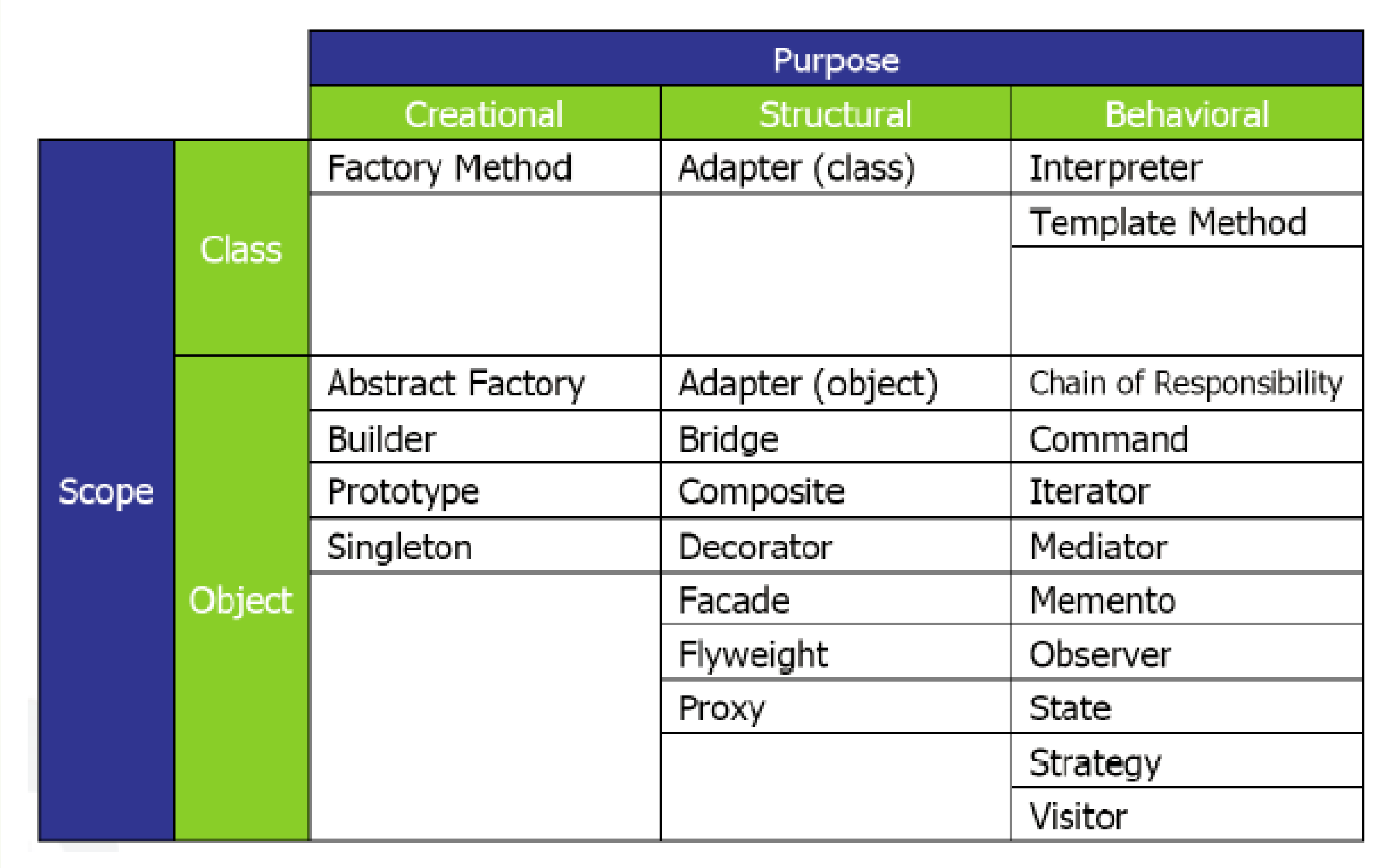
Design Patterns

Lecturer: Adel Vahdati

General Categories

- **Creational** patterns
  - Deal with initializing and configuring classes and objects.
- **Structural** patterns
  - Deal with decoupling interface and implementation of classes and objects.
- **Behavioral** patterns
  - Deal with dynamic interactions among societies of classes and objects.

GoF Design Patterns: Purpose and Scope



		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Creational Pattern

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Factory Method

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

Problem

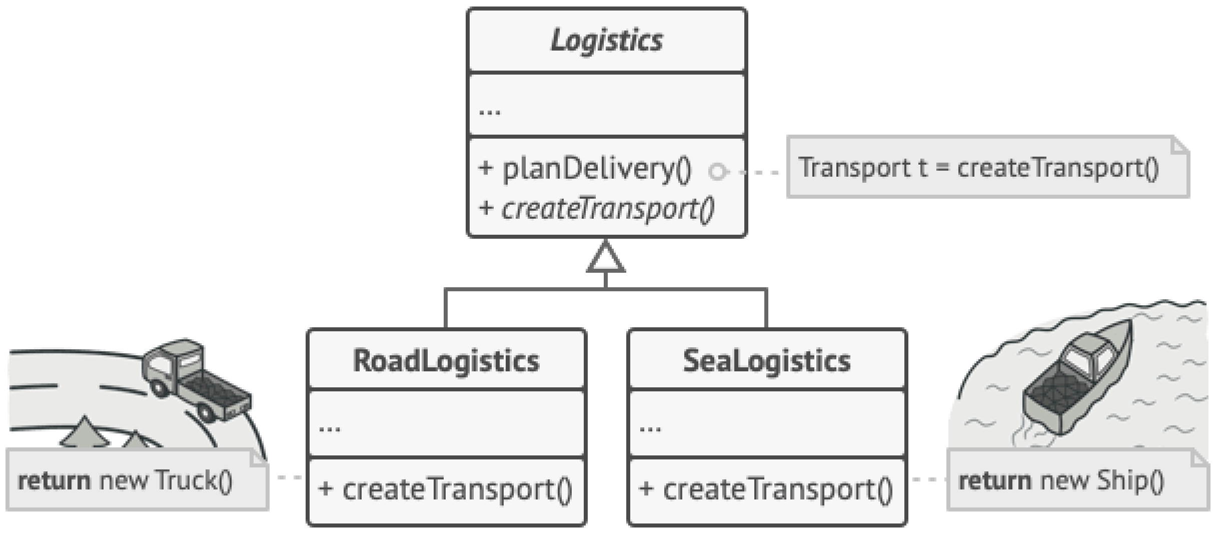
- Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
- At present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase.

LogisticsApp

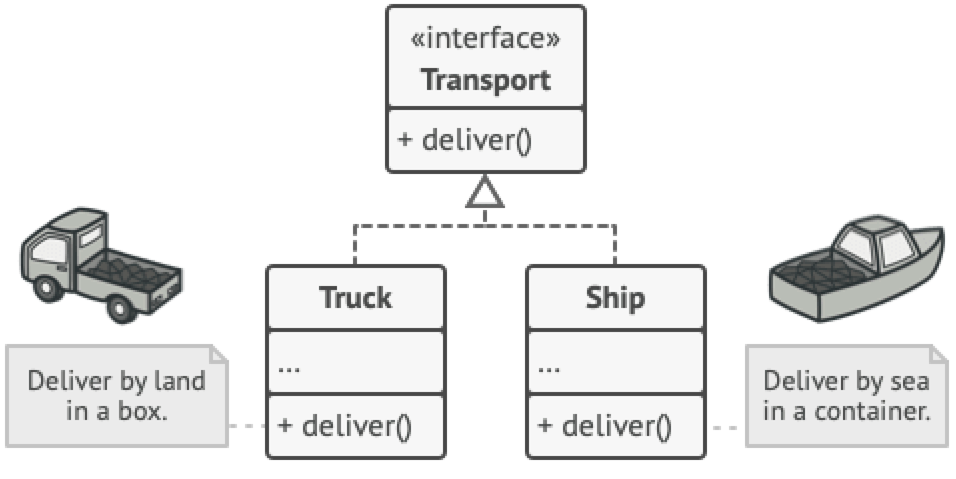
Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Solution

- The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method.



Subclasses can alter the class of objects being returned by the factory method.



All products must follow the same interface.

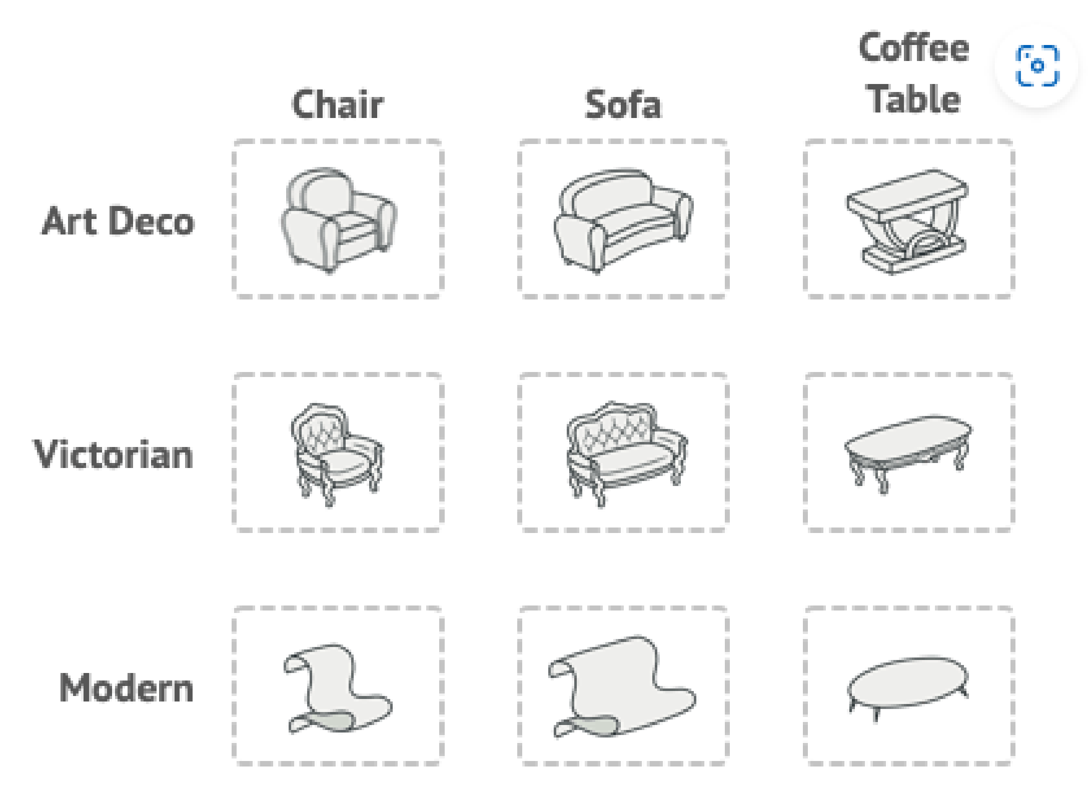
Abstract Factory

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Problem

- Imagine that you’re creating a furniture shop simulator. Your code consists of classes that represent:
- A family of related products, say: Chair + Sofa + Coffee Table.
- Several variants of this family. For example, products Chair + Sofa + Coffee Table are available in these variants: Modern, Victorian, ArtDeco.
- You need a way to create individual furniture objects so that they match other objects of the same family.

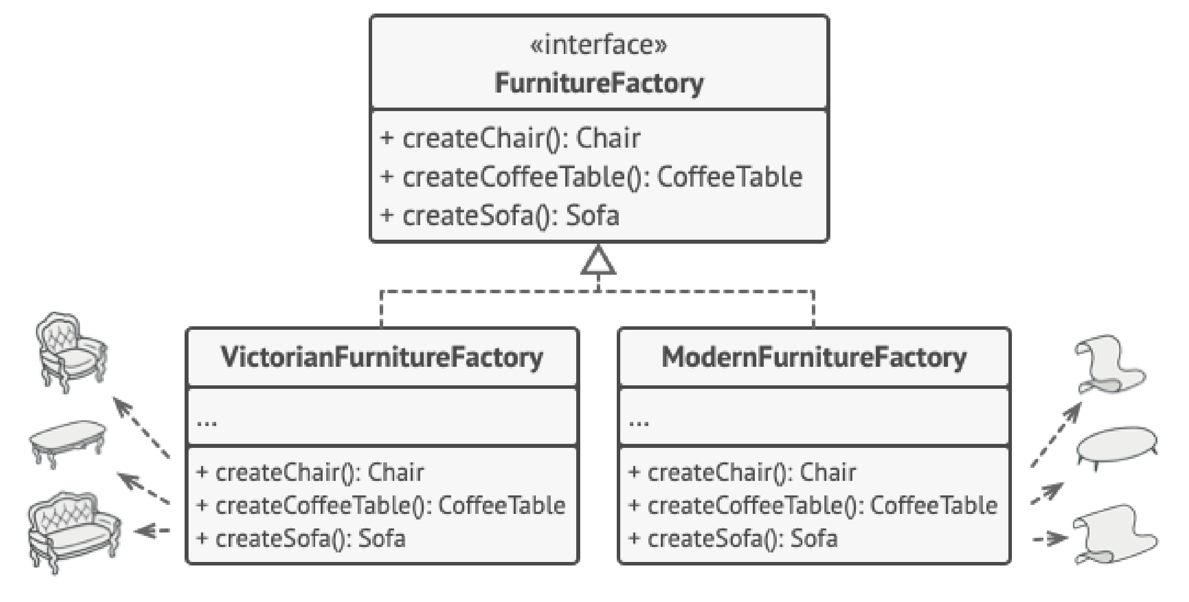
Problem



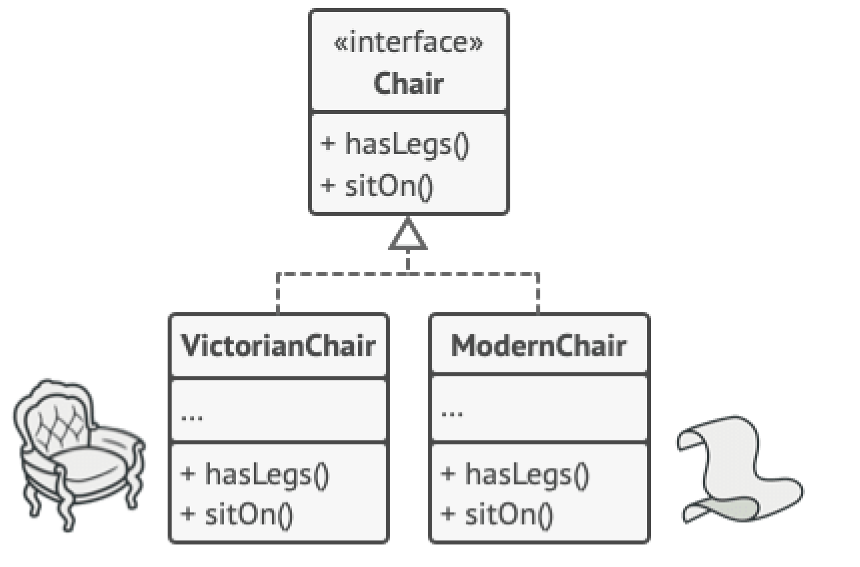
Product families and their variants.

Solution

- The first thing the Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table). Then you can make all variants of products follow those interfaces.



Each concrete factory corresponds to a specific product variant.



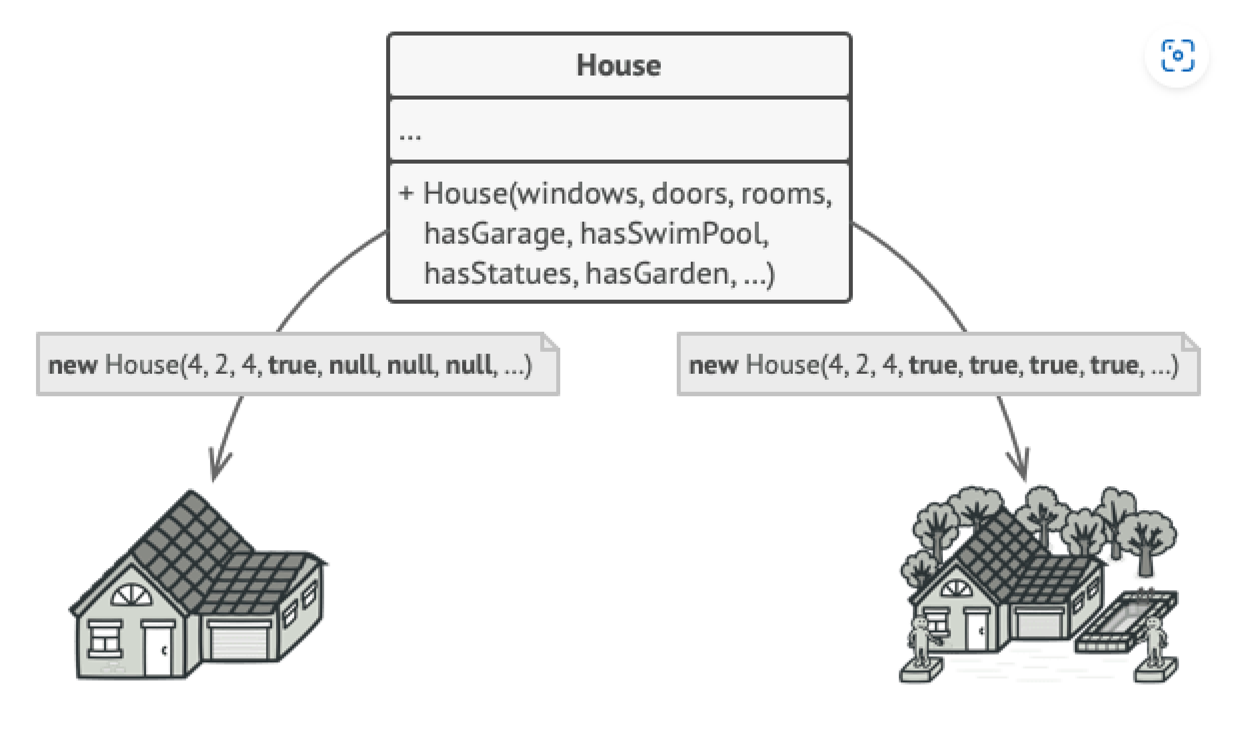
All variants of the same object must be moved to a single class hierarchy.

Builder

- Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Problem

- Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects.



```
+ House(windows, doors, rooms,
  hasGarage, hasSwimPool,
  hasStatues, hasGarden, ...)
```

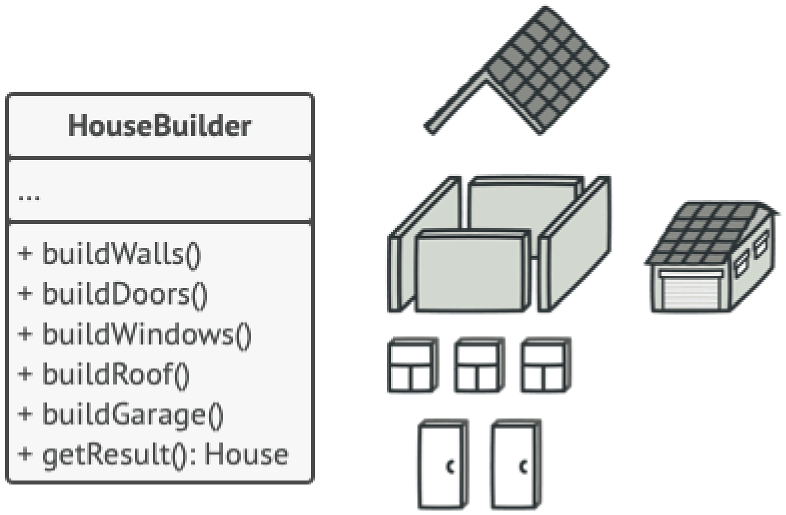
```
new House(4, 2, 4, true, null, null, null, ...)
```

```
new House(4, 2, 4, true, true, true, true, ...)
```

The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

Solution

- The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.



The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

Prototype

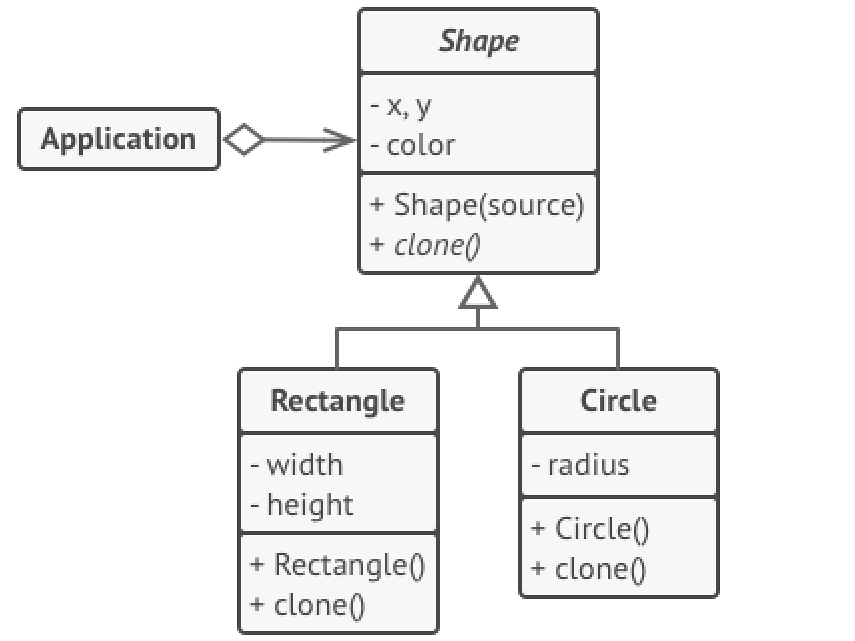
- **Prototype** is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
-

Problem

- You have an object, and you want to create an exact copy of it.
- You have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.
- Not all objects can be copied that way because some of the object’s fields may be private and not visible from outside of the object itself.
- Since you have to know the object’s class to create a duplicate, your code becomes dependent on that class.
- Sometimes you only know the interface that the object follows, but not its concrete class

Solution

- The Prototype pattern delegates the cloning process to the actual objects that are being cloned.
- The pattern declares a common interface for all objects that support cloning.
- This interface lets you clone an object without coupling your code to the class of that object.
- Usually, such an interface contains just a single clone method.

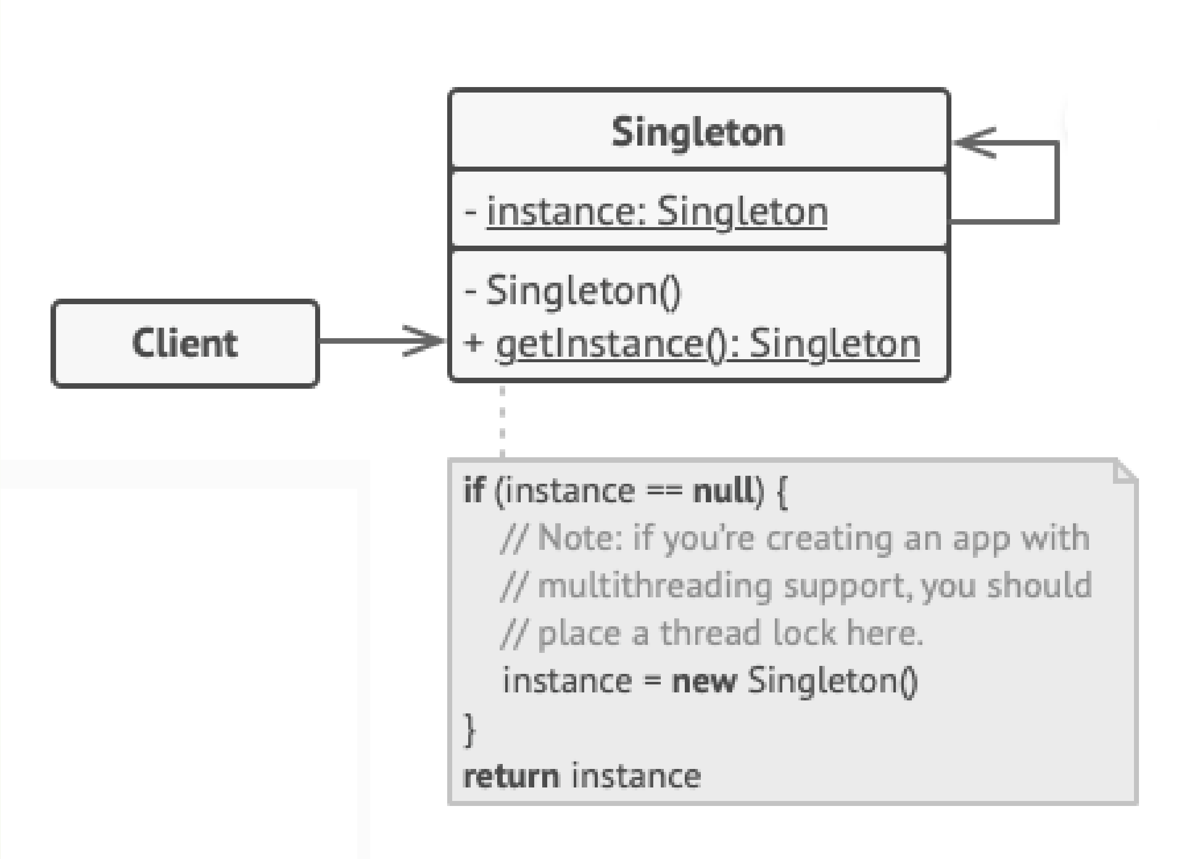


Cloning a set of objects that belong to a class hierarchy.

Singleton

- Ensure a class only has one instance, and provide a global point of access to it.
- All implementations of the Singleton have these two steps in common:
  - Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
  - Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

Singleton



```
if (instance == null) {
   // Note: if you're creating an app with
   // multithreading support, you should
   // place a thread lock here.
   instance = new Singleton()
}
return instance
```

Structural Pattern

- **Adapter:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge:** Decouple an abstraction from its implementation so that the two can vary independently.
- **Composite:** Compose objects into tree structures to represent whole part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator:** Attach additional responsibilities to an object dynamically.
- **Façade:** Provide a unified interface to a set of interfaces in a subsystem.
- **Flyweight:** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy:** Provide a surrogate or placeholder for another object control access to it.

Adapter

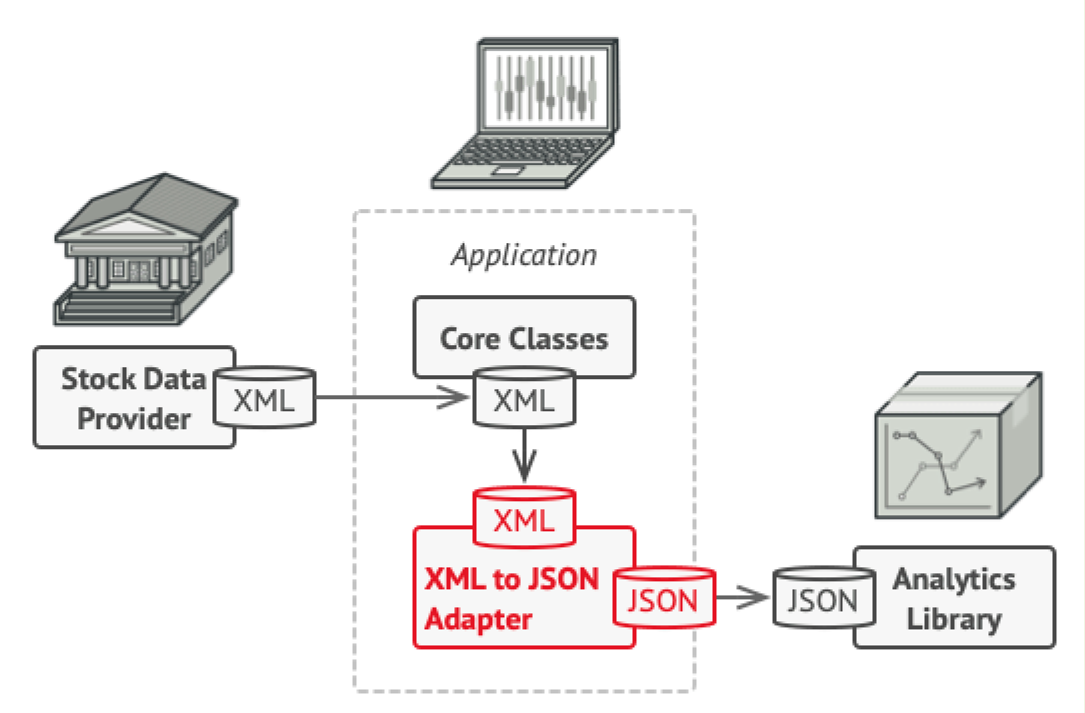
- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.

Solution

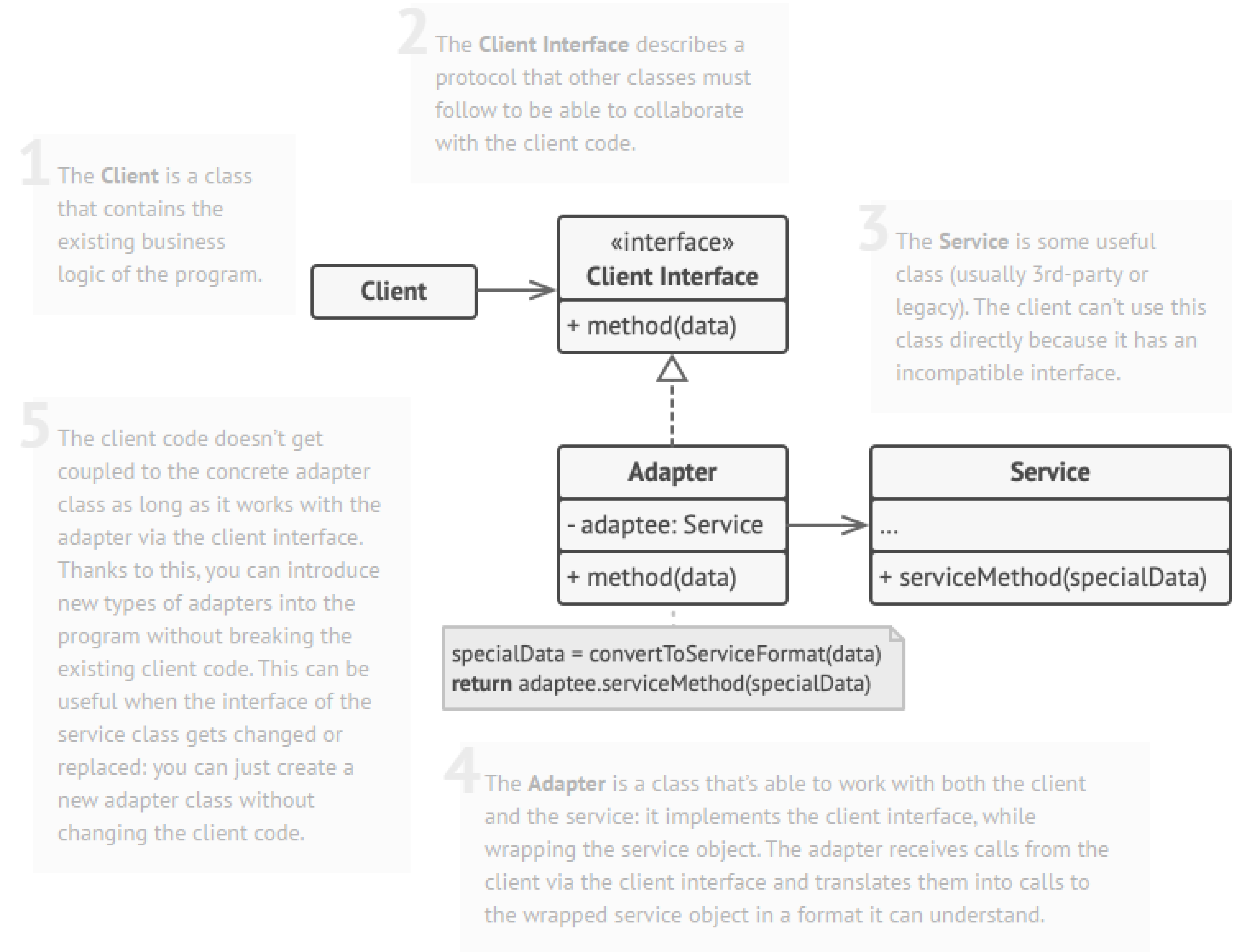
- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter.



Solution

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:
  - The adapter gets an interface, compatible with one of the existing objects.
  - Using this interface, the existing object can safely call the adapter's methods.
  - Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Solution



1 The **Client** is a class that contains the existing business logic of the program.

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

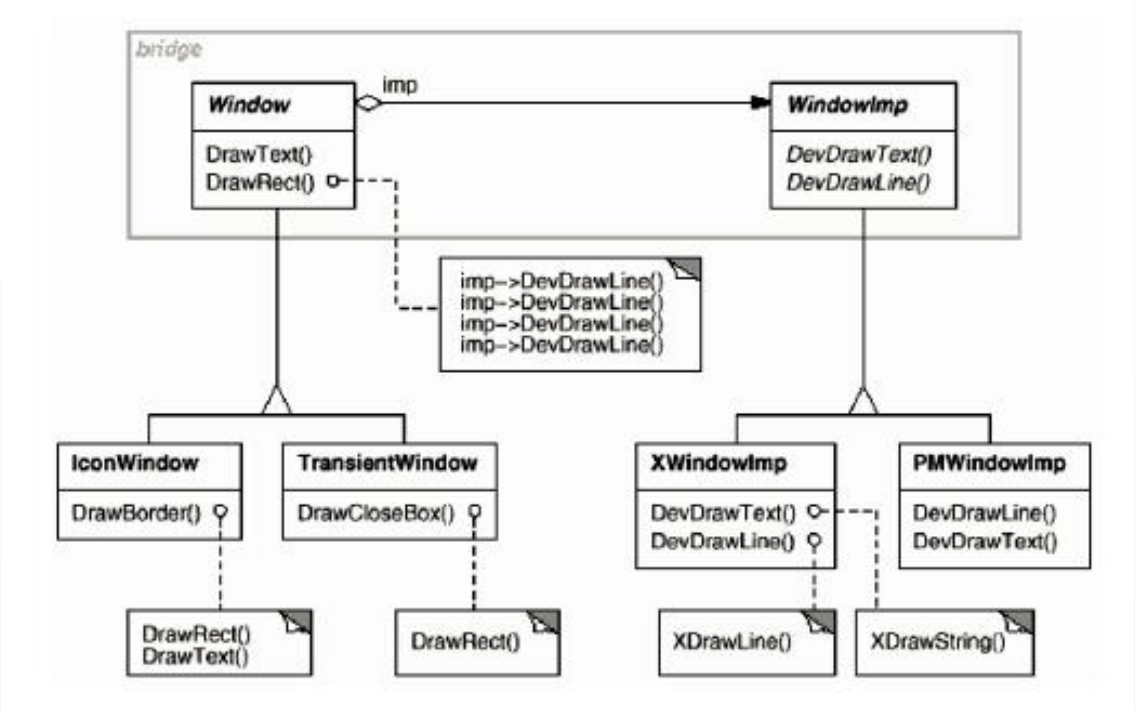
4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

```
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)
```

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.



Bridge: Applicability

- you want to avoid a permanent binding between an abstraction and its implementation; for example, when the implementation must be selected or switched at run-time.
- both the abstractions and their implementations should be extensible by sub-classing; combine different abstractions and implementations and extend them independently.
- you want to share an implementation among multiple objects and this fact should be hidden from the client.

Composite

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

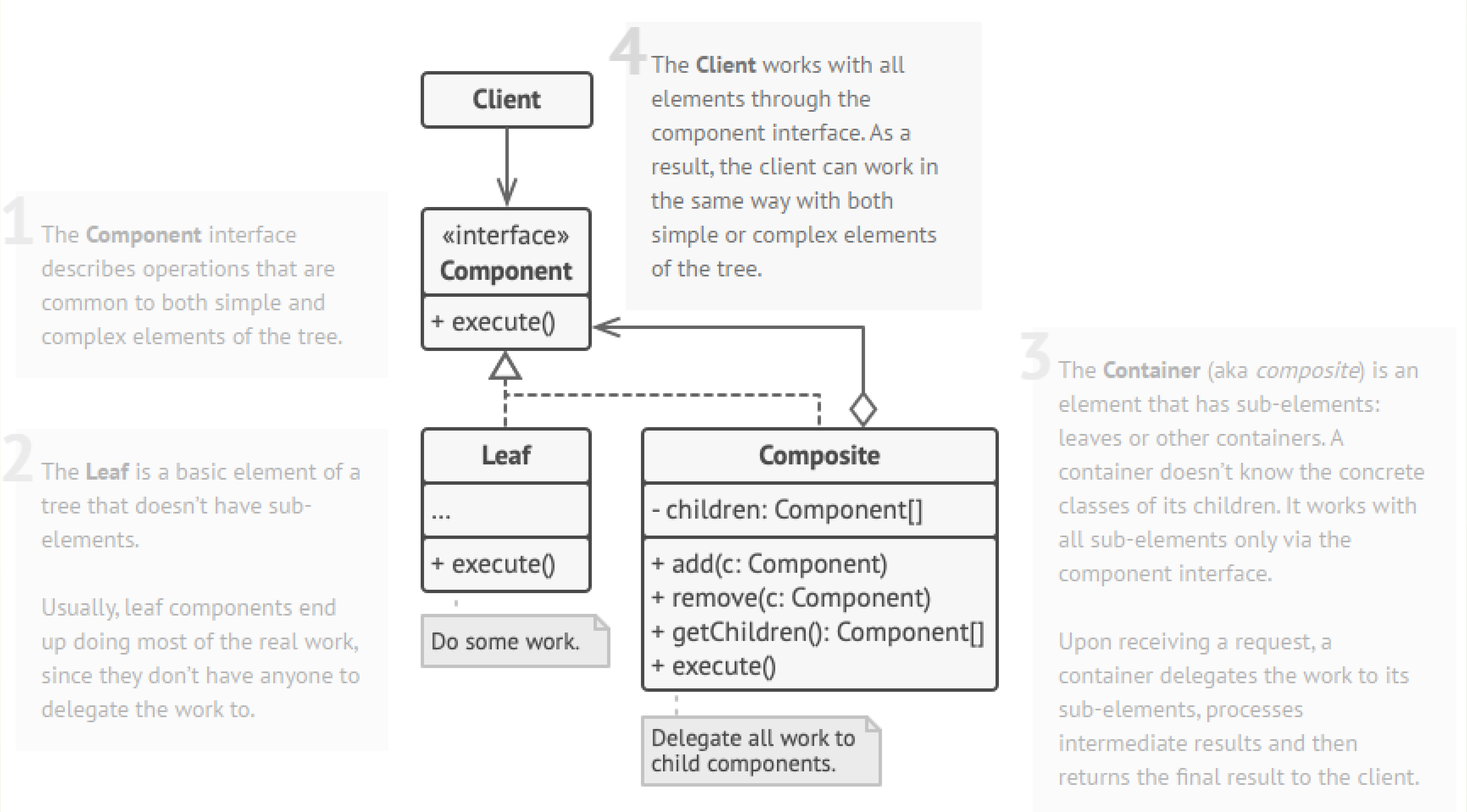
Problem

- imagine that you have two types of objects: Products and Boxes. A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.
- Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order

Solution

- The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.
- How would this method work?
- For a product, it'd simply return the product's price. For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.

Solution



1 The **Component** interface describes operations that are common to both simple and complex elements of the tree.

2 The **Leaf** is a basic element of a tree that doesn't have sub-elements.

Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.

3 The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

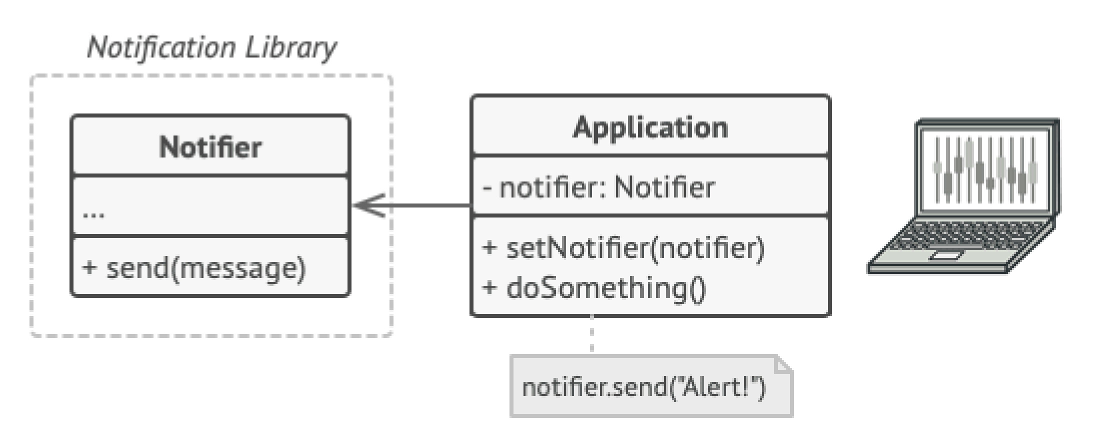
4 The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

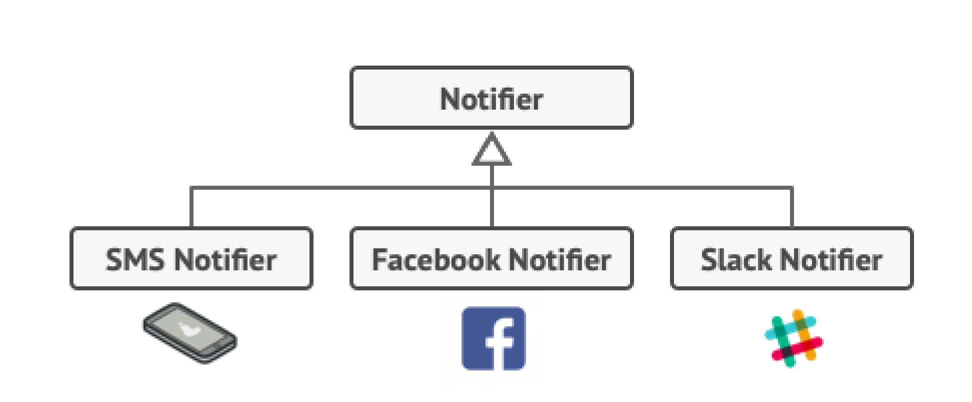
Decorator

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem

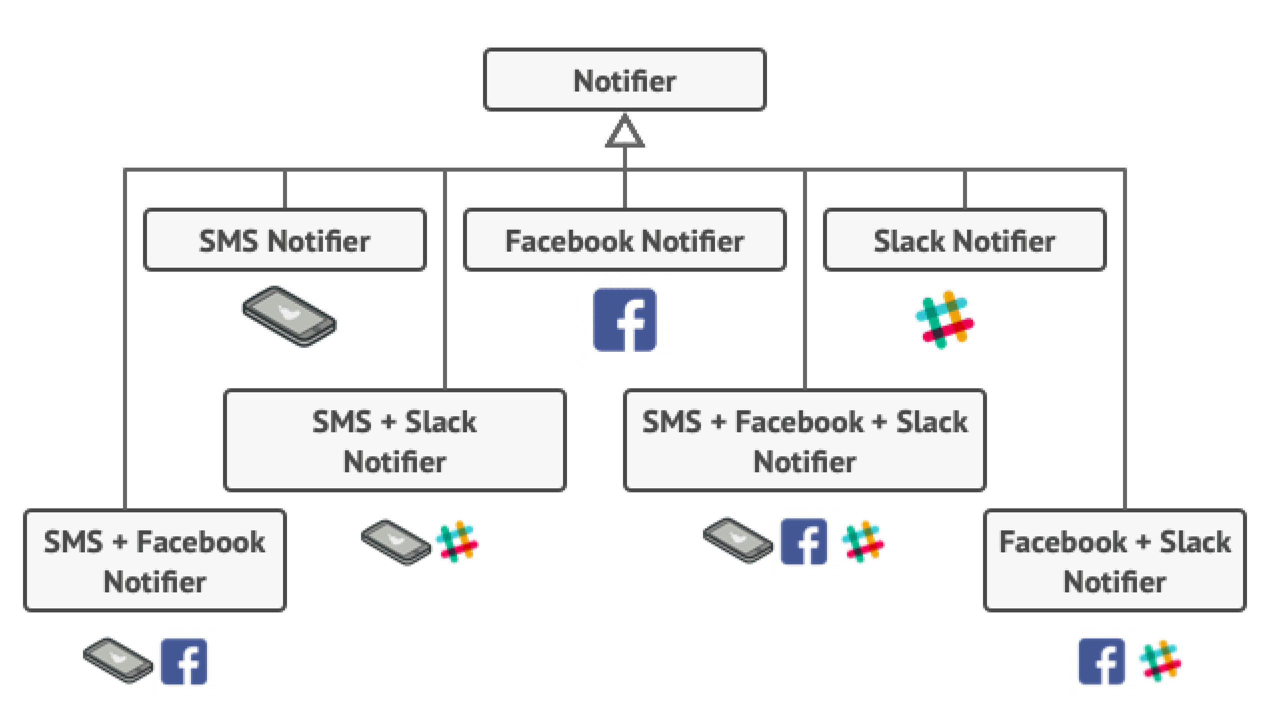
- Extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses.





Problem

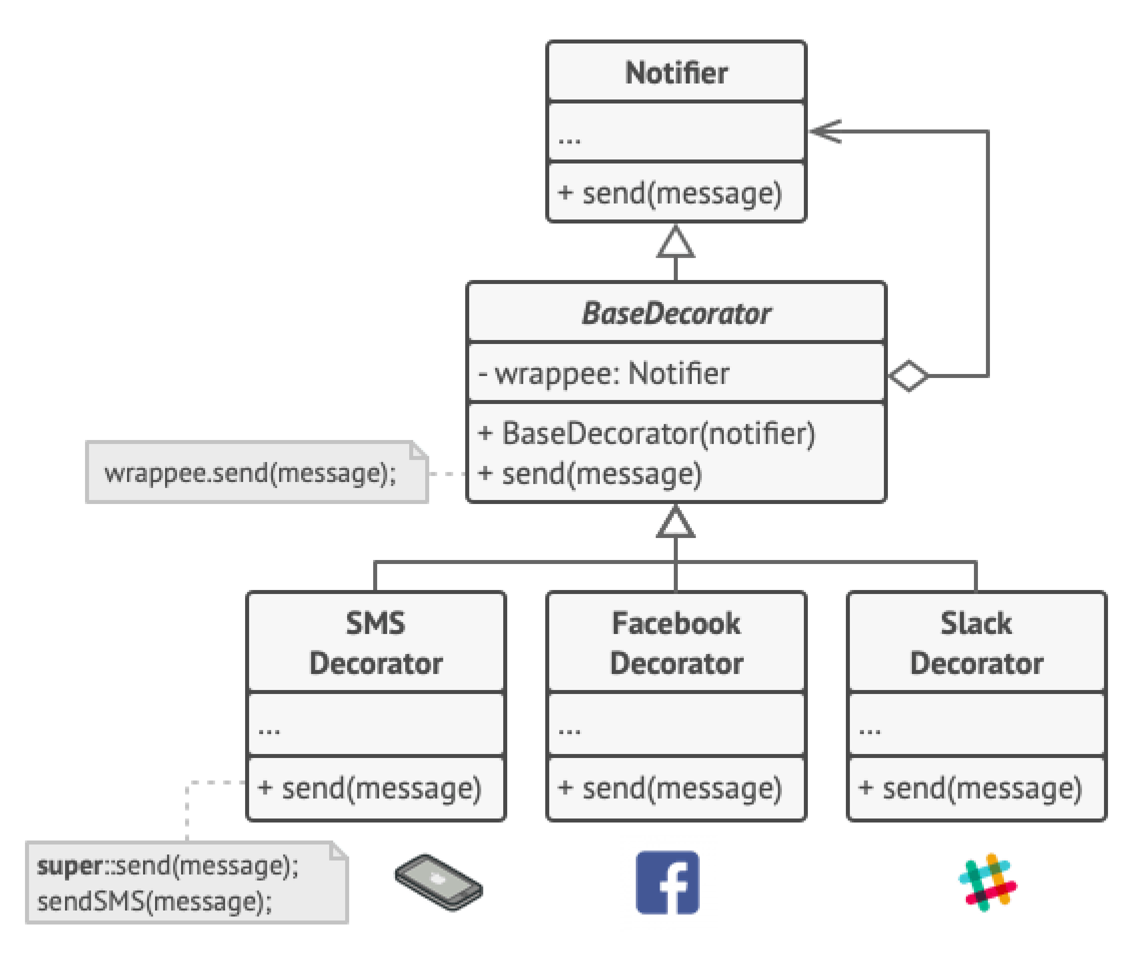
- Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.
- Inheritance is static. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
- Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.



Combinatorial explosion of subclasses.

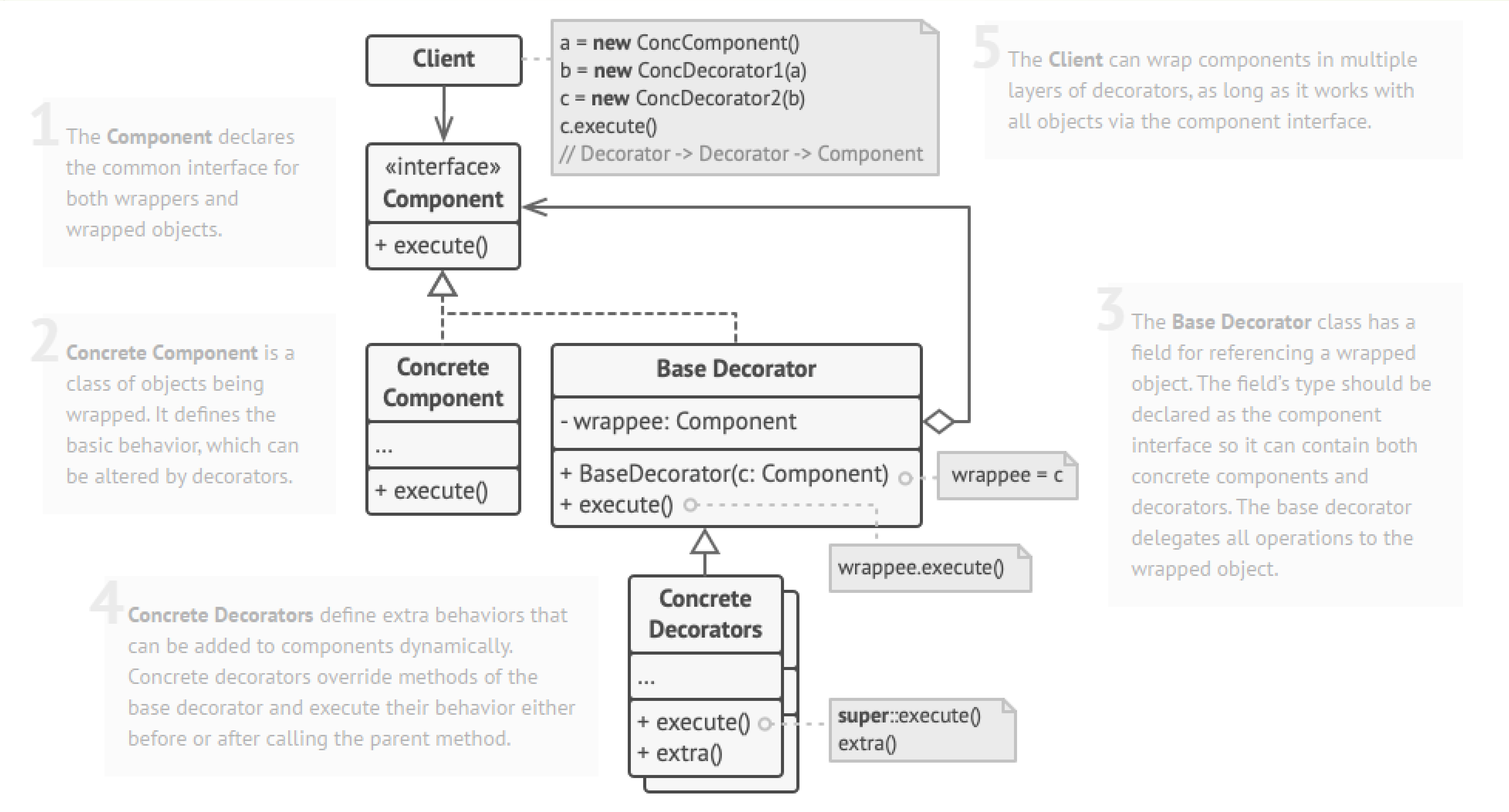
Solution

- One of the ways to overcome these caveats is by using Aggregation or Composition instead of Inheritance.



Various notification methods become decorators.

Solution



```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

1 The **Component** declares the common interface for both wrappers and wrapped objects.

2 **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3 The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

4 **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

5 The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

Façade

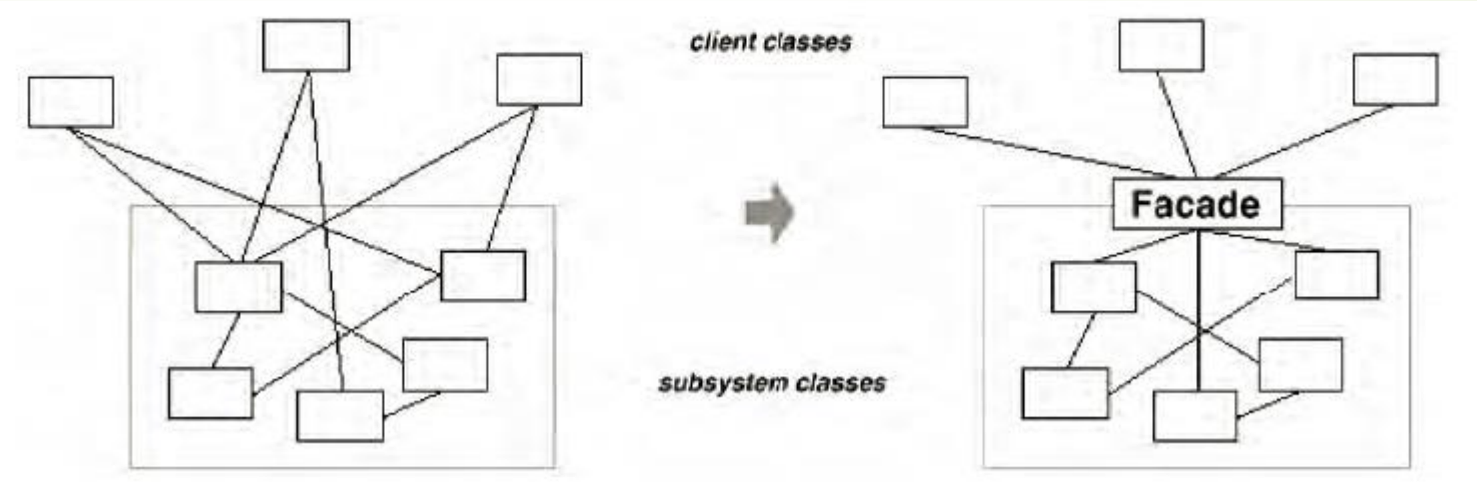
- Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.
- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Problem

- Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework.
- Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.
- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

Solution

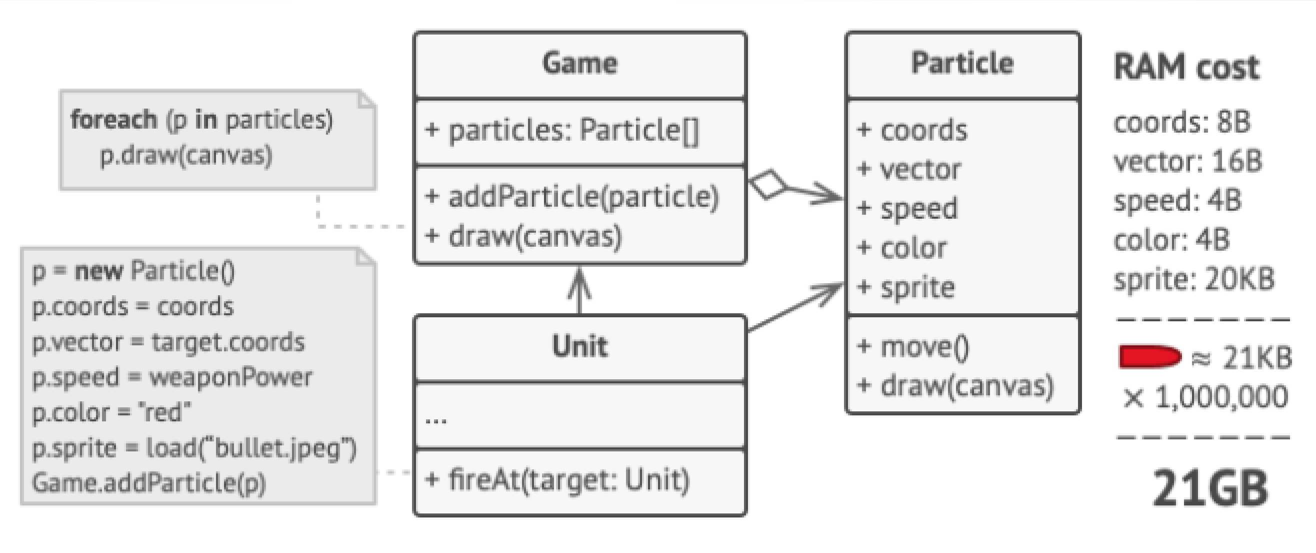
- A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts.
- A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.



Flyweight

- Use sharing to support large numbers of fine-grained objects efficiently.
- Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Problem



```
foreach (p in particles)
    p.draw(canvas)
```

```
p = new Particle()
p.coords = coords
p.vector = target.coords
p.speed = weaponPower
p.color = "red"
p.sprite = load("bullet.jpeg")
Game.addParticle(p)
```

Game

+ particles: Particle[]

+ addParticle(particle)
+ draw(canvas)

Unit

...

+ fireAt(target: Unit)

Particle

+ coords
+ vector
+ speed
+ color
+ sprite

+ move()
+ draw(canvas)

RAM cost

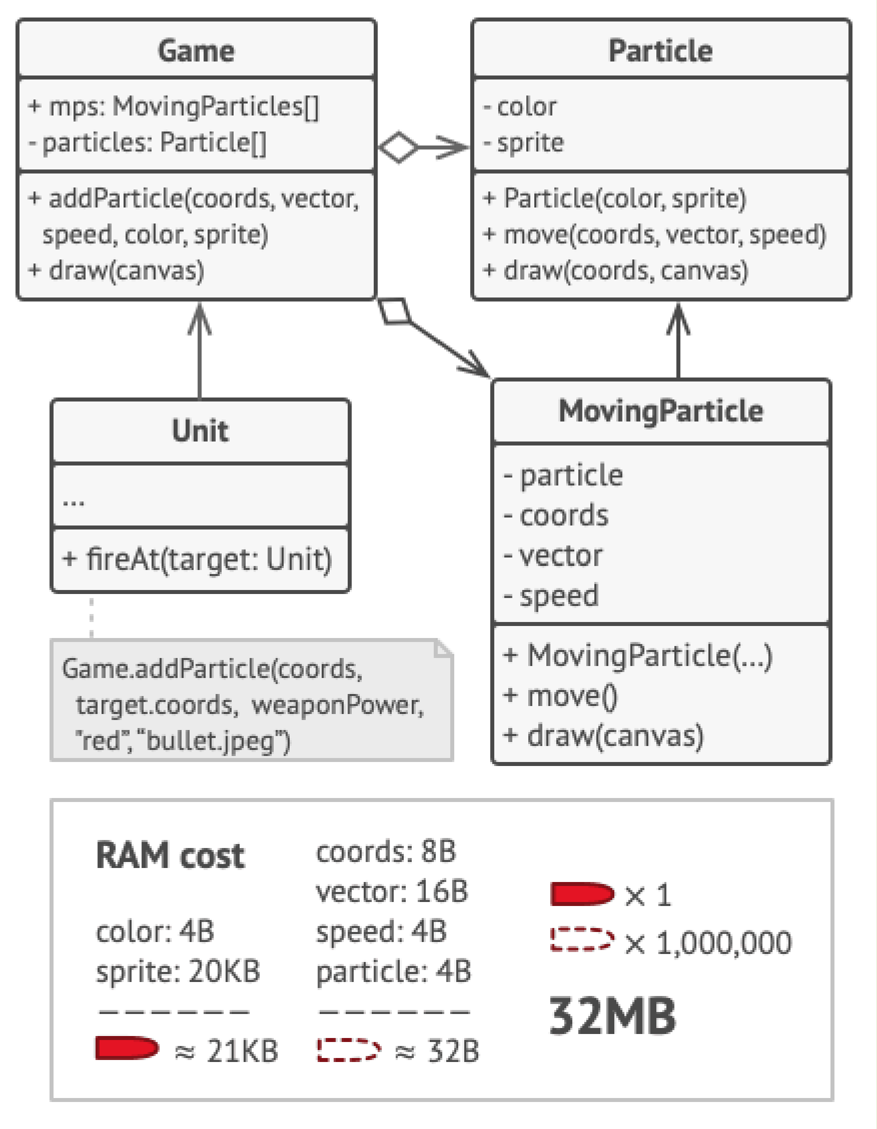
coords: 8B
vector: 16B
speed: 4B
color: 4B
sprite: 20KB

≈ 21KB
× 1,000,000

21GB

Solution

- Two fields store almost identical data across all particles. For example, all bullets have the same color and sprite.
- Other parts of a particle’s state, such as coordinates, movement vector and speed, are unique to each particle.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.



Game
+ mps: MovingParticles[]
- particles: Particle[]
+ addParticle(coords, vector, speed, color, sprite)
+ draw(canvas)

Particle
- color
- sprite
+ Particle(color, sprite)
+ move(coords, vector, speed)
+ draw(coords, canvas)

Unit
...
+ fireAt(target: Unit)

Game.addParticle(coords, target.coords, weaponPower, "red", "bullet.jpeg")

MovingParticle
- particle
- coords
- vector
- speed
+ MovingParticle(...)
+ move()
+ draw(canvas)

RAM cost

color: 4B
sprite: 20KB

🟥 ≈ 21KB

coords: 8B
vector: 16B
speed: 4B
particle: 4B

⬚ ≈ 32B

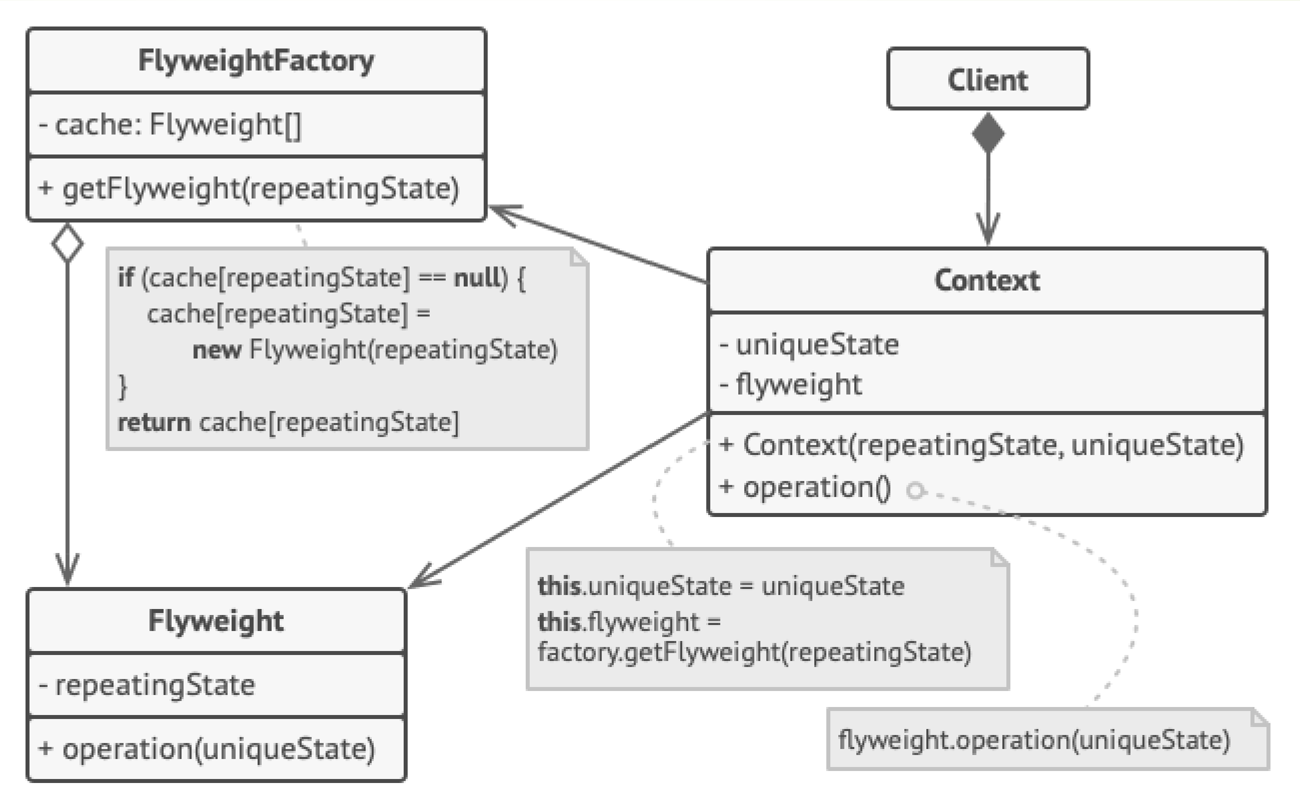
🟥 × 1
⬚ × 1,000,000

32MB

Solution

- This constant data of an object is usually called the intrinsic state. It lives within the object; other objects can only read it, not change it.
- The rest of the object's state, often altered "from the outside" by other objects, is called the extrinsic state.
- The Flyweight pattern suggests that you stop storing the extrinsic state inside the object.
- Instead, you should pass this state to specific methods which rely on it.
- Only the intrinsic state stays within the object, letting you reuse it in different contexts.
- As a result, you'd need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.

Solution



```
if (cache[repeatingState] == null) {
    cache[repeatingState] =
        new Flyweight(repeatingState)
}
return cache[repeatingState]
```

```
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
```

```
flyweight.operation(uniqueState)
```

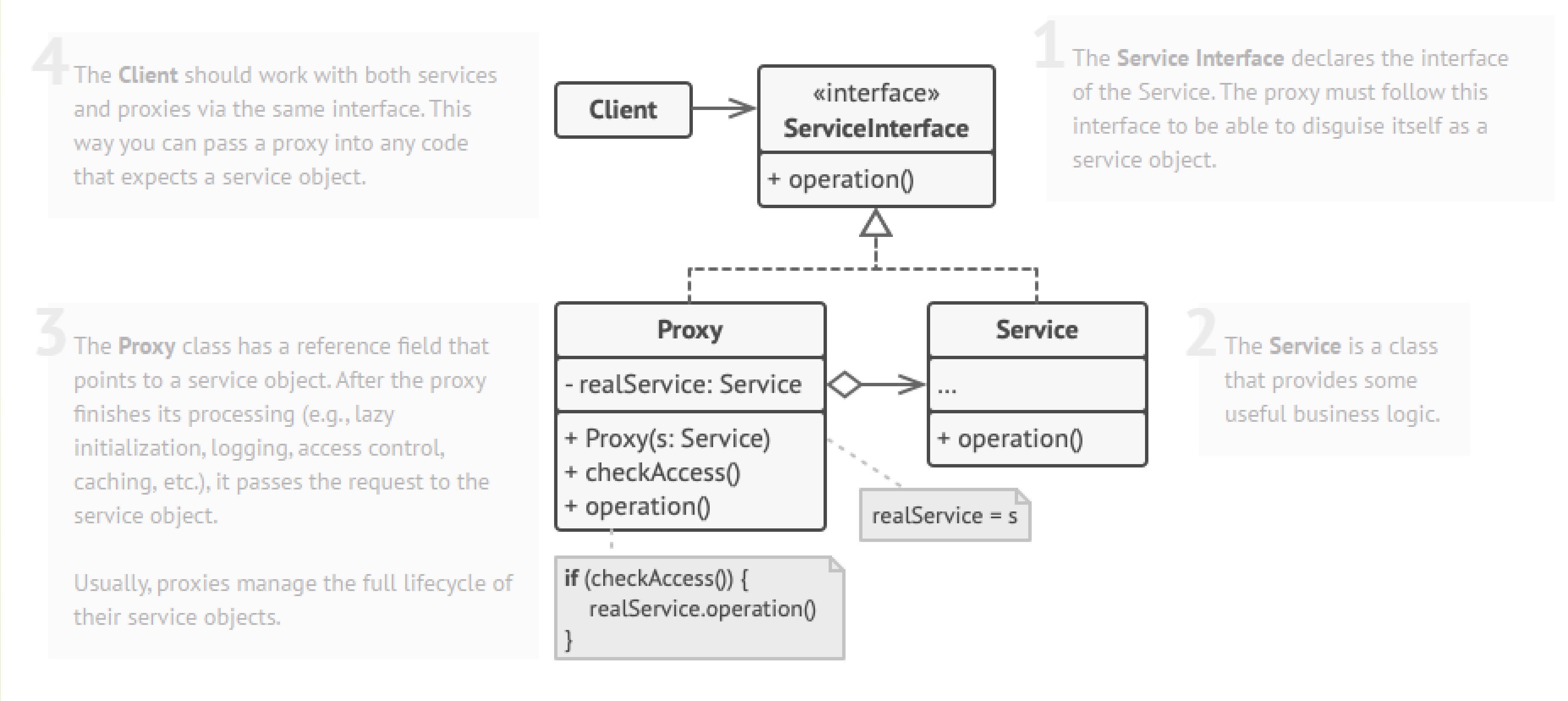
Proxy

- 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.

Solution

- The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object.
- Then you update your app so that it passes the proxy object to all of the original object’s clients.
- Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

Solution



4 The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

1 The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

3 The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

Usually, proxies manage the full lifecycle of their service objects.

2 The **Service** is a class that provides some useful business logic.

```
if (checkAccess()) {
  realService.operation()
}
```

Behavioral Patterns

- **Chain of Responsibility:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Iterator:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator:** Define an object that encapsulates how a set of objects interact; promotes loose coupling by keeping objects from referring to each other explicitly.

Behavioral Patterns (2)

- **Memento:** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State:** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy:** Define a family of algorithms, encapsulate each one, and make them interchangeable; lets the algorithm vary independently from clients that use it
- Visitor: Represent an operation to be performed on the elements of an object structure; lets you define a new operation without changing the classes of the elements.

Chain of Responsibility

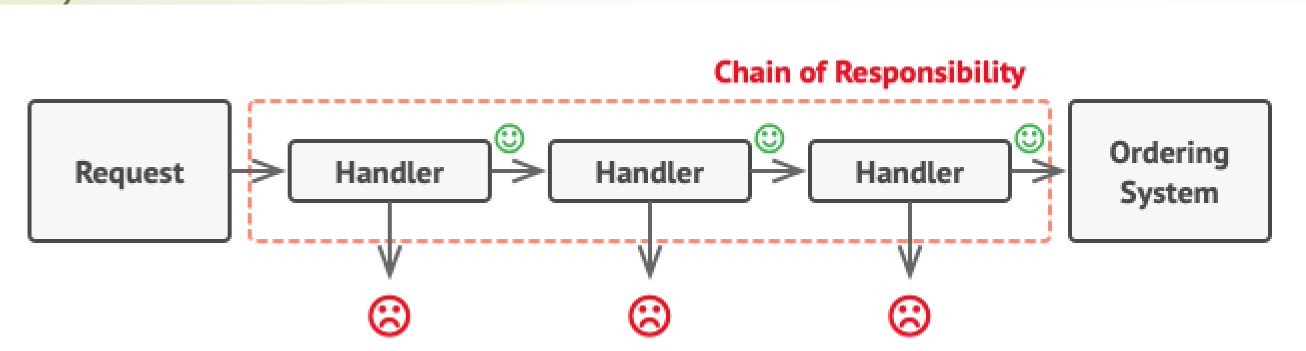
- **Chain of Responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

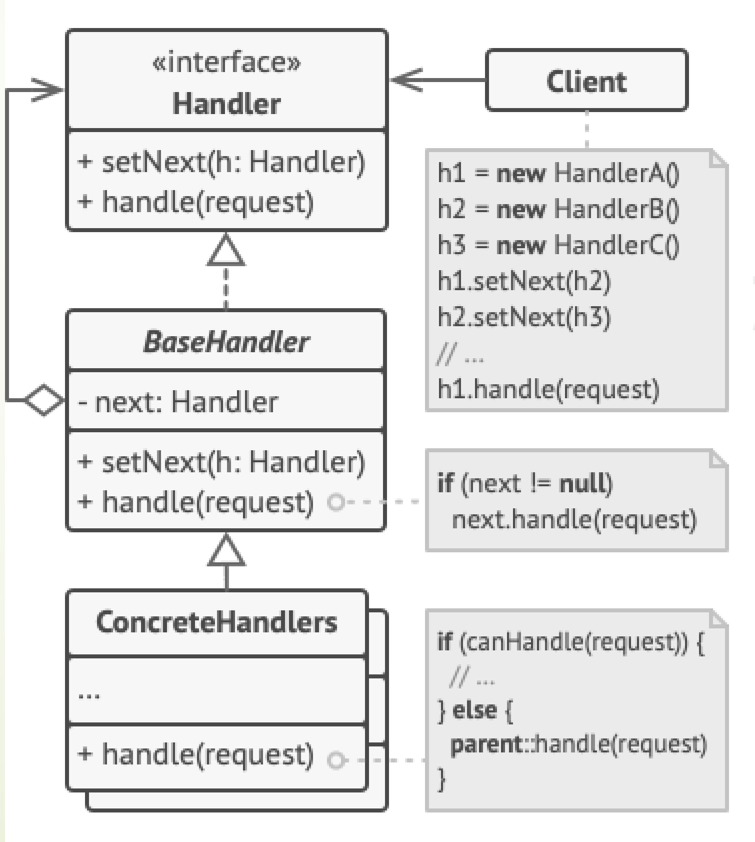
Problem

- Imagine that you’re working on an online ordering system. You want to restrict access to the system so only authenticated users can create orders. Also, users who have administrative permissions must have full access to all orders.

Solution

- In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.





```
h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
```

```
if (next != null)
  next.handle(request)
```

```
if (canHandle(request)) {
  // ...
} else {
  parent::handle(request)
}
```

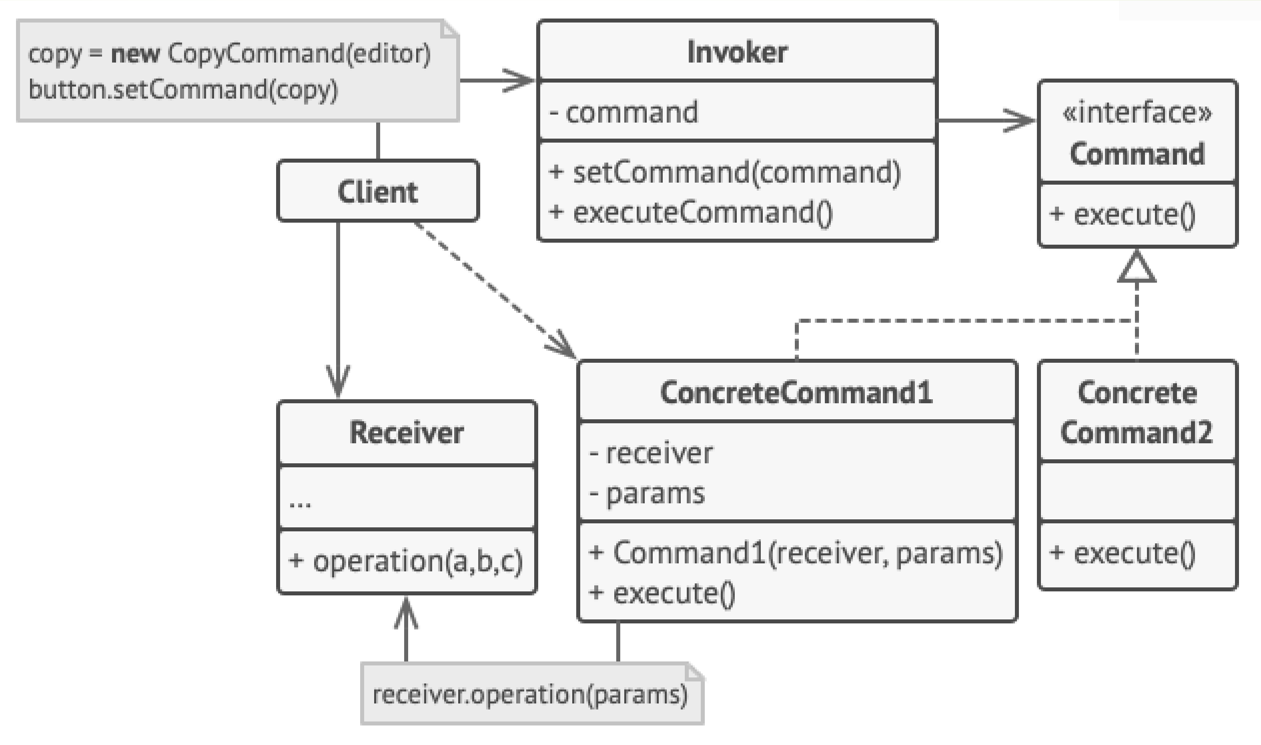
Command

- **Command** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Problem

- Imagine that you’re working on a new text-editor app. Your current task is to create a toolbar with a bunch of buttons for various operations of the editor.

Solution


copy = new CopyCommand(editor)
button.setCommand(copy)
Invoker
- command
+ setCommand(command)
+ executeCommand()
«interface»
Command
+ execute()
Client
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete
Command2
+ execute()
receiver.operation(params)

Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.
- You can assemble commands into a composite command.
- It's easy to add new Commands, because you don't have to change existing classes.

Iterator

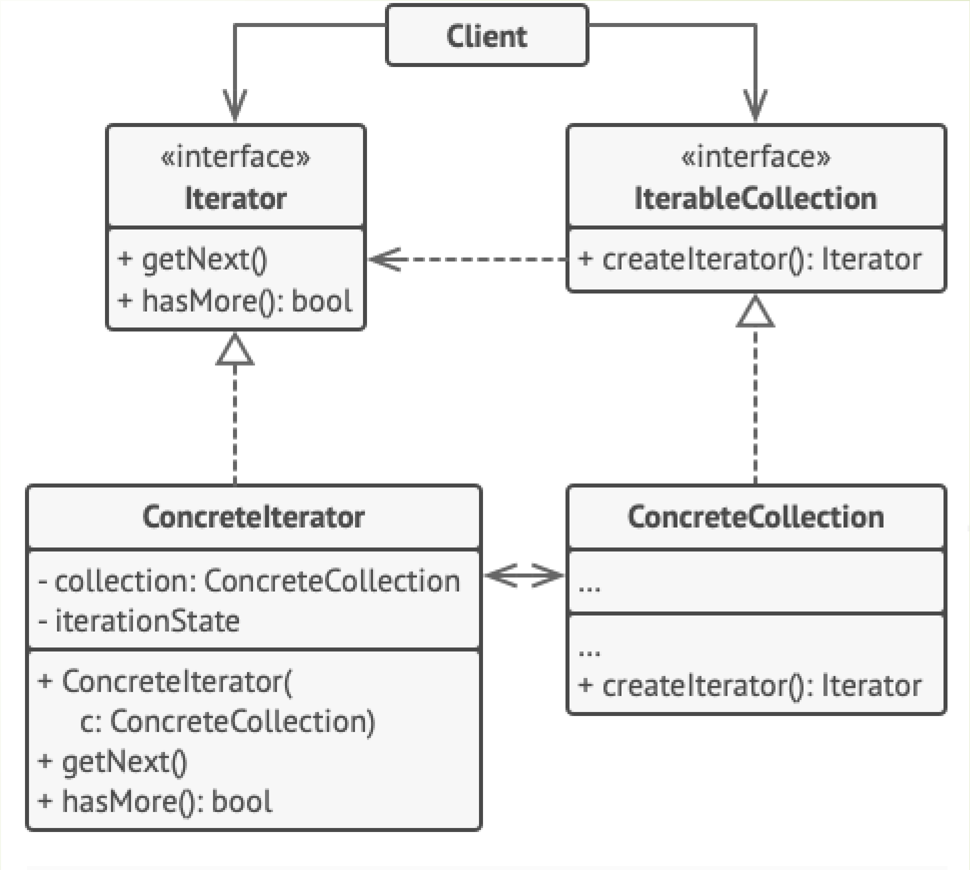
- **Iterator** is a behavioral design pattern that lets you traverse elements of a collection (an aggregate object) without exposing its underlying representation (list, stack, tree, etc.).

Problem

- Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.
- Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.
- But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements.

Solution

- The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.
- It supports variations in the traversal of an aggregate.



Mediator

- **Mediator** is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
- Define an object that encapsulates how a set of objects interact: promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.

Problem

- You have a dialog for creating and editing customer profiles. It consists of various form controls such as text fields, checkboxes, buttons, etc.
- Some of the form elements may interact with others. For instance, selecting the “I have a dog” checkbox may reveal a hidden text field for entering the dog’s name. Another example is the submit button that has to validate values of all fields before saving the data.
- By having this logic implemented directly inside the code of the form elements you make these elements’ classes much harder to reuse in other forms of the app. For example, you won’t be able to use that checkbox class inside another form, because it’s coupled to the dog’s text field.

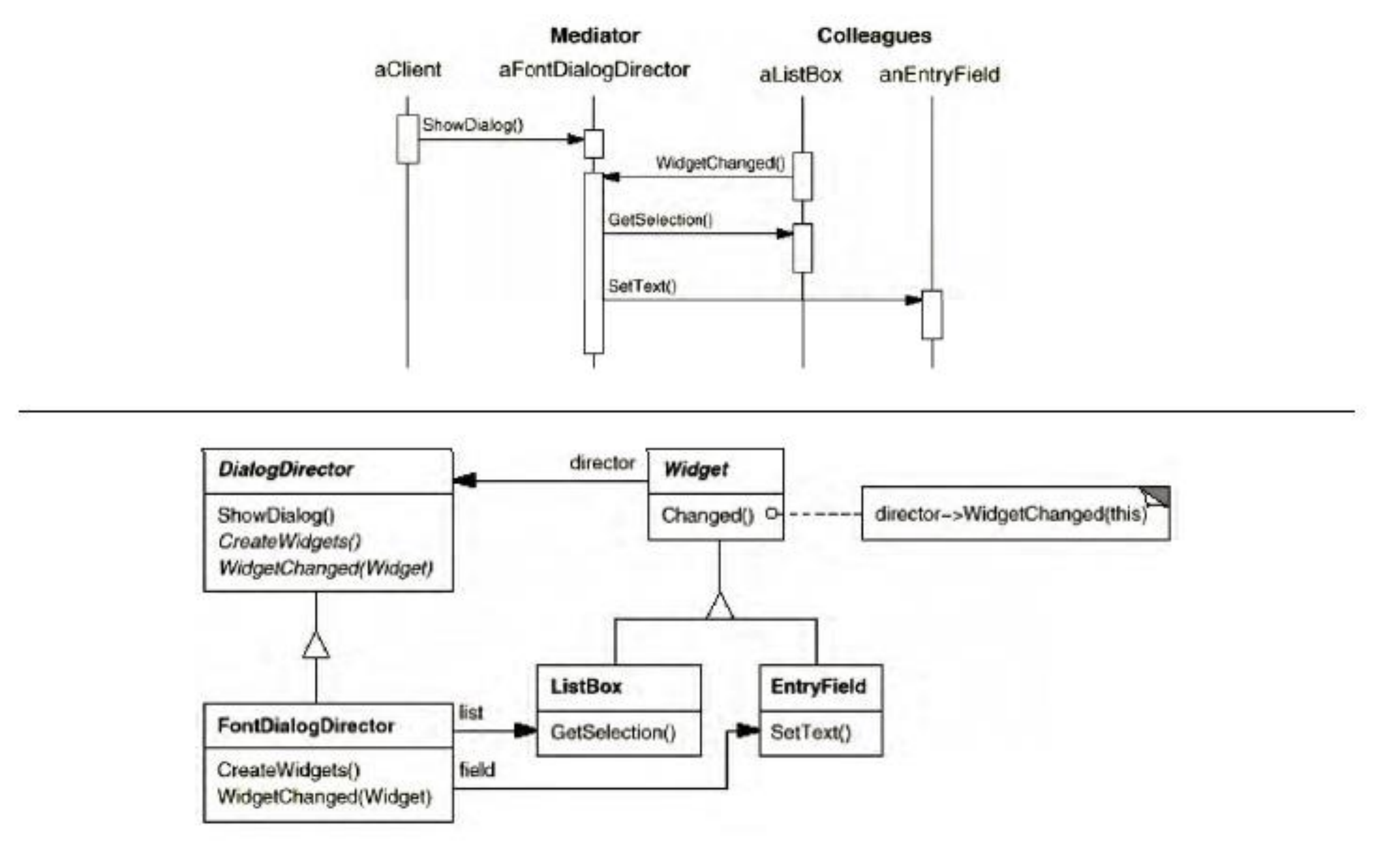
Problem

- A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- a behavior that's distributed between several classes should be customizable without a lot of sub-classing.

Solution

- The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components. As a result, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues.

Solution


Mediator
Colleagues
aClient
aFontDialogDirector
aListBox
anEntryField
ShowDialog()
WidgetChanged()
GetSelection()
SetText()
DialogDirector
ShowDialog()
CreateWidgets()
WidgetChanged(Widget)
director
Widget
Changed()
director->WidgetChanged(this)
FontDialogDirector
CreateWidgets()
WidgetChanged(Widget)
list
field
ListBox
GetSelection()
EntryField
SetText()

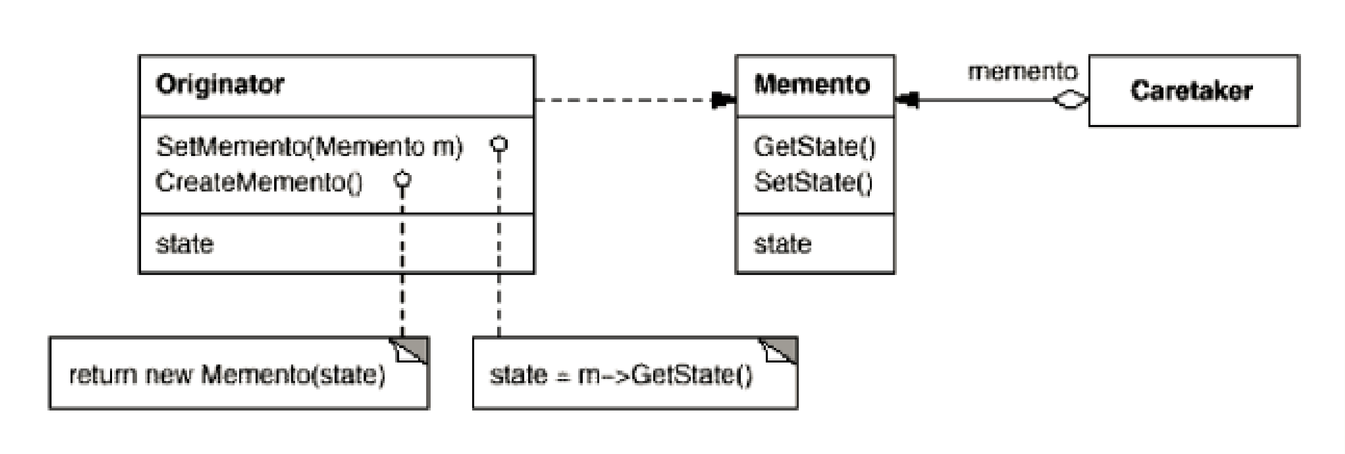
Memento

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

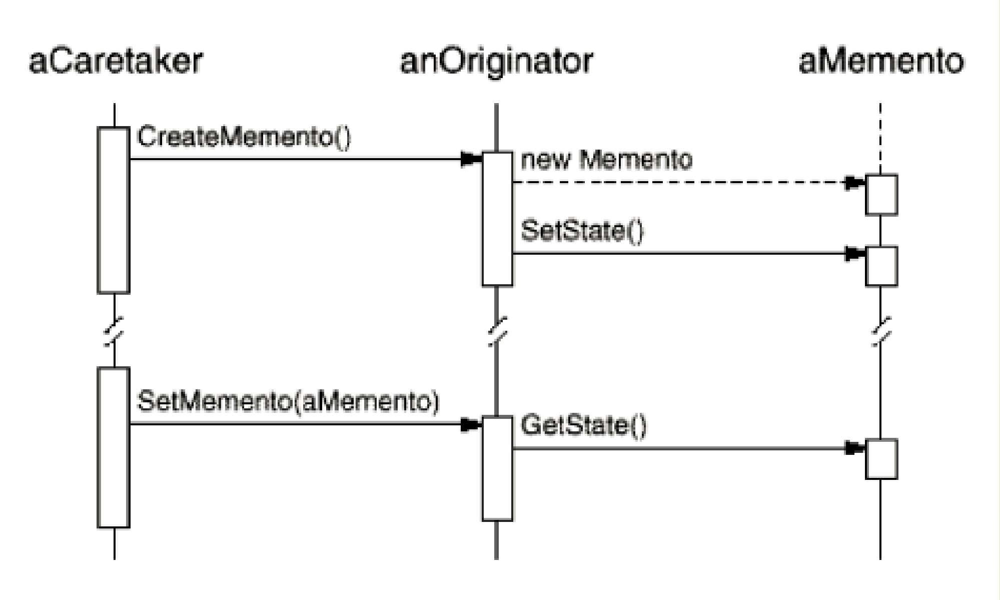
Problem

- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Solution



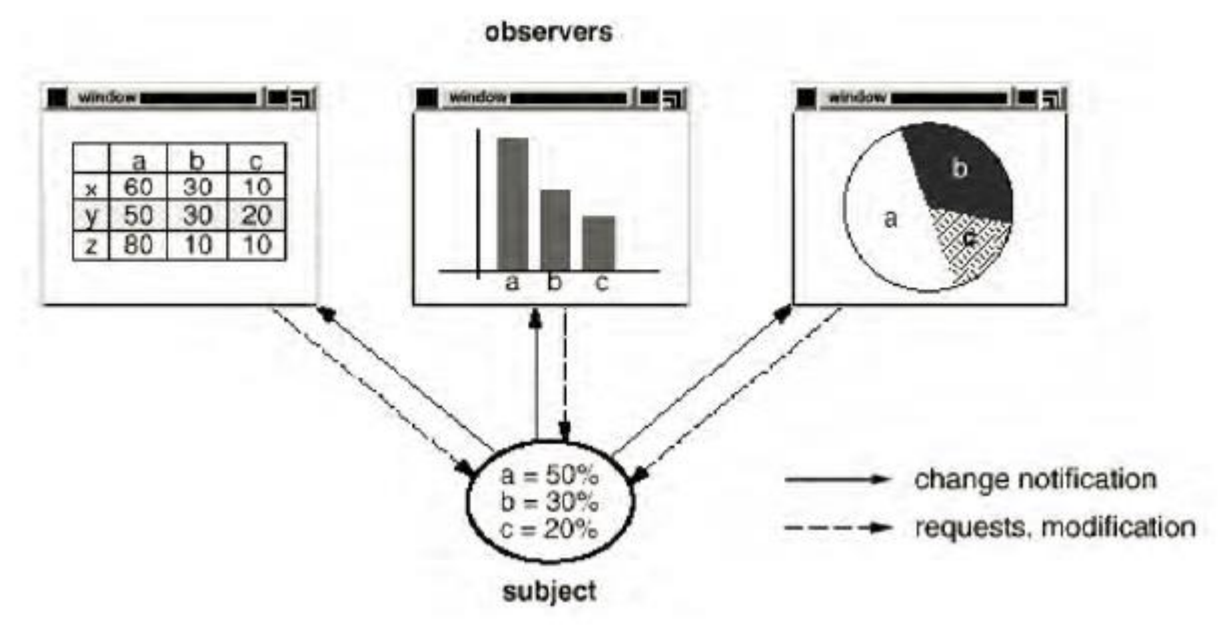
Originator
SetMemento(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
memento
Caretaker
return new Memento(state)
state = m->GetState()



aCaretaker
anOriginator
aMemento
CreateMemento()
new Memento
SetState()
SetMemento(aMemento)
GetState()

Observer

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



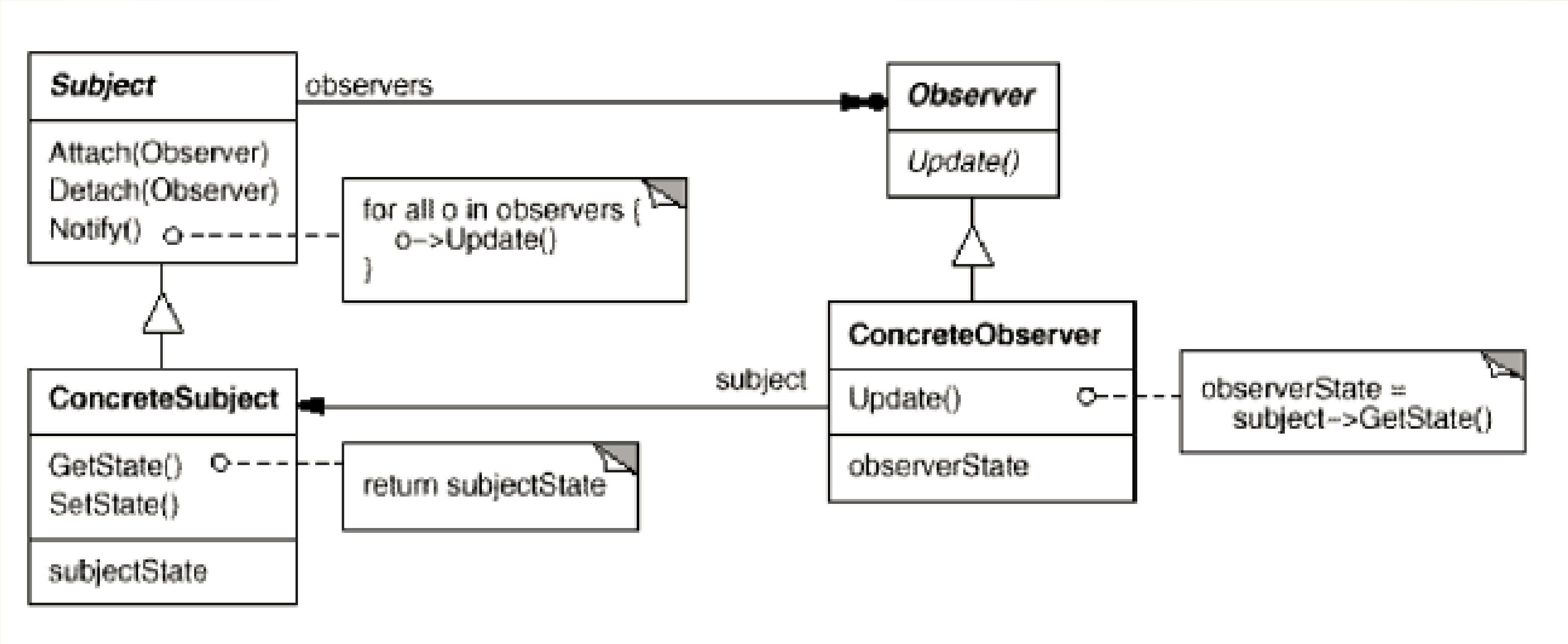
	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

a = 50%
b = 30%
c = 20%

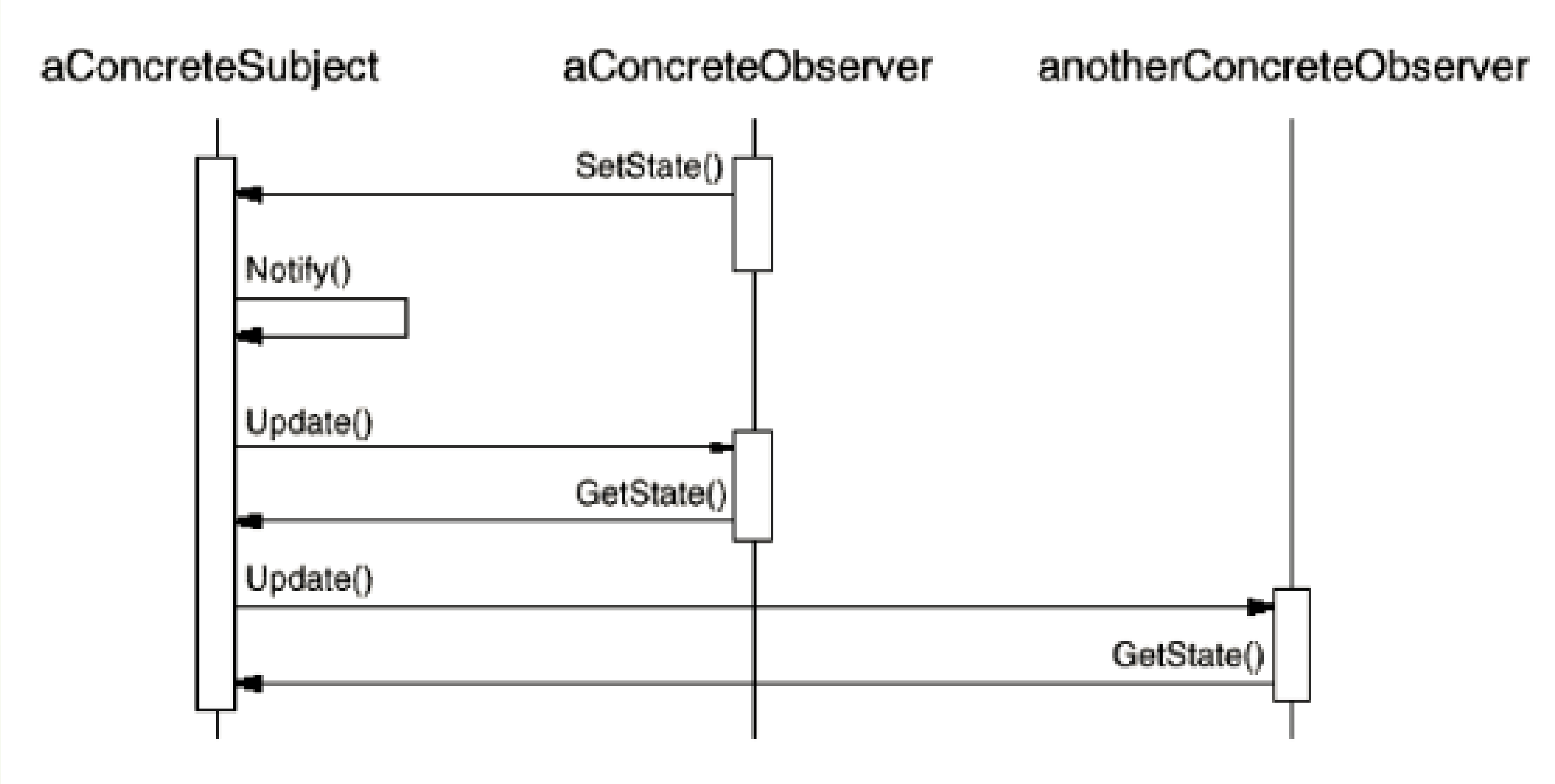
Applicability

- a change to one object requires changing others, and you don't know how many objects need to be changed.
- an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Solution

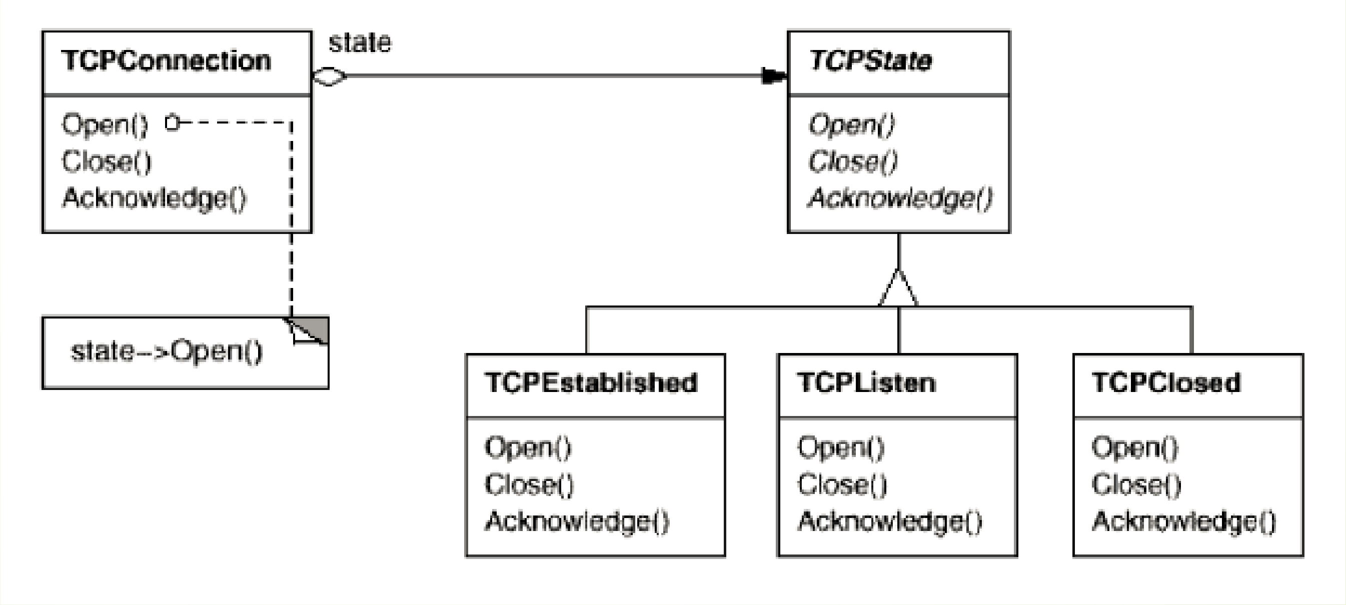

Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for all o in observers {
o->Update()
}
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState
subject
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()

Solution


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()

State

- Allow an object to alter its behavior when its internal state changes.

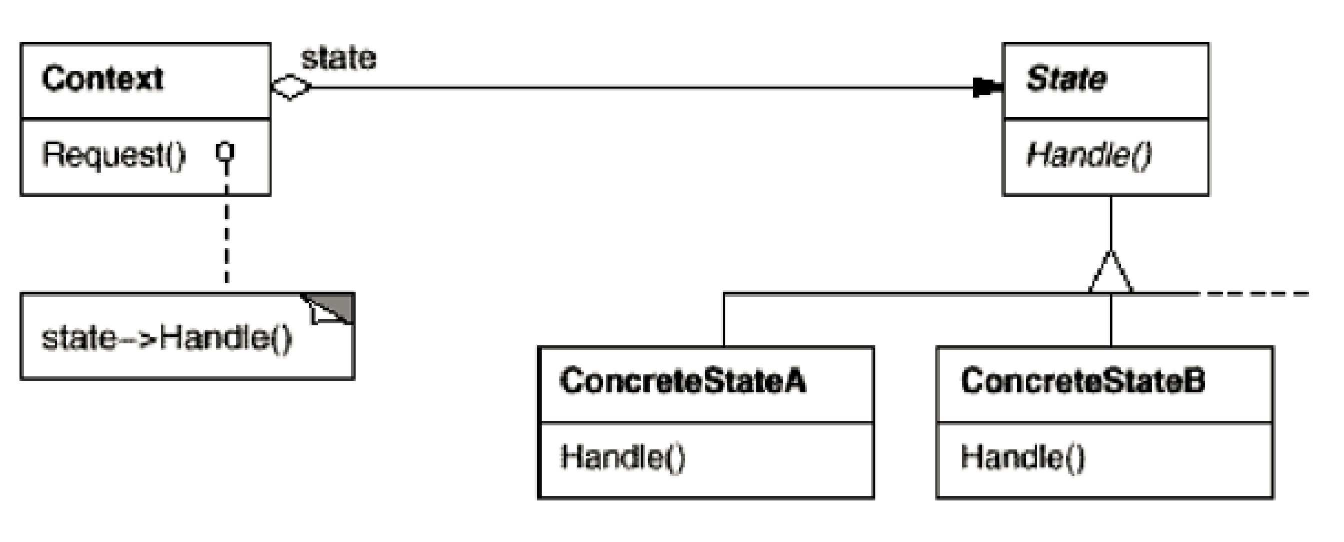


Applicability

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state.

Solution

- It localizes state-specific behavior and partitions behavior for different states. New states and transitions can be added easily by defining new subclasses.



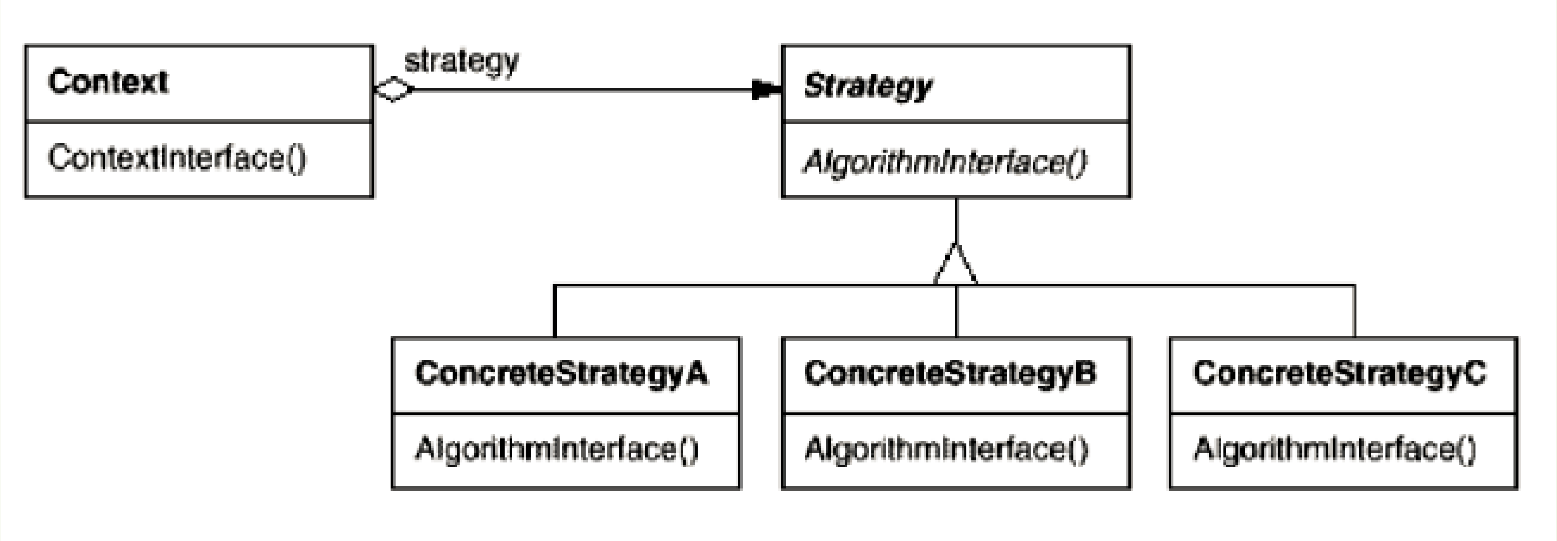
Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Applicability

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time tradeoffs.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations.

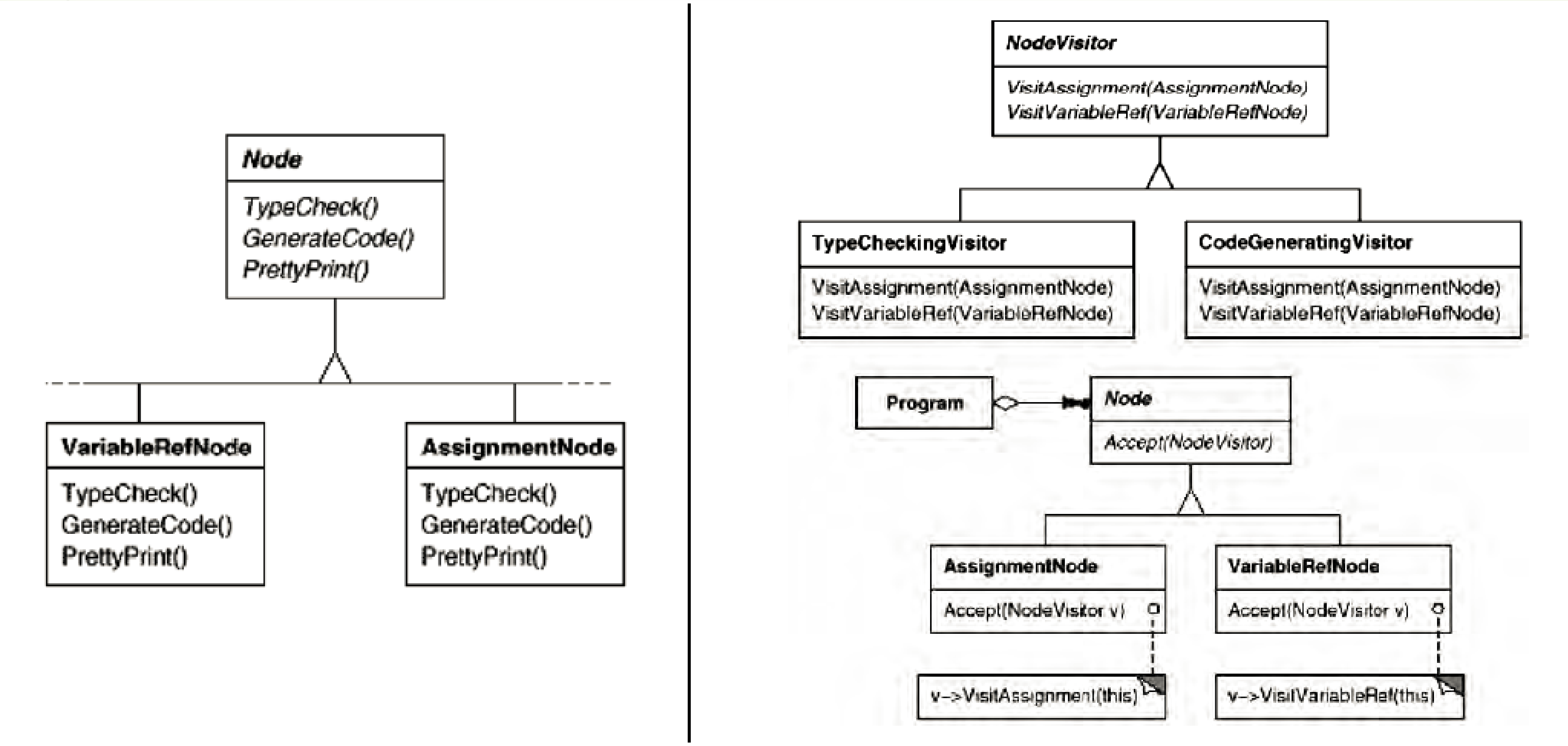
Solution


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Visitor

- Represent an operation to be performed on the elements of an object structure; lets you define a new operation without changing the classes of the elements on which it operates.

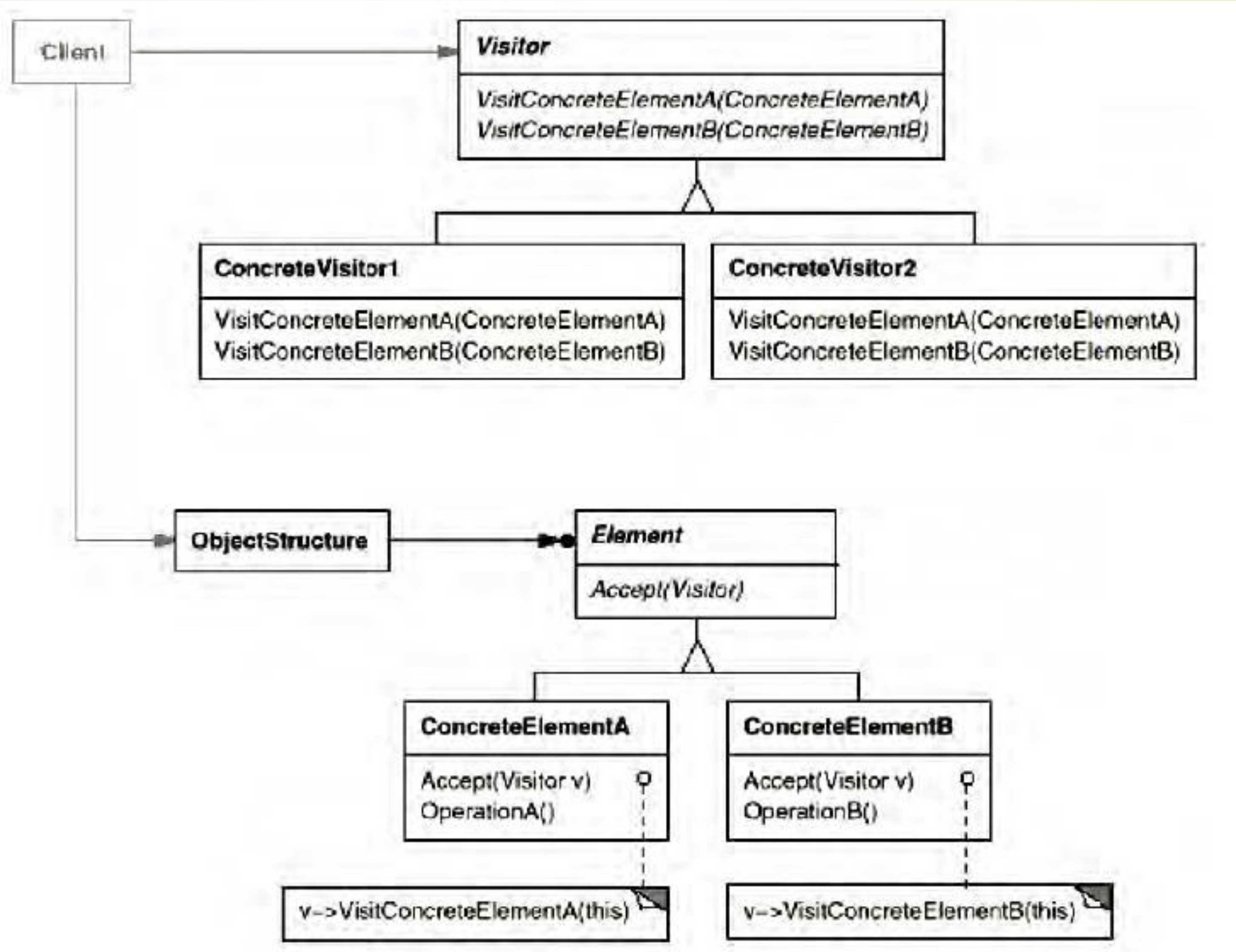
Visitor


Node
TypeCheck()
GenerateCode()
PrettyPrint()
VariableRefNode
TypeCheck()
GenerateCode()
PrettyPrint()
AssignmentNode
TypeCheck()
GenerateCode()
PrettyPrint()
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)

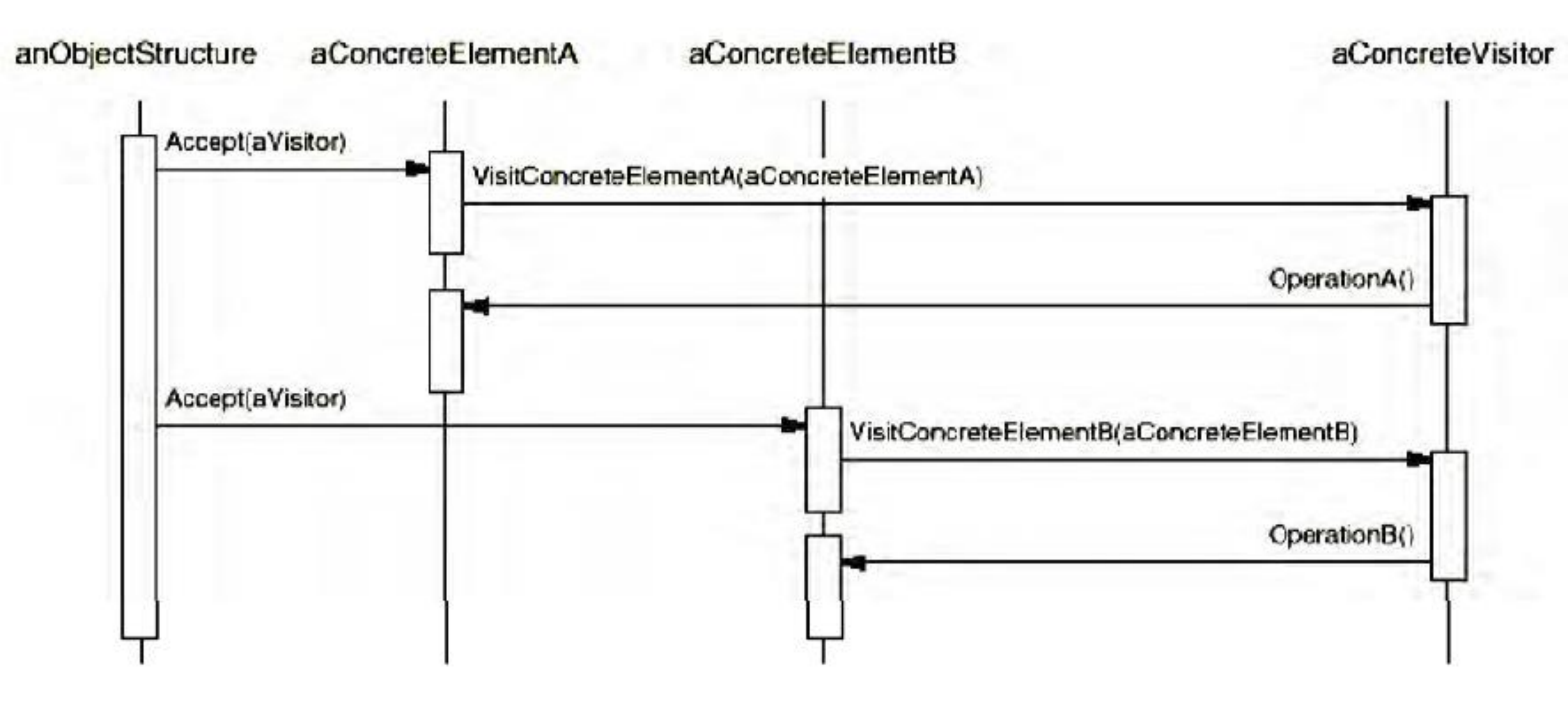
Applicability

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.

Solution


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Solution


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()

References

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995.
- Ramsin, Raman. "Home." Department of Computer Science and Engineering, Sharif University of Technology. Accessed February 15, 2025. https://sharif.edu/~ramsin/index.htm.
- Refactoring.Guru. "Design Patterns." Accessed May 4, 2025. https://refactoring.guru/design-patterns