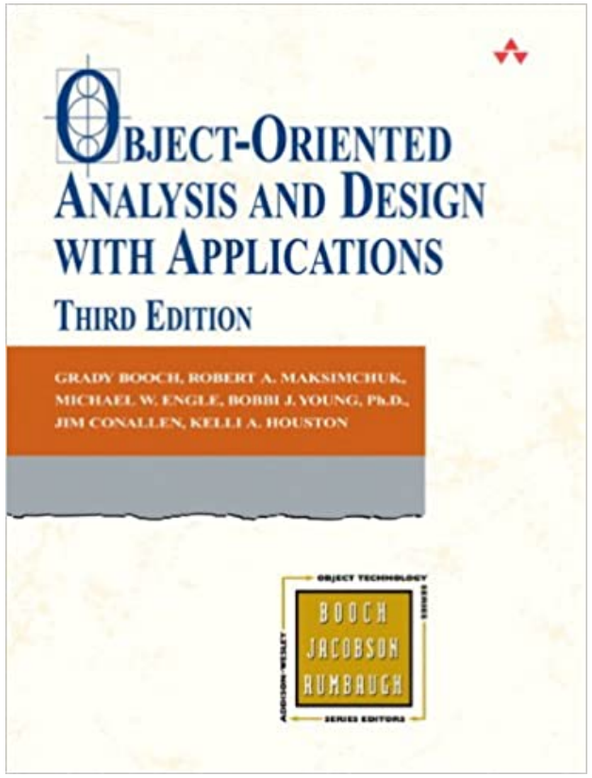
OOP

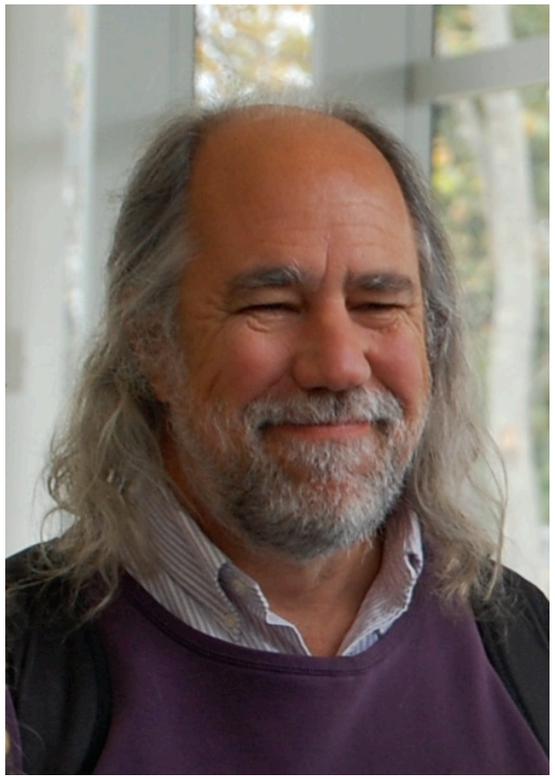
- “Object-Oriented Programming” (OOP) was coined by Alan Kay in 1966.
- The first programming language widely recognized as “Object Oriented” was Simula, specified in 1965.
- Alan Kay developed Smalltalk, which was more object-oriented than Simula.
- According to Alan Kay, the essential ingredients of OOP are: Message passing, Encapsulation and Dynamic binding



OOP according to Grady Booch

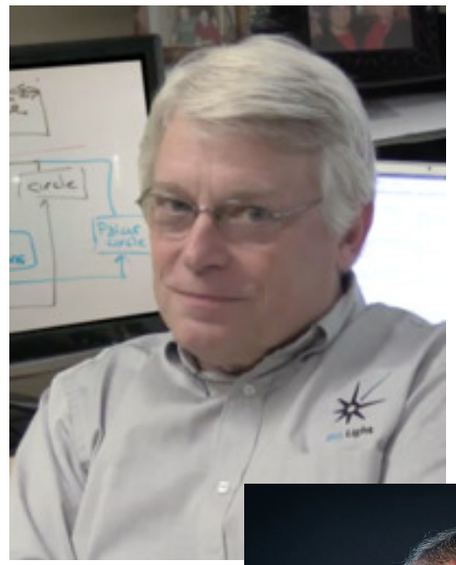
- Major Elements:
  - Abstractions
  - Encapsulation
  - Modularity
  - Hierarchy
- Minor Elements:
  - Typing
  - Persistence
  - Concurrency





SOLID Principles

- Robert C Martin (Uncle Bob) introduced 5 principles in his 2000 paper “Design Principles and Design Patterns”
  - The actual acronym SOLID was, however identified later by Michael Feathers.
- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle



Single Responsibility Principle

- A class must have a single reason to change
- Separation of concerns

Open / Closed Principle

- Software entities must be open for extension but closed for modification

Liskov Substitution Principle

- An object of a sub type must be substitutable to an object of its base type

Interface Segregation Principle

- The dependency of one class to another should depend on the smallest possible interface.
  - Clients should not be forced to implement interfaces they don’t use.
  - Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

Dependency Inversion Principle

- Depend upon abstractions (interfaces) rather than concrete classes.
- Loose coupling

Introduction to Design Patterns

In general, GoF patterns, classifications, complexity, and Usage in C++

Really, what are they?

- Design patterns are typical solutions to common problems in software design
- Each pattern is like a blueprint that you can customize to solve a particular design problem in your application

What are they not?

- You can’t just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries.
- The pattern is not a specific piece of code, but a general concept for solving a particular problem.
- You can follow the pattern details and implement a solution that suits the realities of your own program.
- Patterns are not ALGORITHMS!!

Algorithms

- An algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution.
- An analogy to an algorithm is a cooking recipe:
  - Both have clear steps to achieve a goal

Patterns (design)

- Code of the same pattern applied to two different programs may be different.
- A pattern is more like a blueprint:
  - You can see what the result and its features are, but the exact order of implementation is up to you

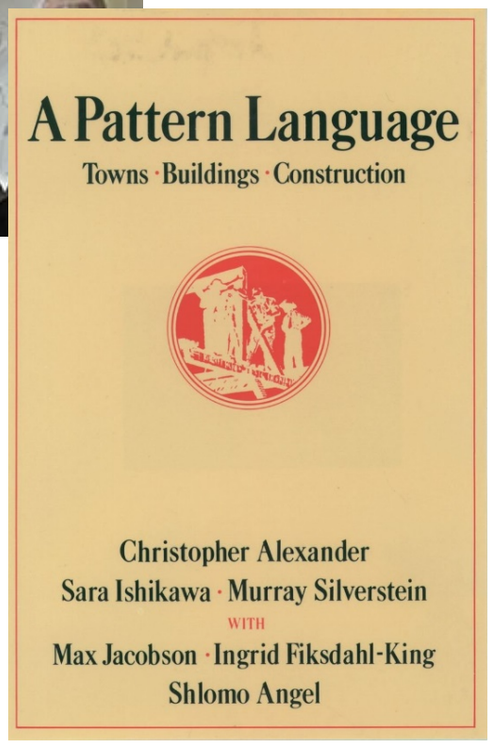
What does the pattern consist of?

- **Intent** of the pattern briefly describes both the problem and solution
- **Motivation** further explains the problem and the solution the pattern makes possible
- **Structure** of classes shows each part of the pattern and how they are related
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern

A bit of history…

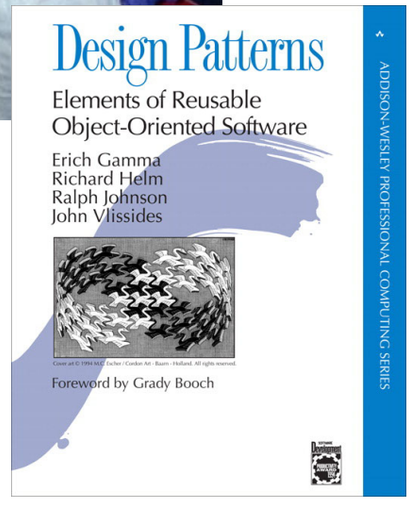
The concept of patterns was first described by

- Christopher Alexander
- “A Pattern language: Towns, Buildings, Construction”
- The book describes a “language” for designing the urban environment.
- The units of this language are patterns.



The idea was picked up by

- Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm.
- Book was published in 1994.
- Features 23 patterns.
- Due to its lengthy name, people started to call it “The book by the gang of four”, which was soon shortened to simply “the GoF book”.



Since then…

- Dozens of other object-oriented patterns have been discovered.
- The “pattern approach” became very popular in other programming fields, so lots of other patterns now exist outside of object-oriented design as well.

Why learn design patterns?

- Design patterns are a toolkit of **tried and tested solutions** to common problems in software design.
- Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.

Why learn design patterns?

- Design patterns define a common language that you and your teammates can use to communicate more efficiently.
- You can say, “Oh, just use the strategy pattern for that”, and everyone will understand the idea behind your suggestion.
  - No need to explain what a strategy pattern is if you know the pattern and its name.

Classification of patterns

- Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed.
- The most basic and low-level patterns are often called idioms.
  - They usually apply only to a single programming language
- The most universal and high-level patterns are architectural patterns.
  - Developers can implement these patterns in virtually any language.
  - Unlike other patterns, they can be used to design the architecture of an entire application.

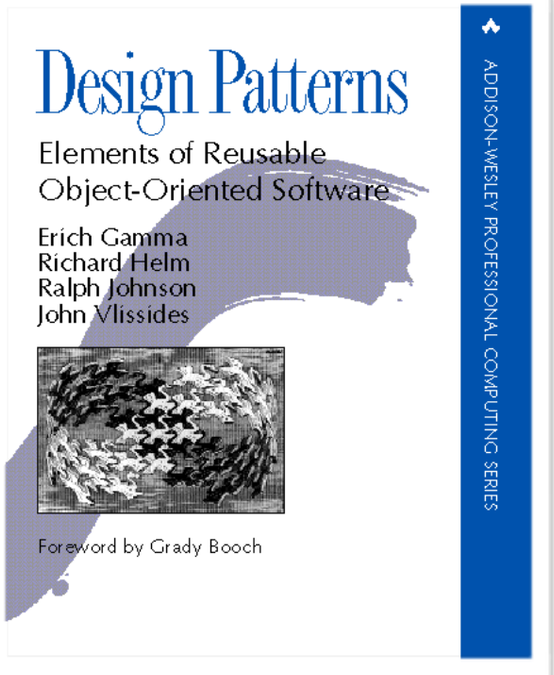
Classification by GoF

- Based on their intent, or purpose, the patterns are grouped into:
- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Gang of Four

- Design patterns gained popularity in computer science after the book "Design Patterns: Elements of Reusable Object-Oriented Software" was published in 1994 by GOF - “Gangs Of Four”

- Ralph Johnson, Erich Gamma, Richard Helm and John Vlissides



Broad categories

- Creational Design Patterns
    - Provide solution to instantiate an object in the best possible way for specific situations.
- Structural Design Patterns
    - Provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.
- Behavioural Design Patterns
    - Provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

Creational Design Patterns

- Singleton
    - A class of which only a single instance can exist
- Factory method
    - Creates an instance of several derived classes
- Abstract Factory
    - Creates instances of several families of classes
- Builder
    - Separates object construction from its representation
- Prototype
    - A fully initialized instance to be copied or cloned

Structural Design Patterns

- Adapter
  - Match interfaces of different classes
- Composite
  - A tree structure of simple and composite objects
- Proxy
  - An object representing another object
- Flyweight
  - A fine-grained instance used for efficient sharing

Structural Design Patterns

- Facade
    - A single class that represents an entire subsystem
- Bridge
    - Separates an object's interface from its implementation
- Decorator
    - Add responsibilities to objects dynamically

Behavioural Design Patterns

- Template Method
    - Defer the exact steps of an algorithm to a subclass
- Mediator
    - Defines simplified communication between classes
- Chain of Responsibility
    - A way of passing a request between a chain of objects
- Observer
    - A way of notifying change to a number of classes

Behavioural Design Patterns

- Strategy
    - Encapsulates an algorithm inside a class
- Command
    - Encapsulate a command request as an object
- State
    - Alter an object's behavior when its state changes
- Visitor
    - Defines a new operation to a class without change

Behavioural Design Patterns

- Iterator
    - Sequentially access the elements of a collection
- Interpreter
    - A way to include language elements in a program
- Memento
    - Capture and restore an object's internal state

Singleton

CREATIONAL PATTERN

Singleton

- Intent:
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated “just-in-time initialization” or “initialization on first use”

Singleton

- Problem:
    - Application needs one, and only one, instance of an object.
    - Additionally, lazy initialization and global access are necessary

Singleton

- Make the class of the single instance object itself, responsible for creation, initialization, access, and enforcement.

- Declare the instance as a private static data member.

- Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

- The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton

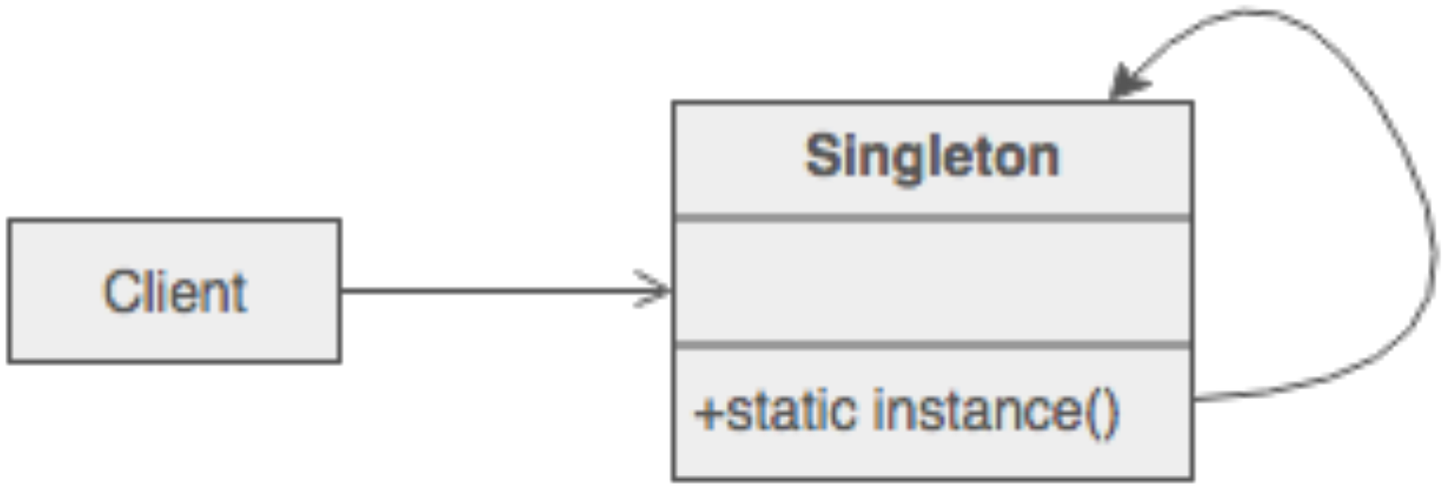
- Singleton should be considered only if all three of the following criteria are satisfied:
    - Ownership of the single instance cannot be reasonably assigned
    - Lazy initialization is desirable
    - Global access is not otherwise provided for

Singleton

- The Singleton pattern can be extended to support access to an application-specific number of instances.
- The "static member function accessor" approach will not support subclassing of the Singleton class.
- Deleting a Singleton class/instance is a non-trivial design problem.

Singleton

- Structure:



Singleton

- Checklist:

    1. Define a private static attribute in the "single instance" class.

    2. Define a public static accessor function in the class.

    3. Do "lazy initialization" (creation on first use) in the accessor function.

    4. Define all constructors to be protected or private.

    5. Clients may only use the accessor function to manipulate the Singleton.

Singleton

- Reflection:
    - Using reflection we can set the private constructor to become accessible at runtime
- How to fix?
    - Throw RuntimeException from the private constructor, if someone tries to make instance in case one instance already exists.

Singleton

- Holder Class (Bill Pugh Method):
    - Bill Pugh came up with a different approach to create the Singleton class using an inner static helper class.

```
public class Singleton {

    private Singleton() {
    }

    static class Holder {
        private static final Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {
        return Holder.instance;
    }
```

```
public static void main(String[] args) {
    Singleton s1 = Singleton.getInstance();
    Singleton s2 = Singleton.getInstance();
    Singleton s3 = Singleton.getInstance();

    System.out.println("s1==s2 is " + (s1 == s2));
    System.out.println("s2==s3 is " + (s2 == s3));
}
```

Factory Method Pattern

Factory method

- Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-classes.

- This pattern takes out the responsibility of instantiation of a class from client program to the factory class.

- We can apply Singleton pattern on Factory class or make the factory method static.

Factory method

- Intent:
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.
  - Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The new operator considered harmful.

Factory method

- Problem:
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Factory method

- A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Builder Pattern

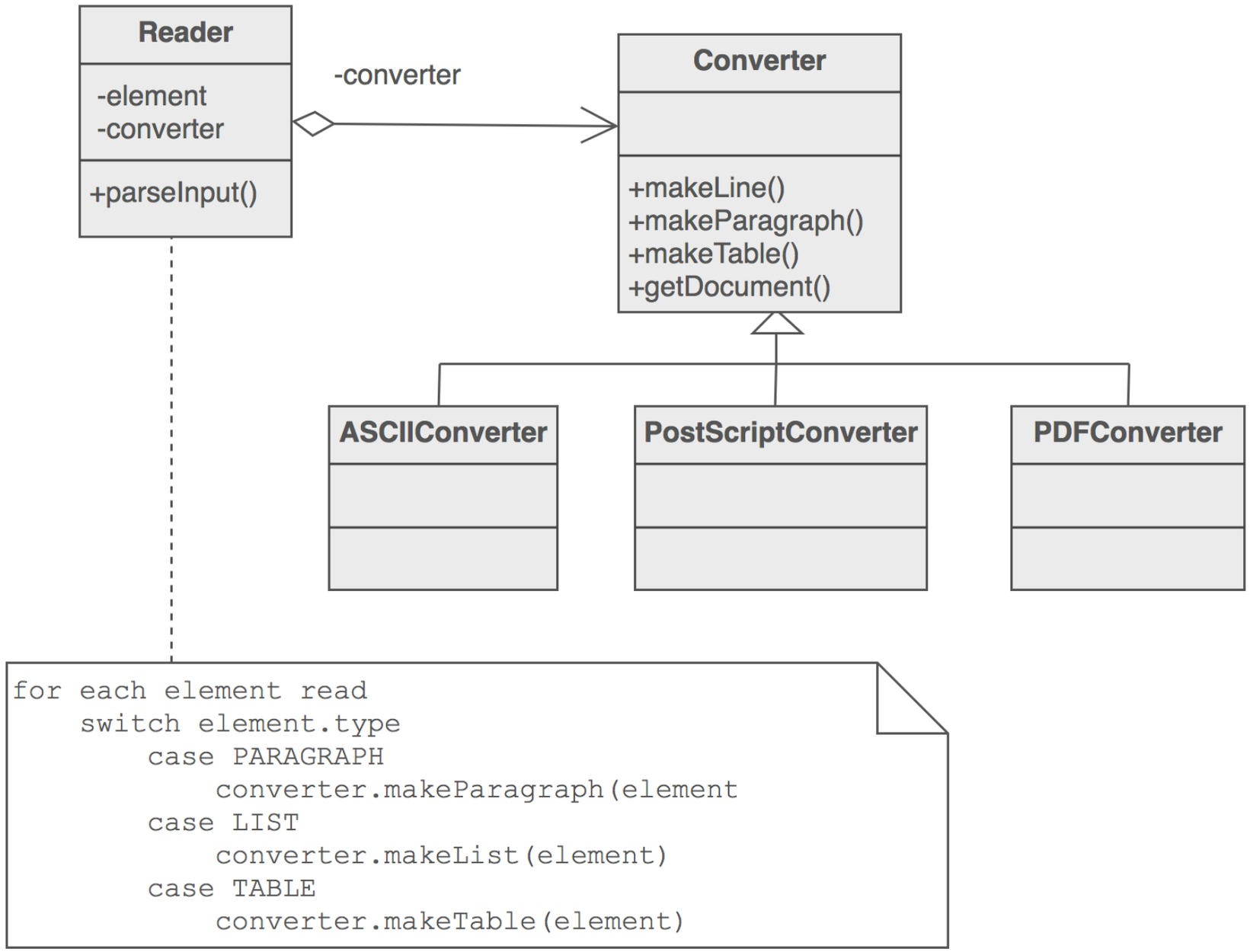
Builder Design Pattern

- Intent
    - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
    - Parse a complex representation, create one of several targets.

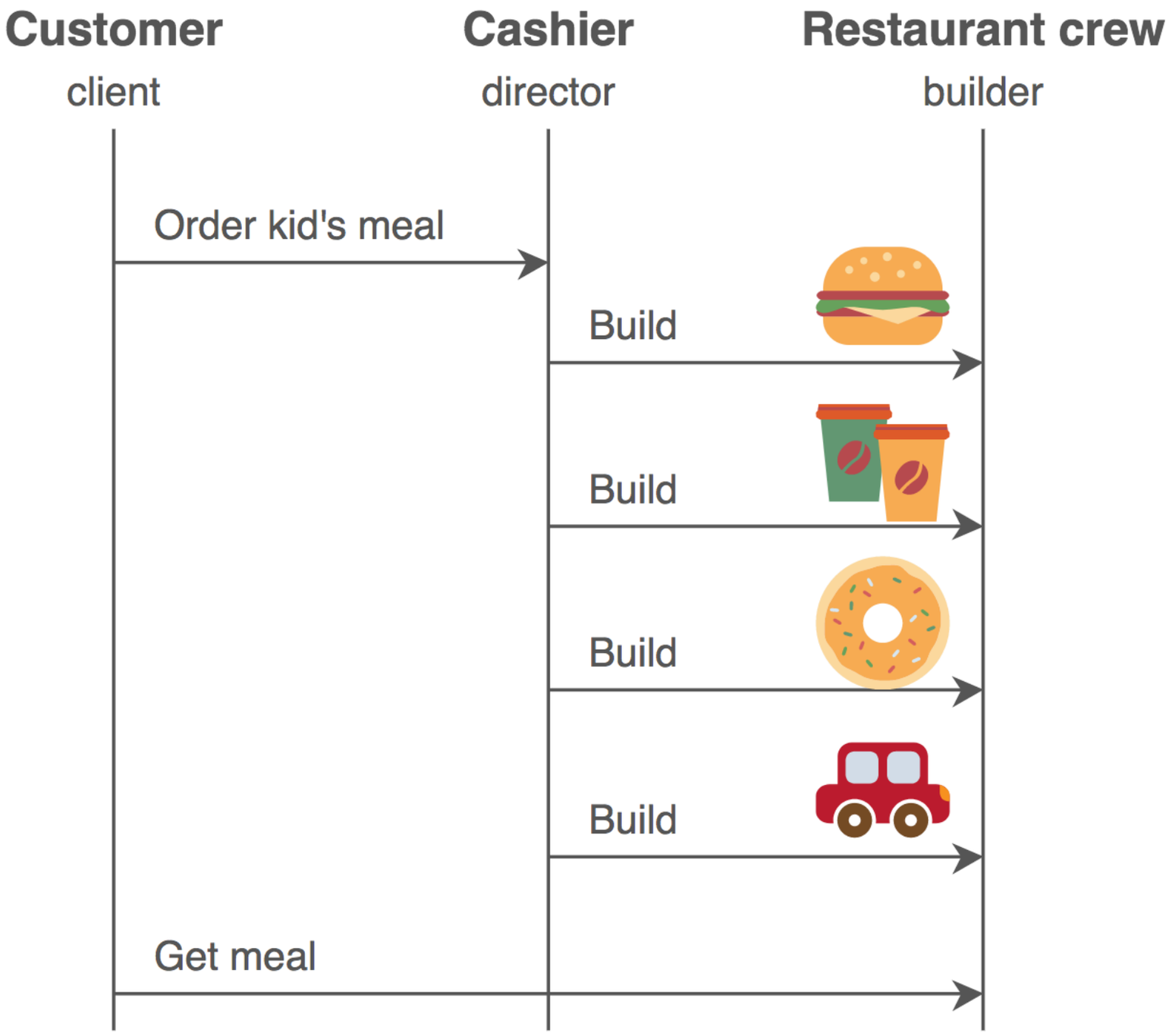
Builder Design Pattern

- Problem
  - An application needs to create the elements of a complex aggregate.
  - The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Builder Design Pattern



Builder Design Pattern



Composite Pattern

Composite Design Pattern

- Intent

  - Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

  - Recursive composition

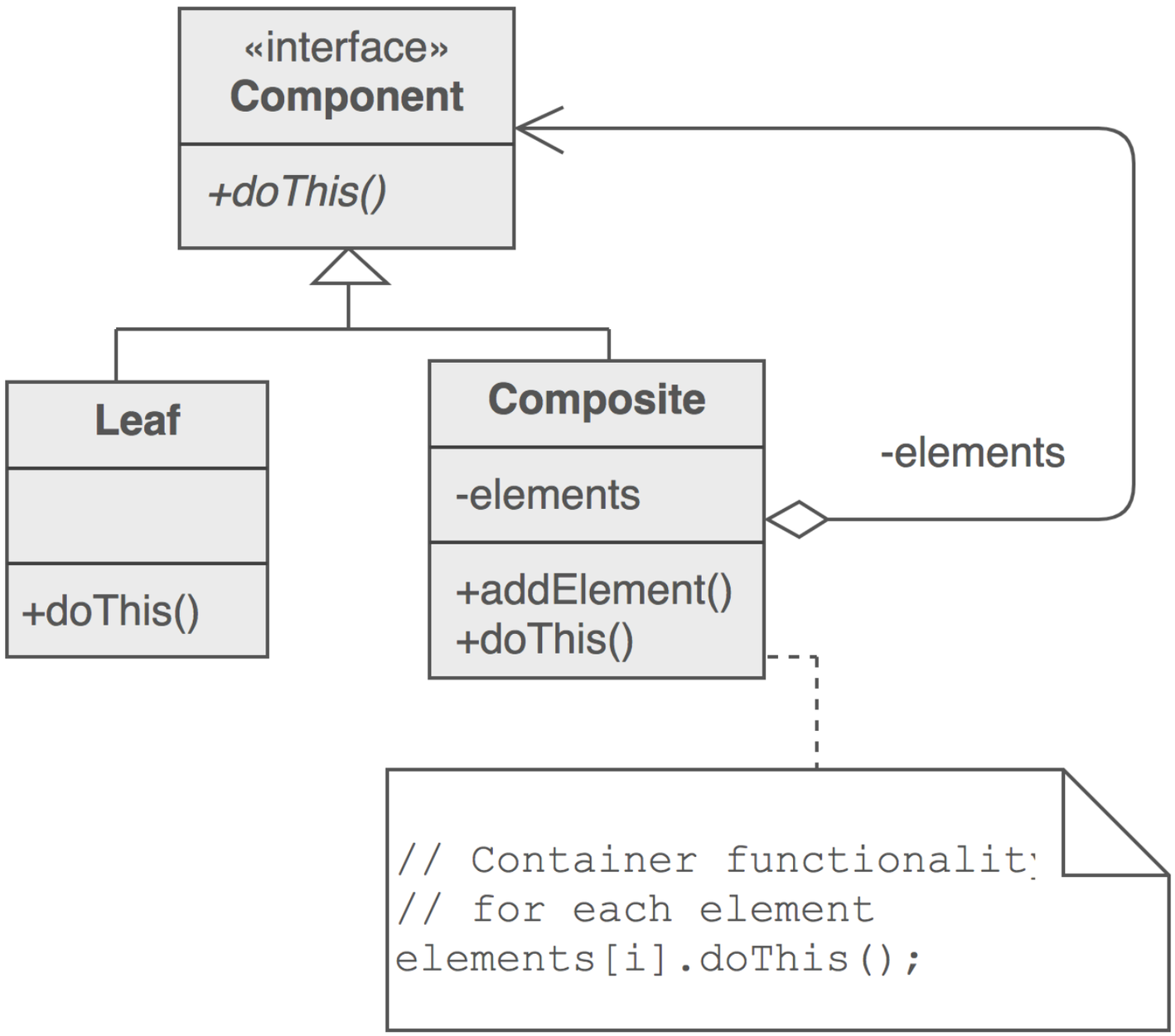
  - "Directories contain entries, each of which could be a directory."

  - 1-to-many "has a" up the "is a" hierarchy

Composite Design Pattern

- Problem
    - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects.
    - Processing of a primitive object is handled one way, and processing of a composite object is handled differently.
    - Having to query the "type" of each object before attempting to process it is not desirable.

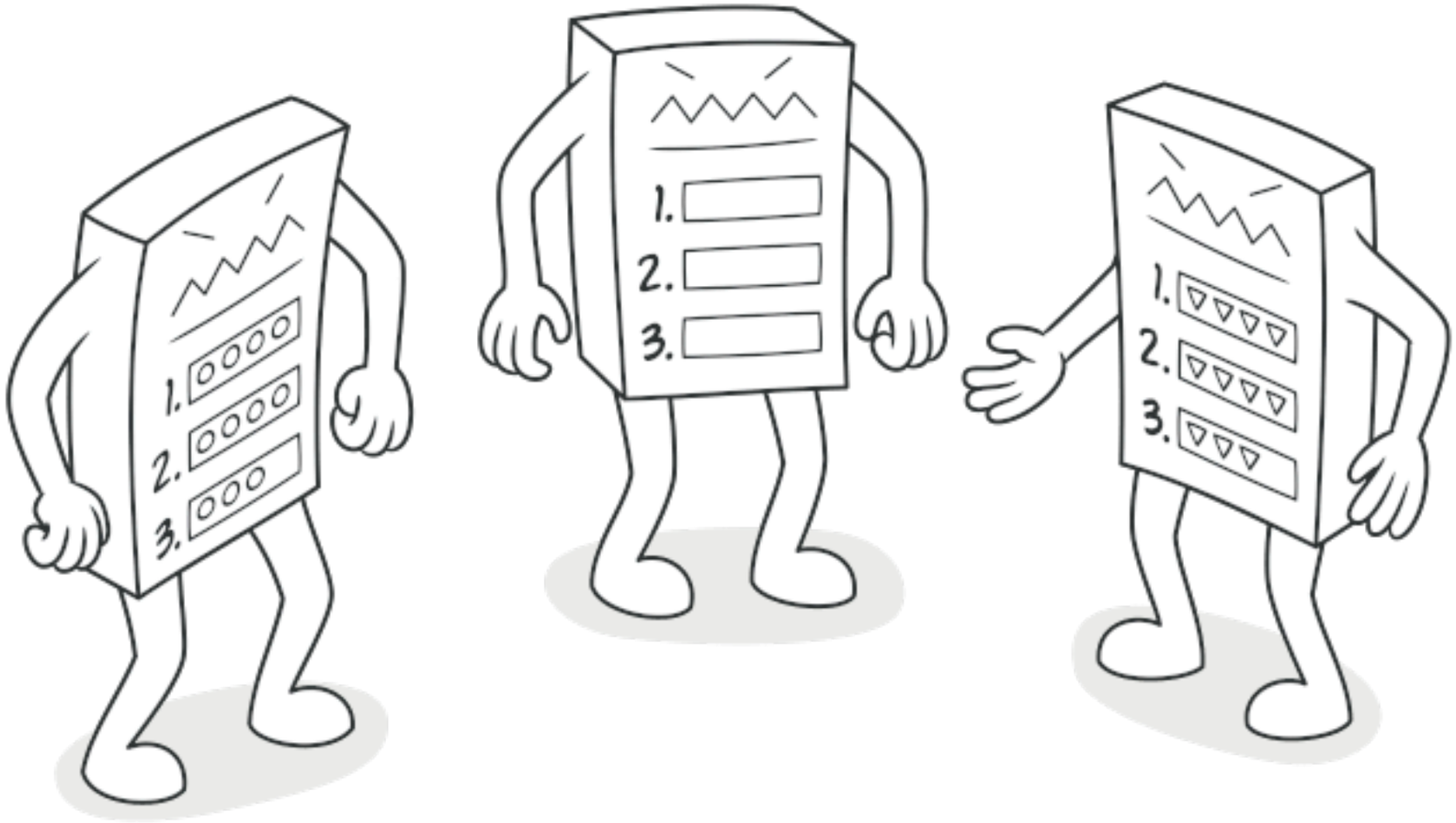
Composite Design Pattern



Template Method Pattern

Template method pattern

- Template Method is a behavioural design pattern that **defines the skeleton of an algorithm** in the superclass but lets **subclasses override specific steps** of the algorithm without changing its structure.



Intent: from GoF book of patterns

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Problem

- Imagine that you're creating a data mining application that analyzes corporate documents.

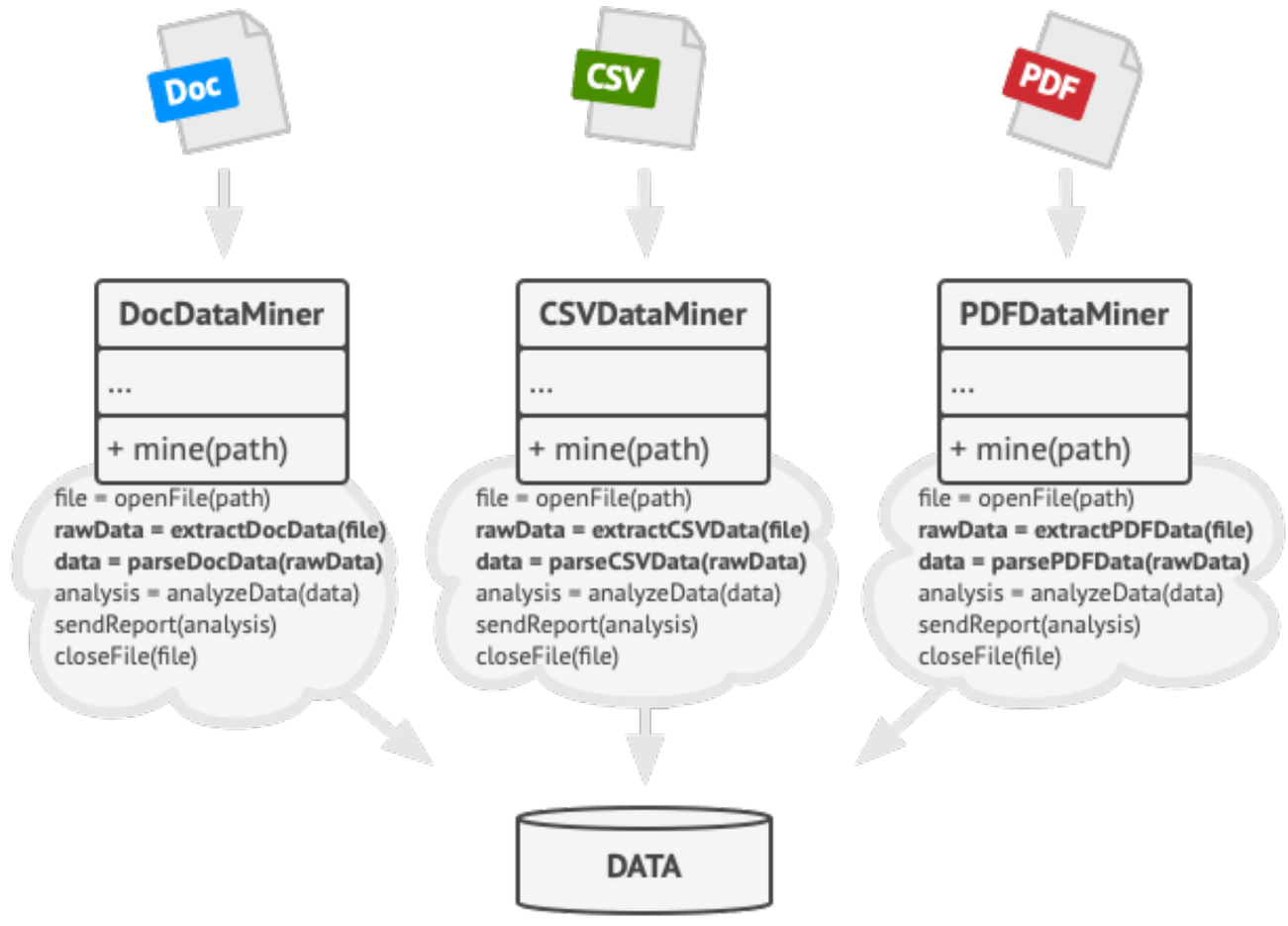
- Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.

Problem

- The first version of the app could work only with DOC files.

- In the following version, it was able to support CSV files.

- A month later, you “taught” it to extract data from PDF files.

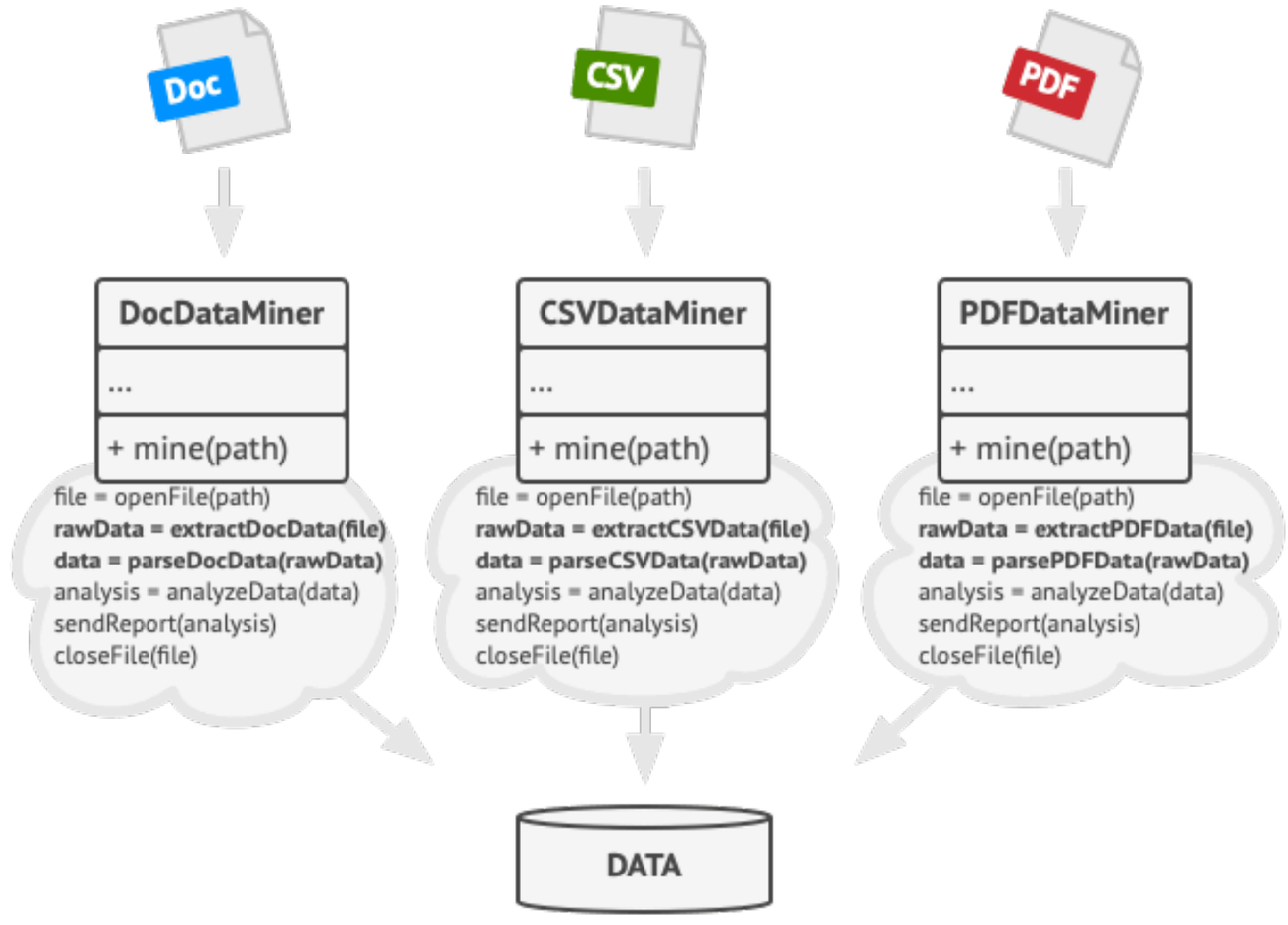


Problem

- At some point, you noticed that all three classes have a lot of similar code.

- While the code for dealing with various data formats was entirely different in all classes, the code for data processing and analysis is almost identical.

- Wouldn’t it be great to get rid of the code duplication, leaving the algorithm structure intact?

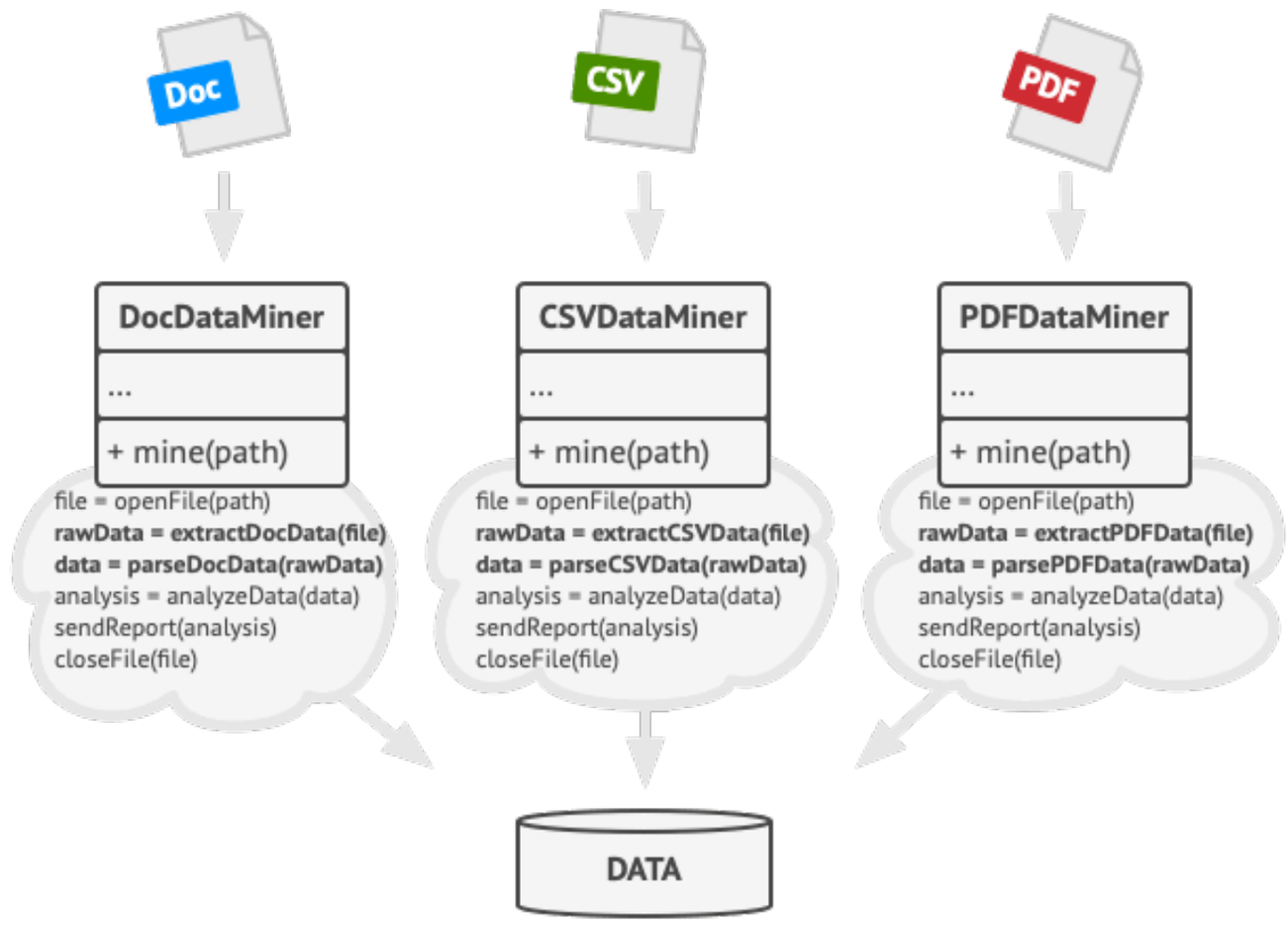


Problem

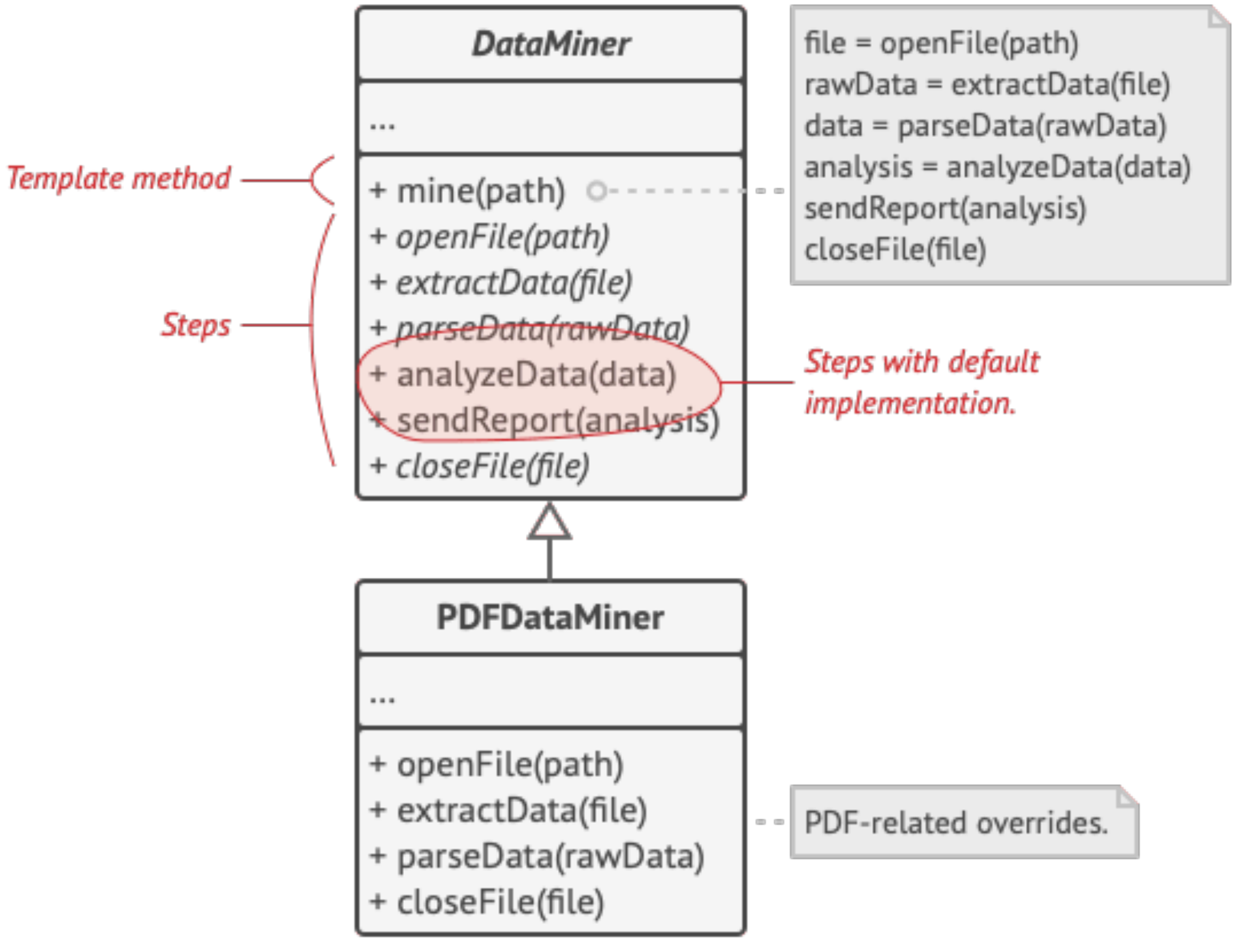
- There was another problem related to client code that used these classes.

- It had lots of conditionals that picked a proper course of action depending on the class of the processing object.

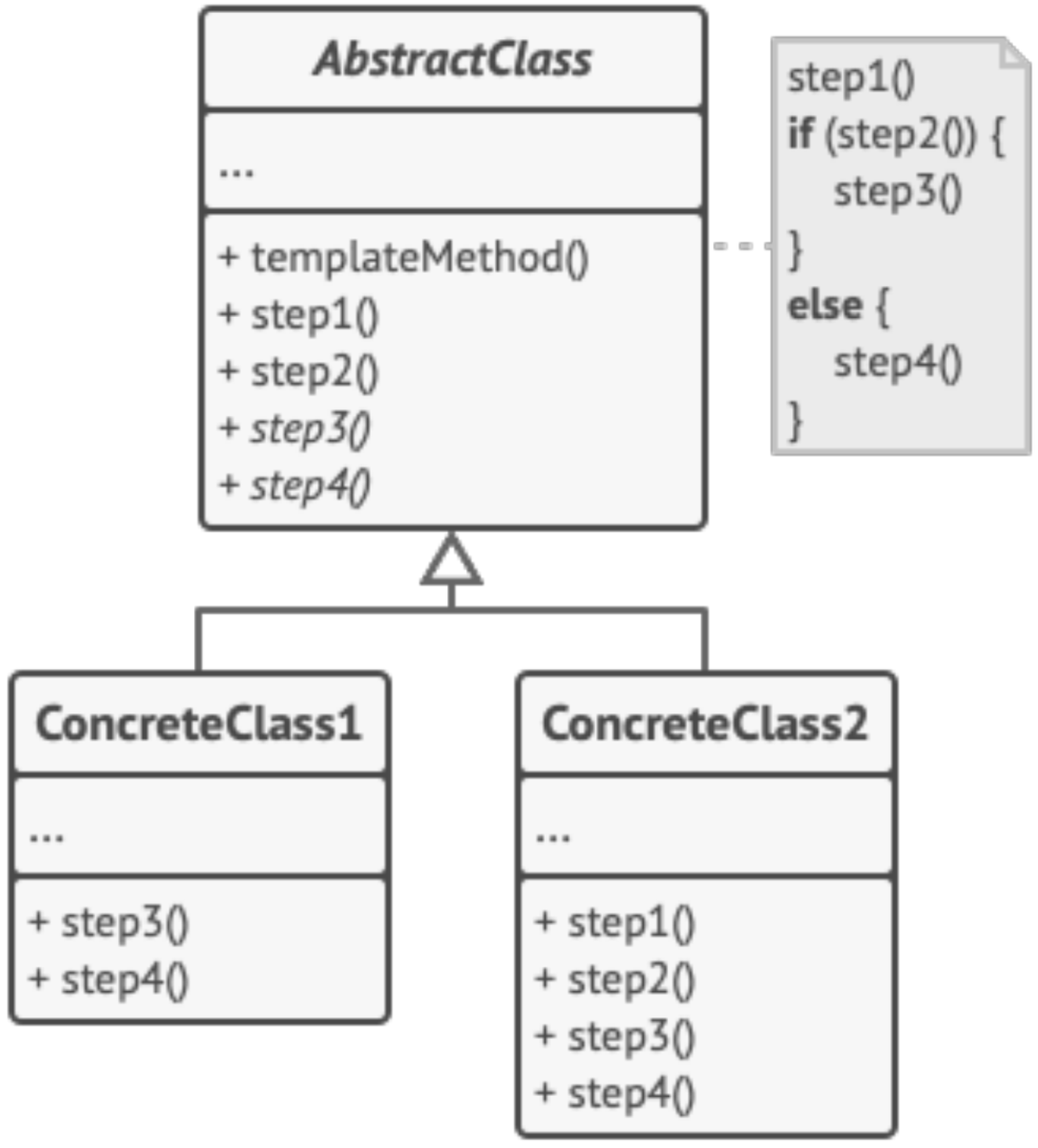
- If all three processing classes had a common interface or a base class, you'd be able to eliminate the conditionals in client code and use polymorphism when calling methods on a processing object.



Solution - Template method



Structure - Template method



Applicability

- Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.

- The Template Method lets you turn a monolithic algorithm into a series of individual steps which can be easily extended by subclasses while keeping intact the structure defined in a superclass.

Applicability

- Use the pattern when you have several classes that contain almost identical algorithms with some minor differences.
    - As a result, you might need to modify all classes when the algorithm changes.
- When you turn such an algorithm into a template method, you can also pull up the steps with similar implementations into a superclass, eliminating code duplication.
    - Code that varies between subclasses can remain in subclasses.

Pros/Cons

- You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
- You can pull the duplicate code into a superclass.
- Some clients may be limited by the provided skeleton of an algorithm.
- You might violate the Liskov Substitution Principle by suppressing a default step implementation via a subclass.
- Template methods tend to be harder to maintain the more steps they have.

Strategy Pattern

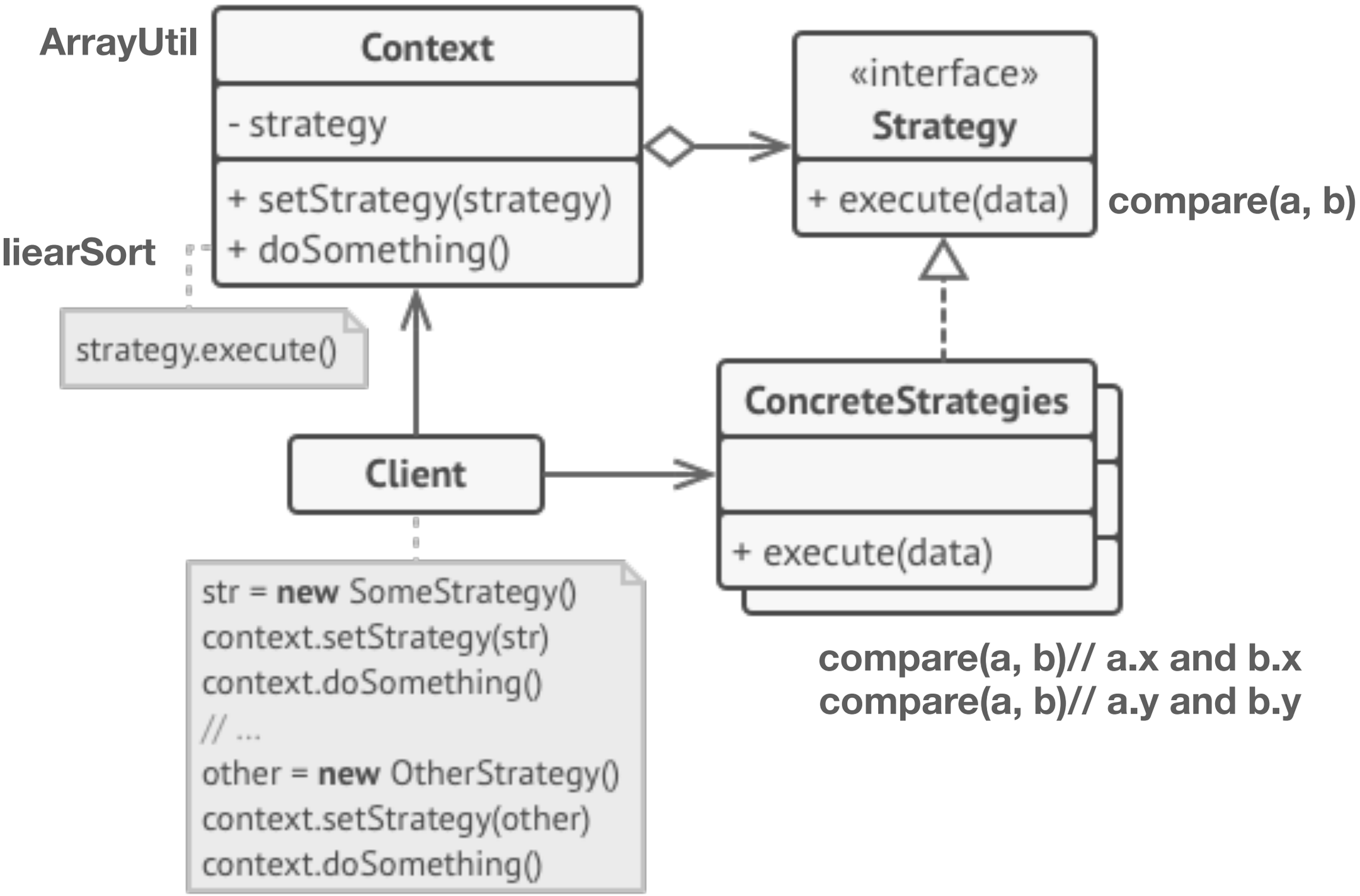
A.K.A. Policy

Intent (GoF book):

Define a family of algorithms, encapsulate each one of them, and make them interchangeable.

Strategy lets the algorithms vary independently from clients that use it.

Structure


ArrayUtil
Context
- strategy
+ setStrategy(strategy)
+ doSomething()
«interface»
Strategy
+ execute(data)
compare(a, b)
liearSort
strategy.execute()
ConcreteStrategies
+ execute(data)
Client
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
compare(a, b)// a.x and b.x
compare(a, b)// a.y and b.y

Applicability

- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.

- The Strategy pattern lets you indirectly alter the object's behaviour at runtime by associating it with different sub-objects which can perform specific sub-tasks in different ways.

Applicability

- Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behaviour.

- The Strategy pattern lets you extract the varying behaviour into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.

Applicability

- Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.

- The Strategy pattern lets you isolate the code, internal data, and dependencies of various algorithms from the rest of the code. Various clients get a simple interface to execute the algorithms and switch them at runtime.

Decorator

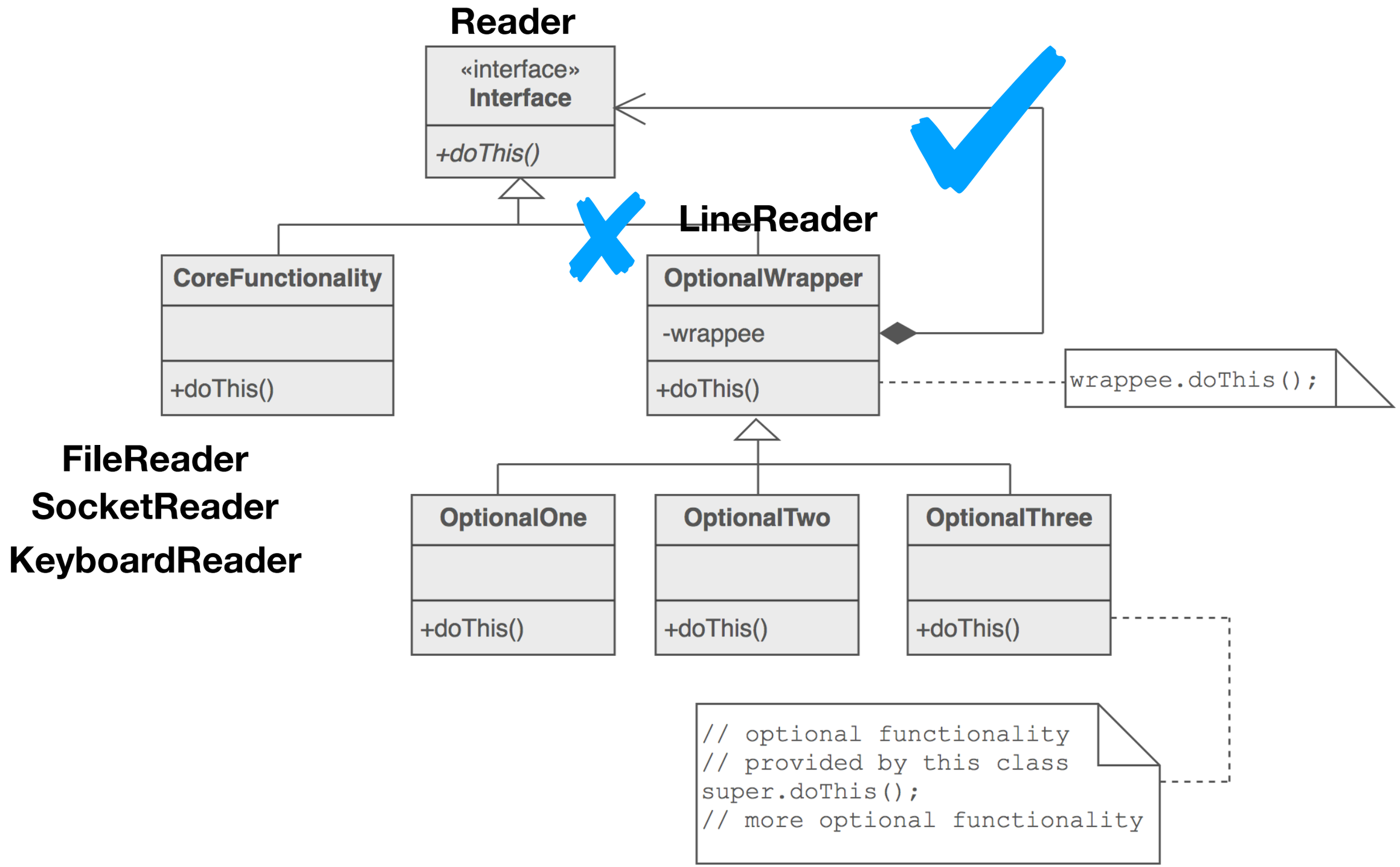
Decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
  - Client-specified embellishment of a core object by recursively wrapping it.
  - Wrapping a gift, putting it in a box, and wrapping the box.

Decorator

- Problem
    - You want to add behavior or state to individual objects at run-time.
    - Inheritance is not feasible because it is static and applies to an entire class.

Decorator


Reader
«interface»
Interface
+doThis()
LineReader
CoreFunctionality
+doThis()
OptionalWrapper
-wrappee
+doThis()
wrappee.doThis();
FileReader
SocketReader
KeyboardReader
OptionalOne
+doThis()
OptionalTwo
+doThis()
OptionalThree
+doThis()
// optional functionality
// provided by this class
super.doThis();
// more optional functionality

Adapter Pattern

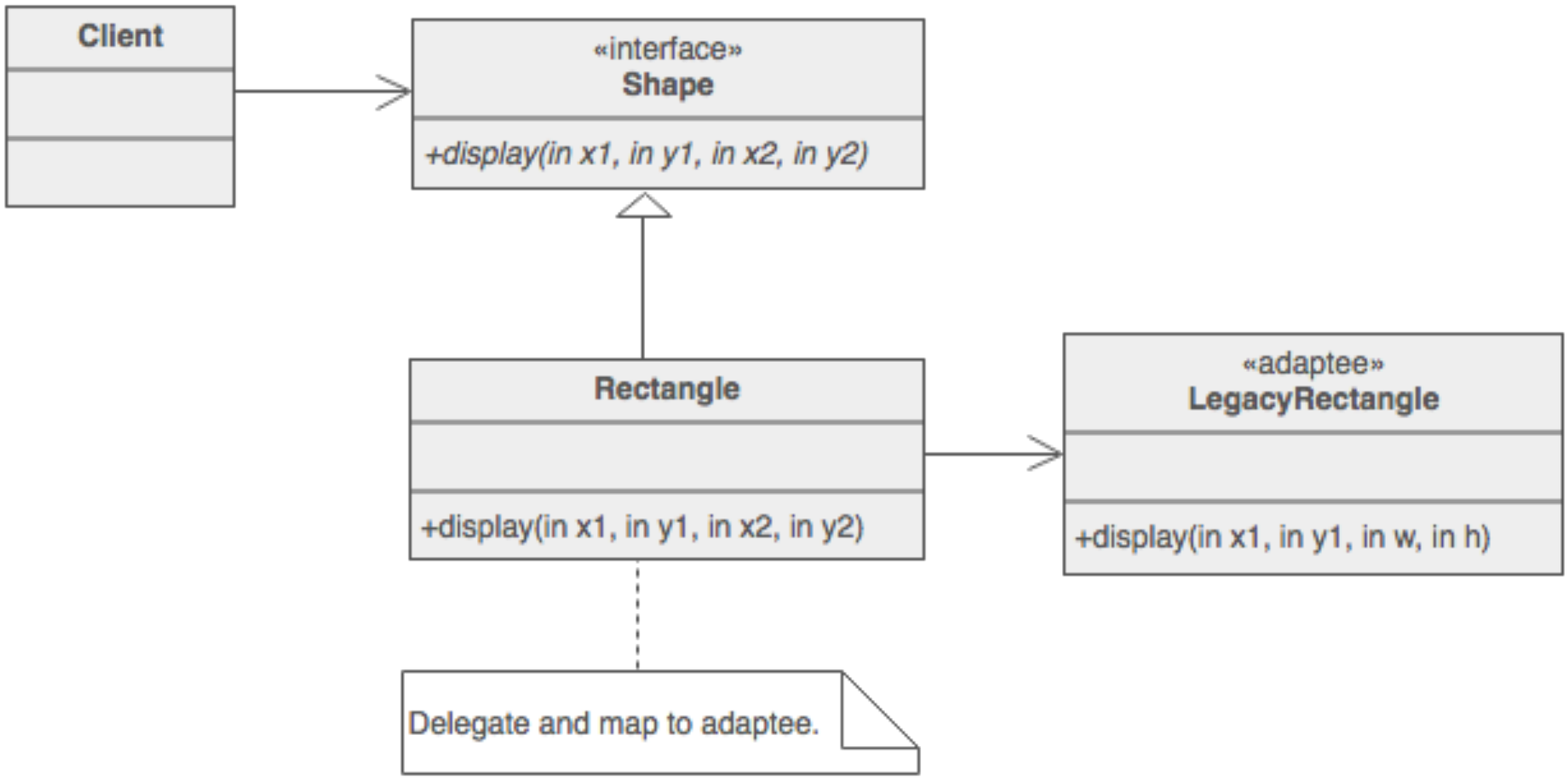
Adapter Design Pattern

- Intent
    - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
    - Wrap an existing class with a new interface.
    - Impedance match an old component to a new system

Adapter Design Pattern

- Problem
    - An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Adapter Design Pattern



Adapter Design Pattern

- Check list

    - Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).

    - Identify the interface that the client requires.

    - Design a "wrapper" class that can "impedance match" the adaptee to the client.

    - The adapter/wrapper class "has a" instance of the adaptee class.

    - The adapter/wrapper class "maps" the client interface to the adaptee interface.

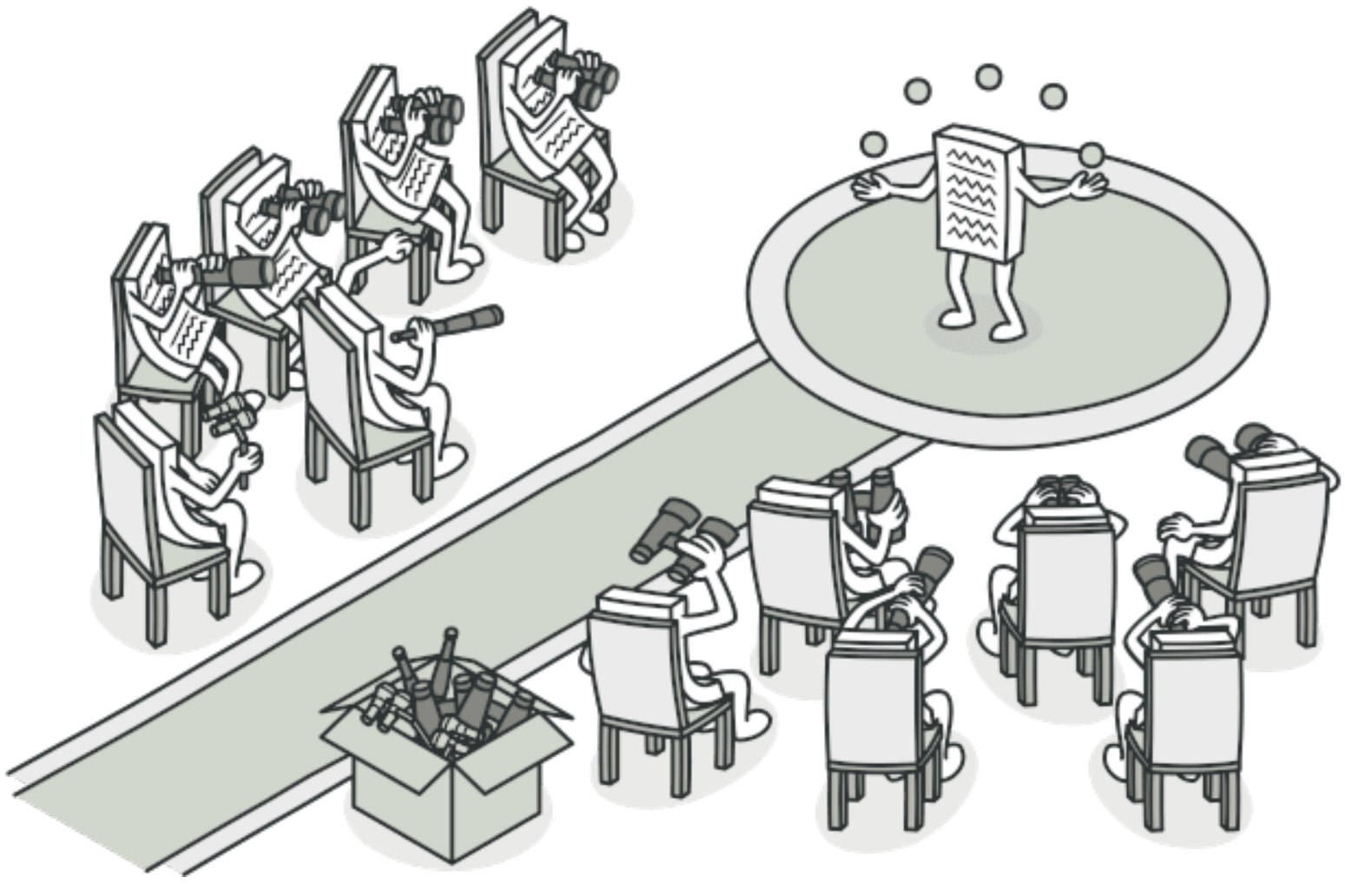
    - The client uses (is coupled to) the new interface

Observer

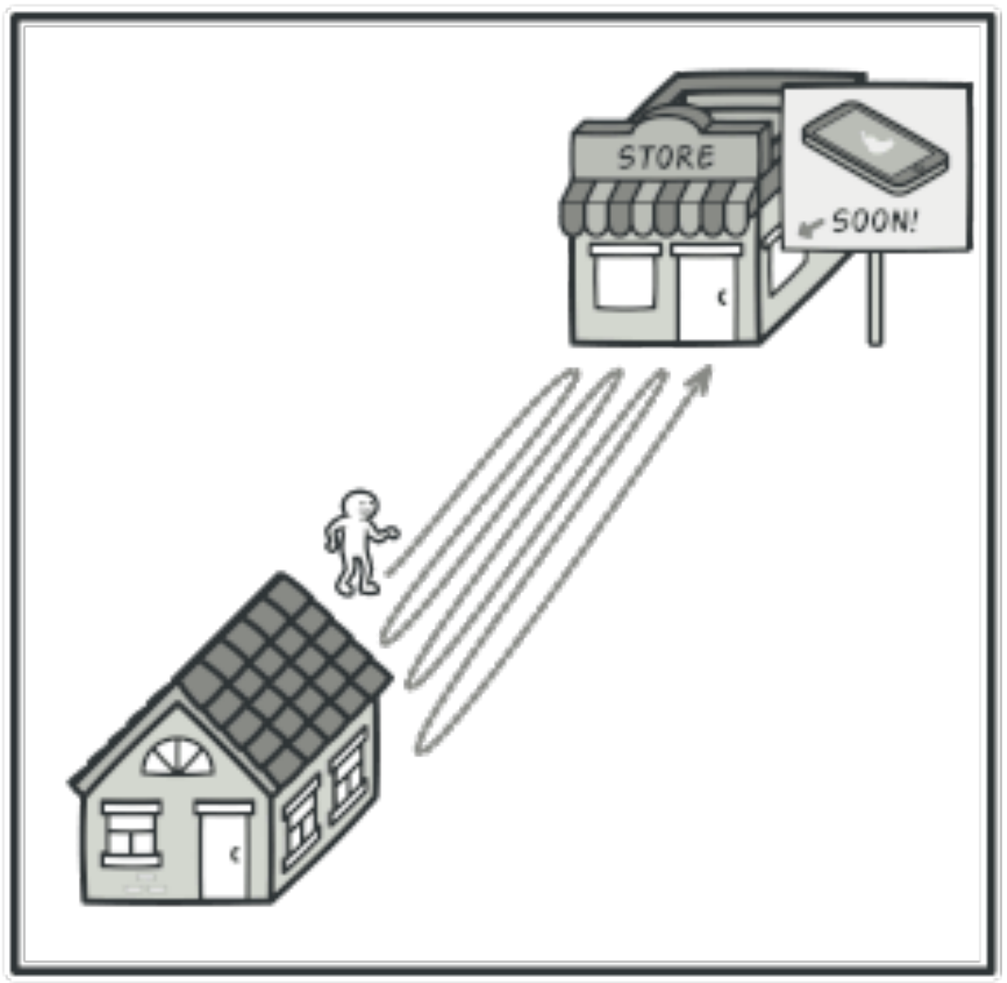
A.K.A. Subject-observer, publish-subscribe, callback

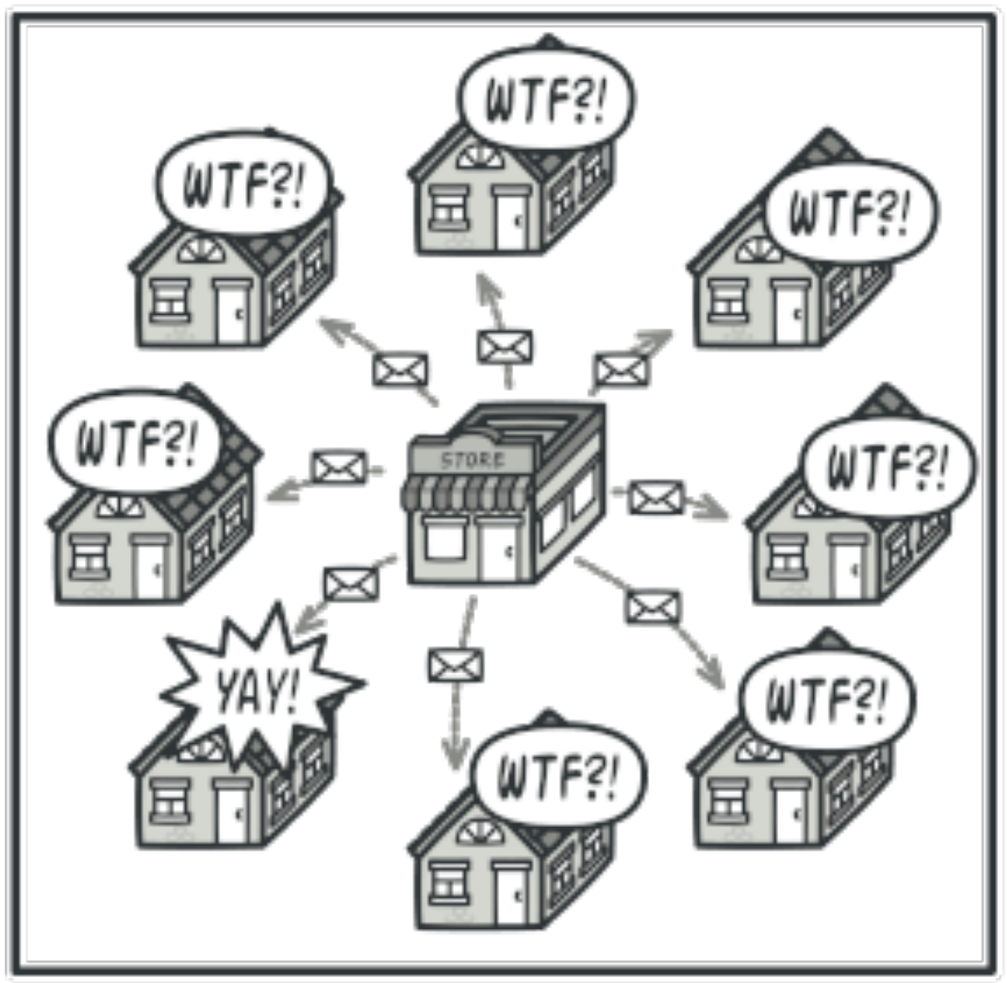
Intent

- Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

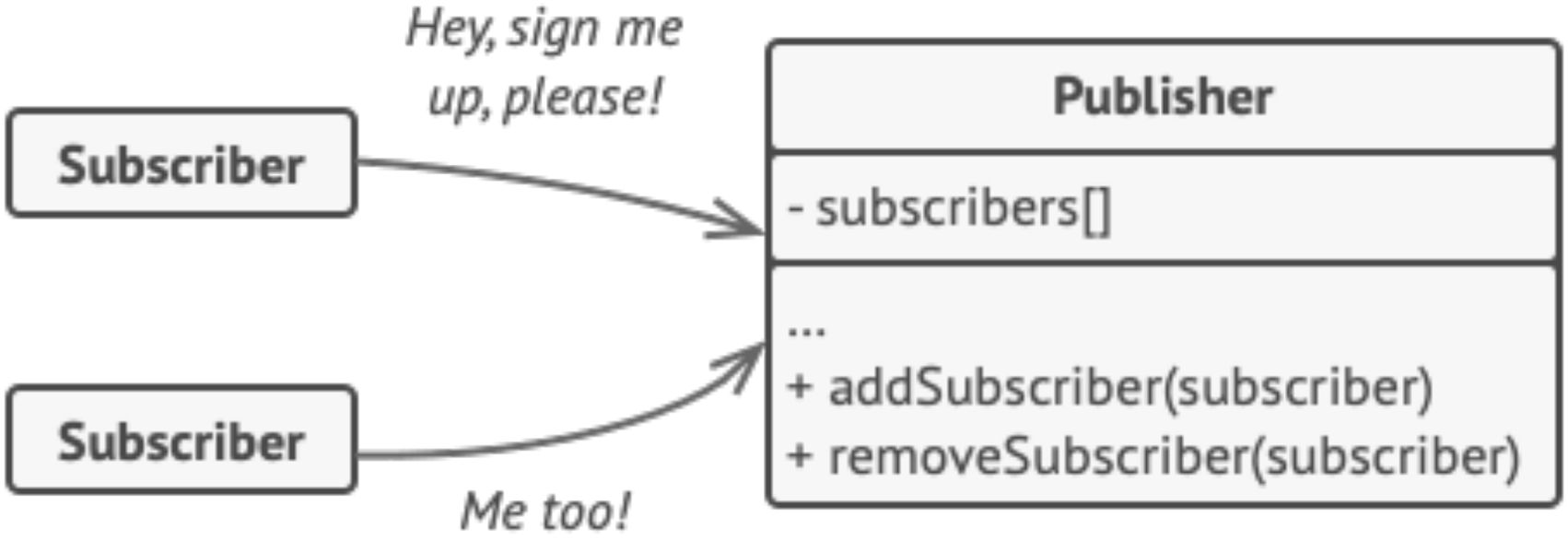


Problem

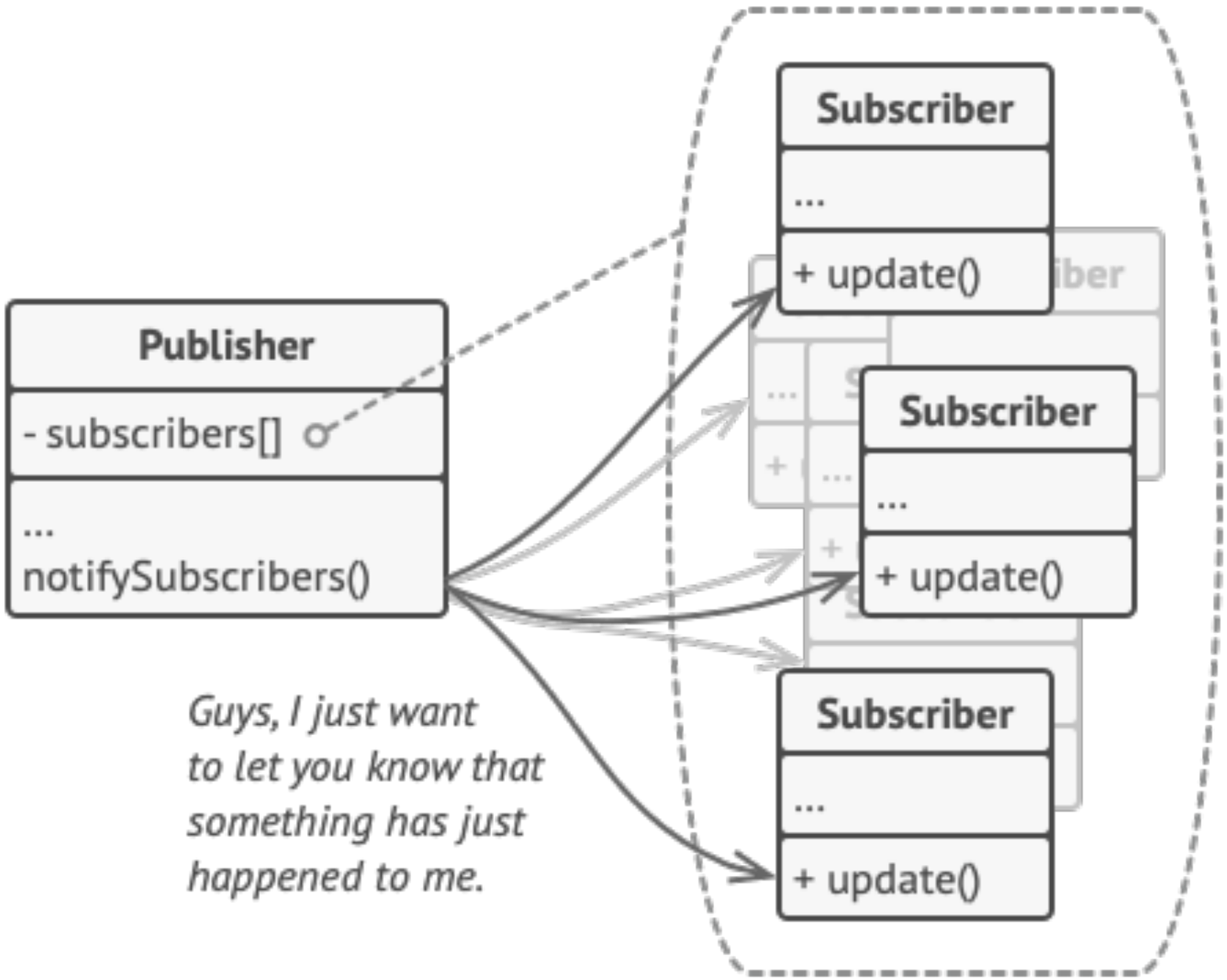




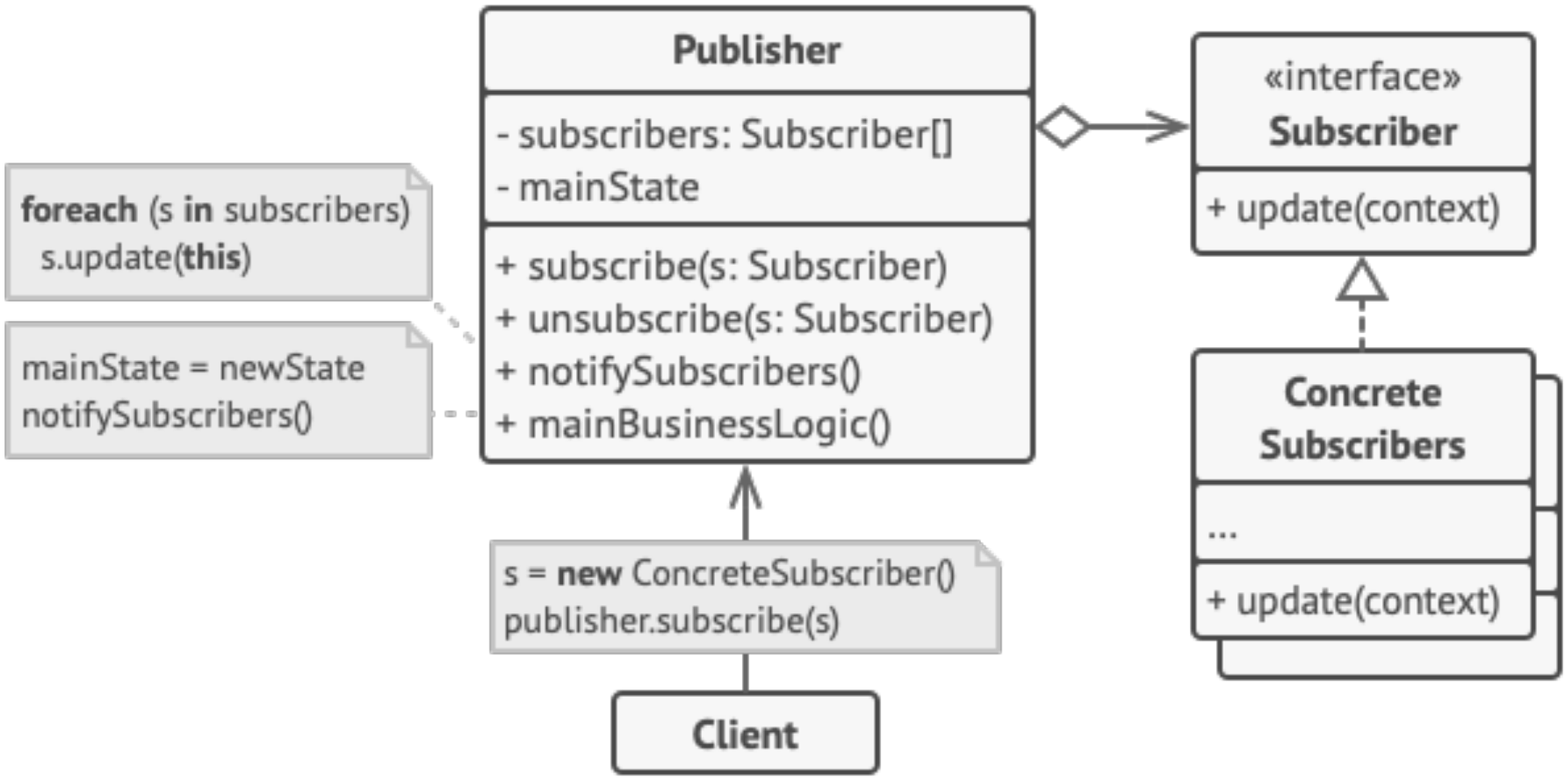
Solution


Hey, sign me up, please!
Subscriber
Subscriber
Me too!
Publisher
- subscribers[]
...
+ addSubscriber(subscriber)
+ removeSubscriber(subscriber)

Solution


Publisher
- subscribers[]
...
notifySubscribers()
Subscriber
...
+ update()
Subscriber
...
+ update()
Subscriber
...
+ update()
Guys, I just want to let you know that something has just happened to me.

Structure


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
«interface»
Subscriber
+ update(context)
Concrete
Subscribers
...
+ update(context)
s = new ConcreteSubscriber()
publisher.subscribe(s)
Client

Proxy

A.K.A. Surrogate, Look-alike

Intent

- Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object.

- A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Proxy pattern

- Simply speaking, a Proxy object is one through which we control access to the actual object on which the functionality lies.

- Depending on the context in which the proxy object is used, the pattern is broadly divided into the following 3 types:

  - Virtual proxy
  - Remote proxy
  - Access proxy

Virtual proxy

- Used for lazy initialization of objects or for lazy processing

  - Suppose you need to support ‘resource-hungry’ objects that involve high amount of I/O or those involved in a database transaction

  - One need not instantiate these objects until they are really required

  - The real object would get created only when the client actually requests for some of its functionality.

  - This is ‘Lazy initialization pattern’

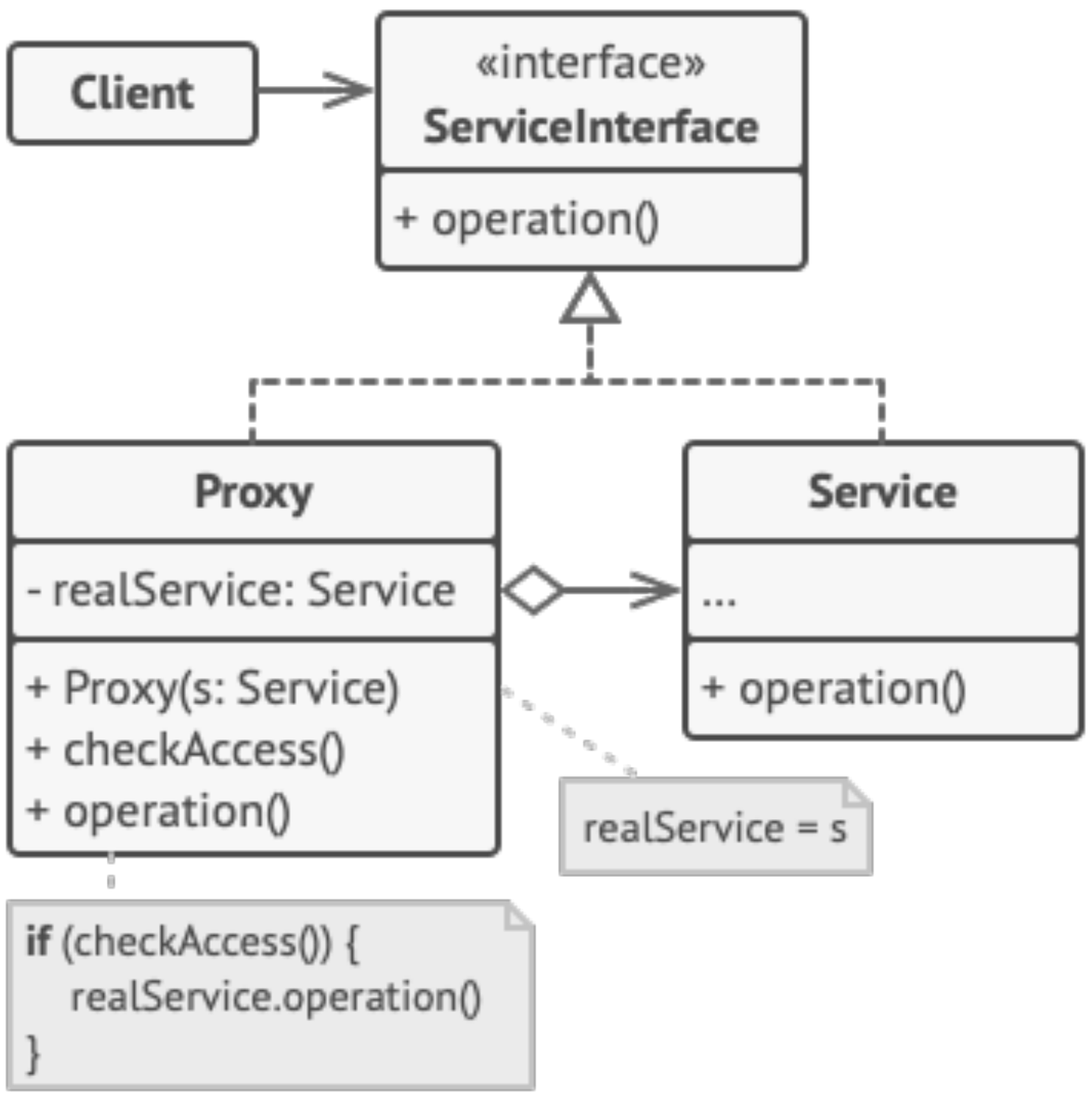
Remote proxy

- Used to hide the communication between mechanisms between remote objects
- In RPC (Remote Procedure Calls) for example, we have the stubs which act as remote proxies for the skeleton.
- This is ‘Remote proxy’

Access proxy

- Used to provide control over a sensitive master object.
- This proxy object could check for the client's access permission before allowing methods to be executed on the actual object.
- This is 'Protection proxy'

Structure


Client
«interface»
ServiceInterface
+ operation()
Proxy
- realService: Service
+ Proxy(s: Service)
+ checkAccess()
+ operation()
Service
...
+ operation()
realService = s
if (checkAccess()) {
realService.operation()
}

Applicability

- Lazy initialization (virtual proxy).
    - This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
    - Instead of creating the object when the app launches, you can delay the object's initialization to a time when it's really needed.

Applicability

- Access control (protection proxy).
    - This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).
    - The proxy can pass the request to the service object only if the client’s credentials match some criteria.

Applicability

- Local execution of a remote service (remote proxy).
    - This is when the service object is located on a remote server.
    - In this case, the proxy passes the client request over the network, handling all of the nasty details of working with the network.

Applicability

- Logging requests (logging proxy).
  - This is when you want to keep a history of requests to the service object.
  - The proxy can log each request before passing it to the service.

Applicability

- Caching request results (caching proxy).
    - This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.
    - The proxy can implement caching for recurring requests that always yield the same results. The proxy may use the parameters of requests as the cache keys.

Applicability

- Smart reference.
  - This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.
  - The proxy can keep track of clients that obtained a reference to the service object or its results. From time to time, the proxy may go over the clients and check whether they are still active. If the client list gets empty, the proxy might dismiss the service object and free the underlying system resources.
  - The proxy can also track whether the client had modified the service object. Then the unchanged objects may be reused by other clients.

Command

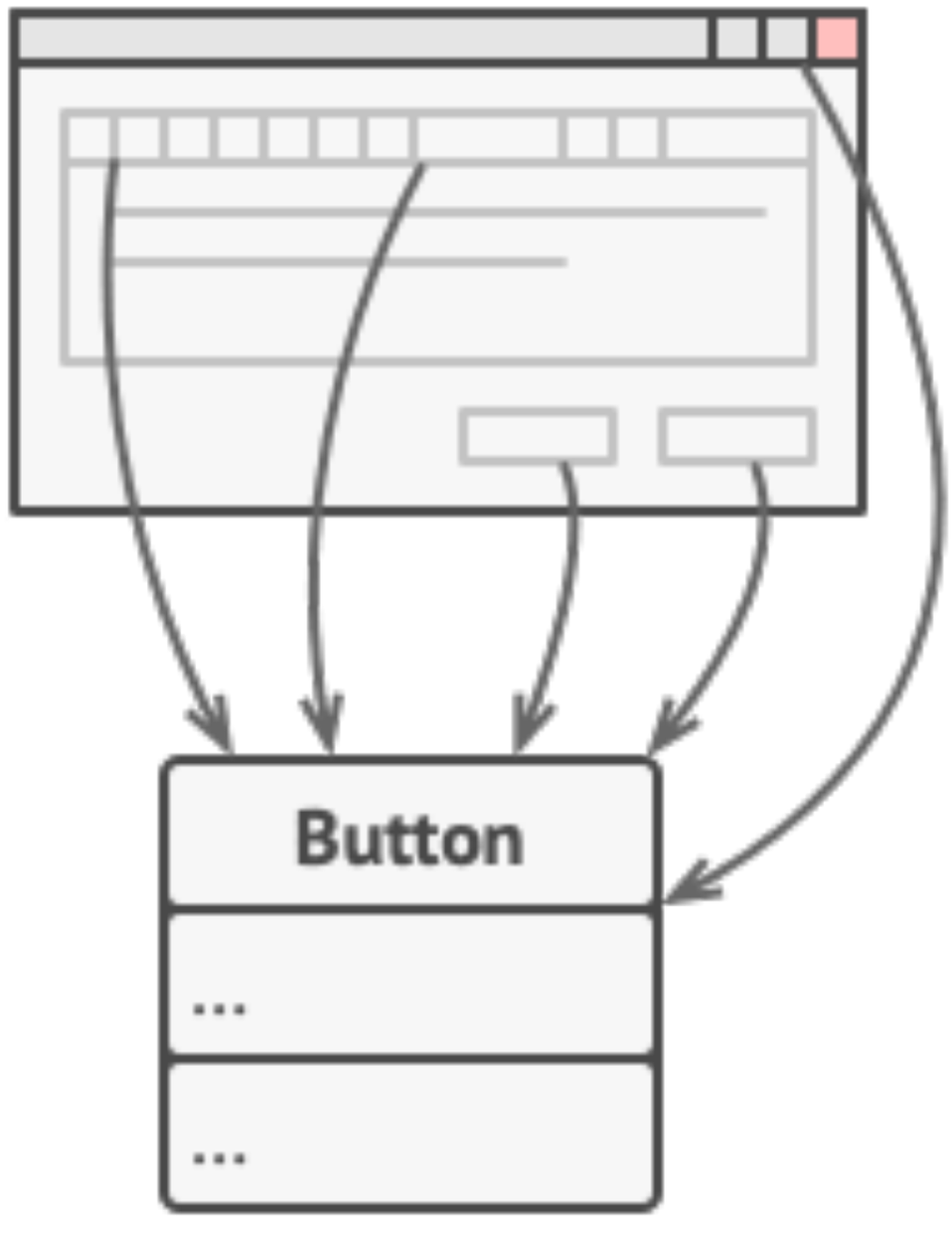
A.K.A. Action, Transaction

Intent

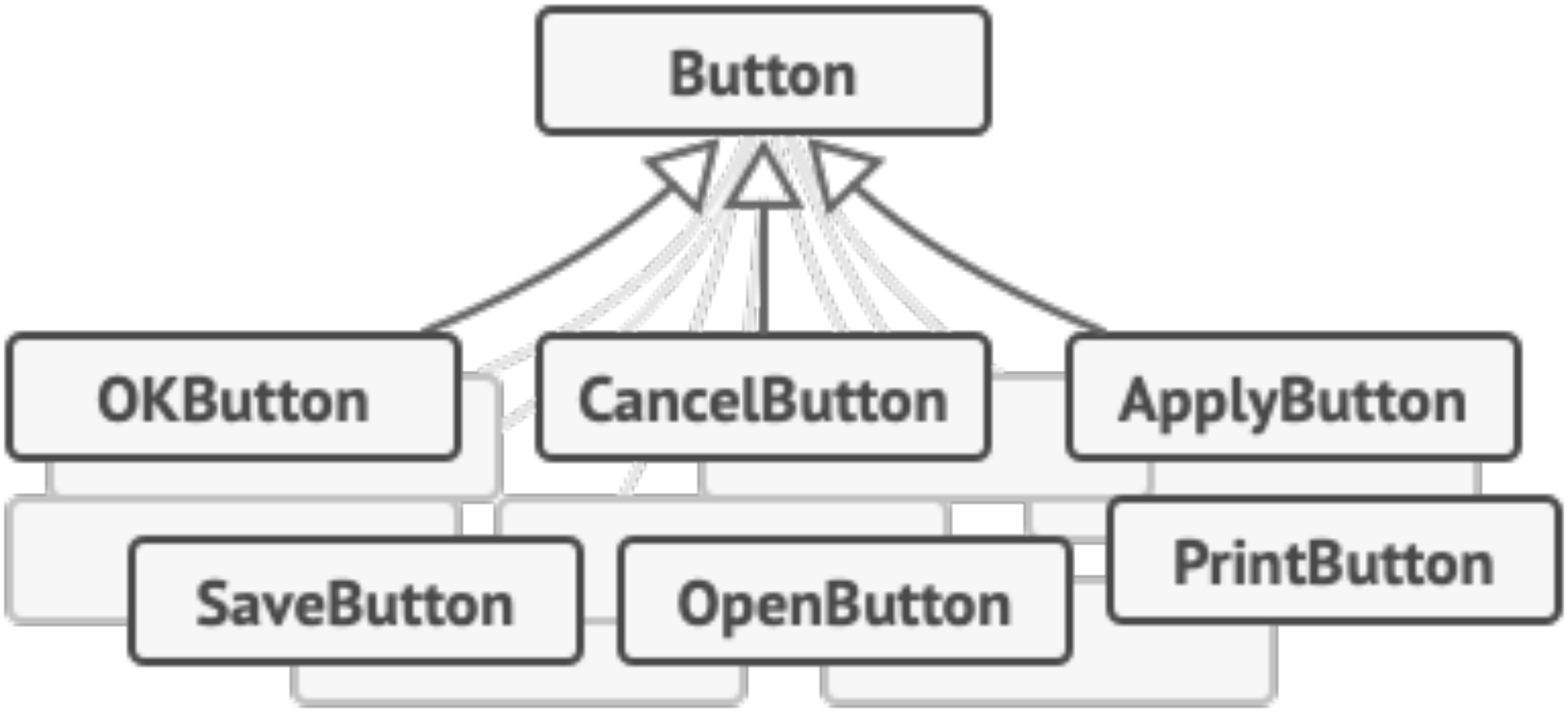
- Command is a behavioural design pattern that turns a request into a stand-alone object that contains all information about the request.

- This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

Problem

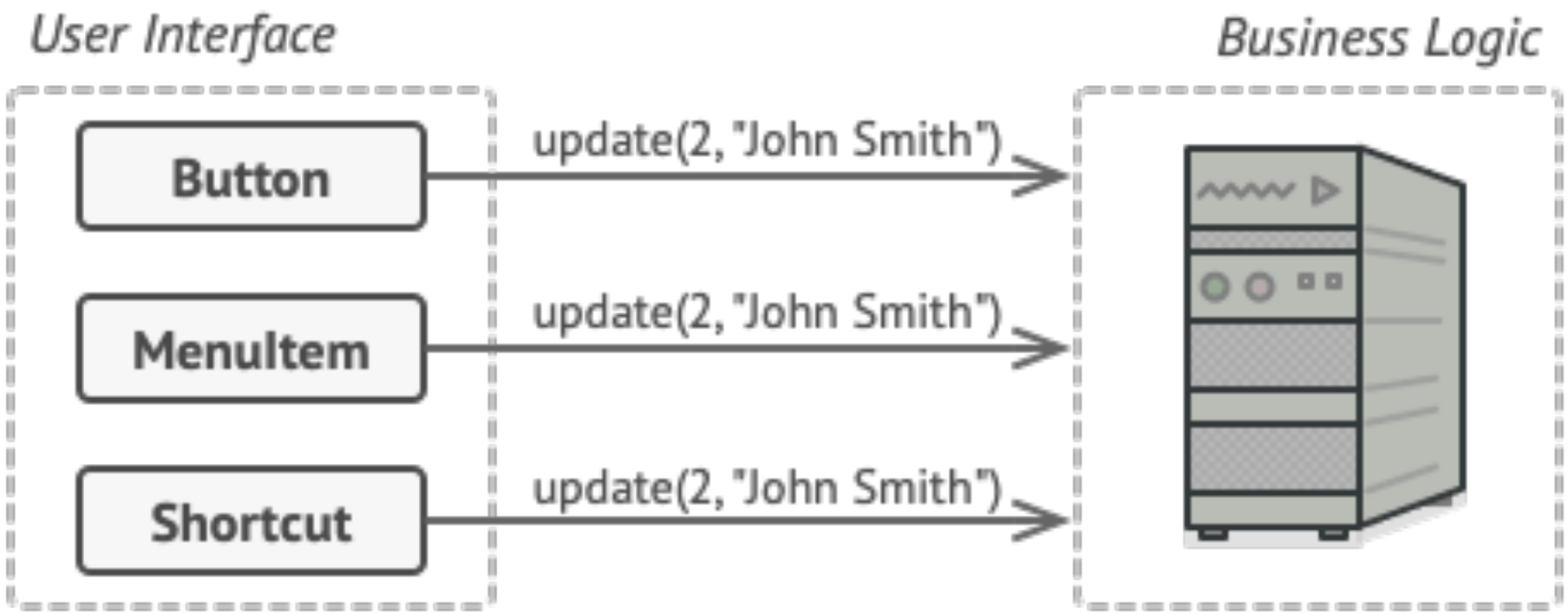


Problem

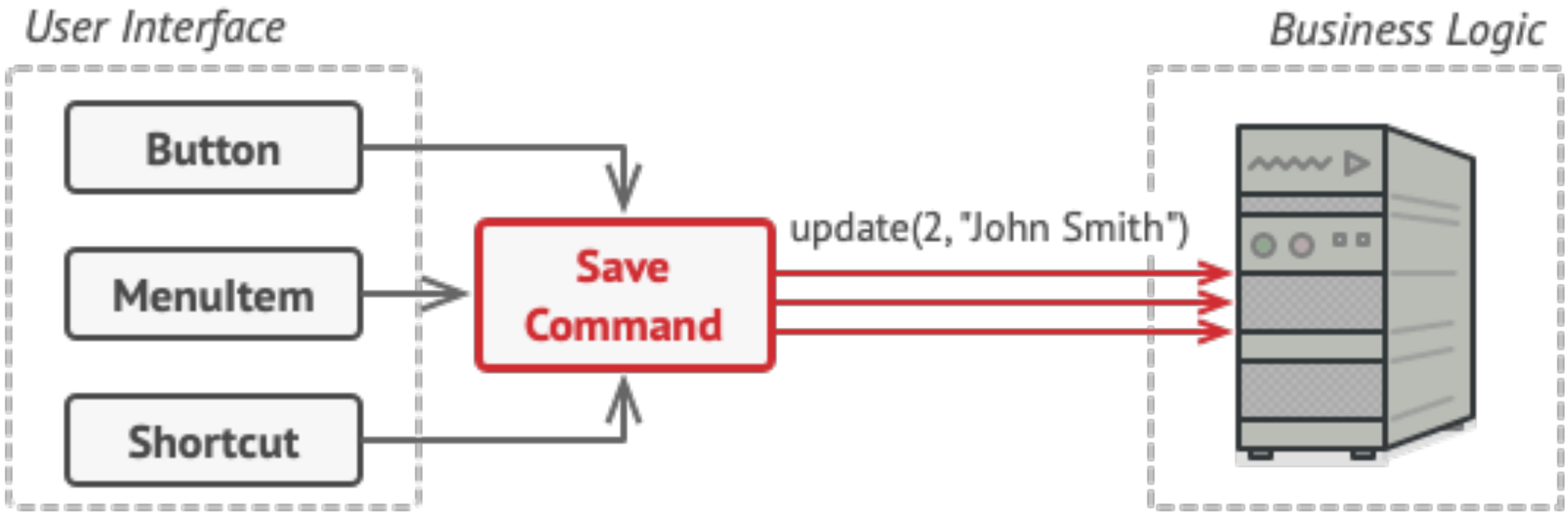

Button
OKButton
CancelButton
ApplyButton
SaveButton
OpenButton
PrintButton

Problem

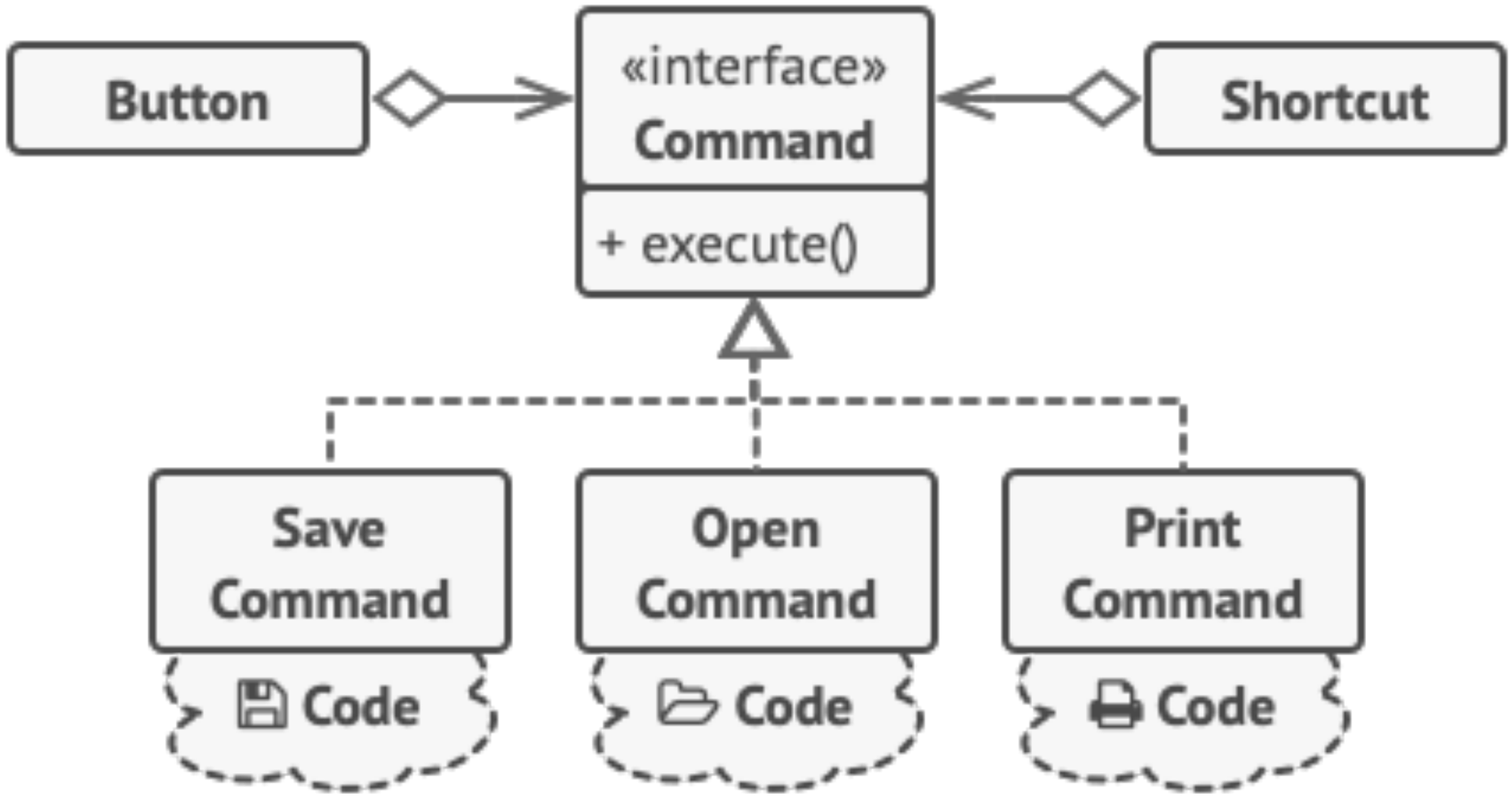
Solution


User Interface
Business Logic
Button
Menultem
Shortcut
update(2, "John Smith")
update(2, "John Smith")
update(2, "John Smith")

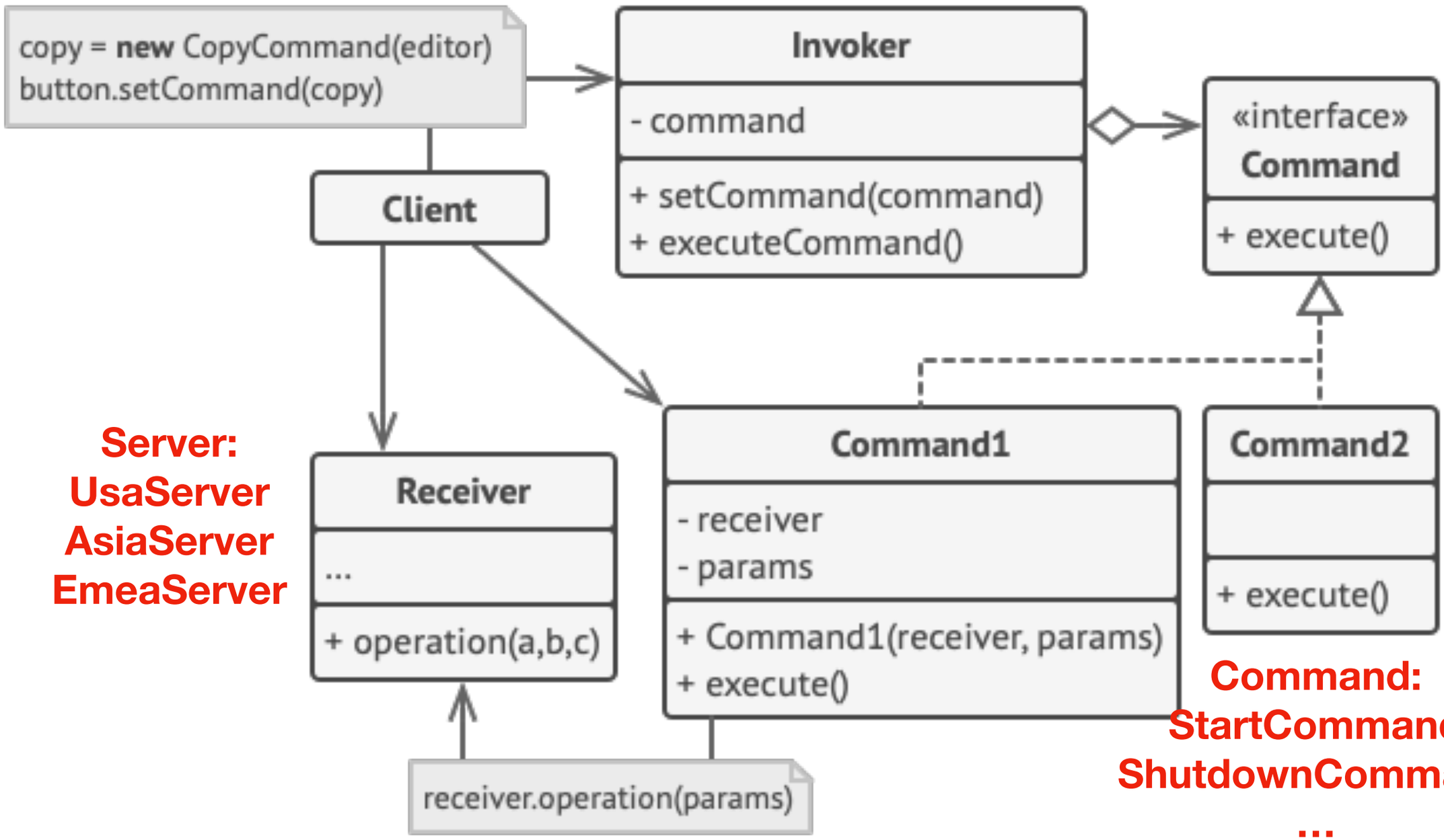
Solution


User Interface
Button
MenuItem
Shortcut
Save Command
update(2, "John Smith")
Business Logic

Solution


Button
«interface»
Command
+ execute()
Shortcut
Save
Command
Code
Open
Command
Code
Print
Command
Code

Structure



Server:
UsaServer
AsiaServer
EmeaServer

Command:
StartCommand
ShutdownCommand
...

Applicability

- Use the Command pattern when you want to parametrize objects with operations.
    - The Command pattern can turn a specific method call into a stand-alone object.
        - This change opens up a lot of interesting uses:
            - you can pass commands as method arguments, store them inside other objects, switch linked commands at runtime, etc.

Applicability

- Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.
    - As with any other object, a command can be serialized, which means converting it to a string that can be easily written to a file or a database.
- Later, the string can be restored as the initial command object.
- Thus, you can delay and schedule command execution.
    - But there’s even more! In the same way, you can queue, log or send commands over the network.

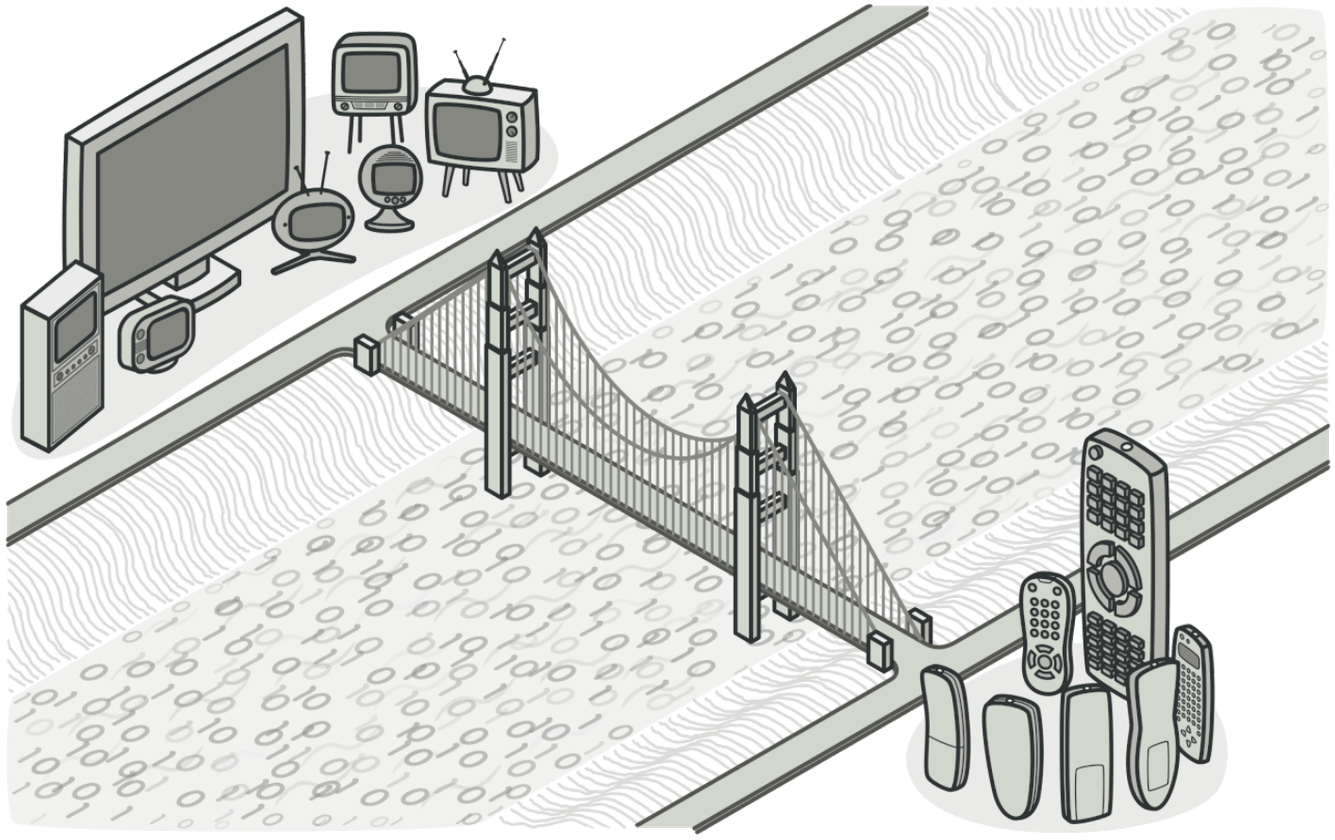
Applicability

- Use the Command pattern when you want to implement reversible operations.
    - Although there are many ways to implement undo/redo, the Command pattern is perhaps the most popular of all.
    - To be able to revert operations, you need to implement the history of performed operations. The command history is a stack that contains all executed command objects along with related backups of the application's state.
    - This method has two drawbacks. First, it isn't that easy to save an application's state because some of it can be private. This problem can be mitigated with the Memento pattern.
    - Second, the state backups may consume quite a lot of RAM. Therefore, sometimes you can resort to an alternative implementation: instead of restoring the past state, the command performs the inverse operation. The reverse operation also has a price: it may turn out to be hard or even impossible to implement.

Bridge

Bridge

- Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Intent

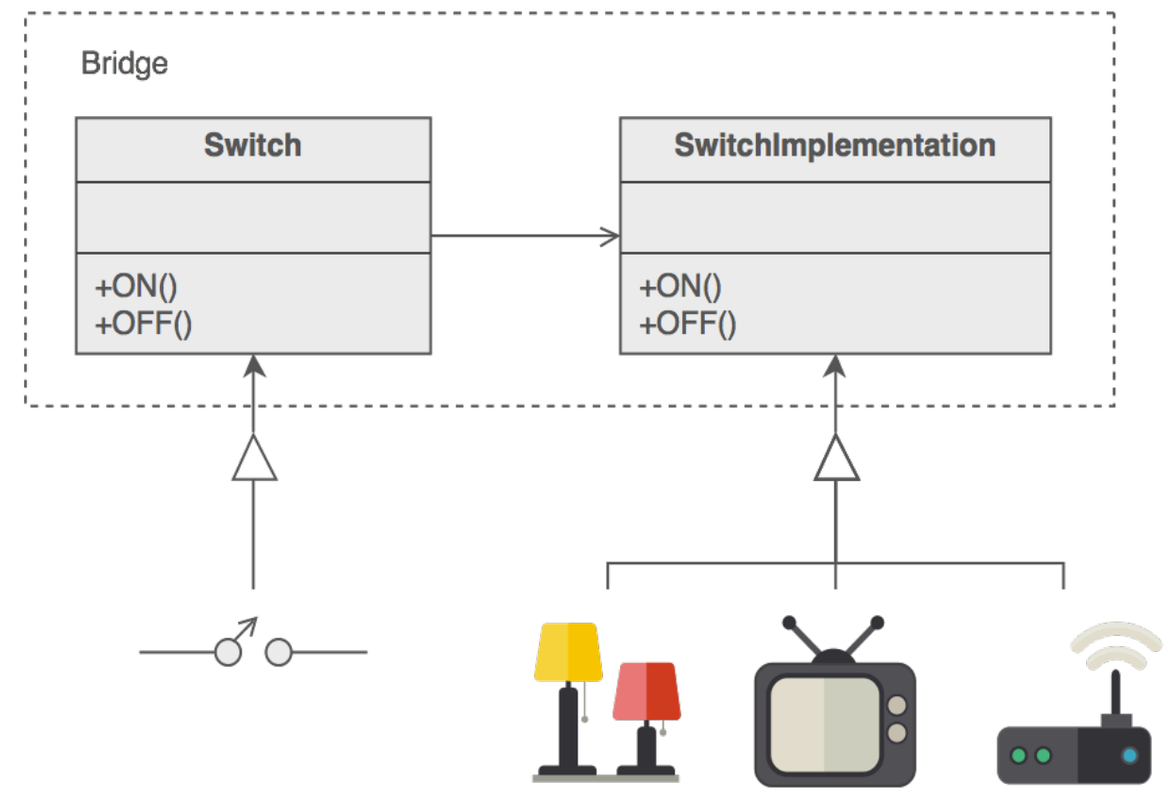
- Decouple an abstraction from its implementation so that the two can vary independently.

- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.

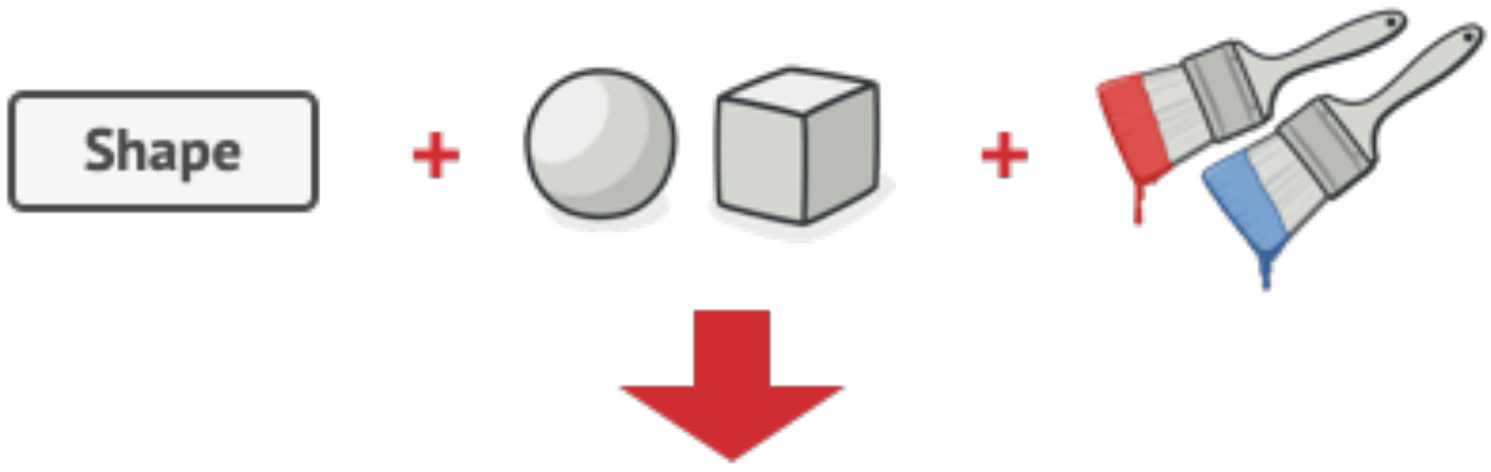
- Beyond encapsulation, to insulation

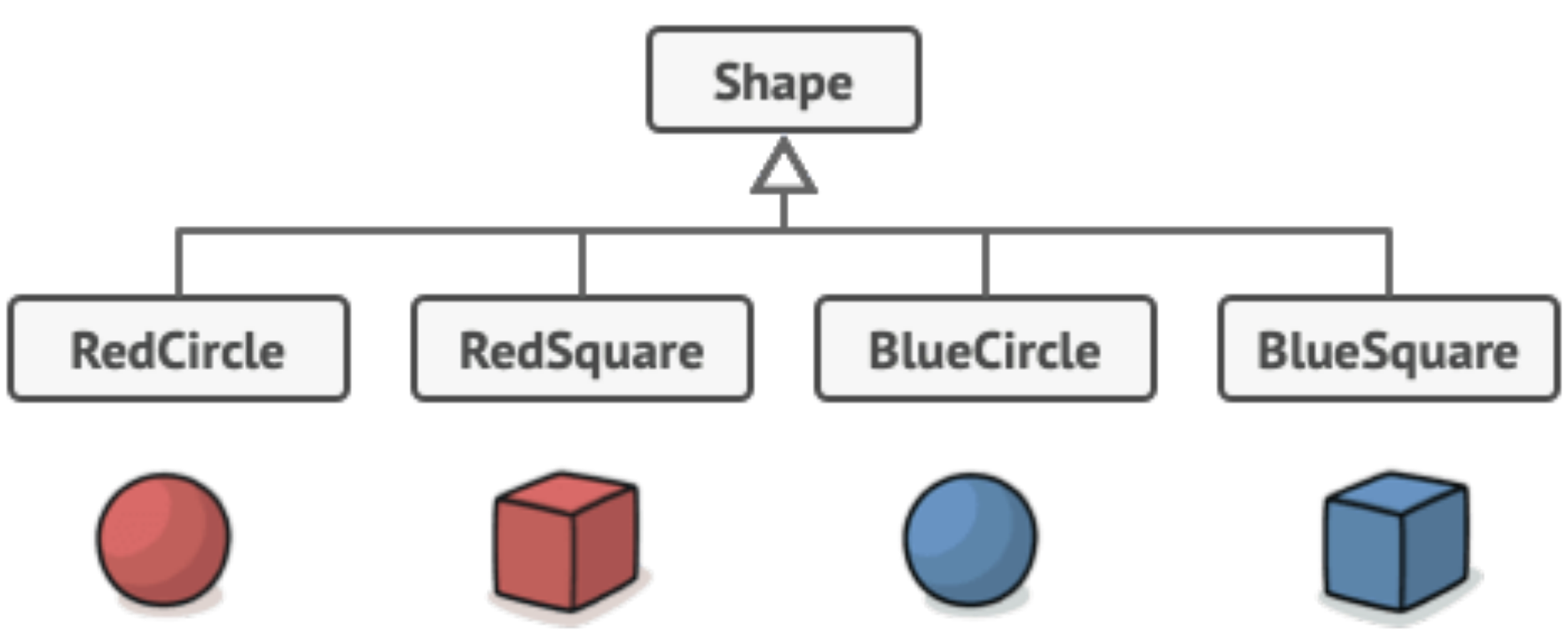
Bridge

- A household switch controlling lights, ceiling fans, etc. is an example of the Bridge.
- The purpose of the switch is to turn a device on or off.
- The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.

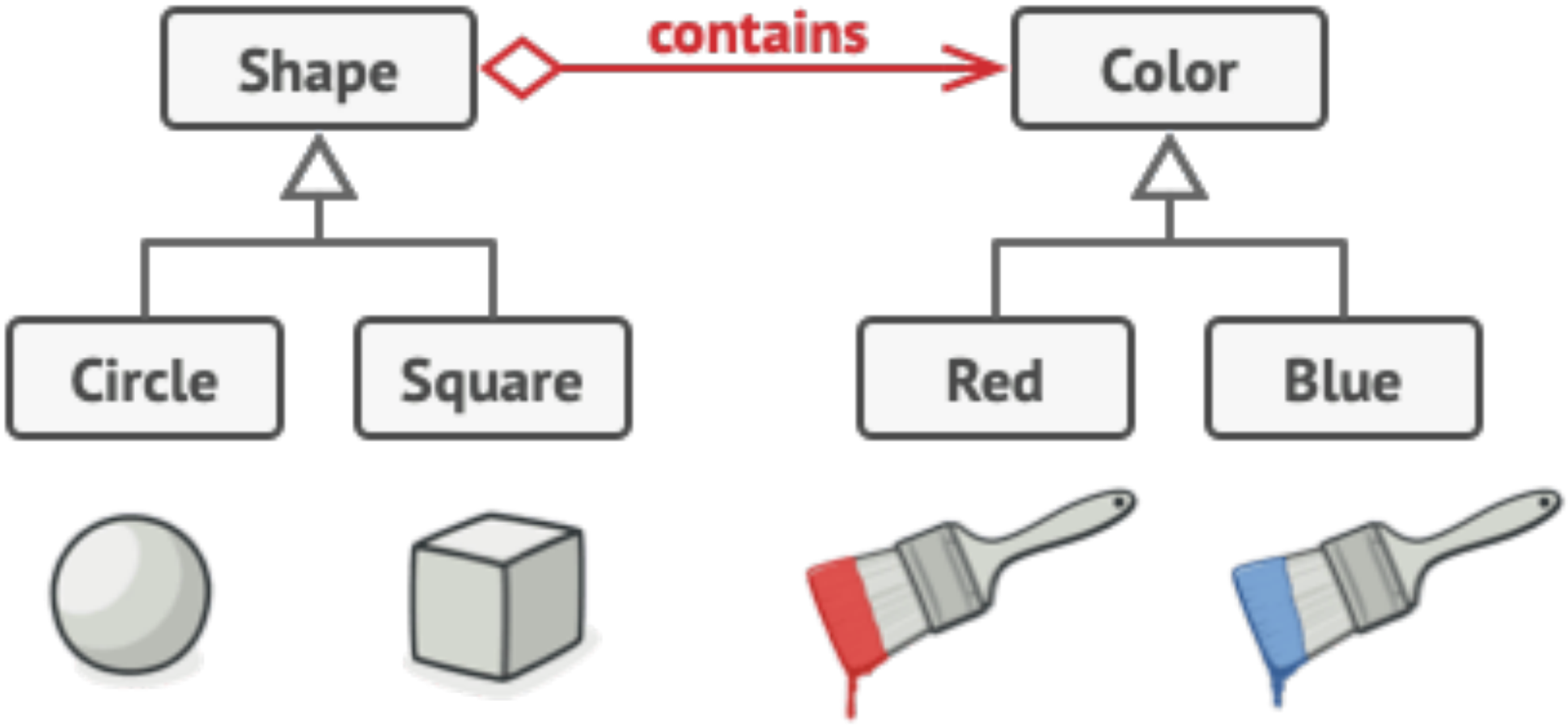


Problem



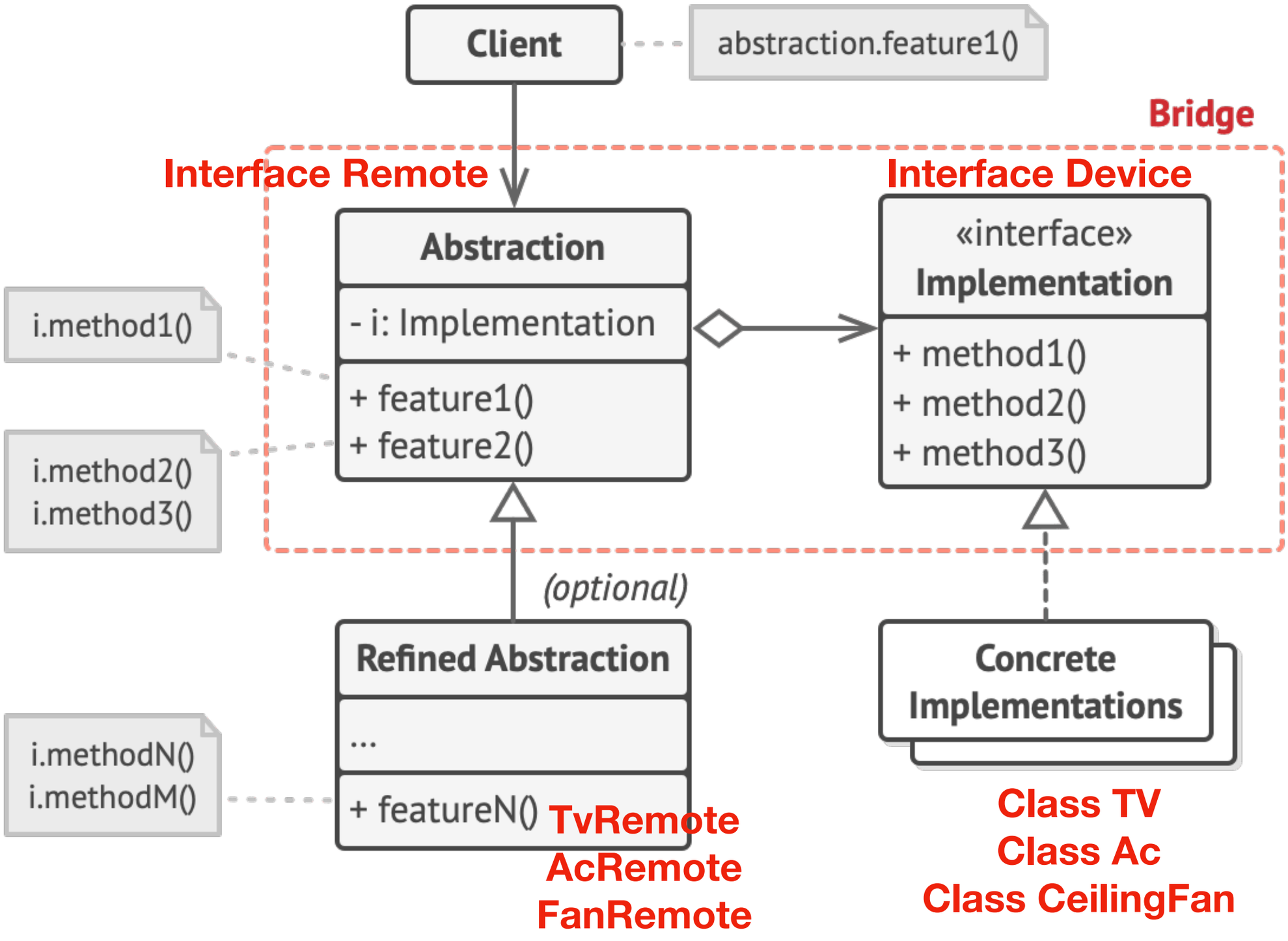


Solution

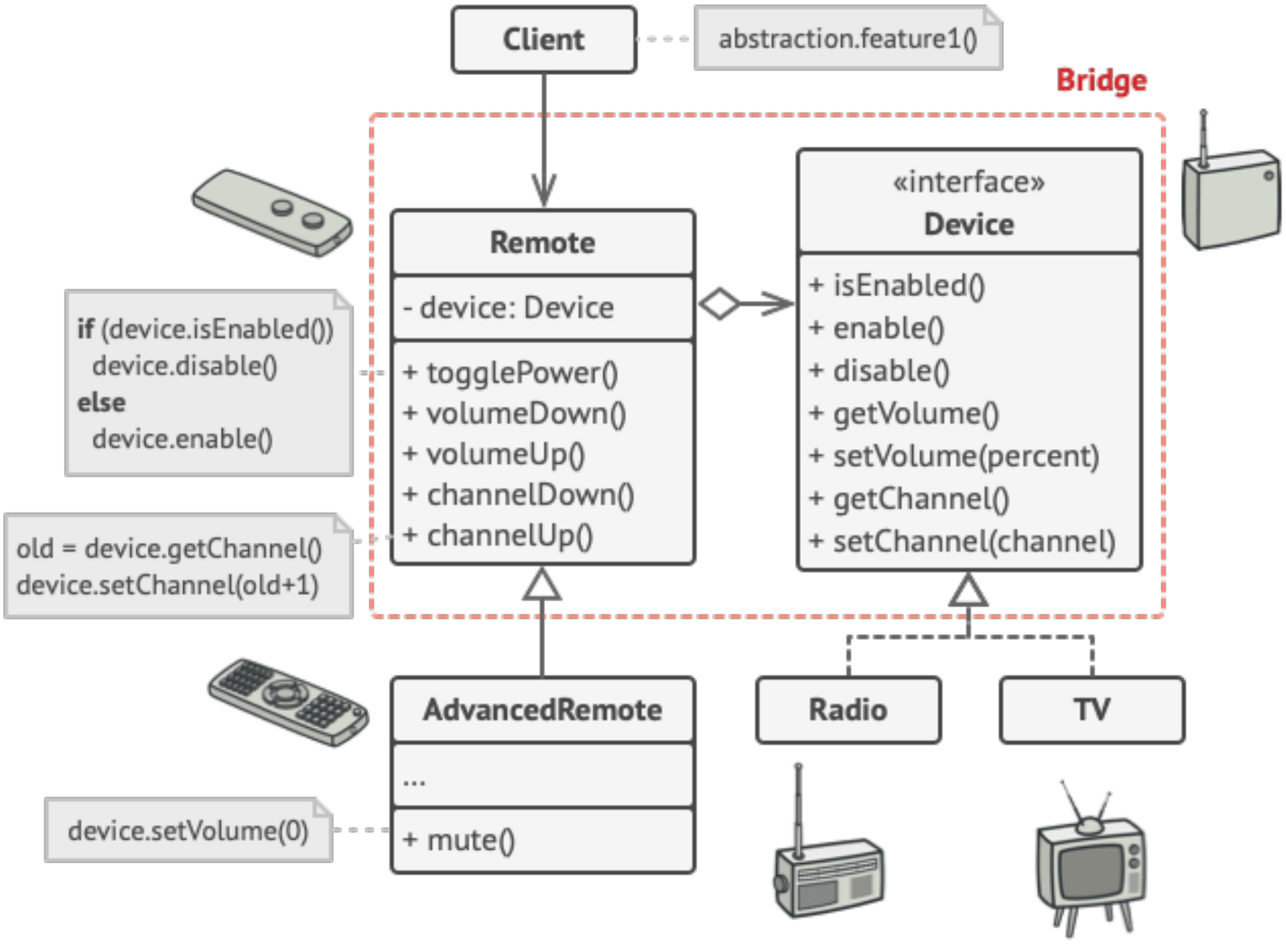


You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.

Structure


Client
abstraction.feature1()
Bridge
Interface Remote
Interface Device
Abstraction
- i: Implementation
+ feature1()
+ feature2()
«interface»
Implementation
+ method1()
+ method2()
+ method3()
i.method1()
i.method2()
i.method3()
(optional)
Refined Abstraction
...
+ featureN()
i.methodN()
i.methodM()
Concrete
Implementations
TvRemote
AcRemote
FanRemote
Class TV
Class Ac
Class CeilingFan

Example


Client
abstraction.feature1()
Bridge
«interface»
Device
+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)
Remote
- device: Device
+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()
if (device.isEnabled())
device.disable()
else
device.enable()
old = device.getChannel()
device.setChannel(old+1)
AdvancedRemote
...
+ mute()
device.setVolume(0)
Radio
TV

Applicability

- Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).

    - The bigger a class becomes, the harder it is to figure out how it works, and the longer it takes to make a change. The changes made to one of the variations of functionality may require making changes across the whole class, which often results in making errors or not addressing some critical side effects.

    - The Bridge pattern lets you split the monolithic class into several class hierarchies. After this, you can change the classes in each hierarchy independently of the classes in the others. This approach simplifies code maintenance and minimizes the risk of breaking existing code.

Applicability

- Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.

    - The Bridge suggests that you extract a separate class hierarchy for each of the dimensions. The original class delegates the related work to the objects belonging to those hierarchies instead of doing everything on its own.

Applicability

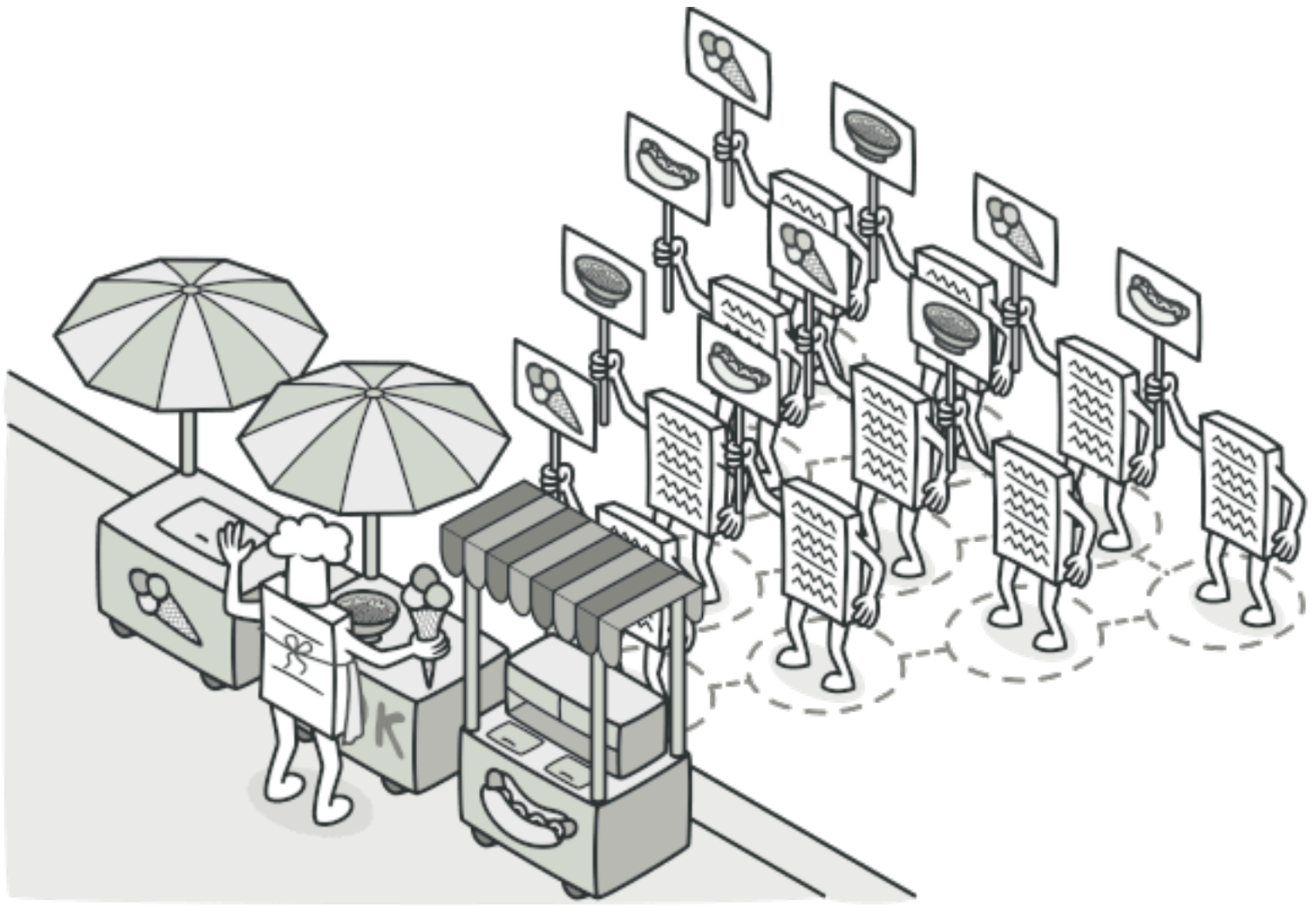
- Use the Bridge if you need to be able to switch implementations at runtime.
    - Although it’s optional, the Bridge pattern lets you replace the implementation object inside the abstraction. It’s as easy as assigning a new value to a field.

Visitor

Based on - Double Dispatch

Visitor

- Visitor is a behavioural design pattern that lets you separate algorithms from the objects on which they operate.

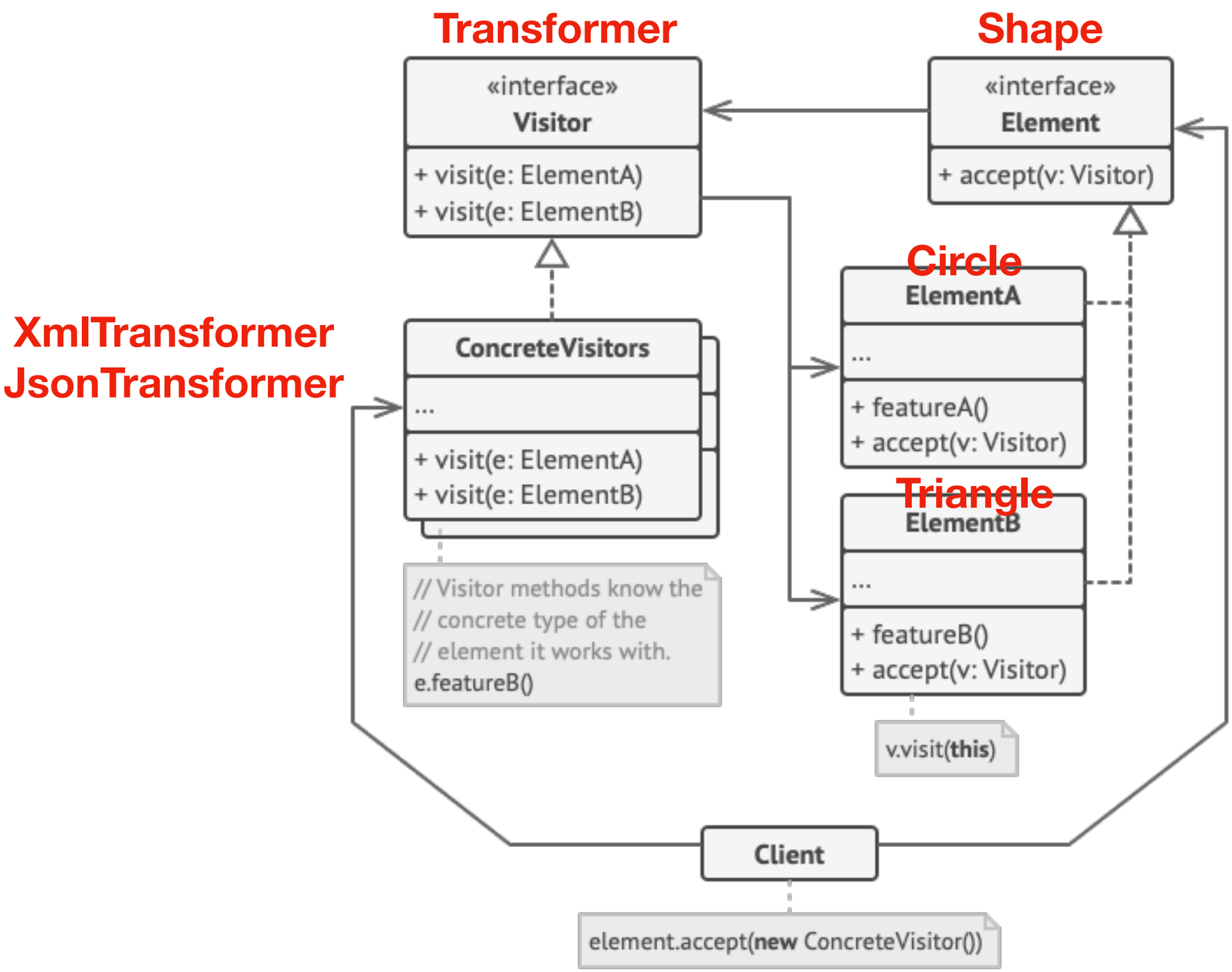


Visitor

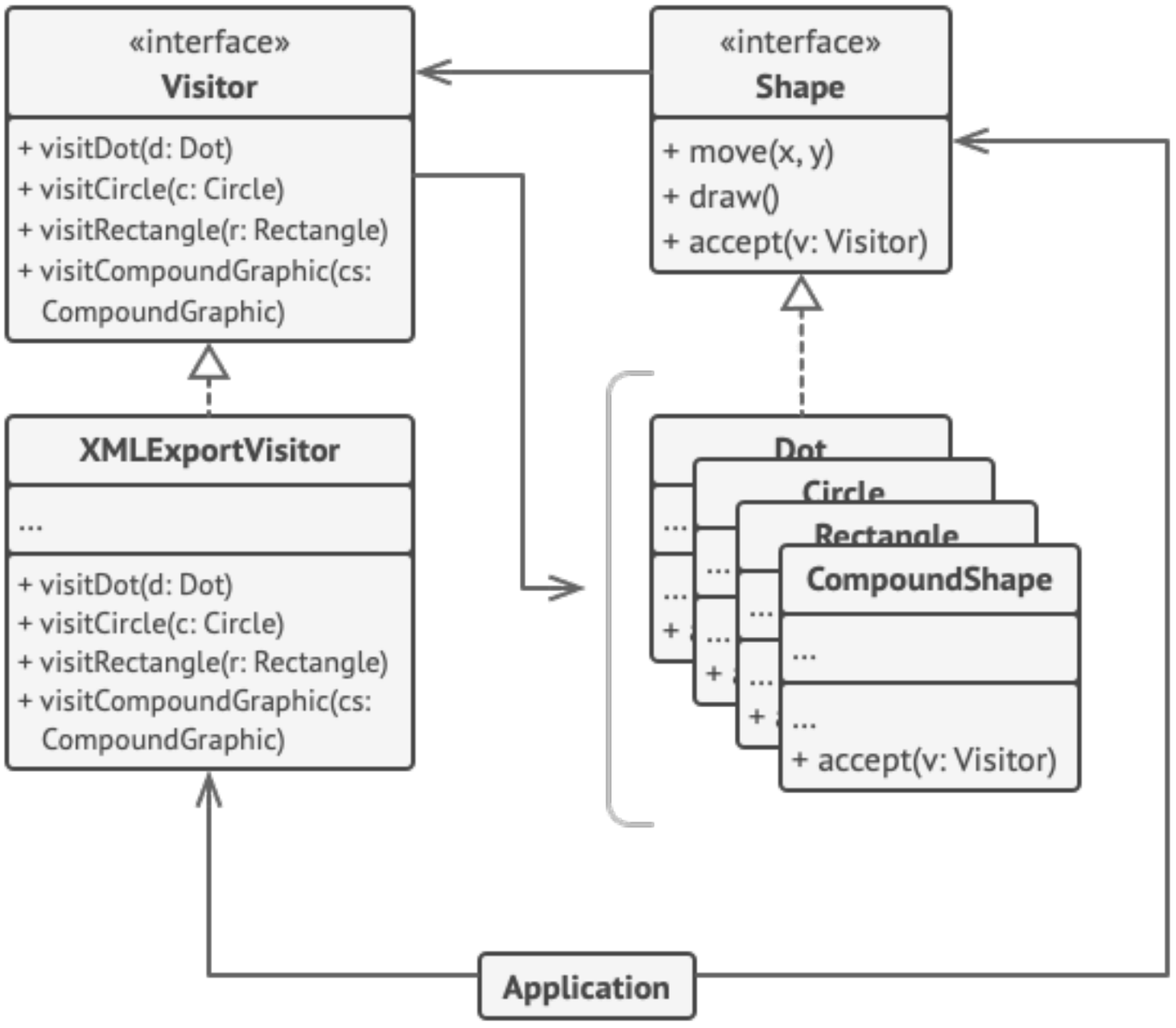
In object-oriented programming and software engineering, the **visitor** design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the open/closed principle.

In essence, the visitor allows adding new virtual functions to a family of classes, without modifying the classes. Instead, a visitor class is created that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

Structure


Transformer
Shape
«interface»
Visitor
+ visit(e: ElementA)
+ visit(e: ElementB)
«interface»
Element
+ accept(v: Visitor)
Circle
XmlTransformer
JsonTransformer
ConcreteVisitors
...
+ visit(e: ElementA)
+ visit(e: ElementB)
ElementA
...
+ featureA()
+ accept(v: Visitor)
Triangle
ElementB
...
+ featureB()
+ accept(v: Visitor)
// Visitor methods know the
// concrete type of the
// element it works with.
e.featureB()
v.visit(this)
Client
element.accept(new ConcreteVisitor())

Example


«interface»
Visitor
+ visitDot(d: Dot)
+ visitCircle(c: Circle)
+ visitRectangle(r: Rectangle)
+ visitCompoundGraphic(cs: CompoundGraphic)
«interface»
Shape
+ move(x, y)
+ draw()
+ accept(v: Visitor)
XMLExportVisitor
...
+ visitDot(d: Dot)
+ visitCircle(c: Circle)
+ visitRectangle(r: Rectangle)
+ visitCompoundGraphic(cs: CompoundGraphic)
Dot
Circle
Rectangle
CompoundShape
...
...
+ accept(v: Visitor)
Application

RAII

Resource Acquisition Is Instantiation

RAII

- Resource Acquisition Is Initialization or RAII, is a C++ programming technique which binds the life cycle of a resource that must be acquired before use to the lifetime of an object.

  - Resource - allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply

RAII

- RAII guarantees that the resource is available to any function that may access the object (resource availability is a class invariant, eliminating redundant runtime tests).

- It also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition.

- Likewise, if resource acquisition fails (the constructor exits with an exception), all resources acquired by every fully-constructed member and base subobject are released in reverse order of initialization.

- This leverages the core language features (object lifetime, scope exit, order of initialization and stack unwinding) to eliminate resource leaks and guarantee exception safety.

- Another name for this technique is Scope-Bound Resource Management (SBRM), after the basic use case where the lifetime of an RAII object ends due to scope exit.