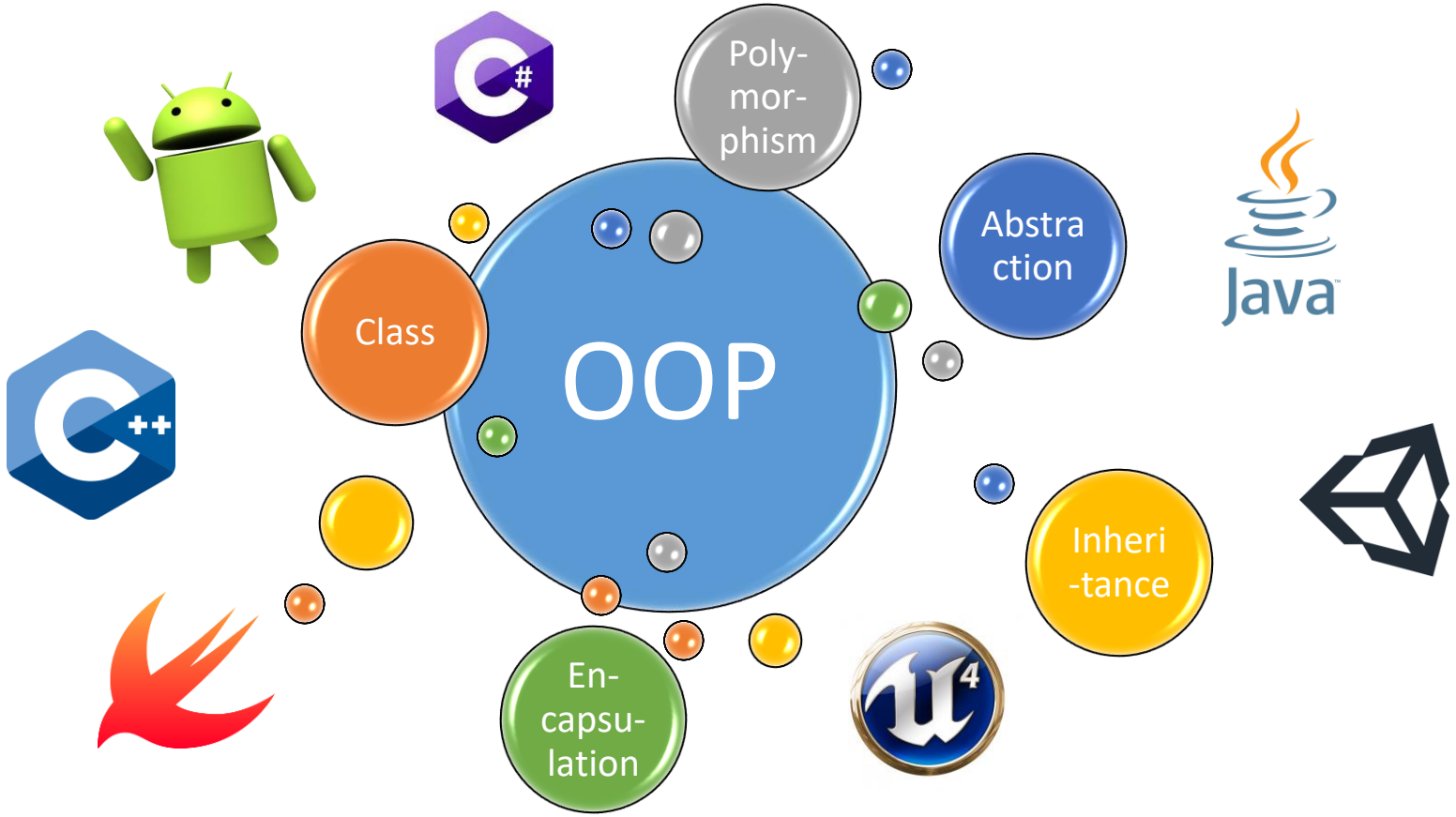


Behavioral Design Patterns

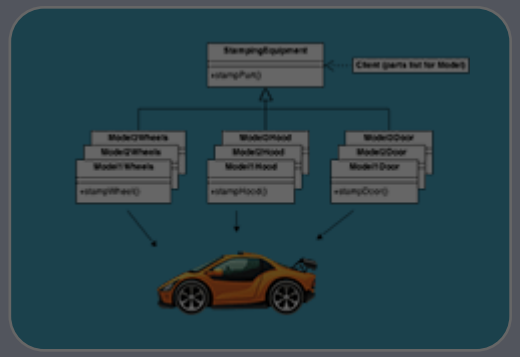
Kuan-Ting Lai

2019/4/20

Behavioral Design Patterns

Creational Design Patterns

Initialize objects or create new classes

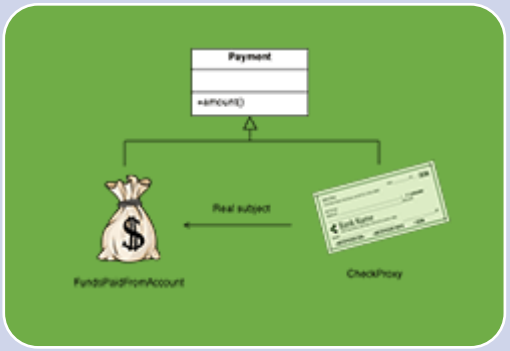


Structural Design Patterns

Compose objects to get new functions

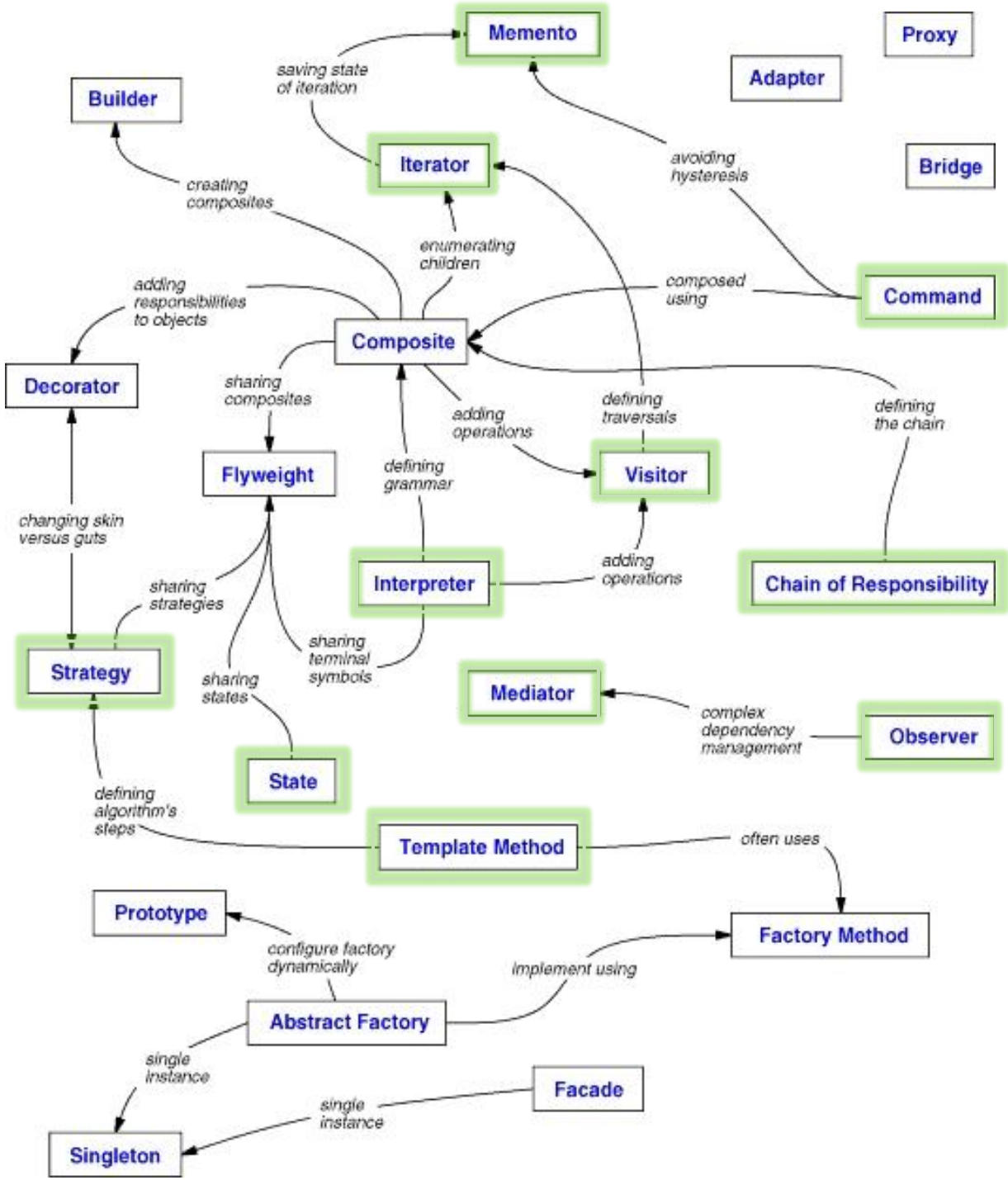
Behavioral Design Patterns

Communication between objects



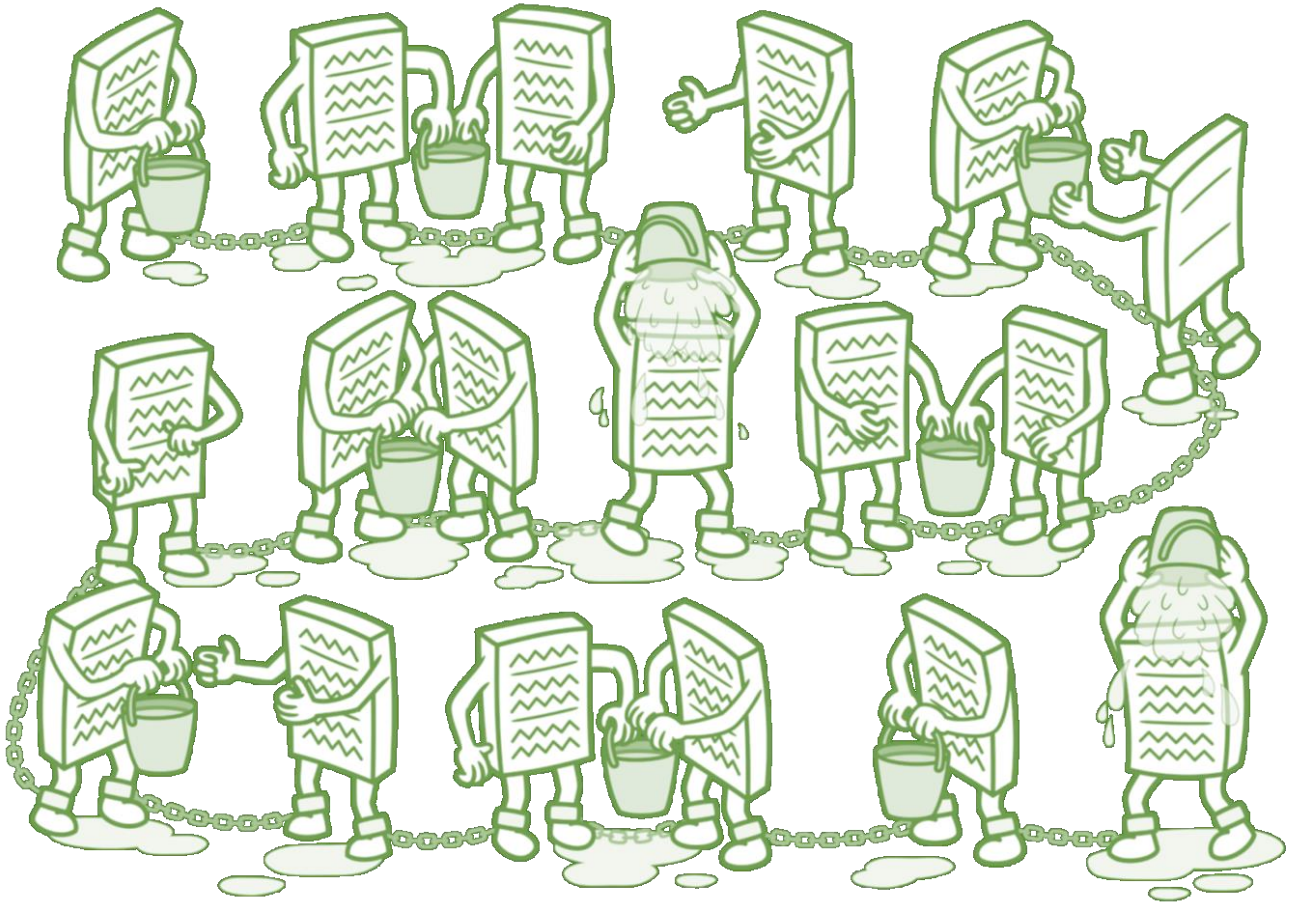
Behavioral Design Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template
11. Visitor


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

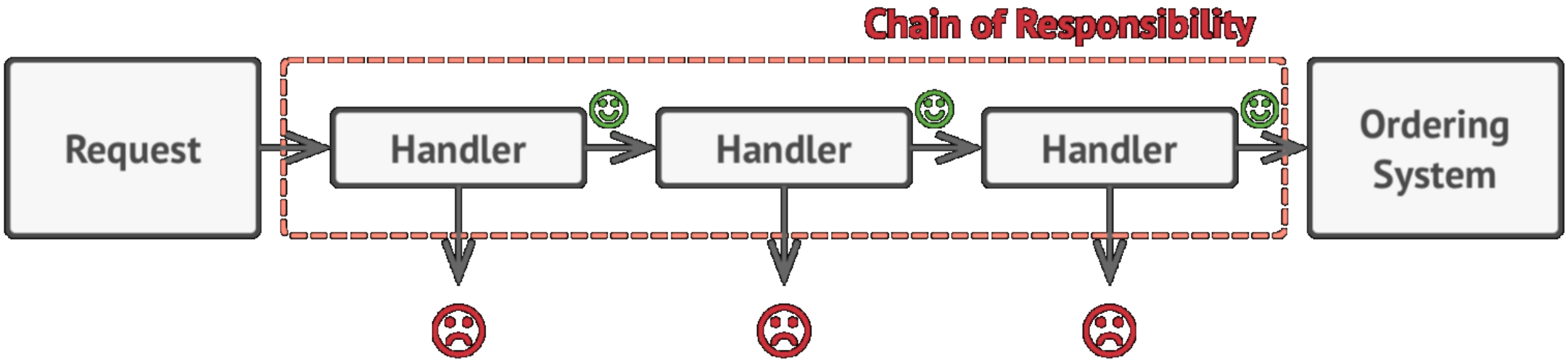
1. Chain of Responsibility

- Pass requests to the chain of handlers



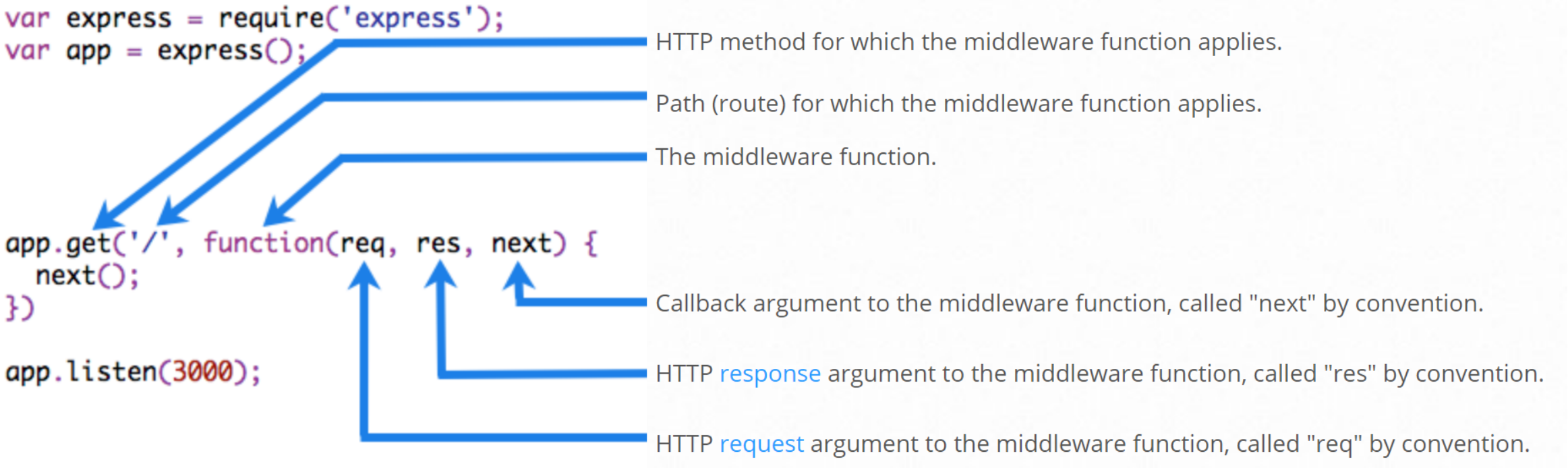
Transform Behavior into “handlers”

- Example: node.js



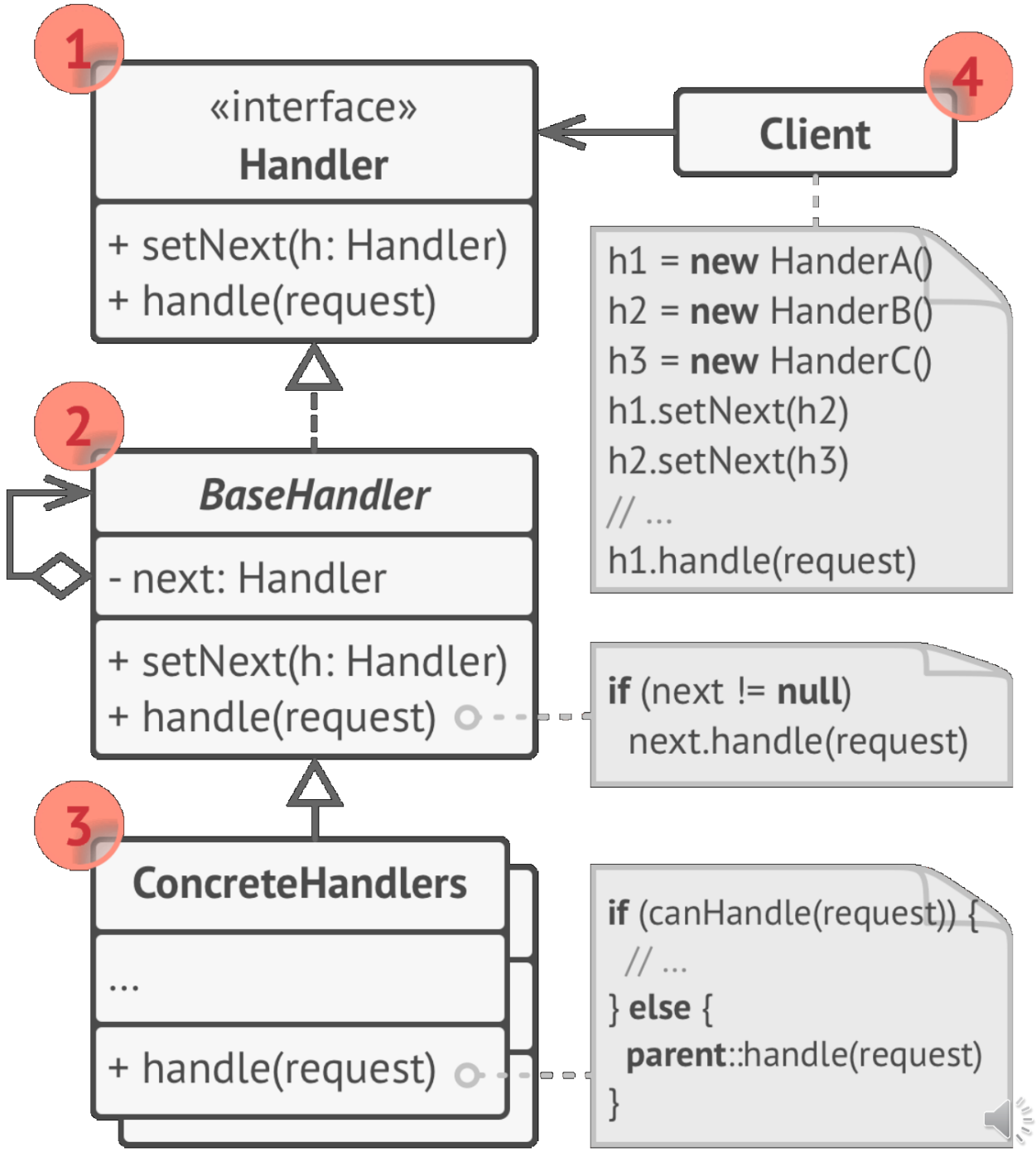
Example: node.js

- Callback function: next()

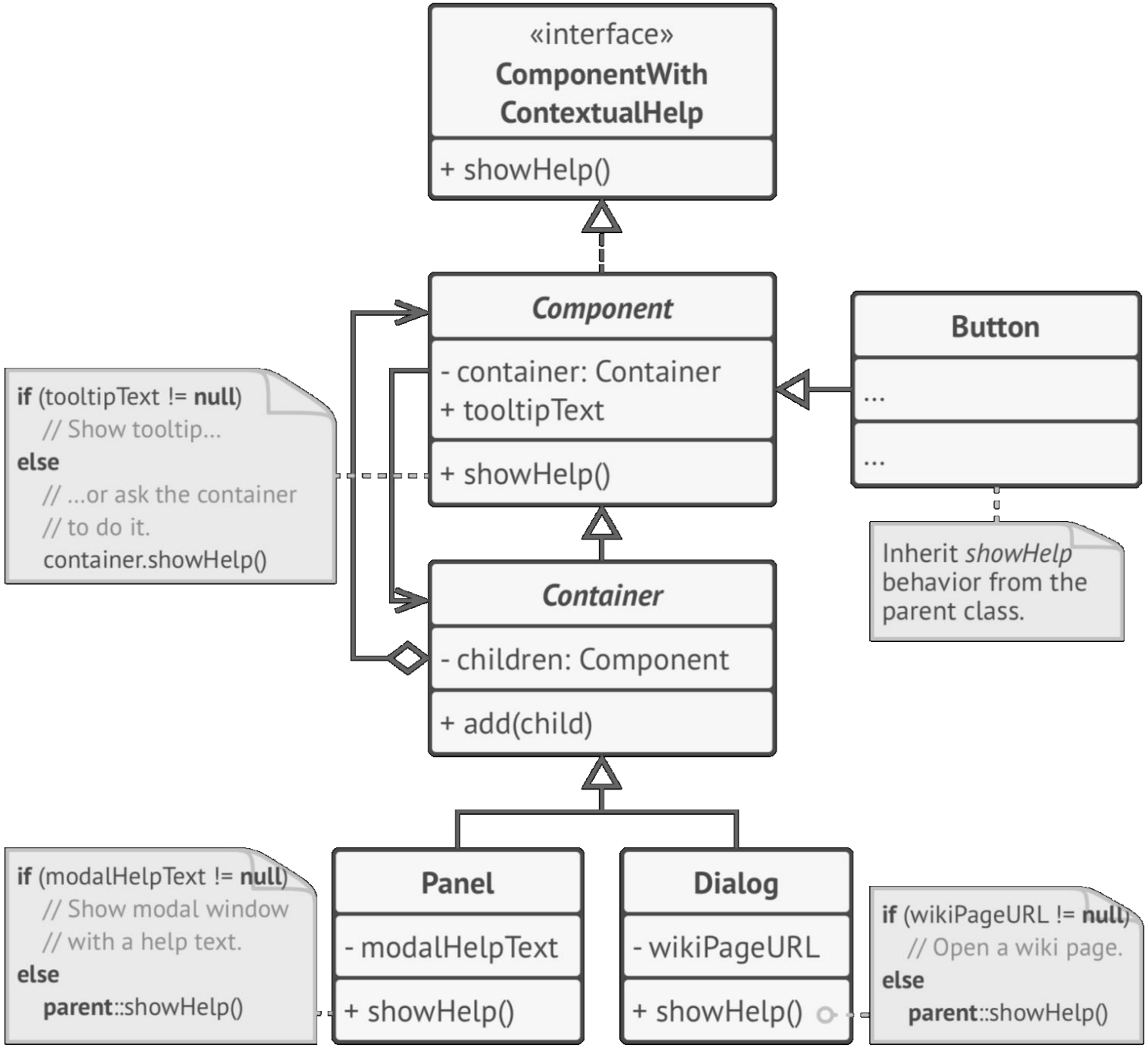


Chain of Responsibility Structure

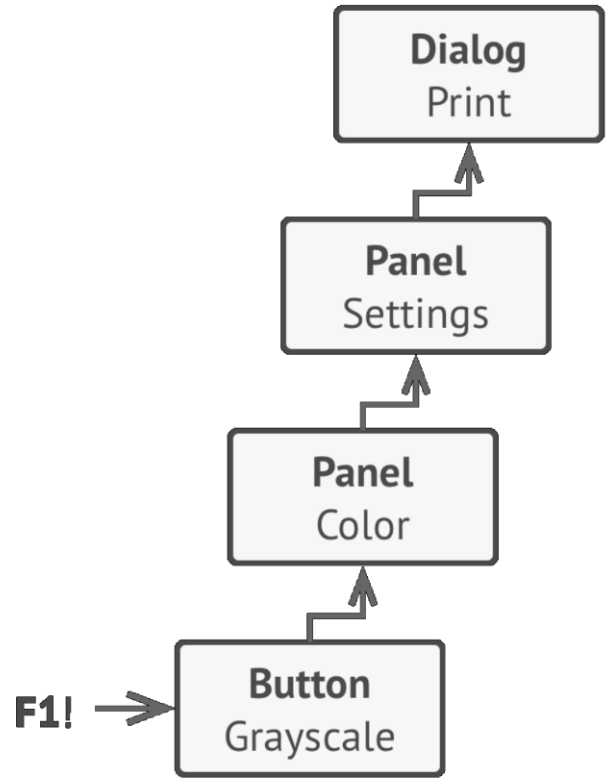
- **Handler** declares the interface, common for all concrete handlers
- **Base Handler** is an optional class where you can put the boilerplate code
- **Concrete Handlers** contain the actual code for processing requests



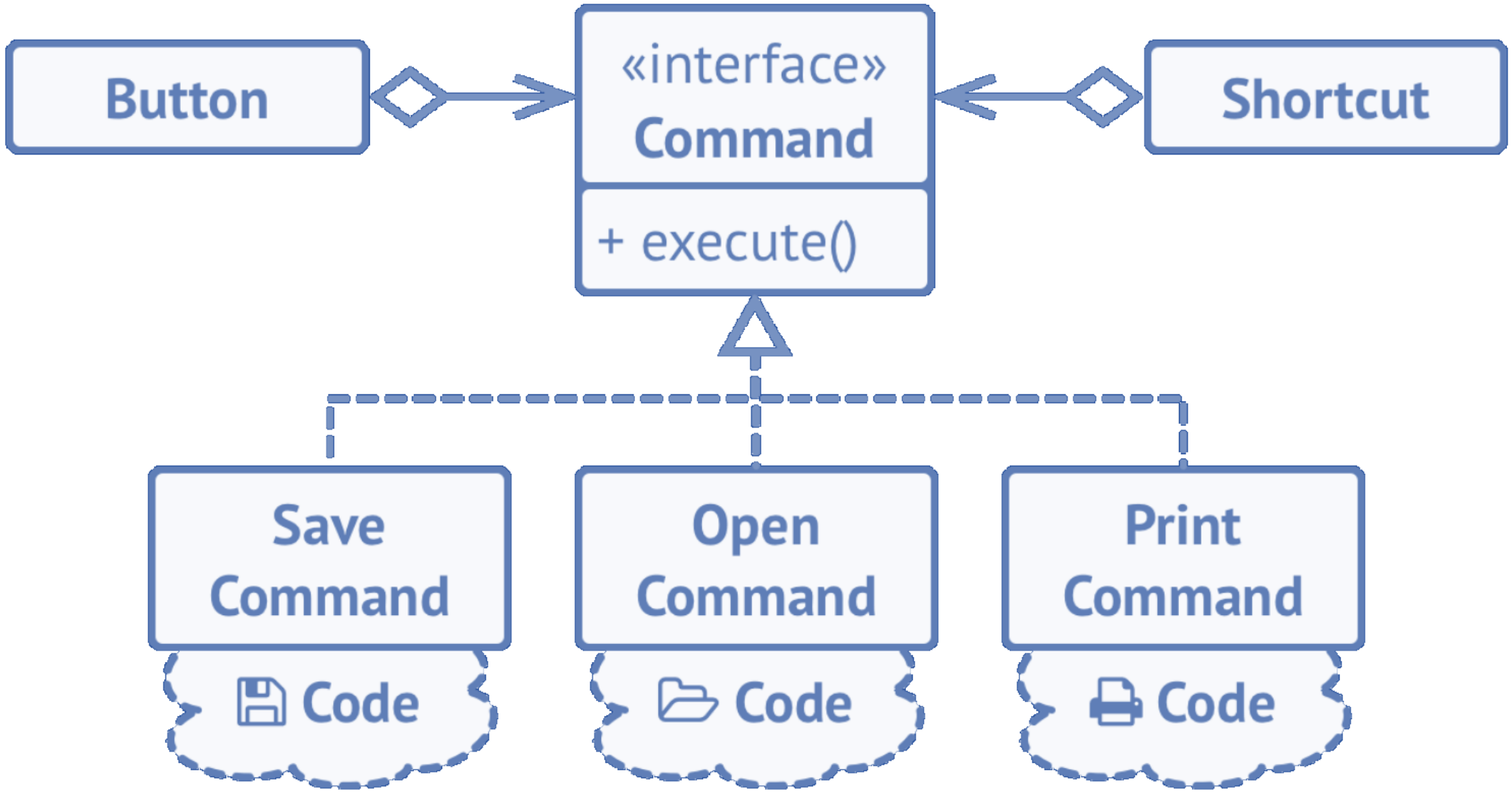
Working with Composite Pattern



- Find the right class to do showHelp()

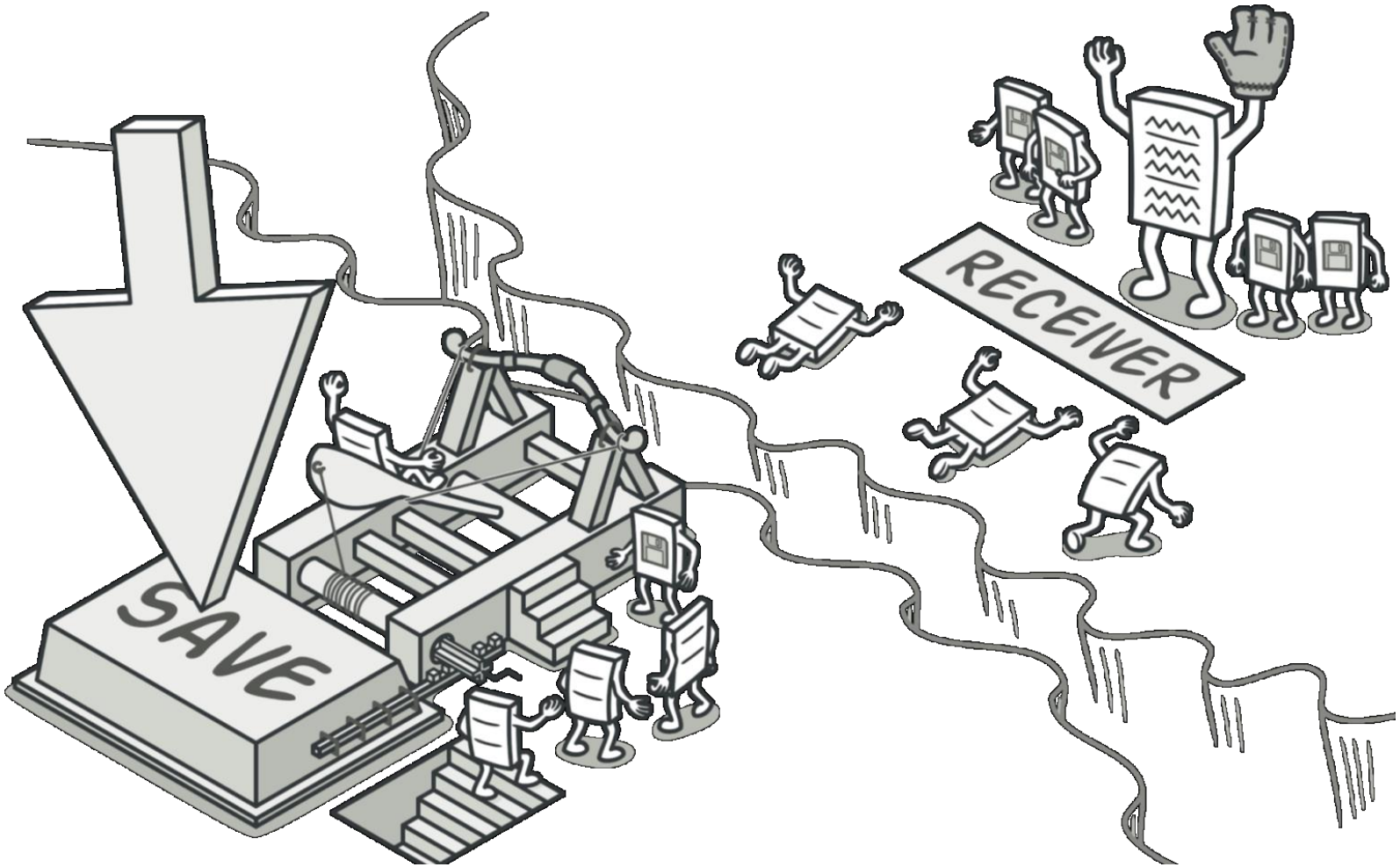


2. COMMAND


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

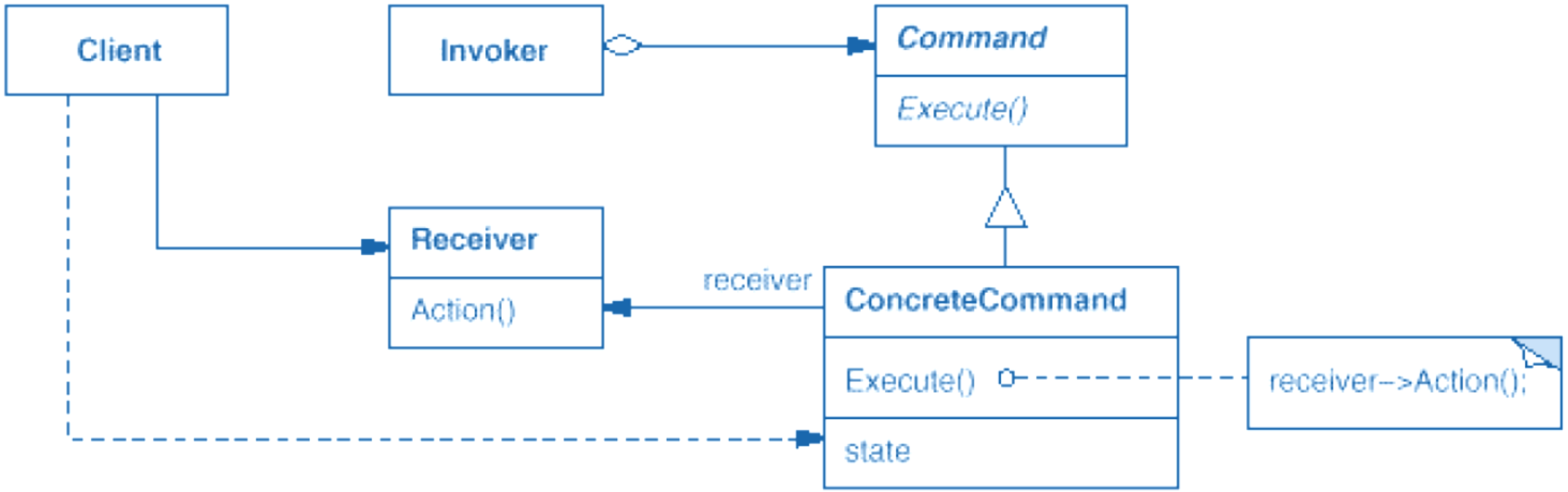
Command Pattern

- Turn a request into a stand-alone object that contains all information

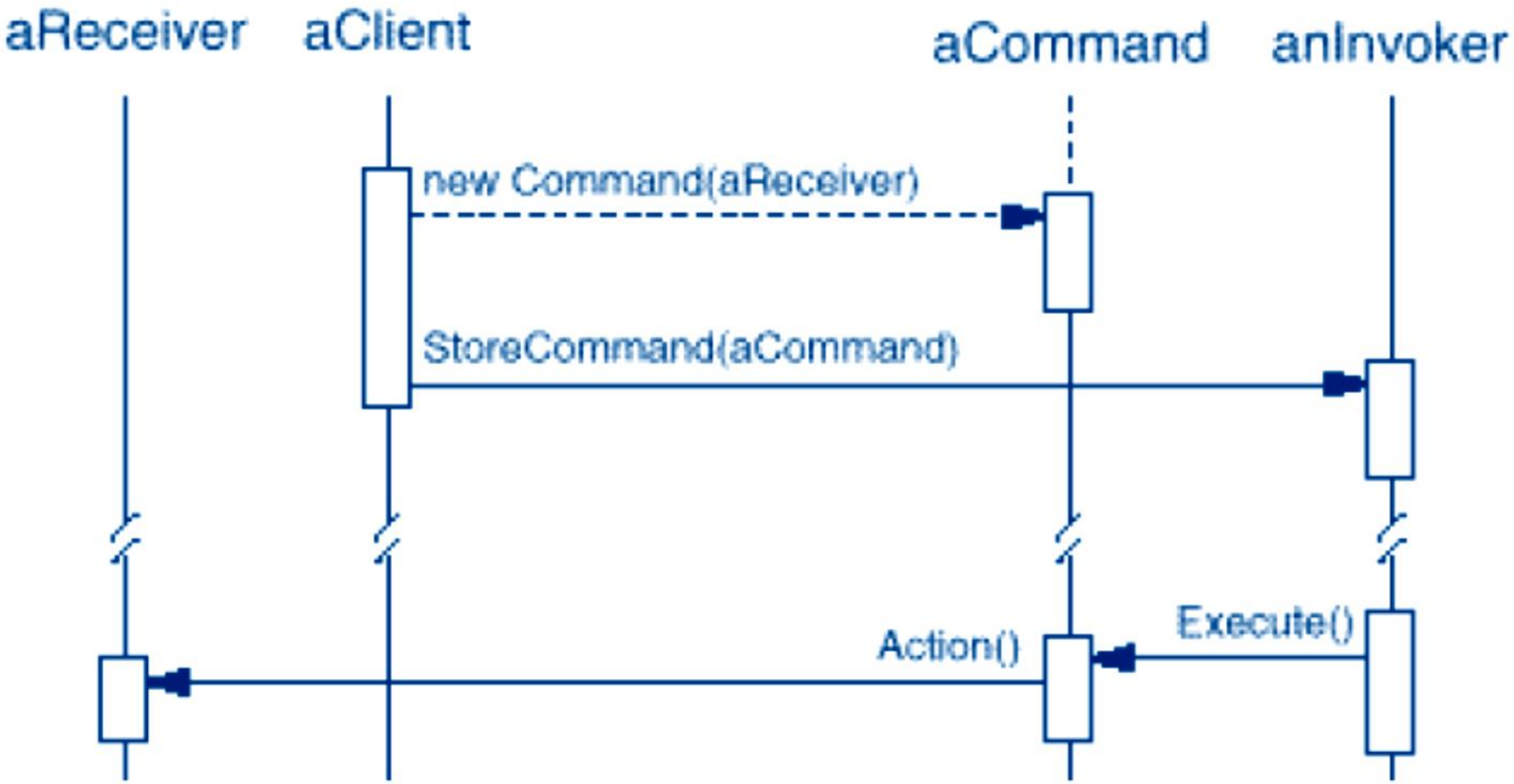


Command Structure

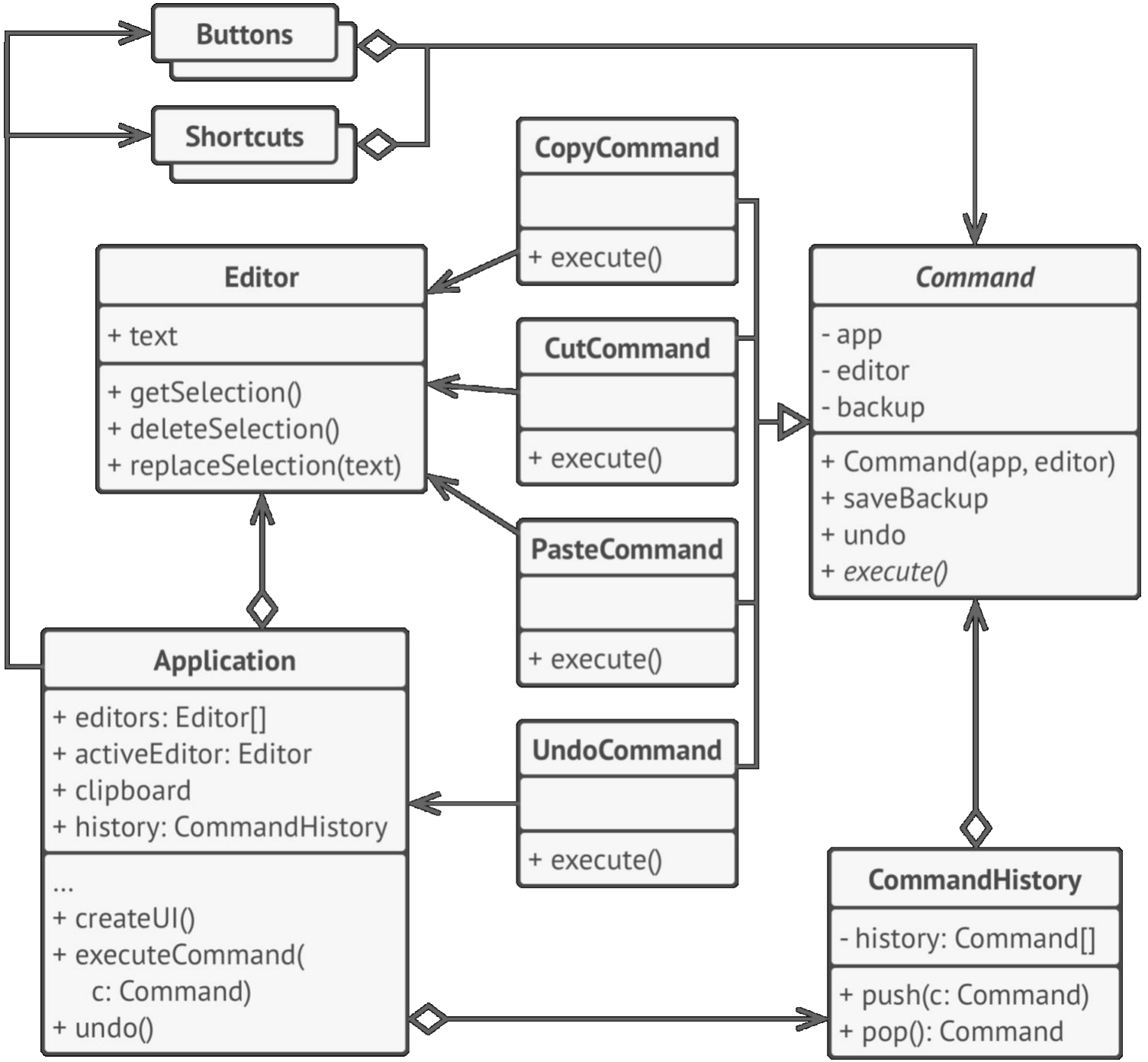
- ConcreteCommand
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.



Collaboration

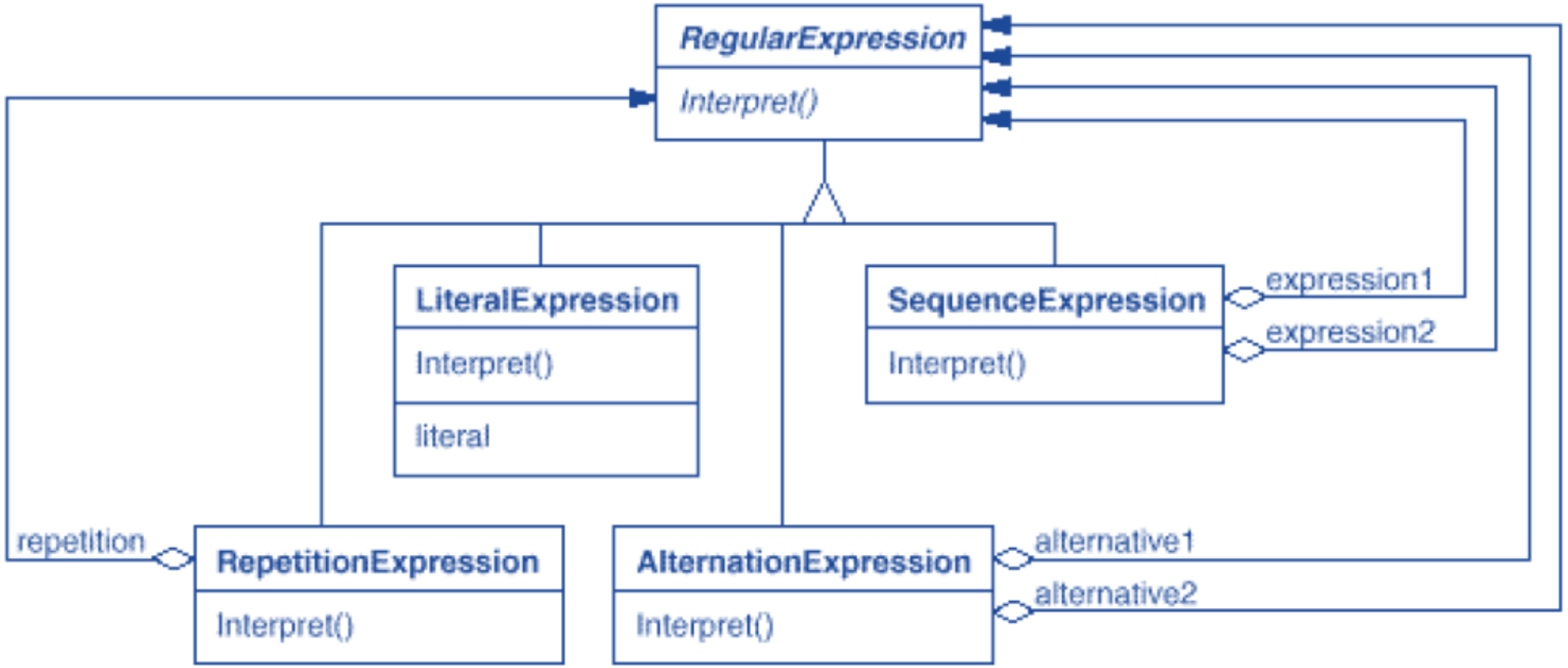


Undo


Buttons
Shortcuts
CopyCommand
+ execute()
Editor
+ text
+ getSelection()
+ deleteSelection()
+ replaceSelection(text)
CutCommand
+ execute()
Command
- app
- editor
- backup
+ Command(app, editor)
+ saveBackup
+ undo
+ execute()
PasteCommand
+ execute()
Application
+ editors: Editor[]
+ activeEditor: Editor
+ clipboard
+ history: CommandHistory
...
+ createUI()
+ executeCommand(
c: Command)
+ undo()
UndoCommand
+ execute()
CommandHistory
- history: Command[]
+ push(c: Command)
+ pop(): Command

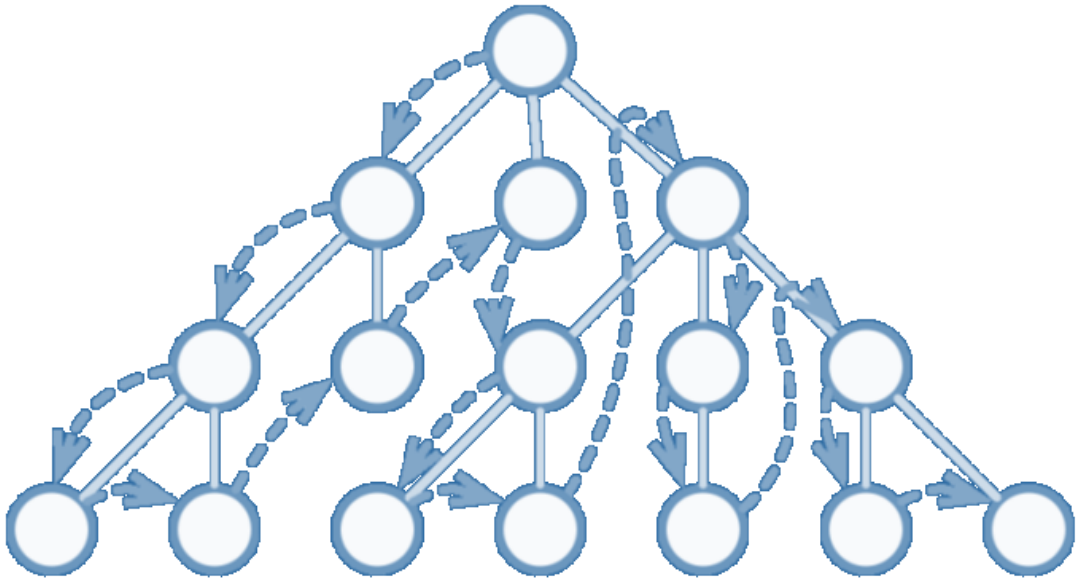
3. Interpreter

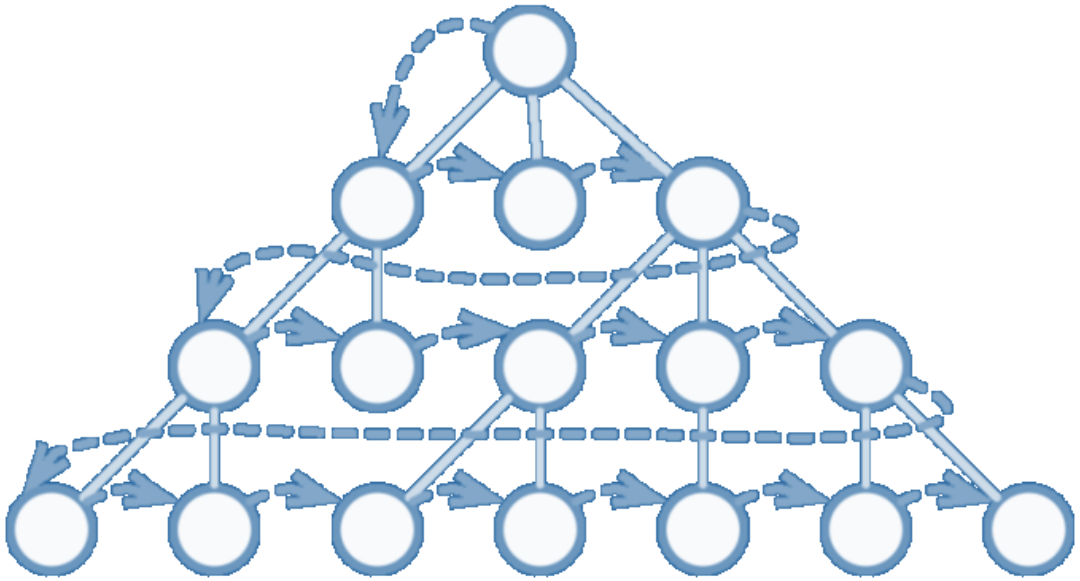
- Given a language, define a representation for its grammar with an interpreter that uses the representation to interpret sentences in the language

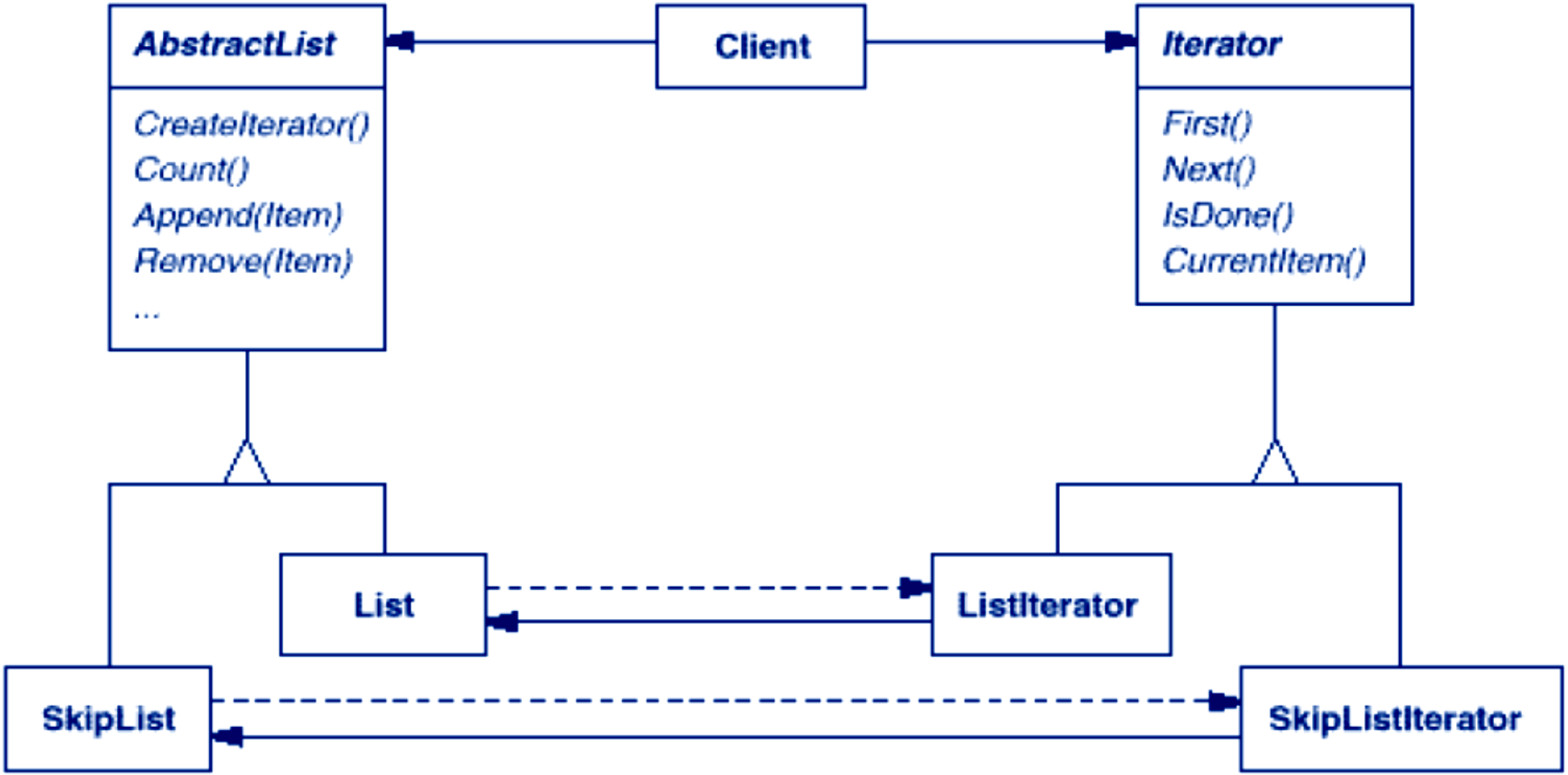


4. Iterator

- A pattern that traverses elements of a collection






AbstractList
CreateIterator()
Count()
Append(Item)
Remove(Item)
...
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator

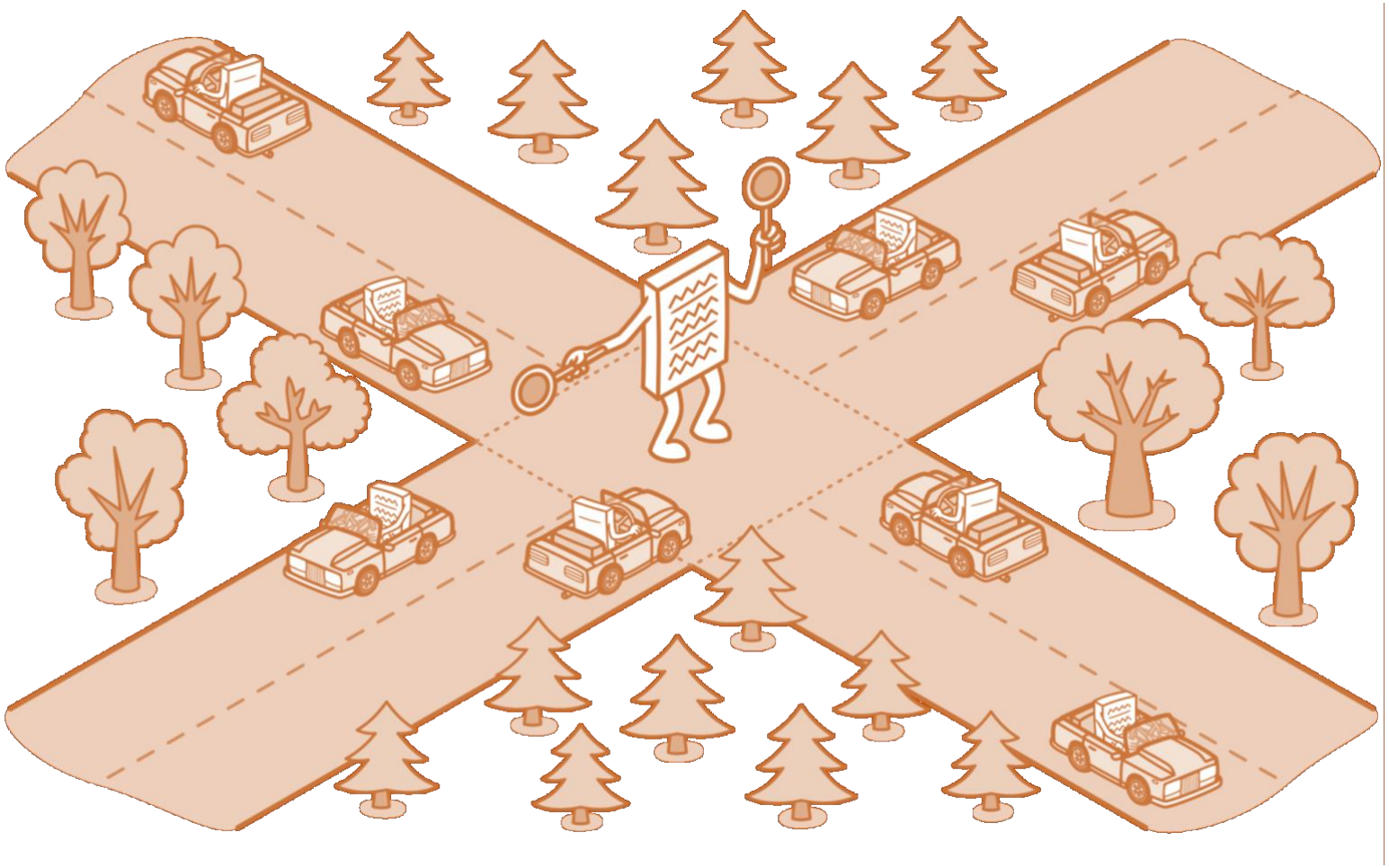
```cpp
// std::iterator example
#include <iostream>     // std::cout
#include <iterator>     // std::iterator, std::input_iterator_tag

class MyIterator : public std::iterator<std::input_iterator_tag, int>
{
        int* p;
public:
    MyIterator(int* x) :p(x) {}
    MyIterator(const MyIterator& mit) : p(mit.p) {}
    MyIterator& operator++() { ++p; return *this; }
    MyIterator operator++(int) { MyIterator tmp(*this); operator++(); return tmp; }
    bool operator==(const MyIterator& rhs) const { return p == rhs.p; }
    bool operator!=(const MyIterator& rhs) const { return p != rhs.p; }
    int& operator*() { return *p; }
};

int main() {
    int numbers[] = { 10,20,30,40,50 };
    MyIterator from(numbers);
    MyIterator until(numbers + 5);
    for (MyIterator it = from; it != until; it++)
    std::cout << *it << ' ';
    std::cout << '\n';

    return 0;
}
```

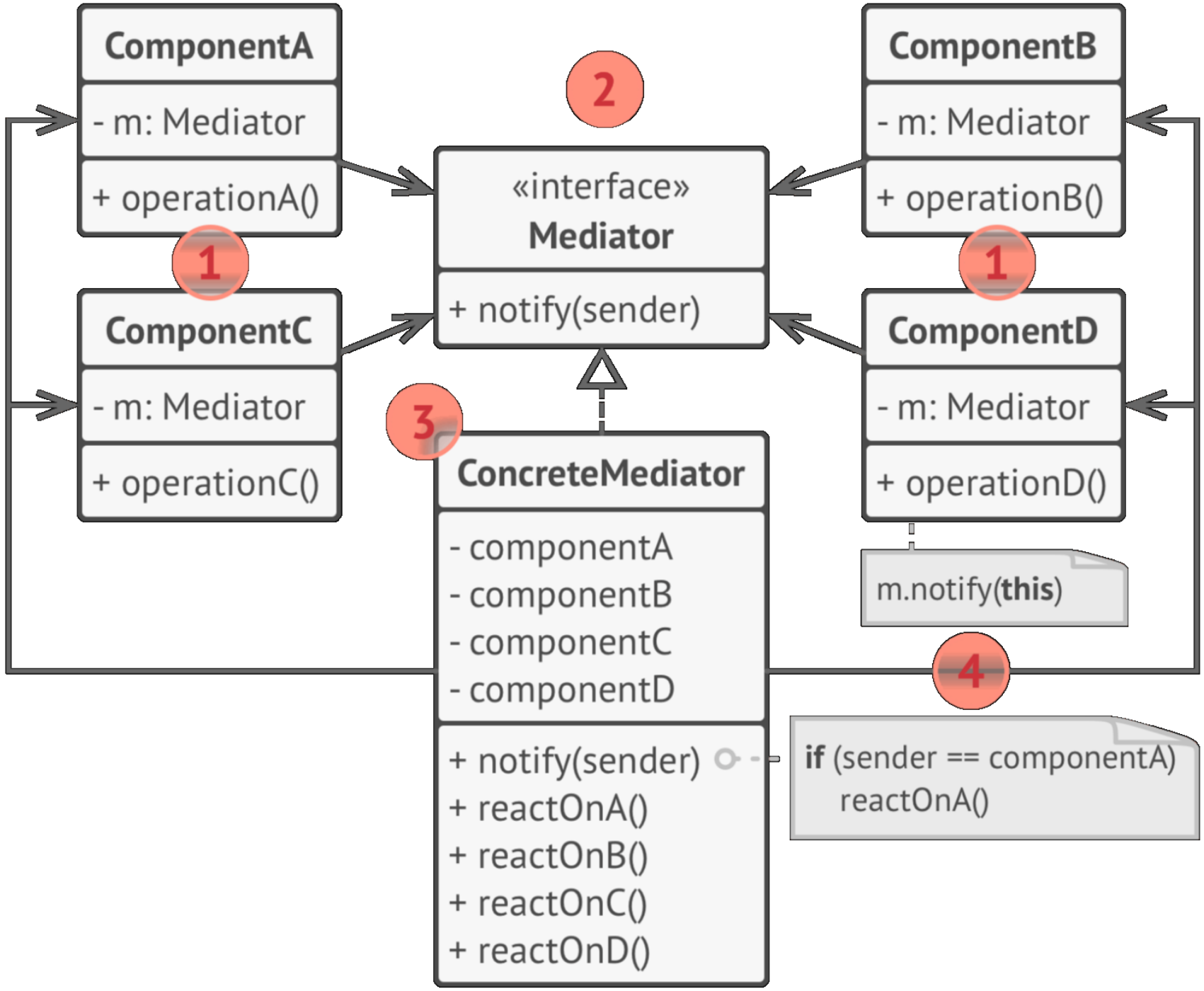
Or

it != from.end()

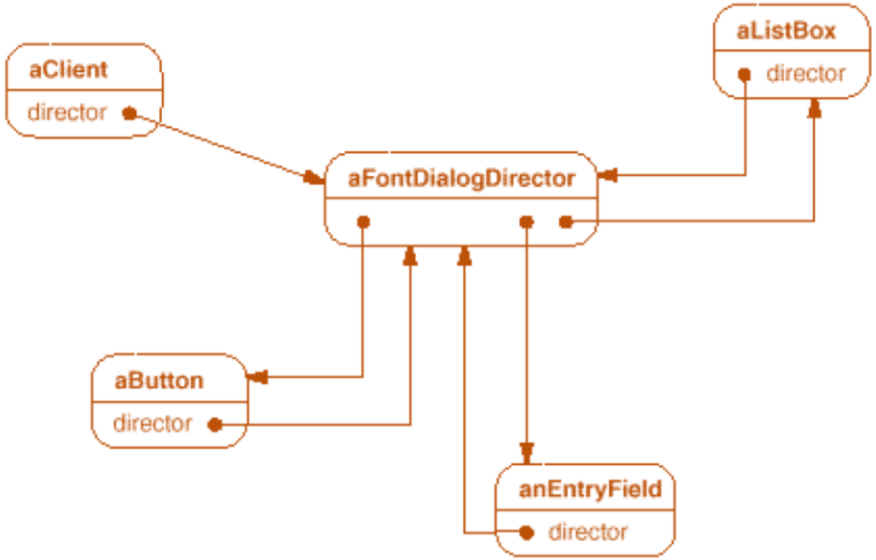
5. Mediator (a.k.a. Intermediary, Controller)

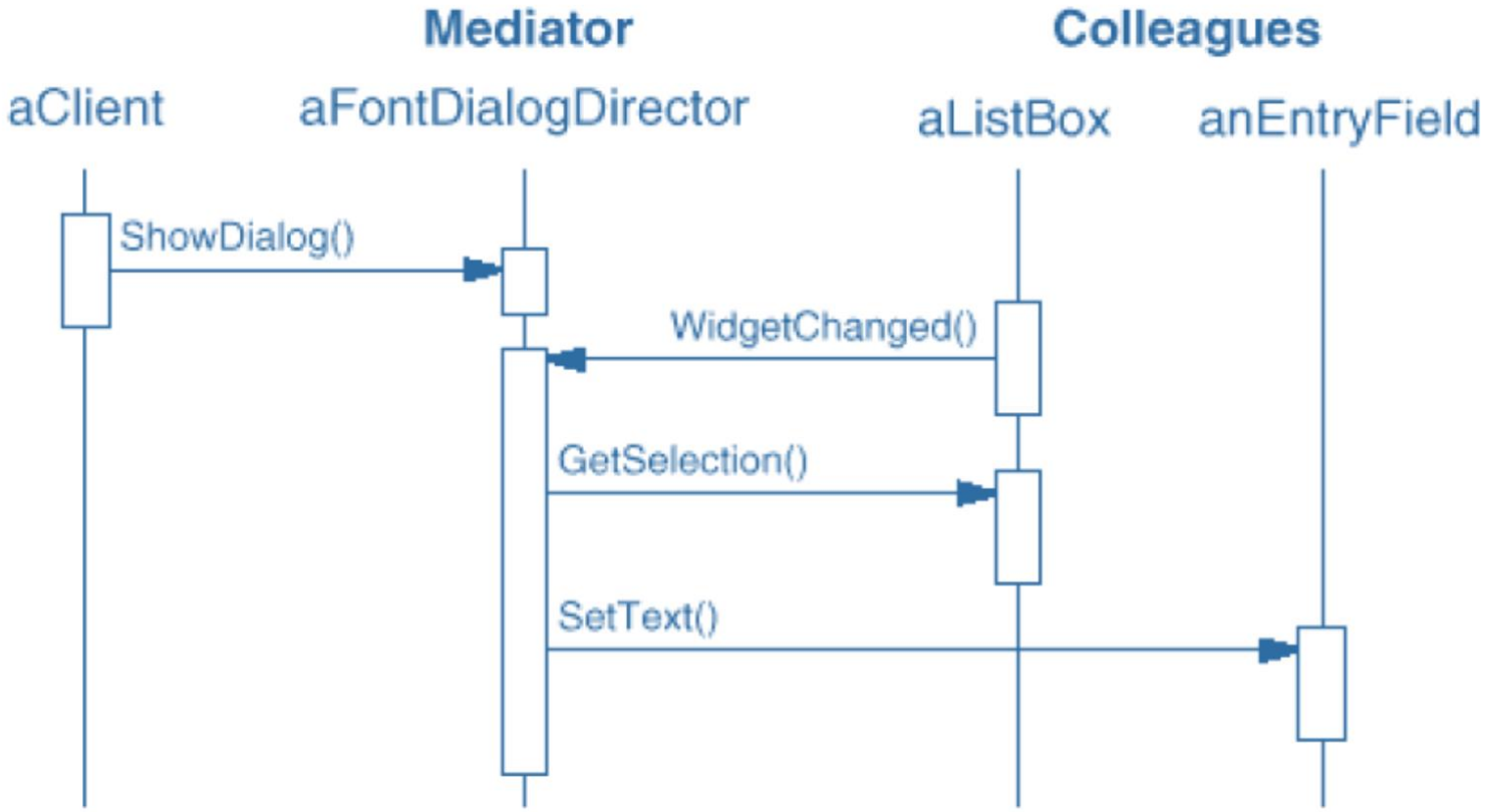
- Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently




ComponentA
- m: Mediator
+ operationA()
ComponentB
- m: Mediator
+ operationB()
2
«interface»
Mediator
+ notify(sender)
1
1
ComponentC
- m: Mediator
+ operationC()
ComponentD
- m: Mediator
+ operationD()
3
ConcreteMediator
- componentA
- componentB
- componentC
- componentD
+ notify(sender)
+ reactOnA()
+ reactOnB()
+ reactOnC()
+ reactOnD()
m.notify(this)
4
if (sender == componentA)
reactOnA()

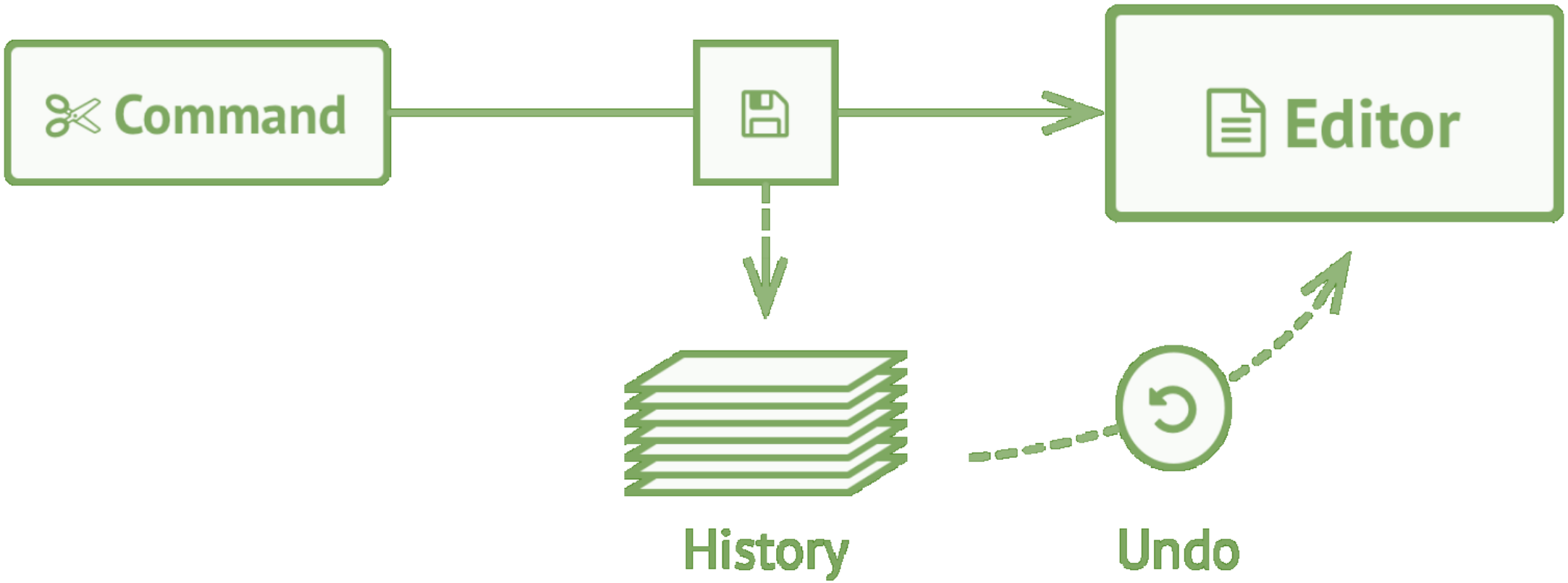
Example: Font Dialog





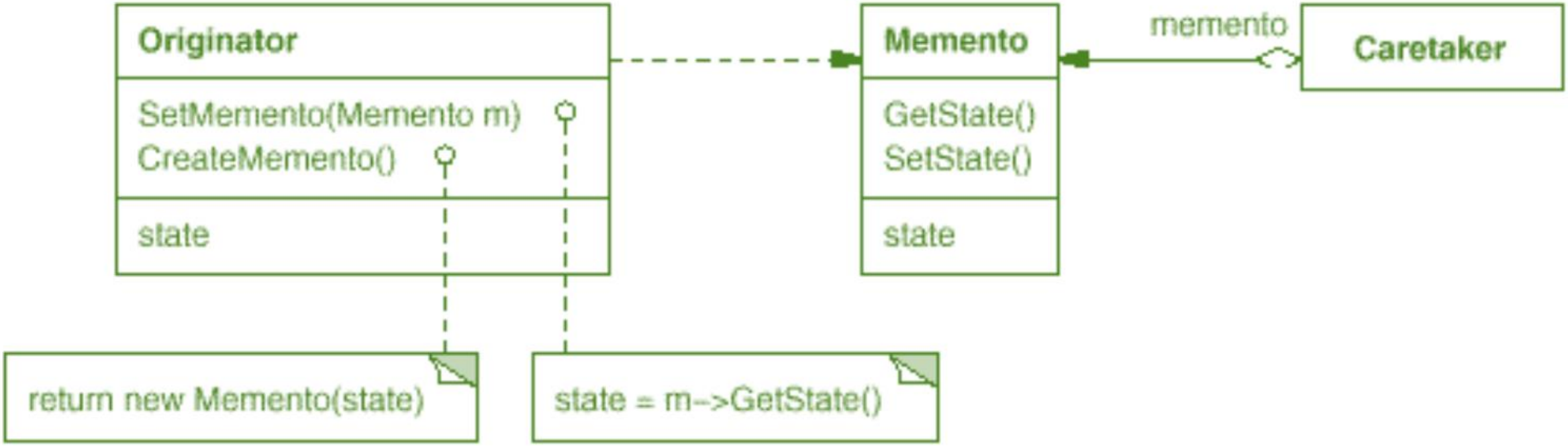
6. Memento

- Save and restore the previous state of an object without revealing the details of its implementation

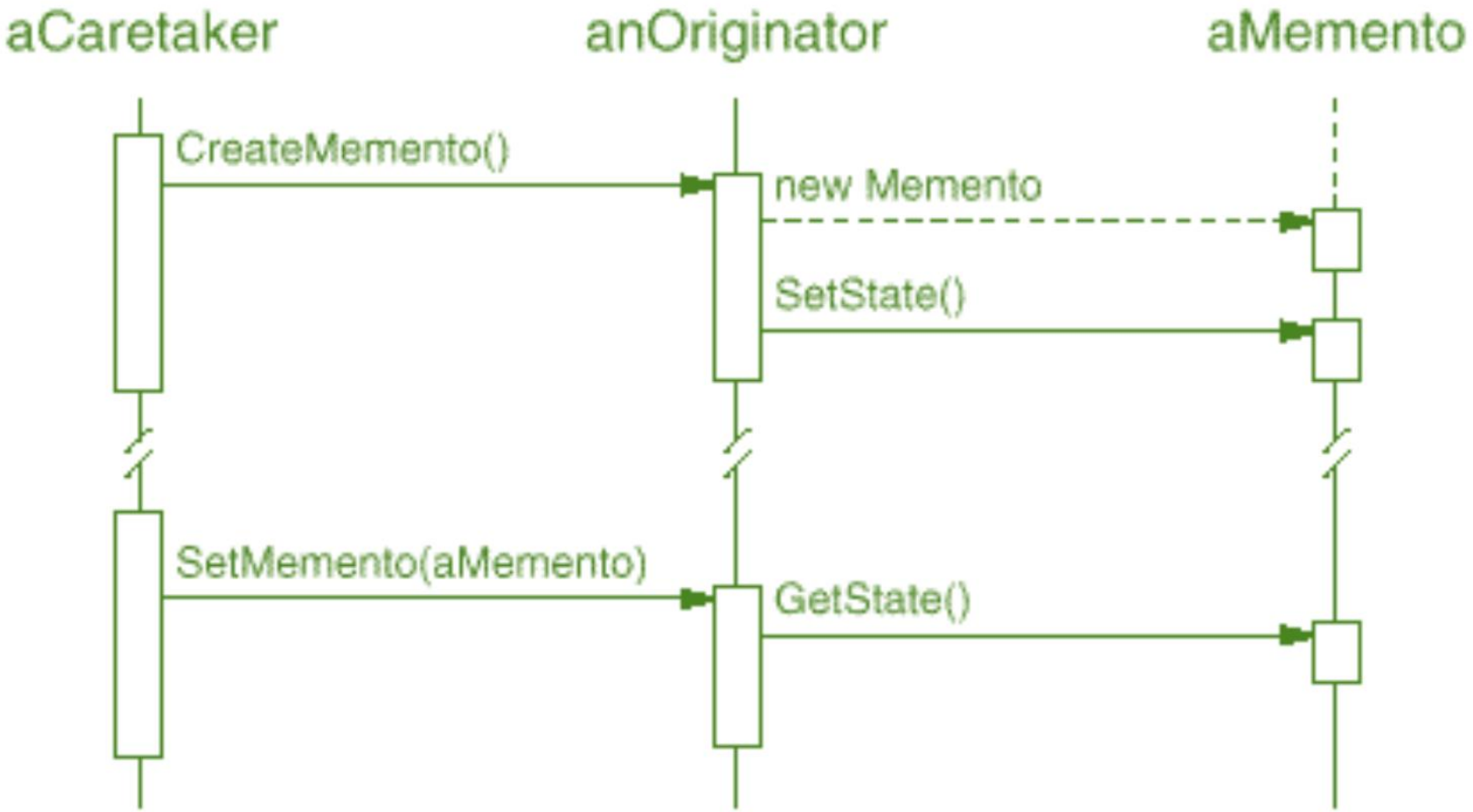


Memento Structure

- Memento: stores the internal state of the Originator
- Originator: creates a memento with a snapshot of its current state
- Caretaker: for memento's safekeeping

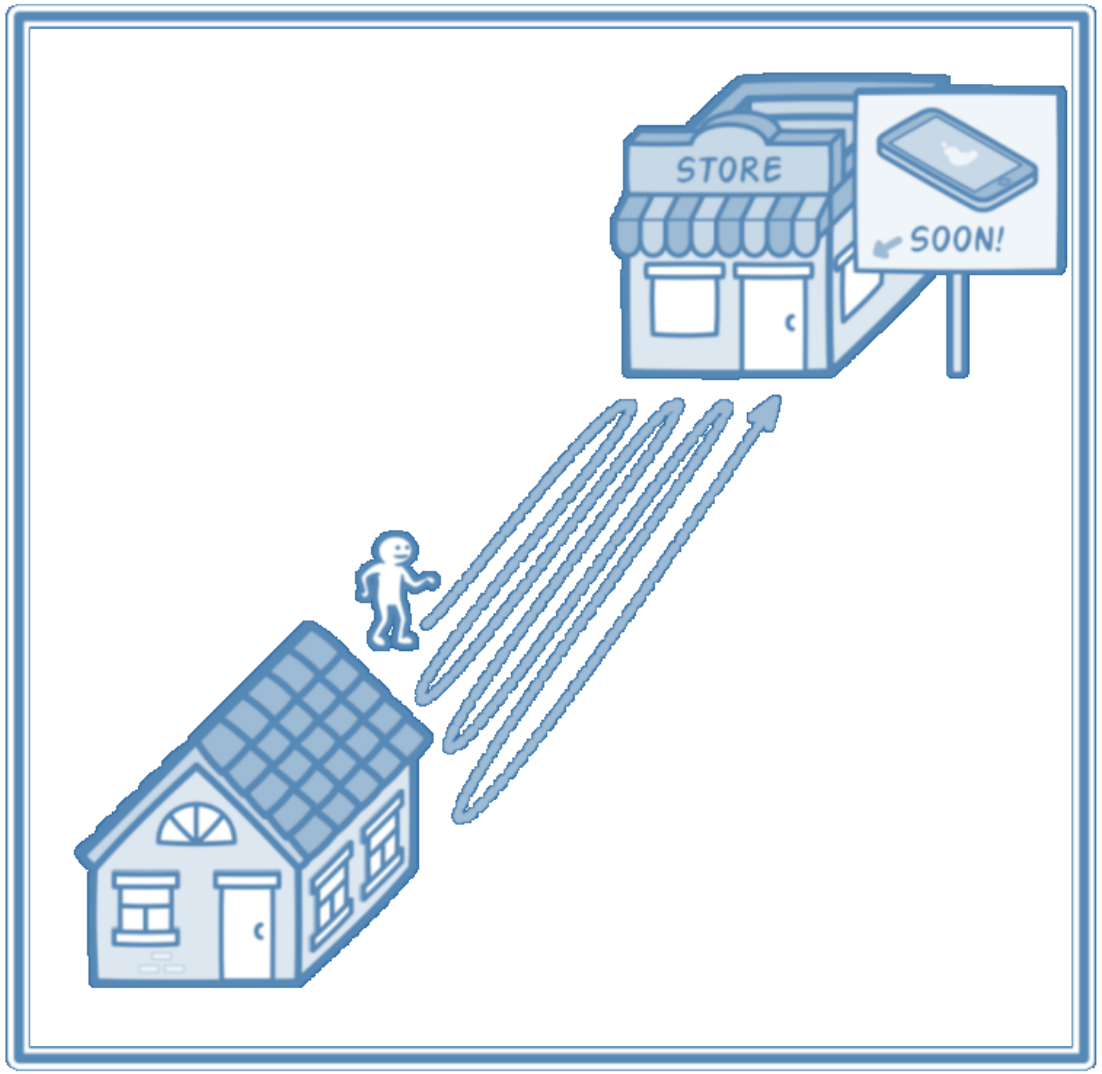


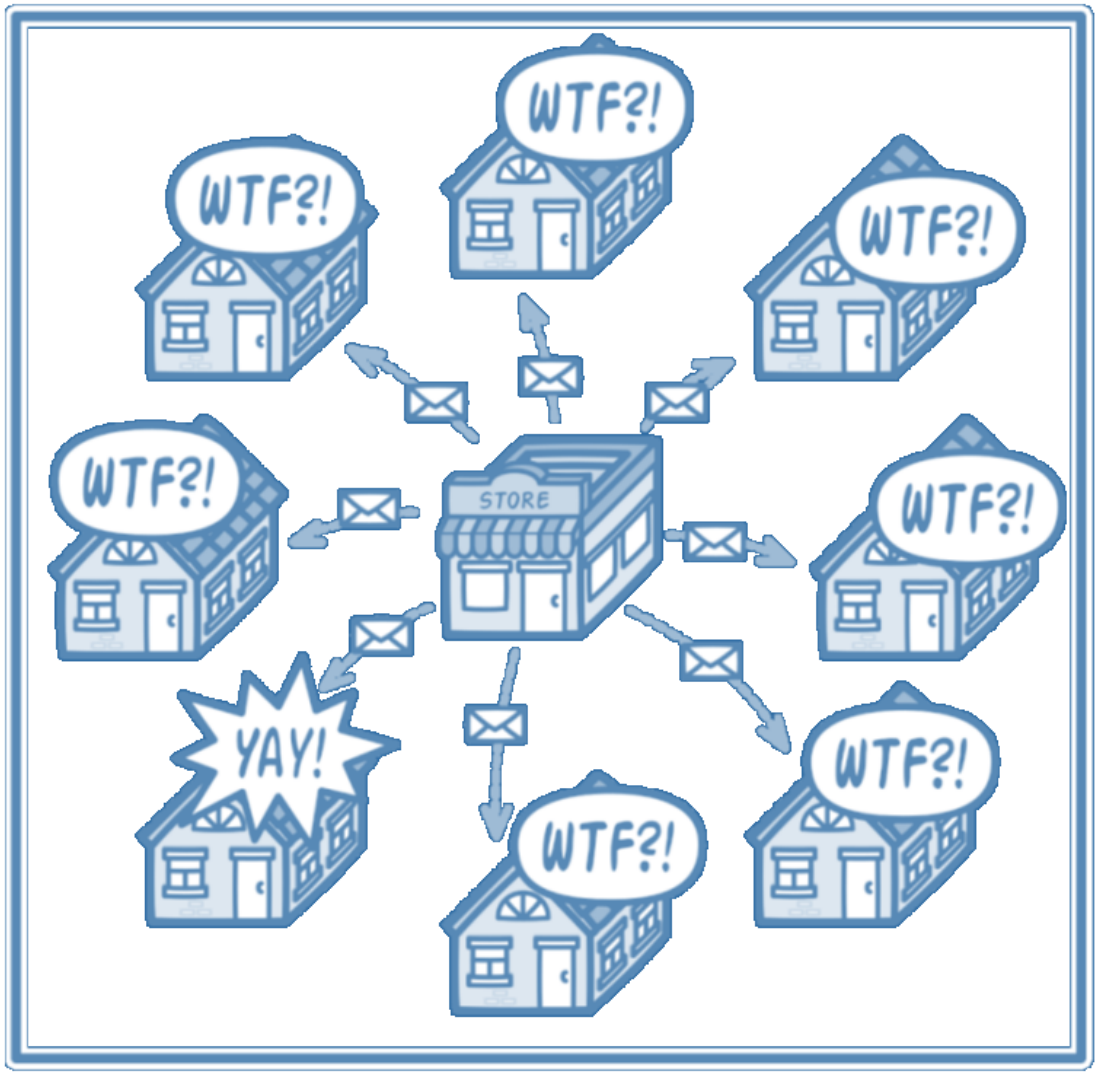
Memento Collaborations

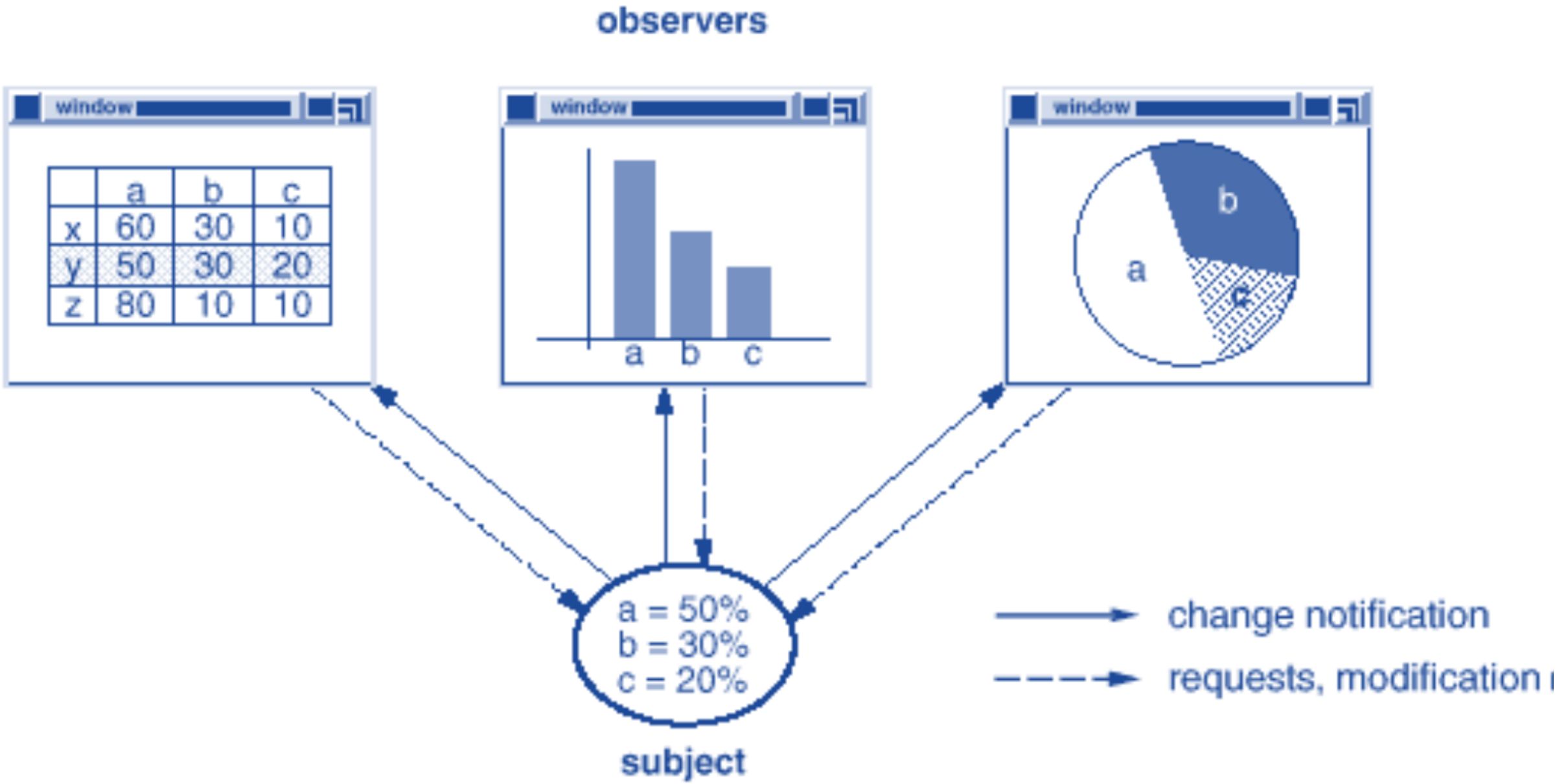


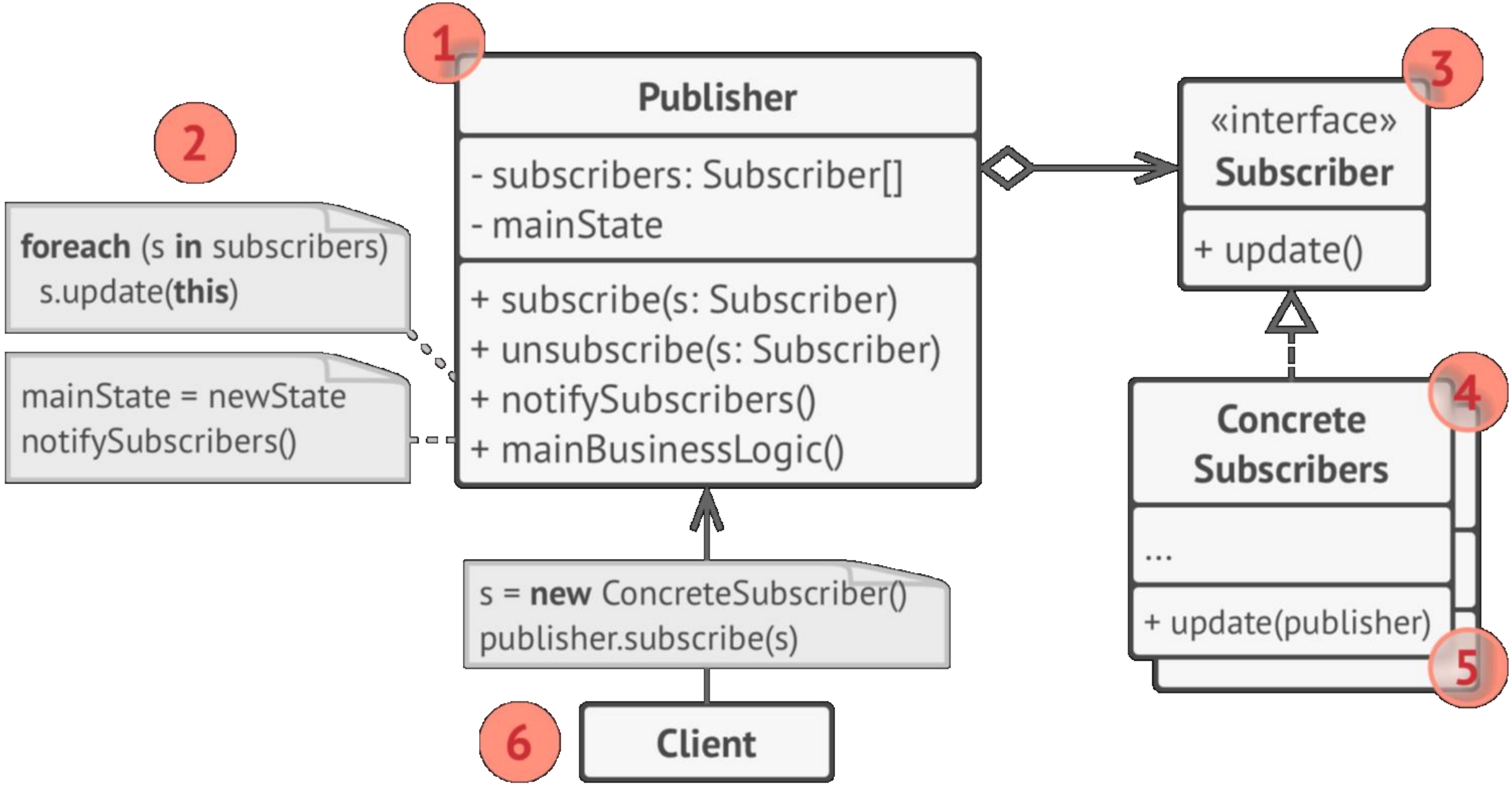
7. Observer

- Define a subscription mechanism to notify multiple objects

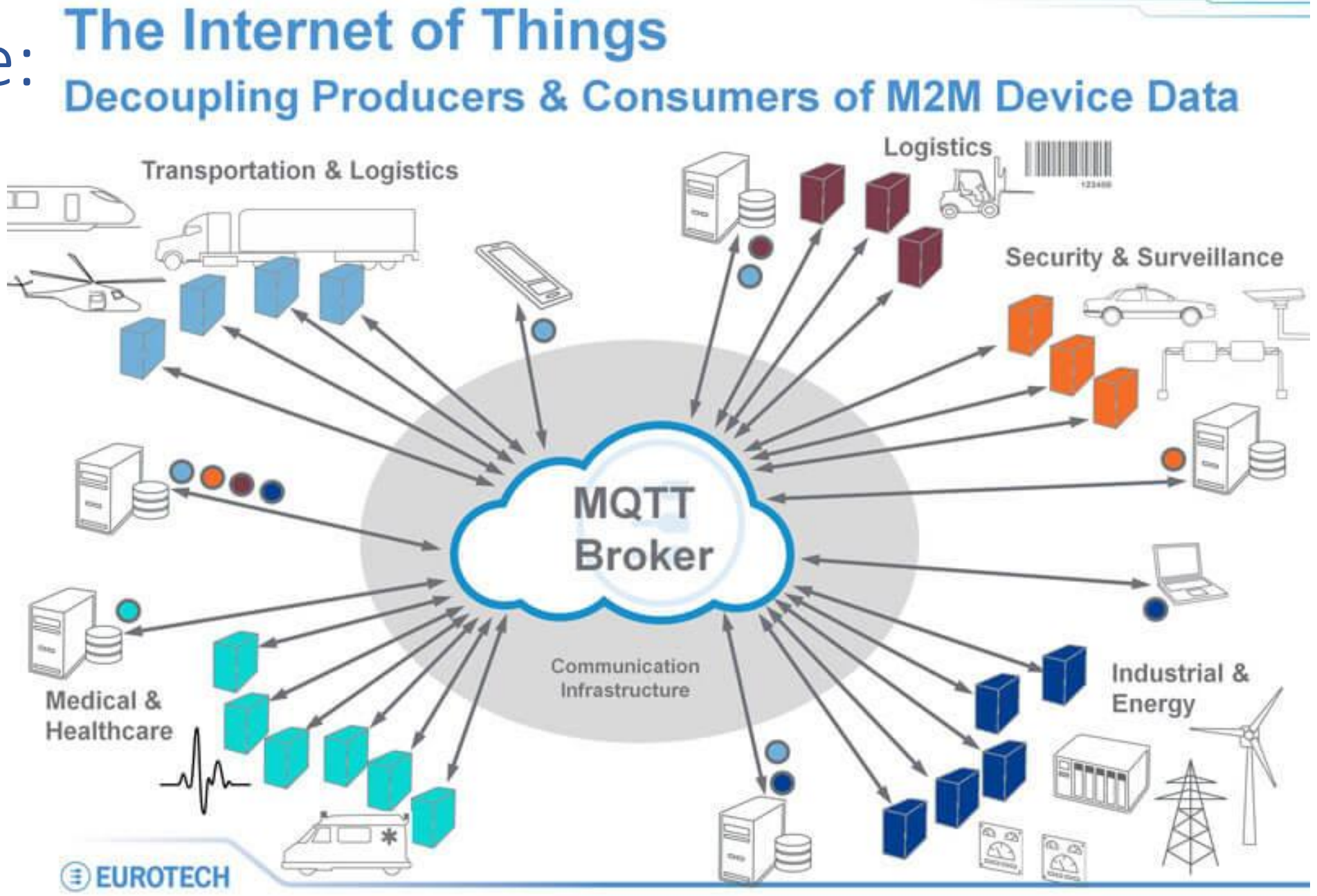





observers
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
b
a
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

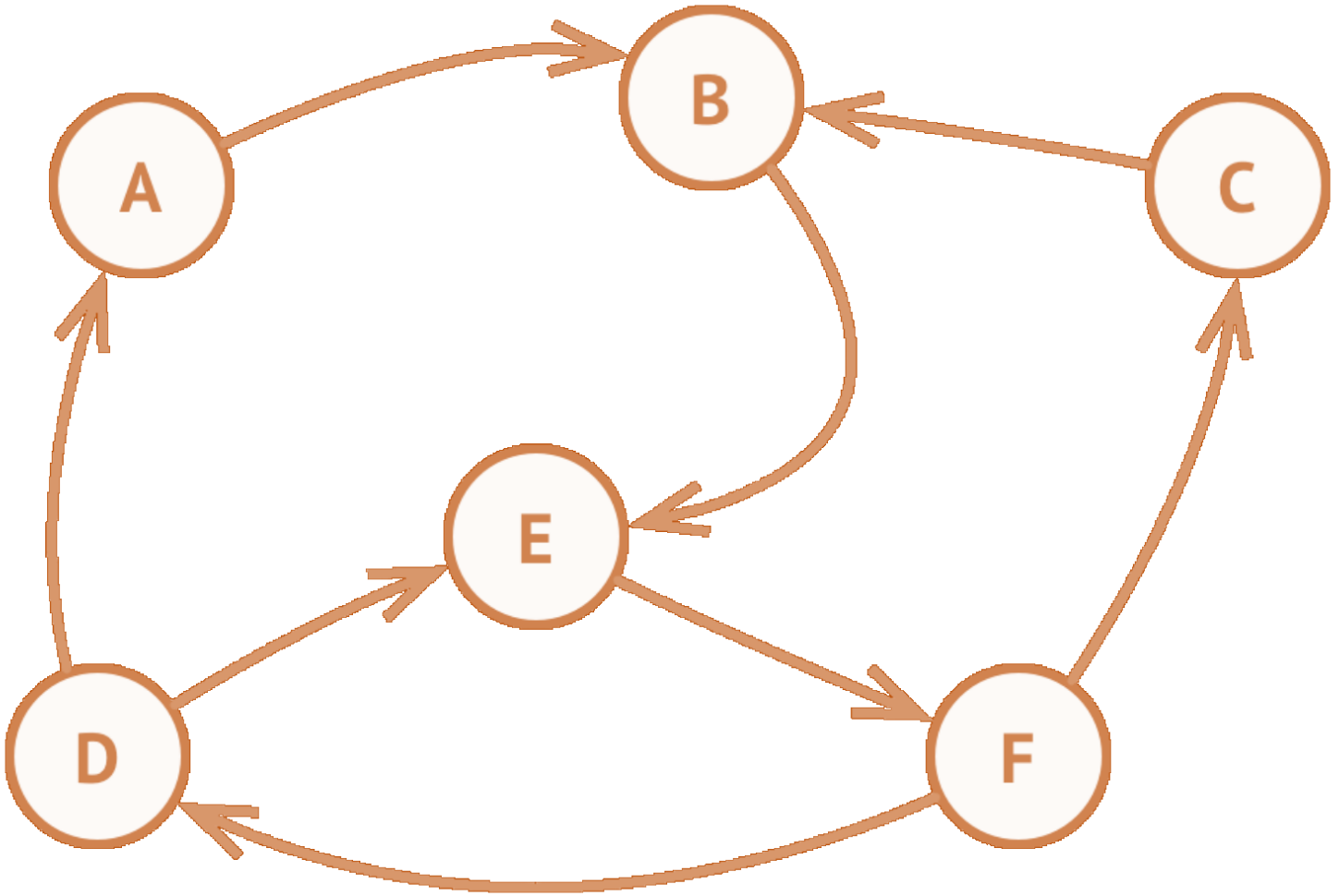

1
Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
2
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
3
«interface»
Subscriber
+ update()
4
Concrete
Subscribers
...
+ update(publisher)
5
s = new ConcreteSubscriber()
publisher.subscribe(s)
6
Client

Example:

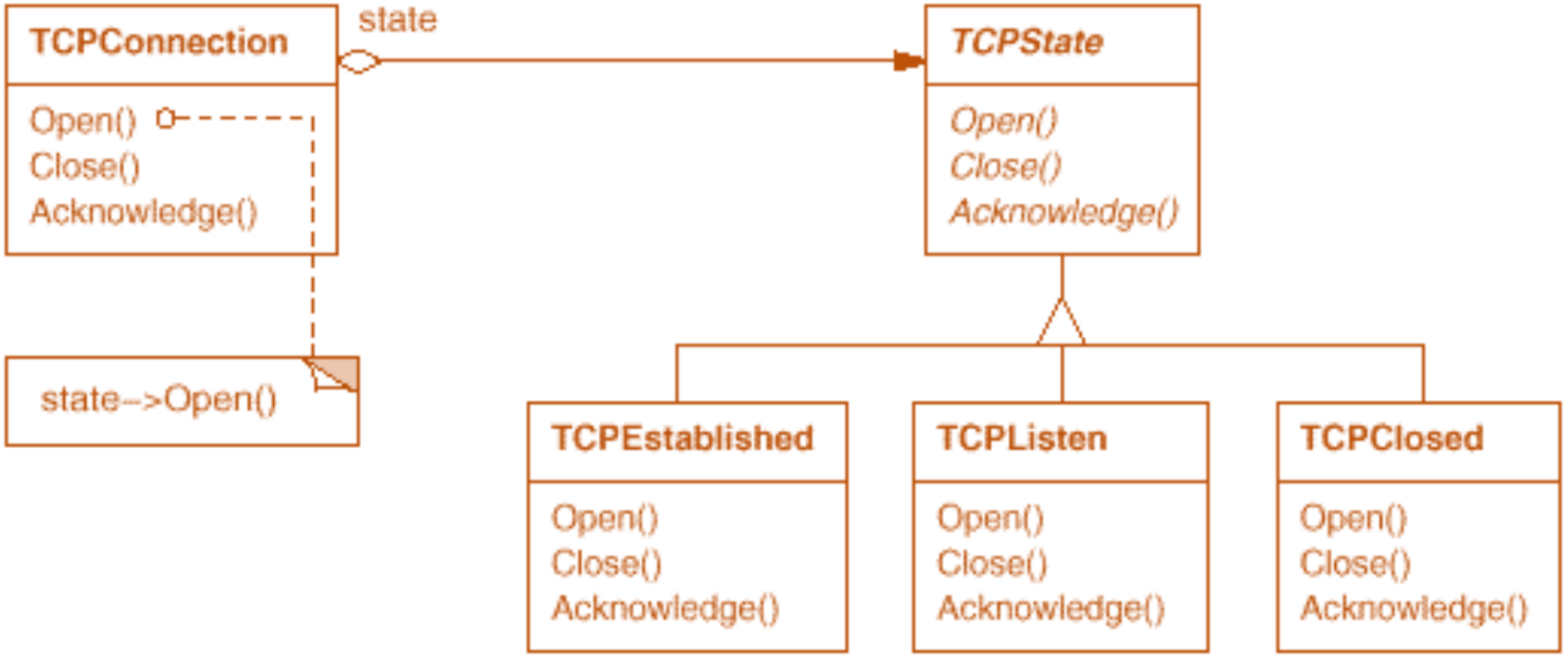

The Internet of Things
Decoupling Producers & Consumers of M2M Device Data
Transportation & Logistics
Logistics
Security & Surveillance
MQTT
Broker
Communication
Infrastructure
Medical &
Healthcare
Industrial &
Energy
EUROTECH

8. STATE

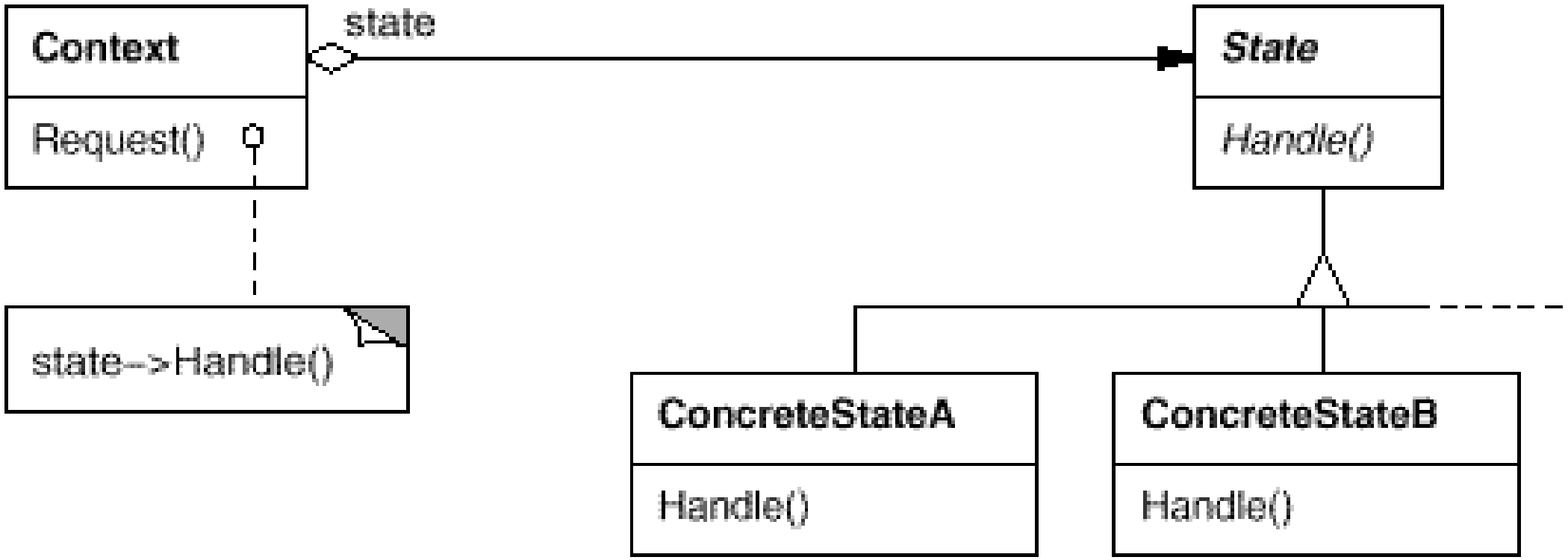
- Let an object alter its behavior when its internal state changes



Example: TCP Connection

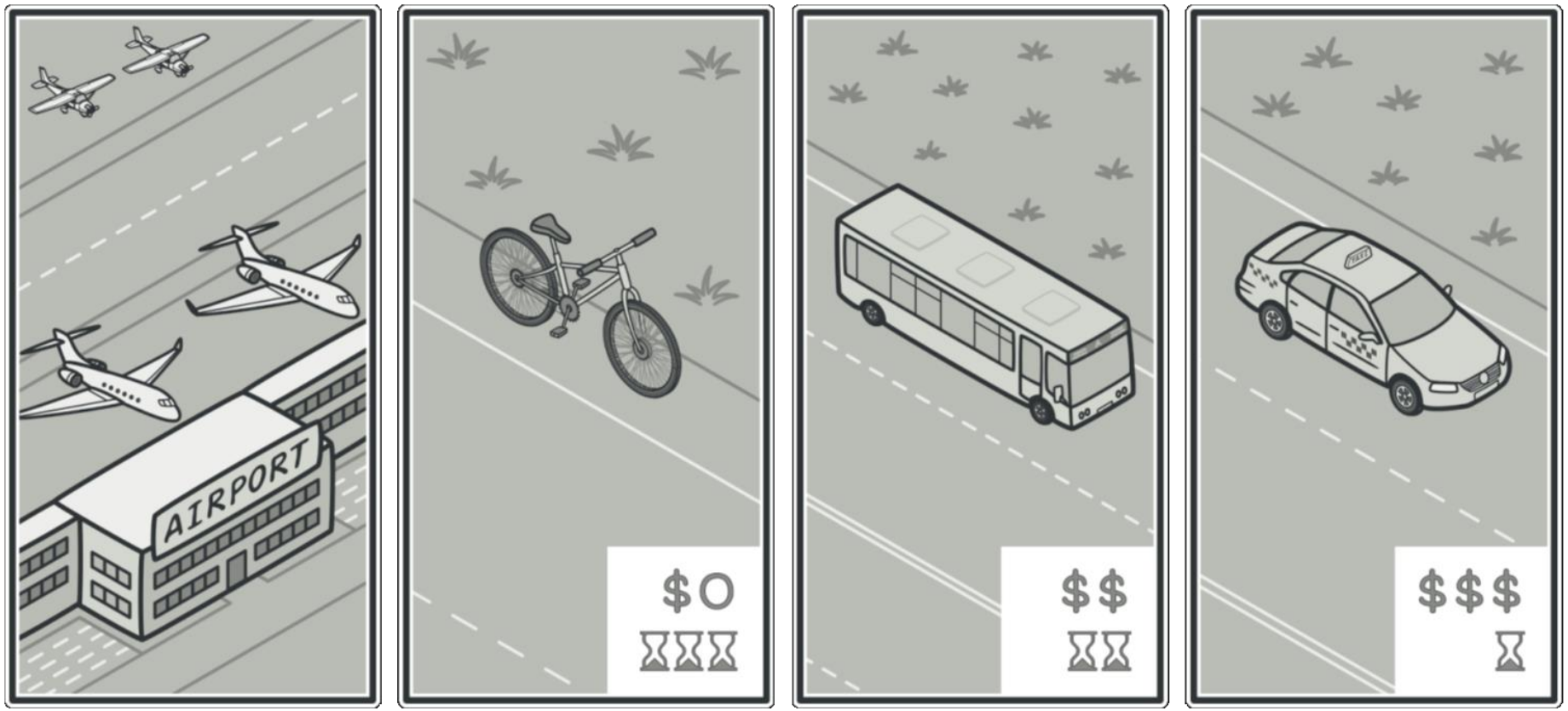


State Structure

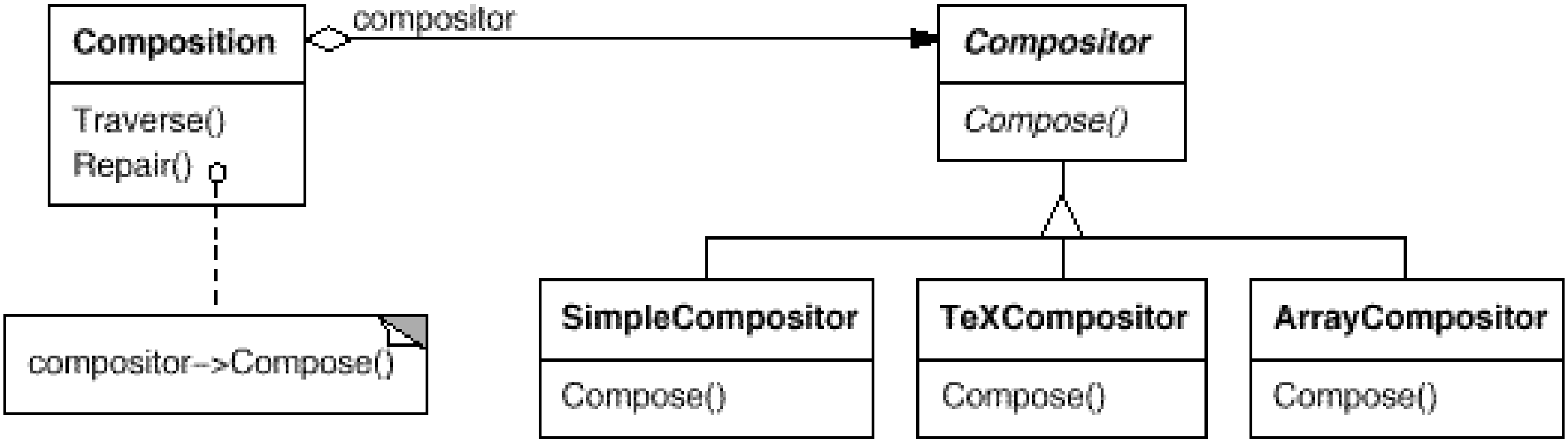


9. Strategy

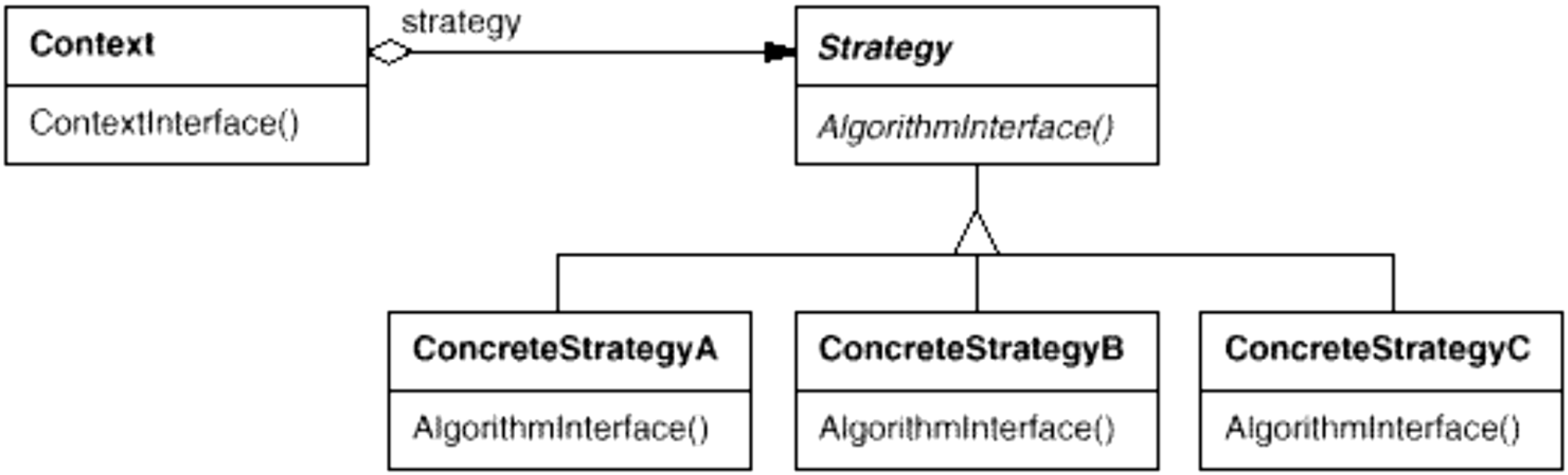
- Define a family of algorithms, put each of them into a separate class, and make their objects interchangeable



Example: Text Editor

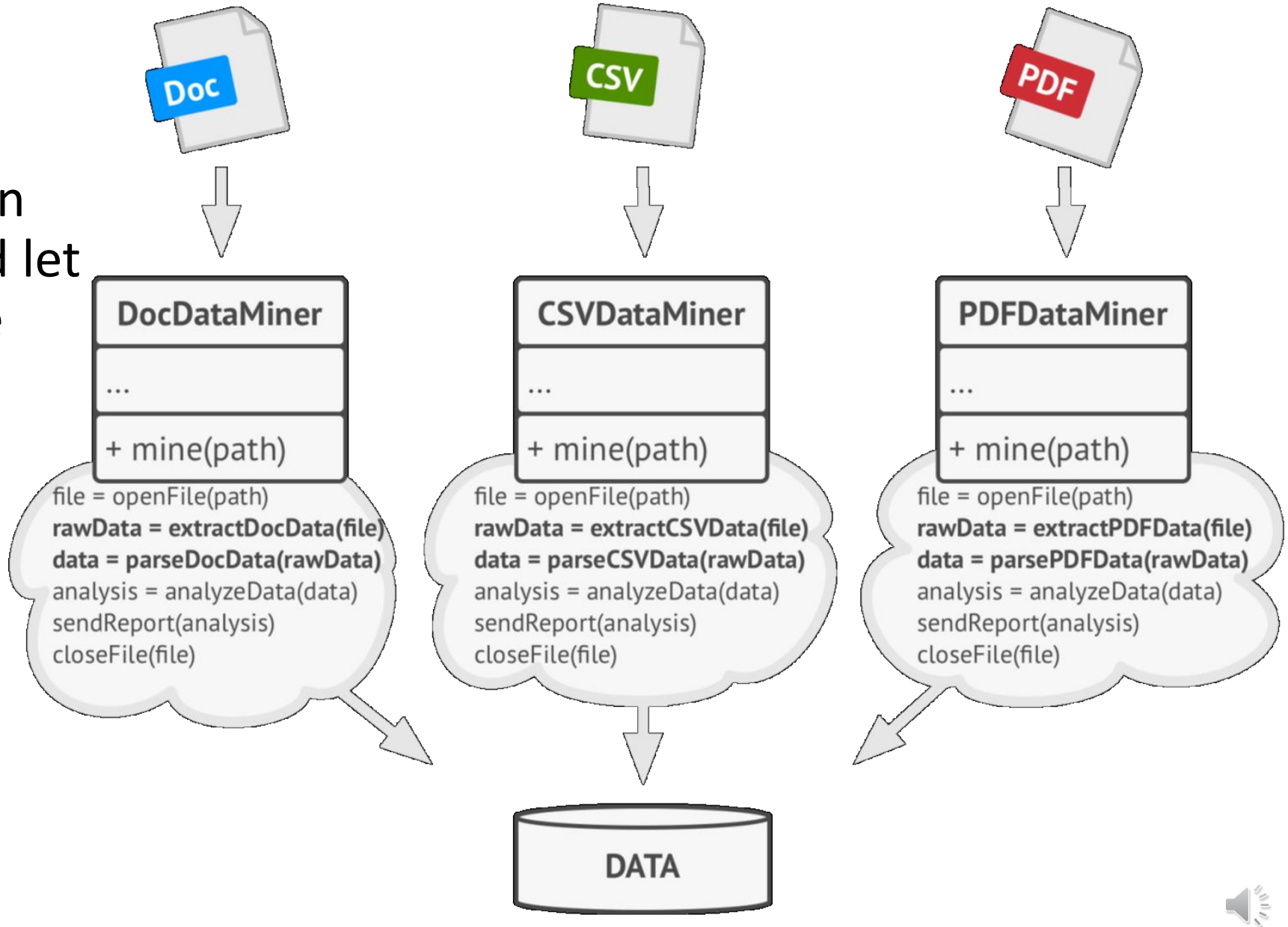


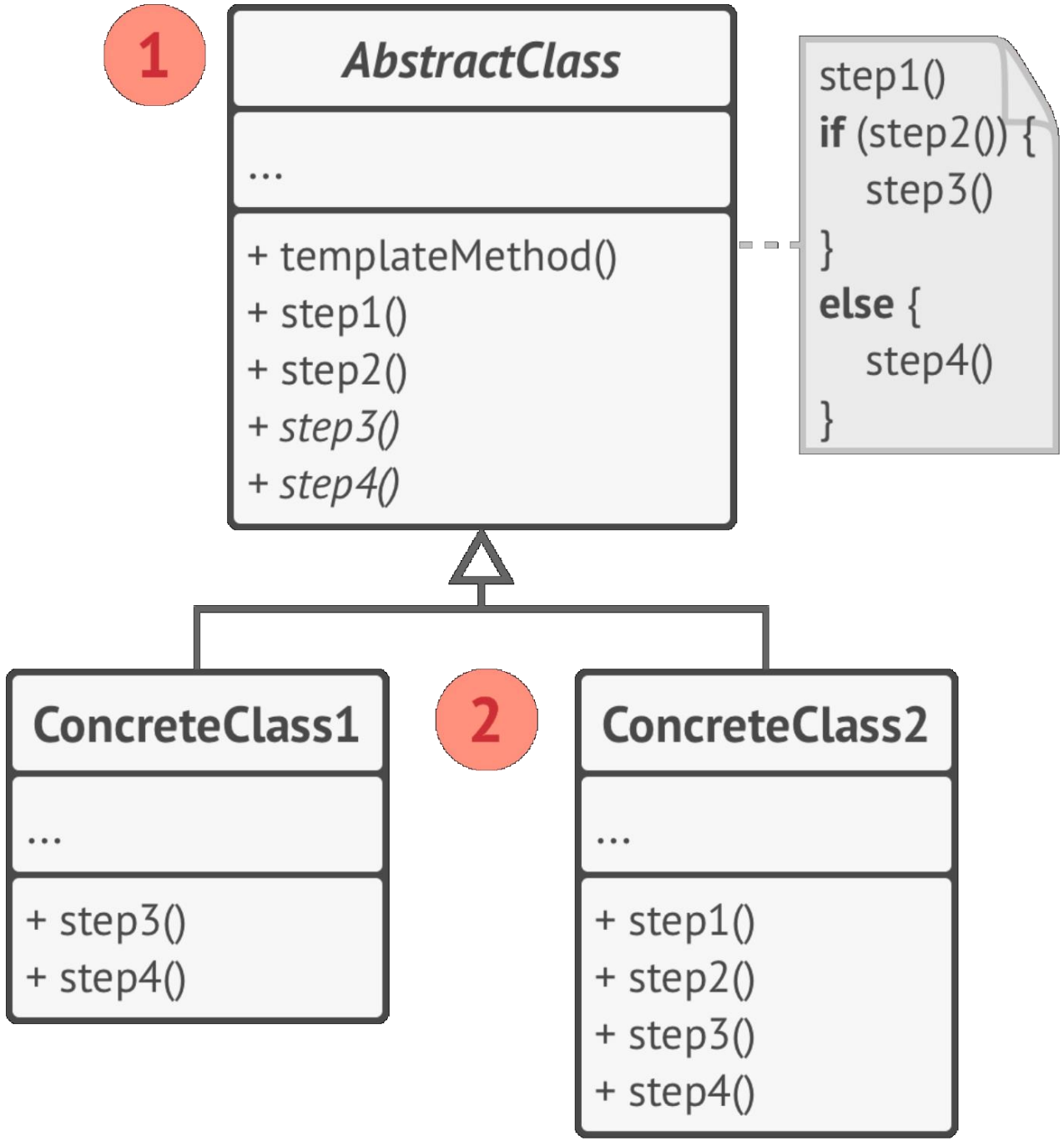
Strategy Structure



10. Template

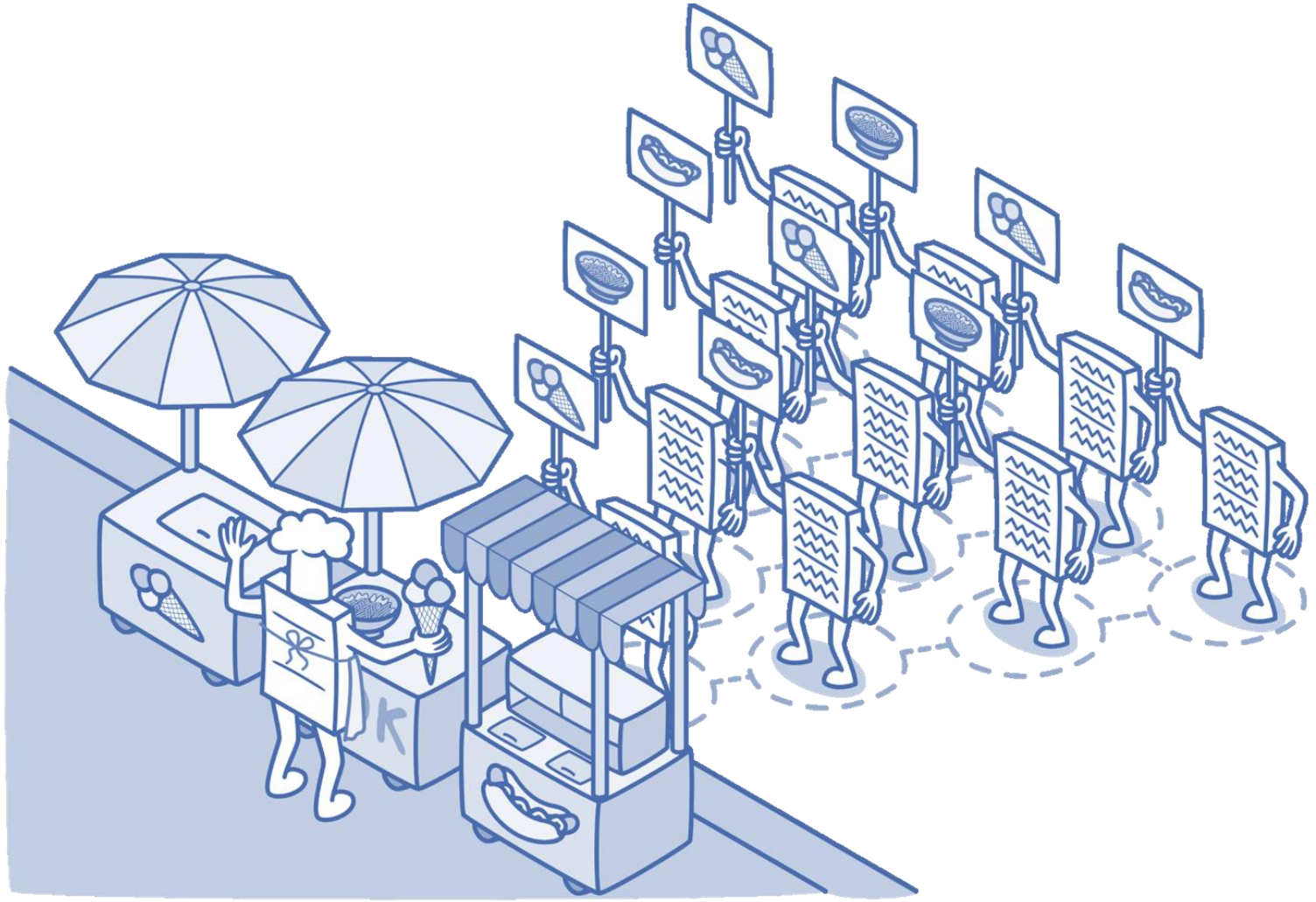
- Defines the skeleton of an algorithm and let subclasses override specific steps




1
AbstractClass
...
+ templateMethod()
+ step1()
+ step2()
+ step3()
+ step4()
step1()
if (step2()) {
step3()
}
else {
step4()
}
ConcreteClass1
...
+ step3()
+ step4()
2
ConcreteClass2
...
+ step1()
+ step2()
+ step3()
+ step4()

11. Visitor

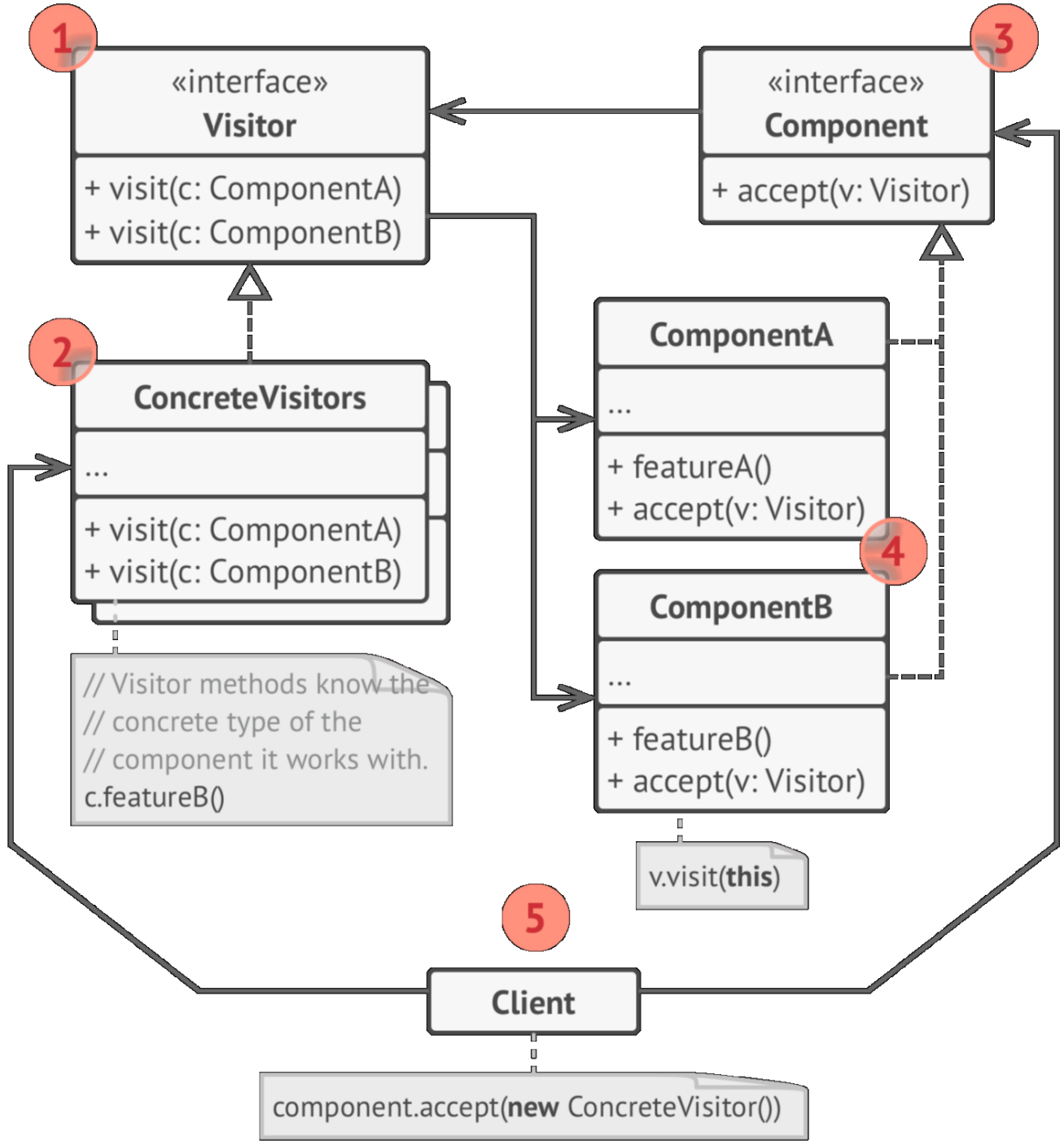
- Separate algorithms from the objects on which they operate



Visitor vs. Iterator

- Visitor Pattern is used to perform an action on a structure of elements

```
public void VisitorExample()
{
    MyVisitorImplementation visitor = new MyVisitorImplementation();
    List<object> myListToHide = GetList();

    //Here you hide that the aggregate is a List<object>
    ConcreteIterator i = new ConcreteIterator(myListToHide);

    IAcceptor item = i.First();
    while (item != null)
    {
    item.Accept(visitor);
    item = i.Next();
    }
    //... do something with the result
}
```


1
«interface»
Visitor
+ visit(c: ComponentA)
+ visit(c: ComponentB)
3
«interface»
Component
+ accept(v: Visitor)
2
ConcreteVisitors
...
+ visit(c: ComponentA)
+ visit(c: ComponentB)
// Visitor methods know the
// concrete type of the
// component it works with.
c.featureB()
ComponentA
...
+ featureA()
+ accept(v: Visitor)
4
ComponentB
...
+ featureB()
+ accept(v: Visitor)
v.visit(this)
5
Client
component.accept(new ConcreteVisitor())

References

- Alexander Shvets, “Dive into Design Patterns,” 2018
- https://www.tutorialspoint.com/design_pattern/index.htm
- Erich Gamma, Richard Helm, Ralph Johnson , John Vlissides, “Design Patterns,” 1994