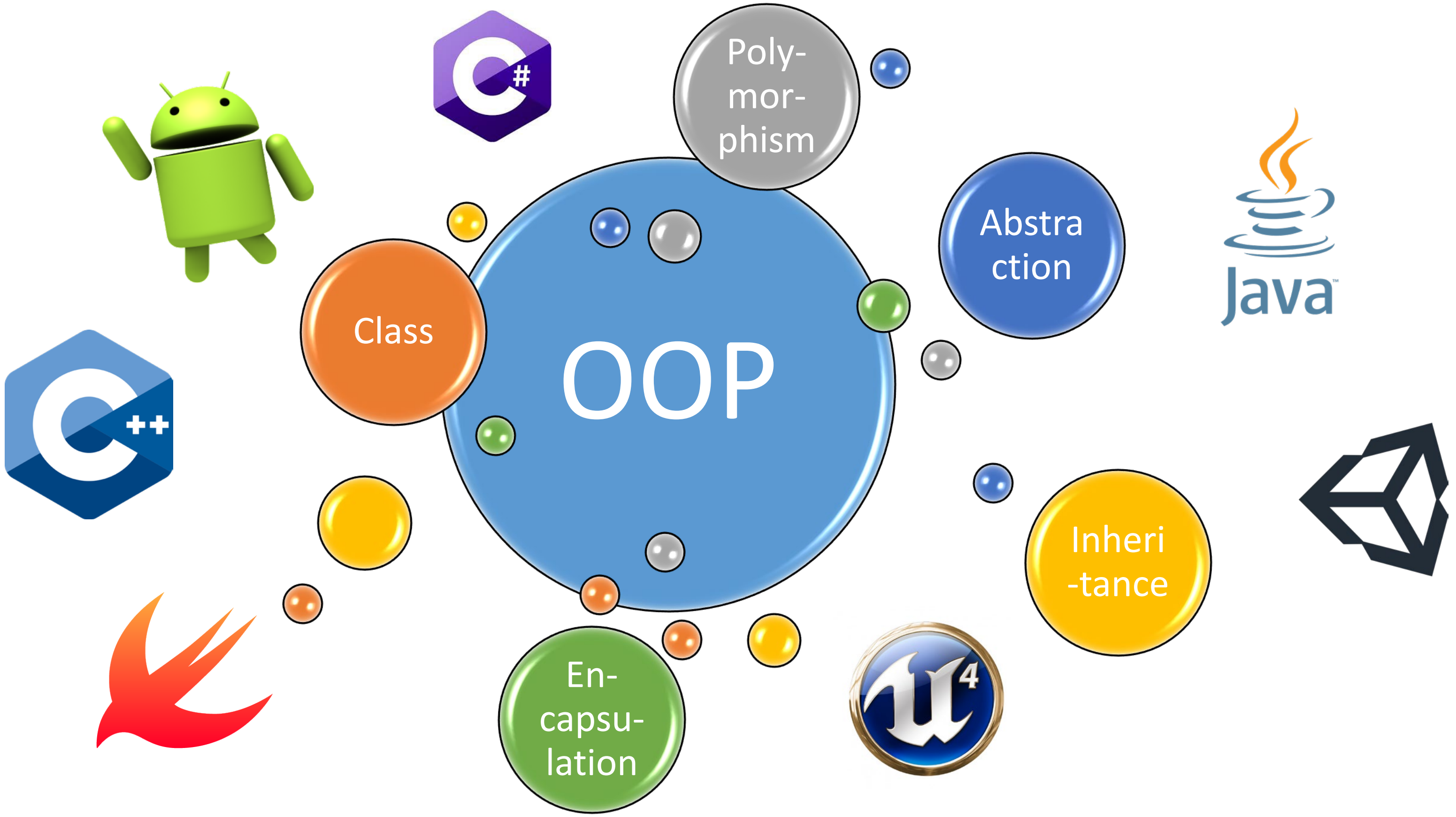


Behavioral Design Patterns

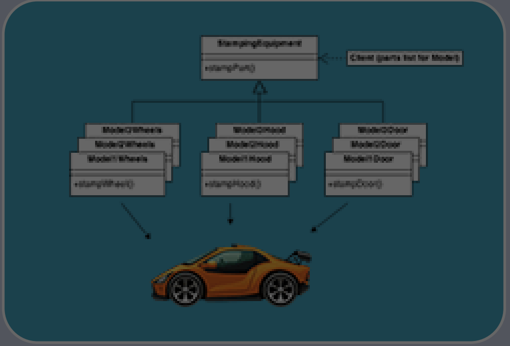
Kuan-Ting Lai

2023/4/7

Behavioral Design Patterns

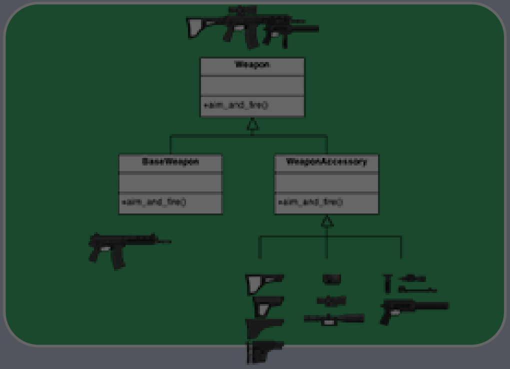
Creational Design Patterns

Initialize objects or create new classes



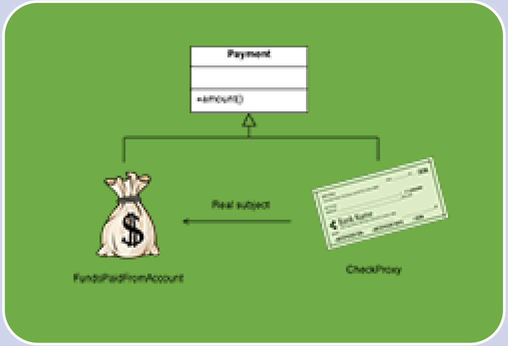
Structural Design Patterns

Compose objects to get new functions



Behavioral Design Patterns

Communication between objects

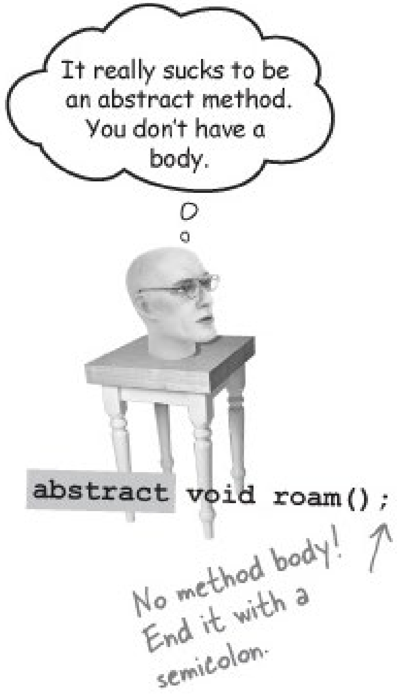


Common Behavioral Design Patterns

1. Strategy
2. Observer
3. State
4. Command
5. Template
6. Iterator
7. Chain of Responsibility

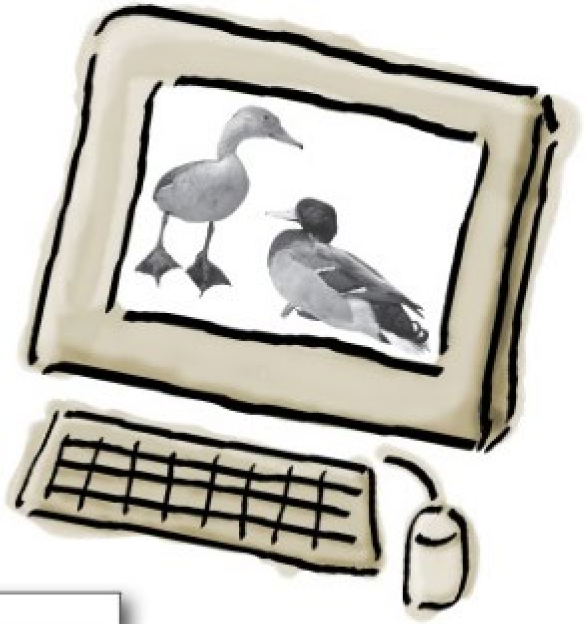
Head First Design Patterns

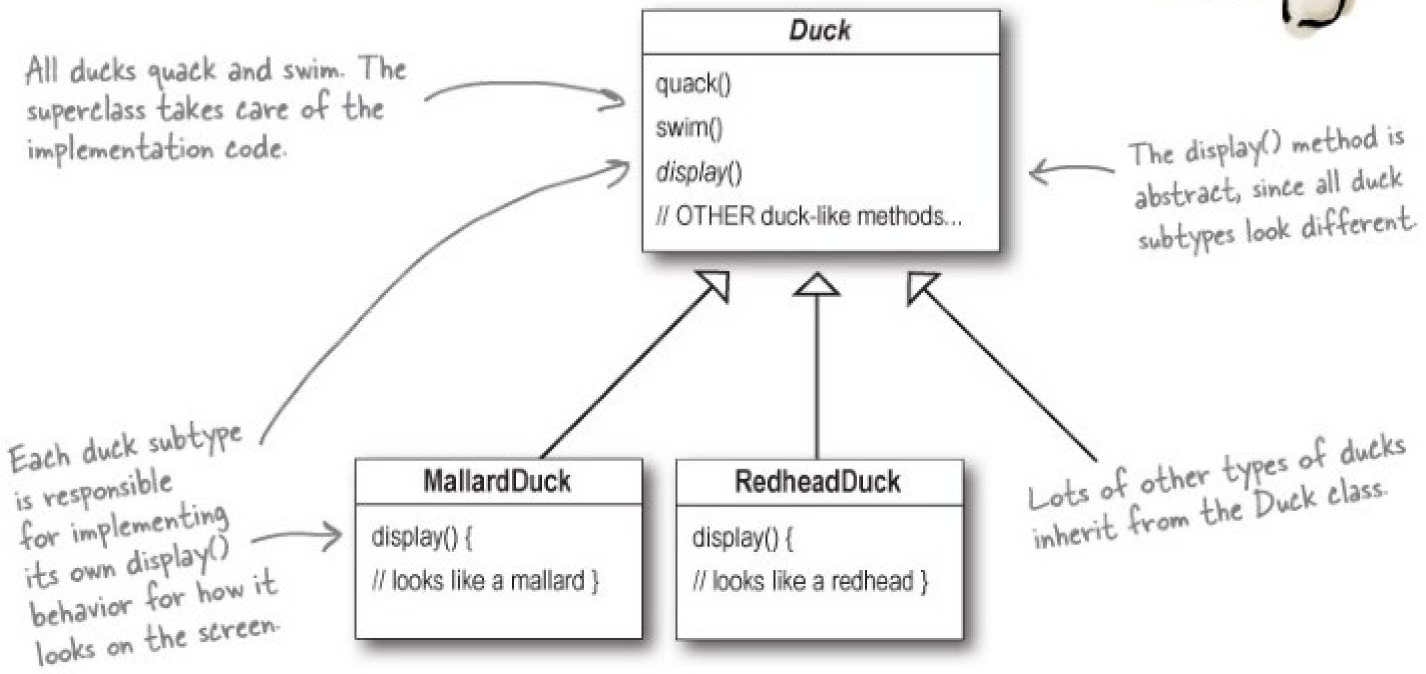
- Freeman, Eric; Robson, Elisabeth; Bates, Bert; Sierra, Kathy. Head First Design Patterns. O'Reilly Media.
- Wonderful examples and modern design patterns



Design a SimUDuck App

- Joe works for a company that makes a highly successful duck pond simulation game, SimUDuck.





We want to make ducks FLY!

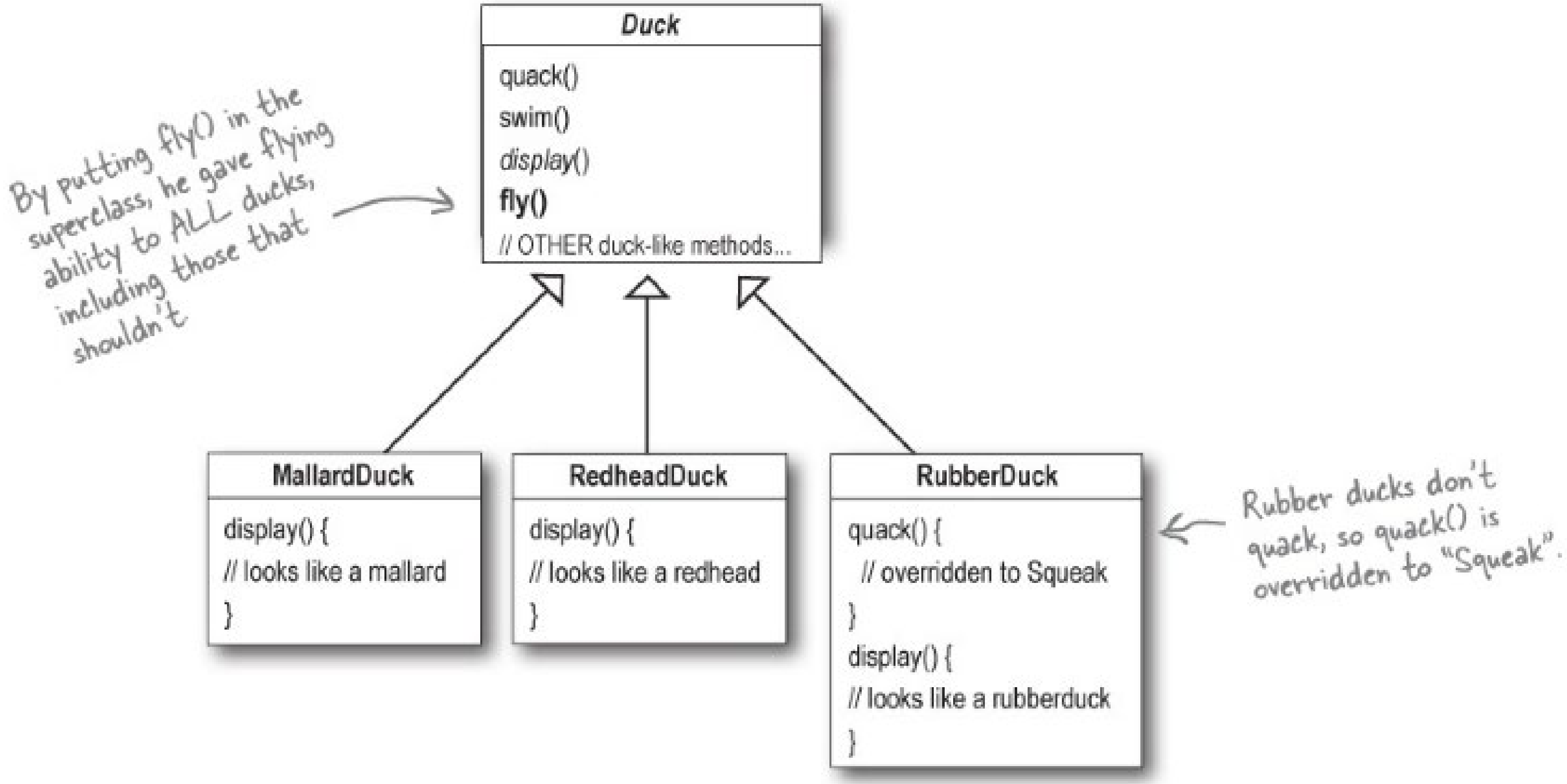
- Add new features to our game
- Let's make ducks fly
- Add a function fly() in parent class Duck

But something went horribly wrong…

- Rubber duckies flying around the screen

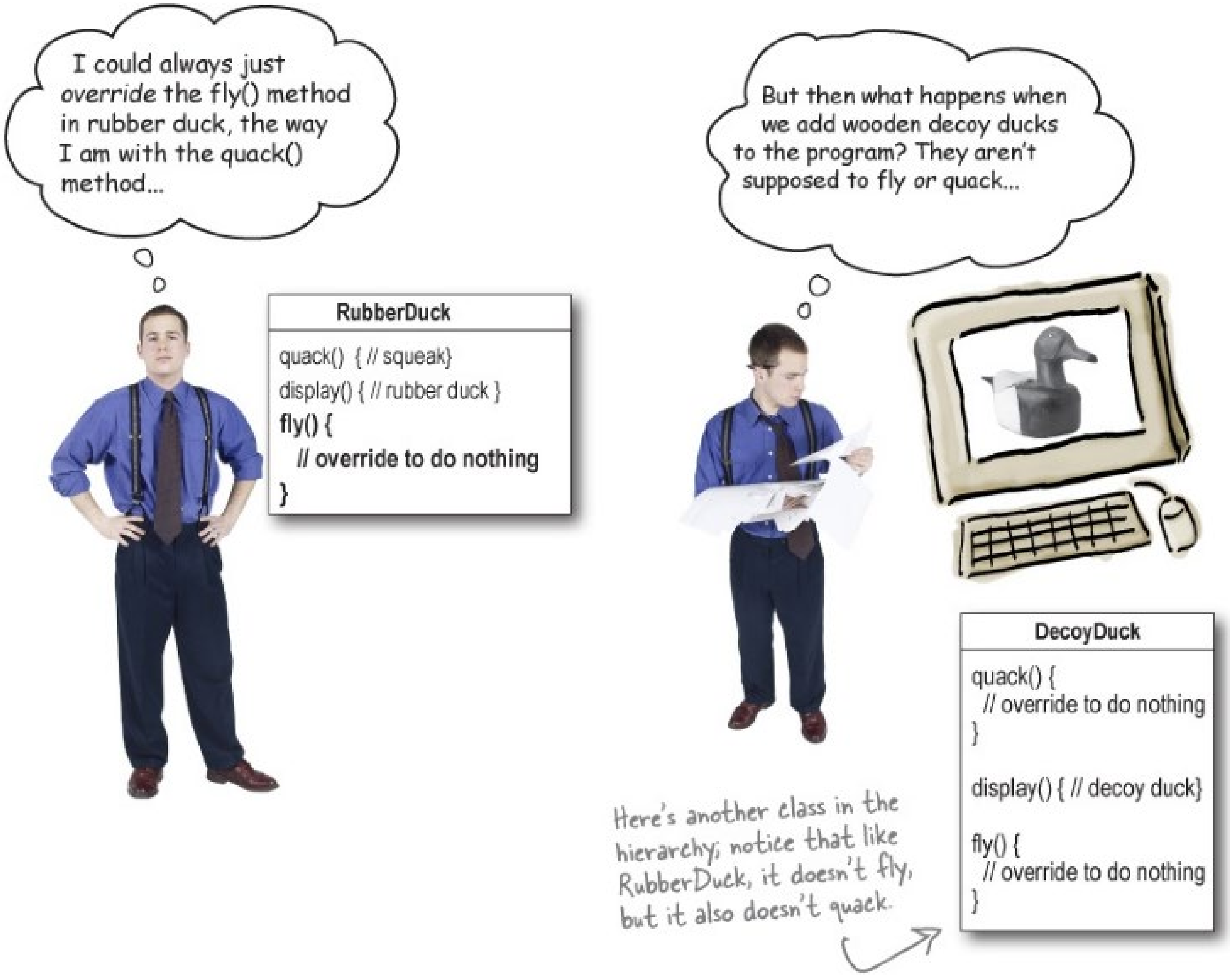
What happened?

- Not all ducks can fly, and not all ducks quack



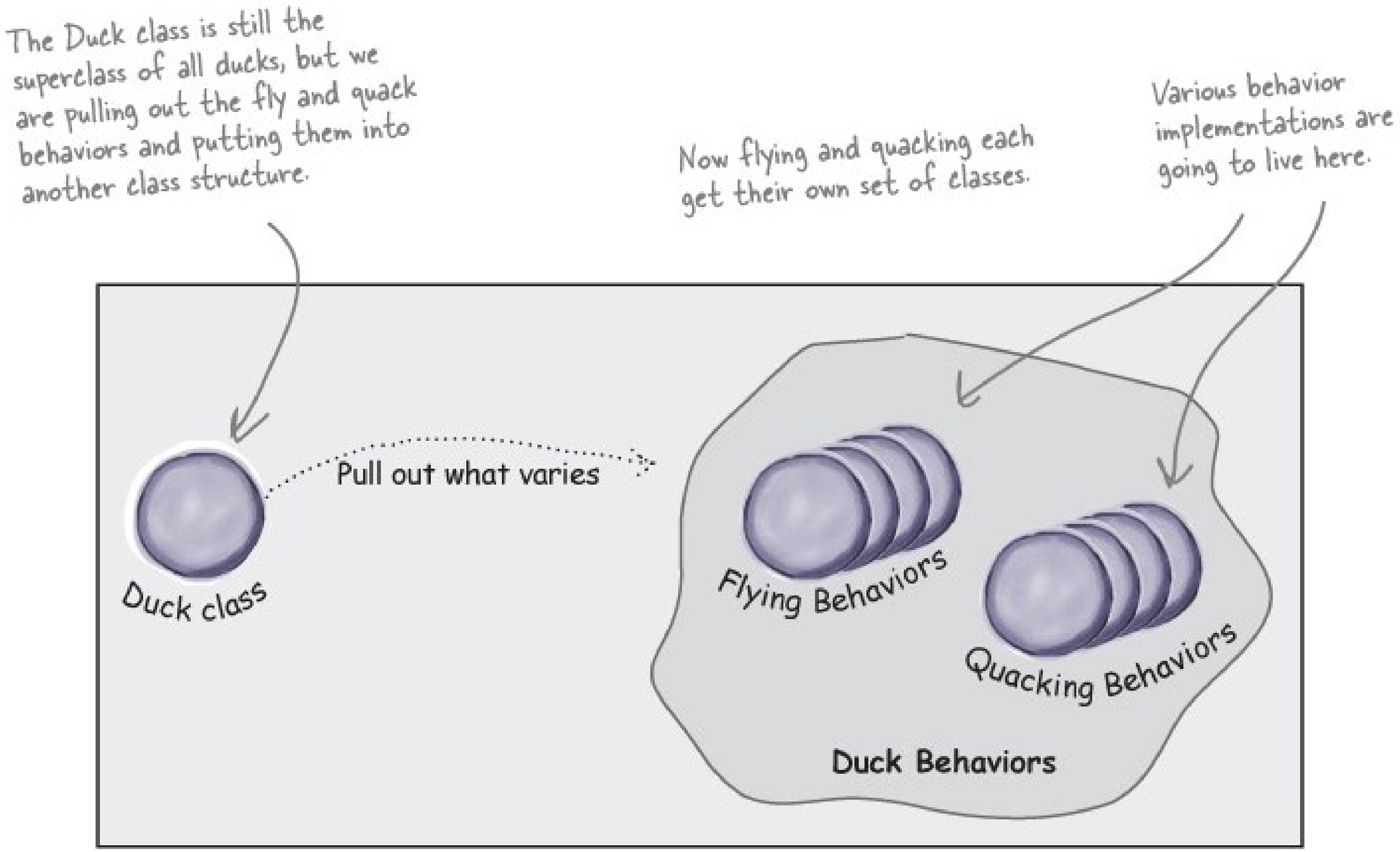
Override?

- Is there a better way than inheritance?
- What if we want to update the product every months?



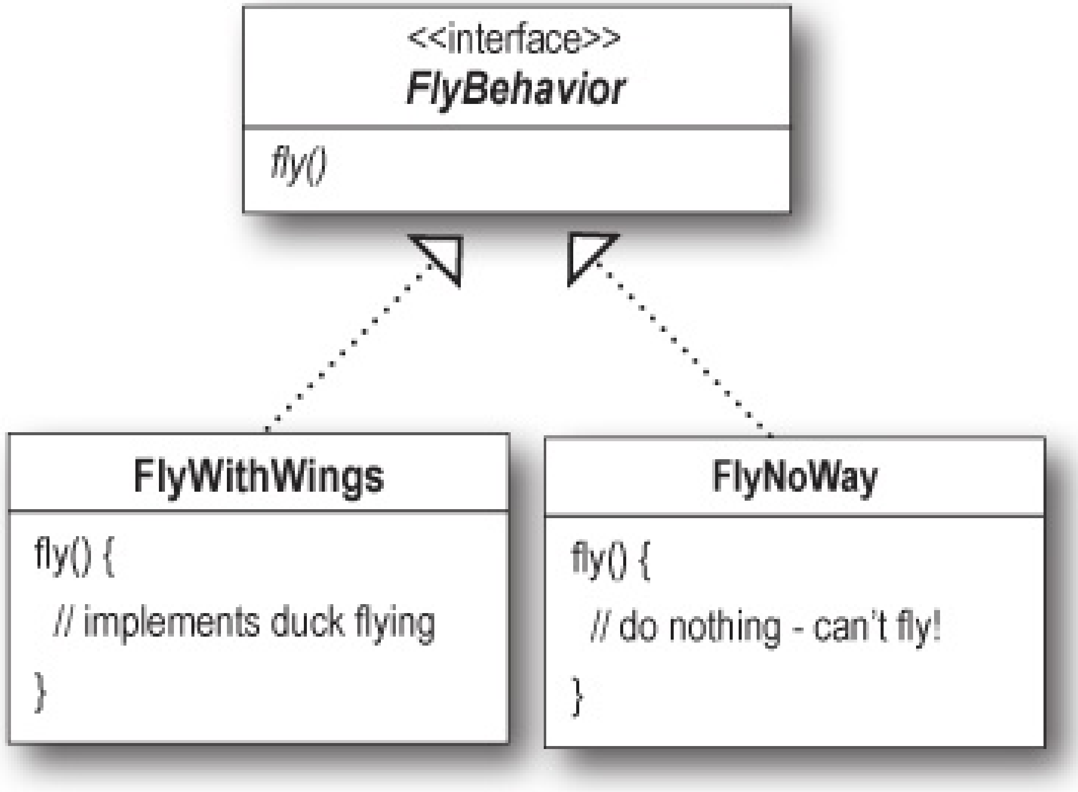
Design new classes for behaviors

- Create new classes for new behaviors
- Add new classes as member variables



In Java, use interface for behaviors

- Java interface == C++ abstract class



Programming to an interface

- Programming to an implementation would be:

    Dog d = new Dog();

    d.bark();

- But programming to an interface/ supertype would be:

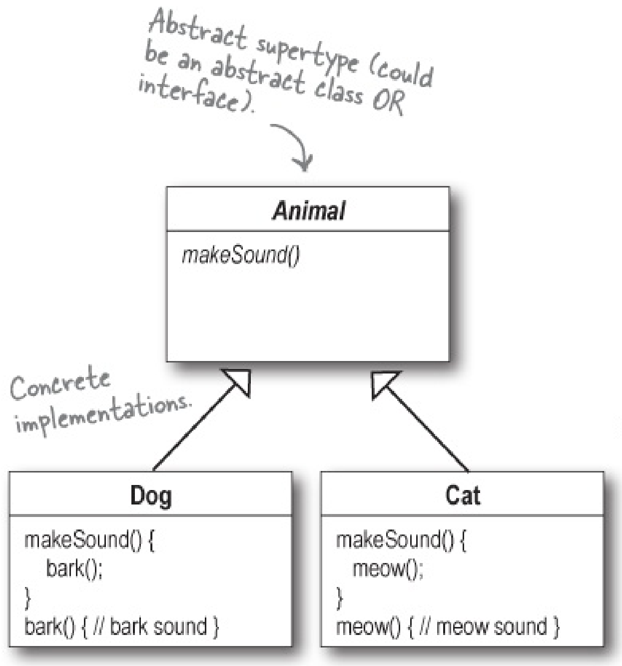
    Animal animal = new Dog();

    animal.makeSound();

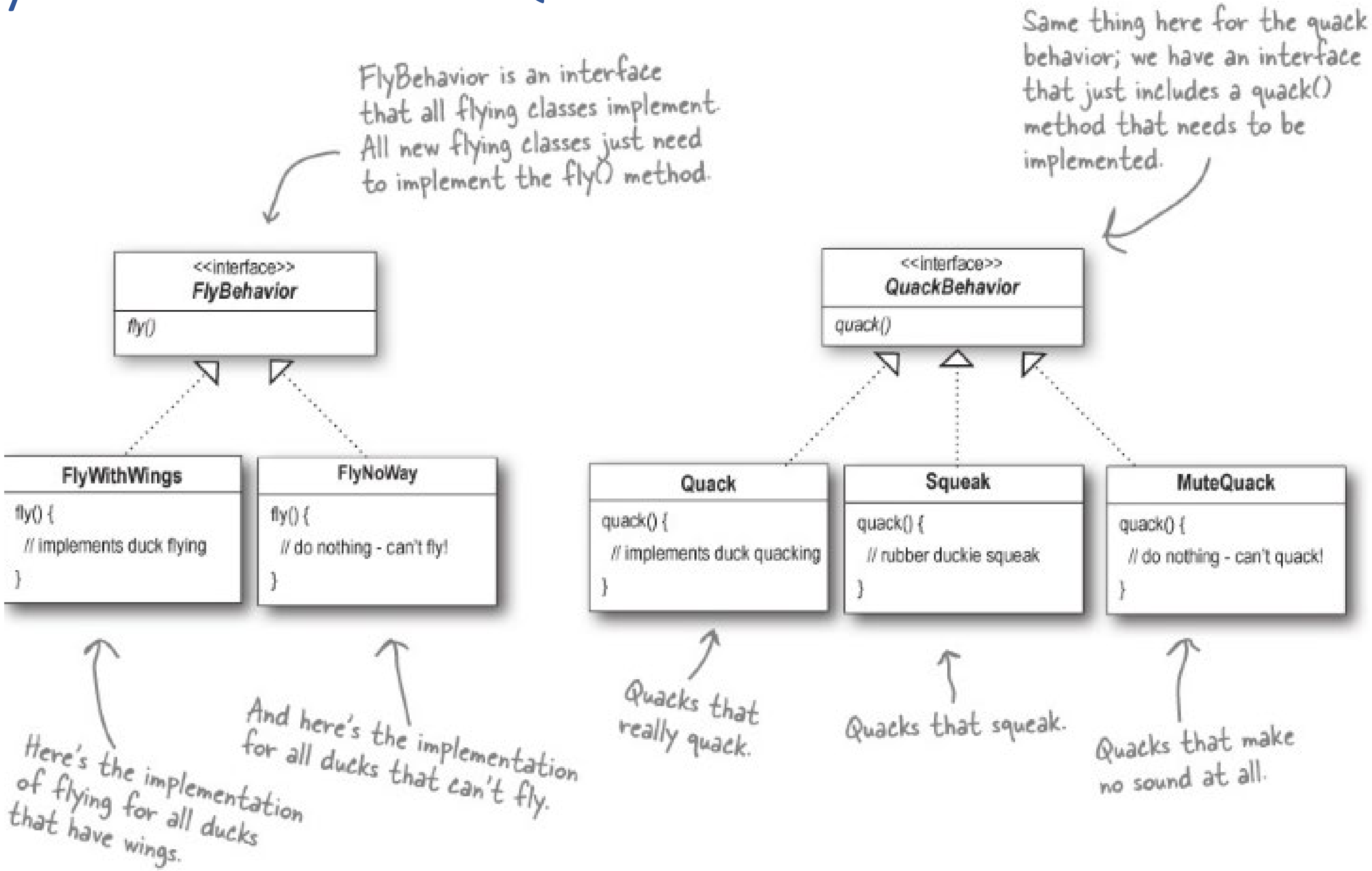
- Even better, we can assign the concrete implementation object at runtime:

    a = getAnimal();

    a.makeSound();

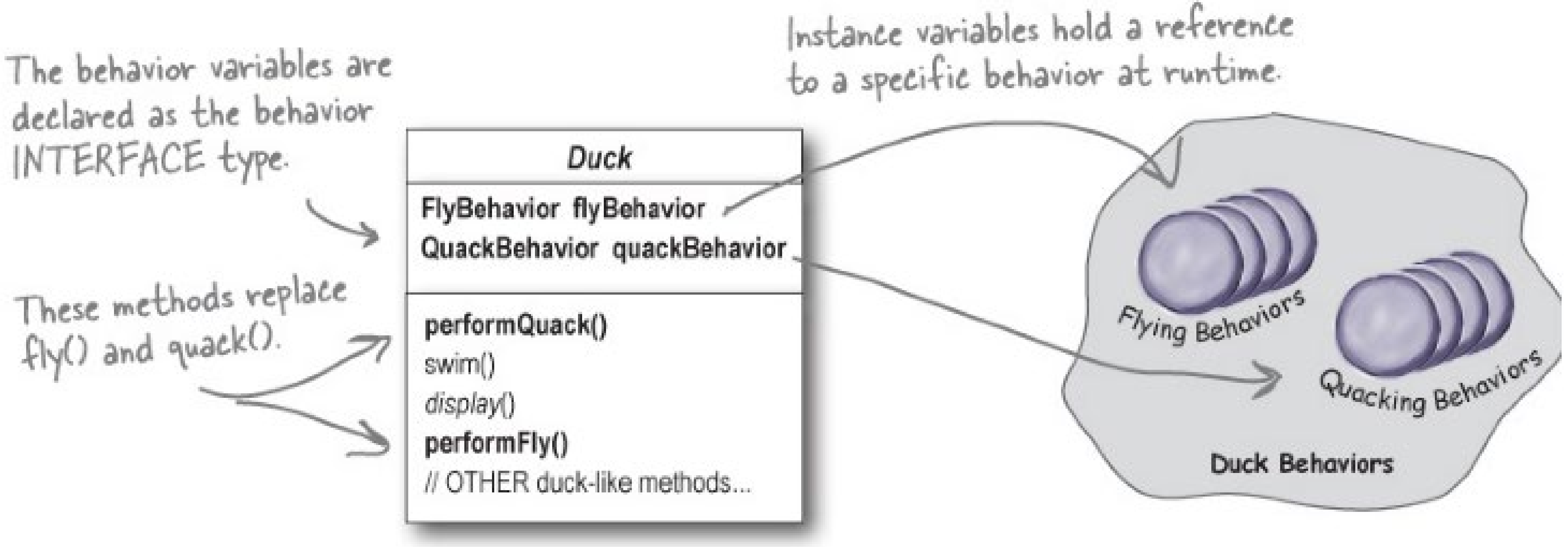


FlyBehavior and QuackBehavior



Delegate flying and quacking behavior

- Make flying and quacking behaviors as member variables, and use performFly() and performQuack to call them.



Inherit the Duck class

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack() is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

Testing the Duck code (1)

- Type and compile the Duck class below (Duck.java), and the MallardDuck class from two pages back (MallardDuck.java)

```java
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() {
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

Delegate to the behavior class.

https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/ducks

Testing the Duck Code (2)

- Type and compile the FlyBehavior interface (FlyBehavior.java) and the two behavior implementation classes (FlyWithWings.java and FlyNoWay.java).

```
public interface FlyBehavior {
   public void fly();
}
```

The interface that all flying behavior classes implement.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

Flying behavior implementation for ducks that DO fly...

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("I can't fly");
   }
}
```

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

Testing the Duck Code (3)

- Type and compile the QuackBehavior interface (QuackBehavior.java) and the 3 behavior implementation classes (Quack.java, MuteQuack.java, and Squeak.java).

```
public interface QuackBehavior {
   public void quack();
}
______________________________________________________________
___
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}
______________________________________________________________
___
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Silence >>");
   }
}
______________________________________________________________
___
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Squeak");
   }
}
```

Testing the Duck Code (4)

- Type and compile the test class (MiniDuckSimulator.java).

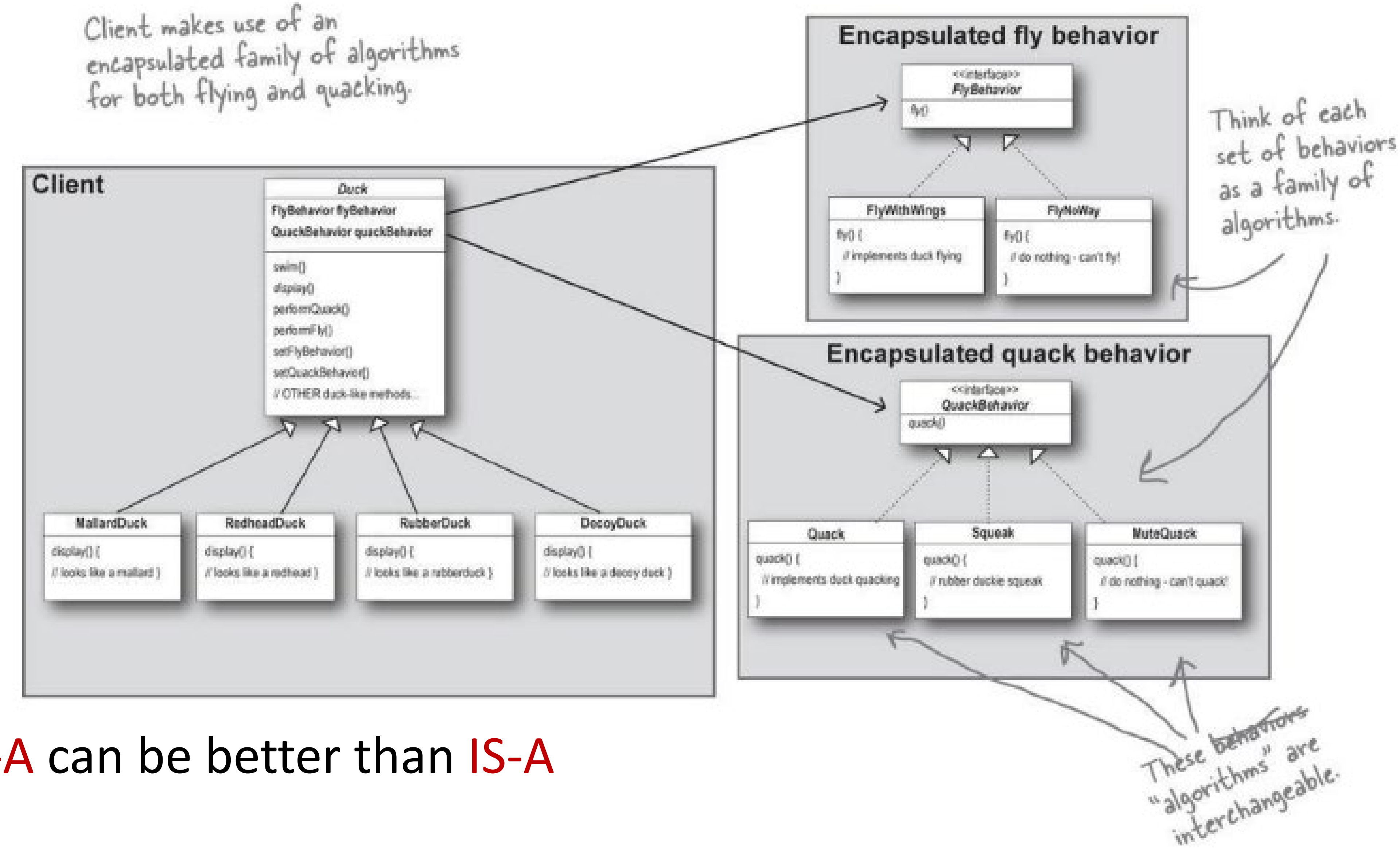
```java
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

This calls the MallardDuck's inherited performQuack() method, which then delegates to the object's QuackBehavior (i.e., calls quack() on the duck's inherited quackBehavior reference).

Then we do the same thing with MallardDuck's inherited performFly() method.

```
File  Edit  Window  Help  Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

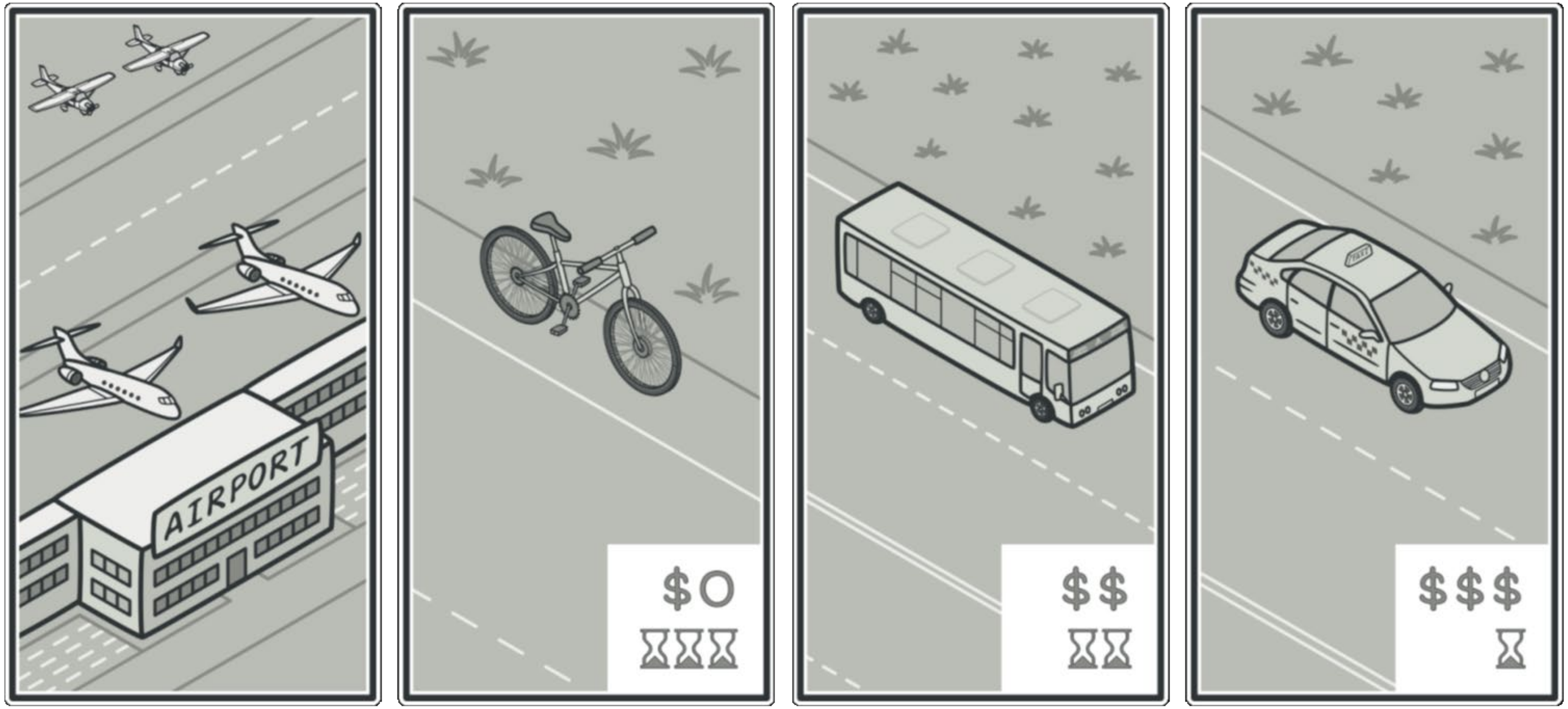
The new Duck OOP diagram



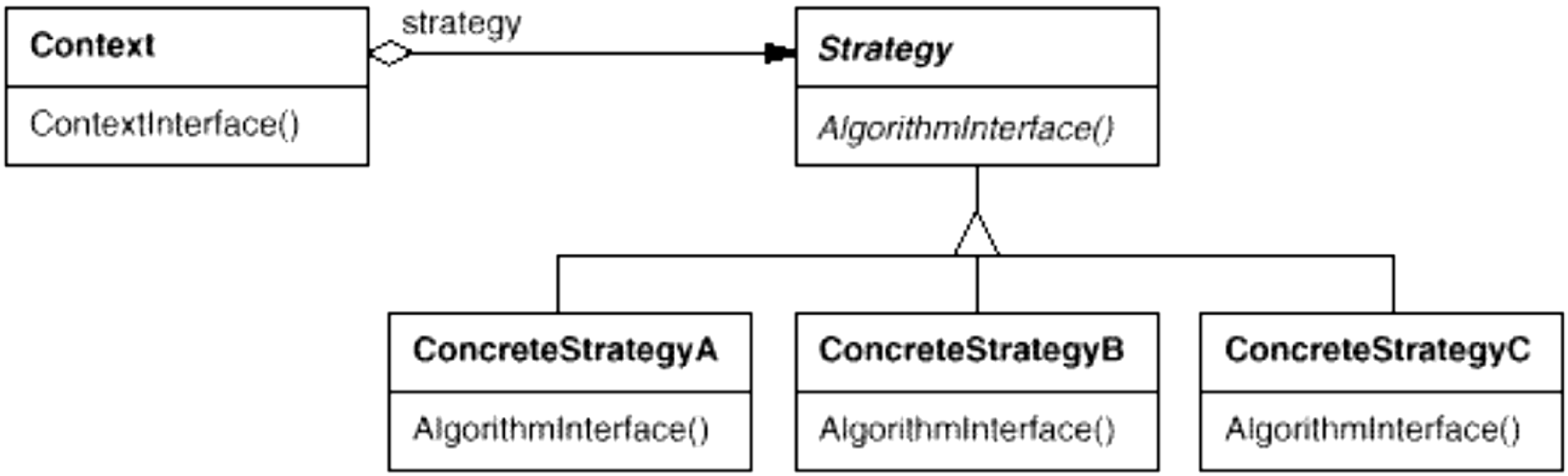
- HAS-A can be better than IS-A

Strategy Pattern

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

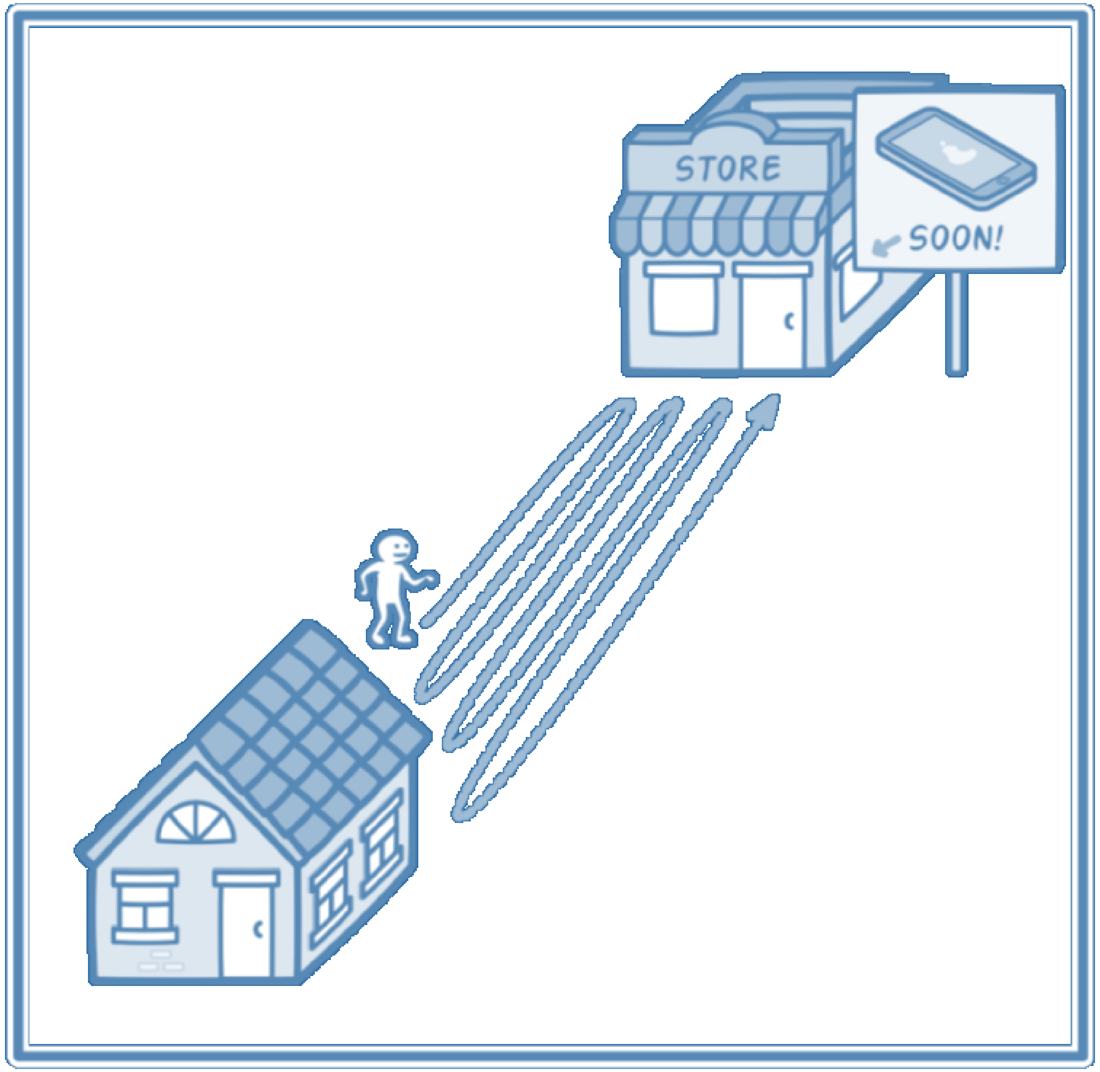


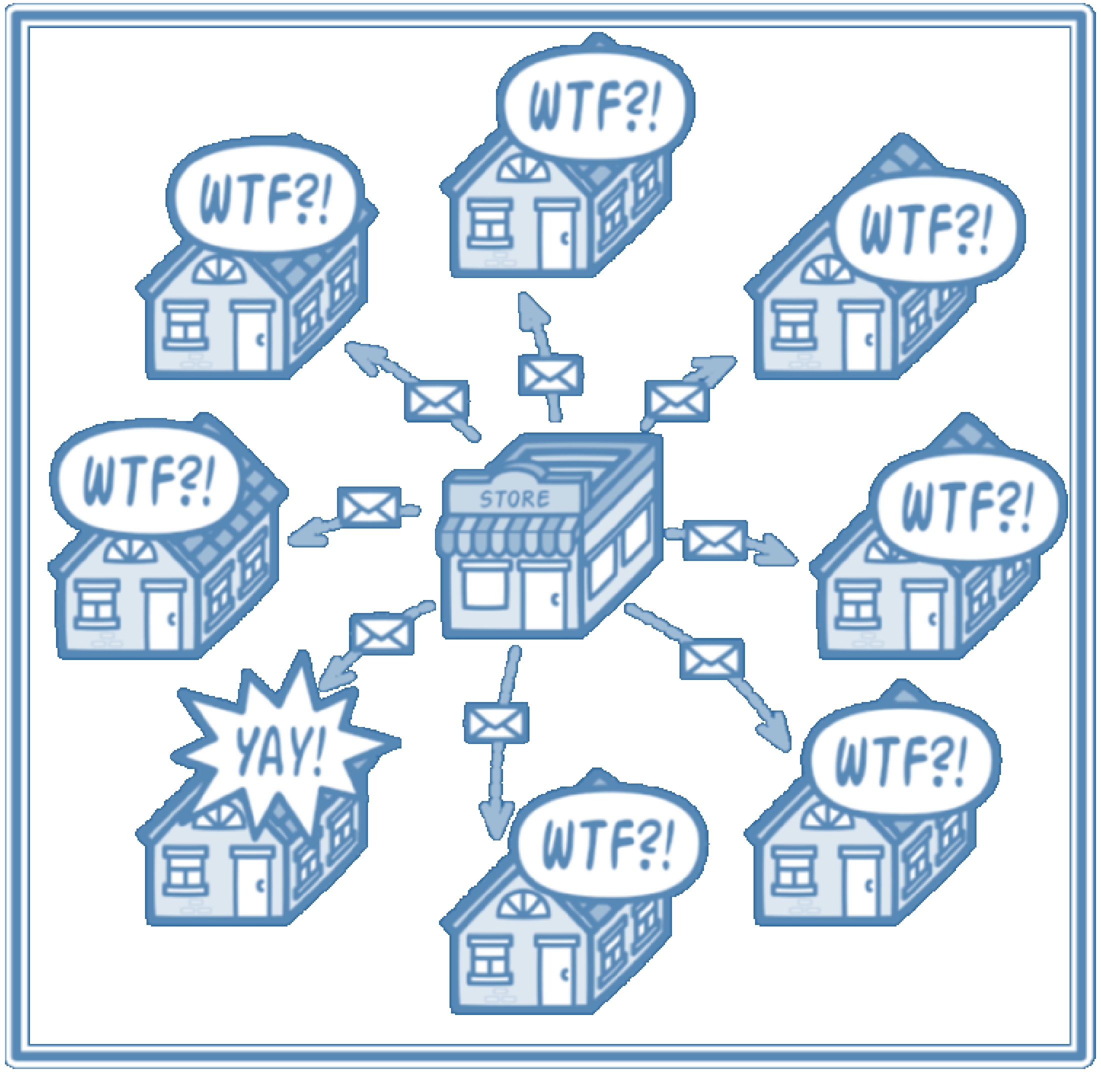
Strategy Structure

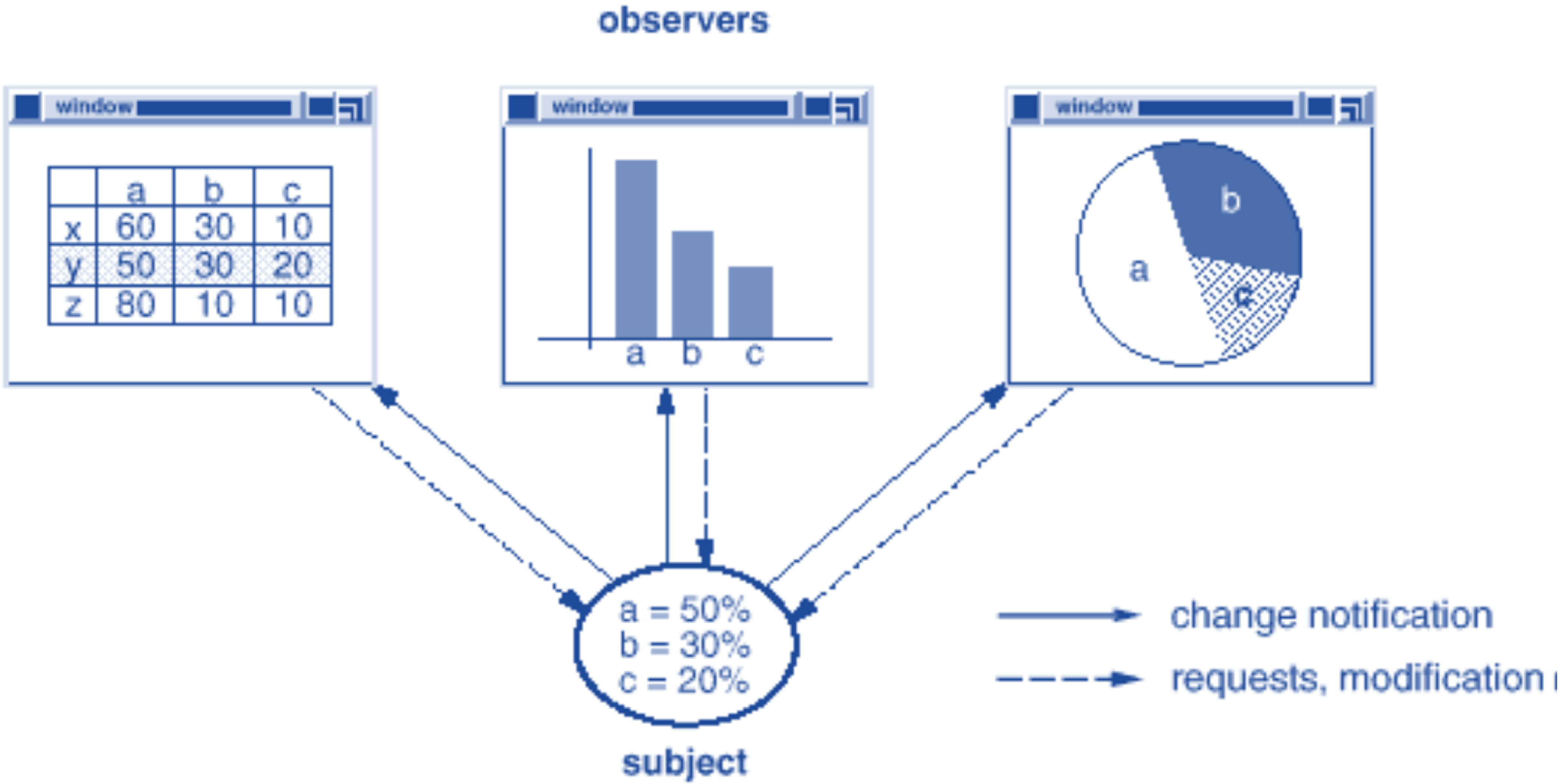


Observer

- Define a subscription mechanism to notify multiple objects



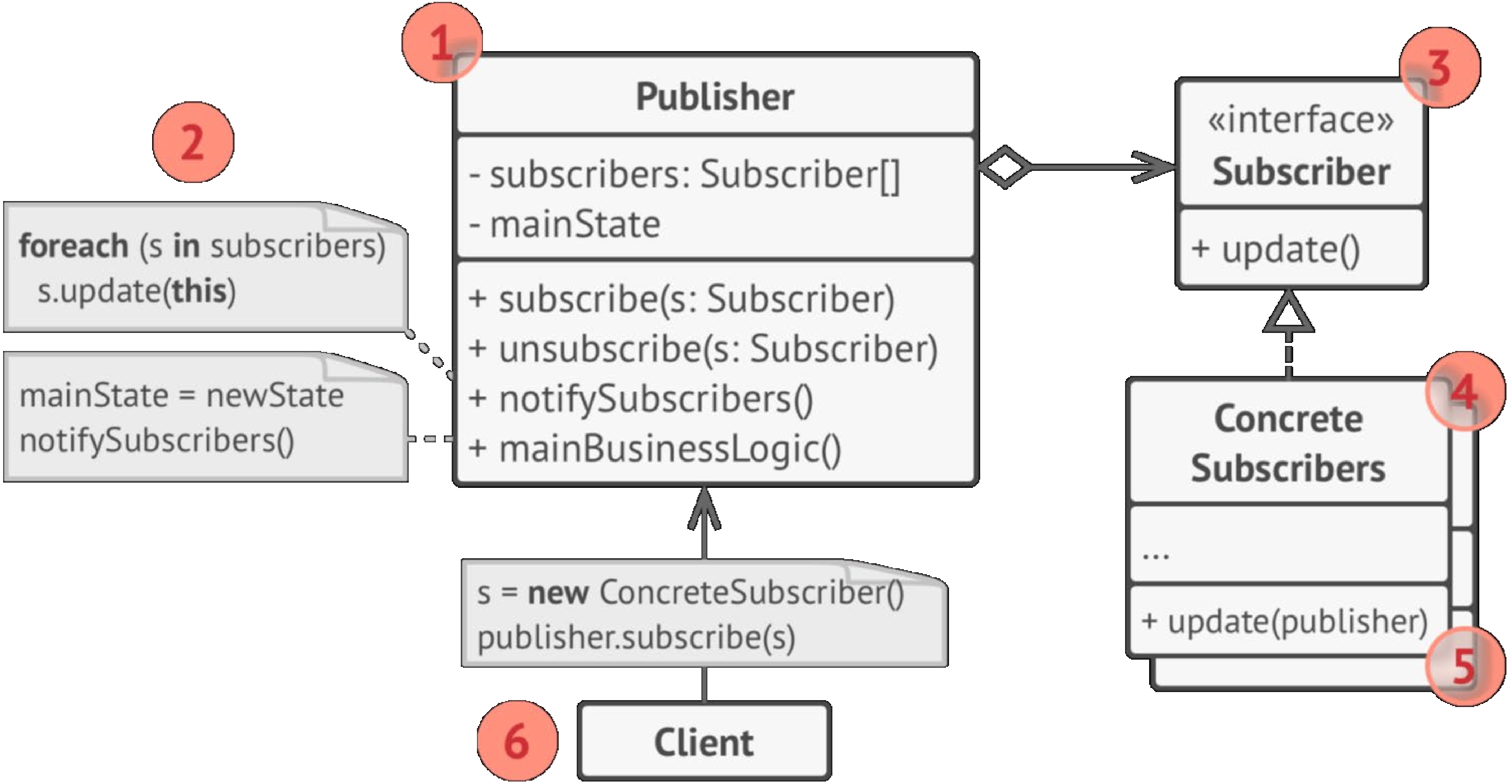



observers
window
window
window
a
b
c
x
60
30
10
y
50
30
20
z
80
10
10
a
b
c
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

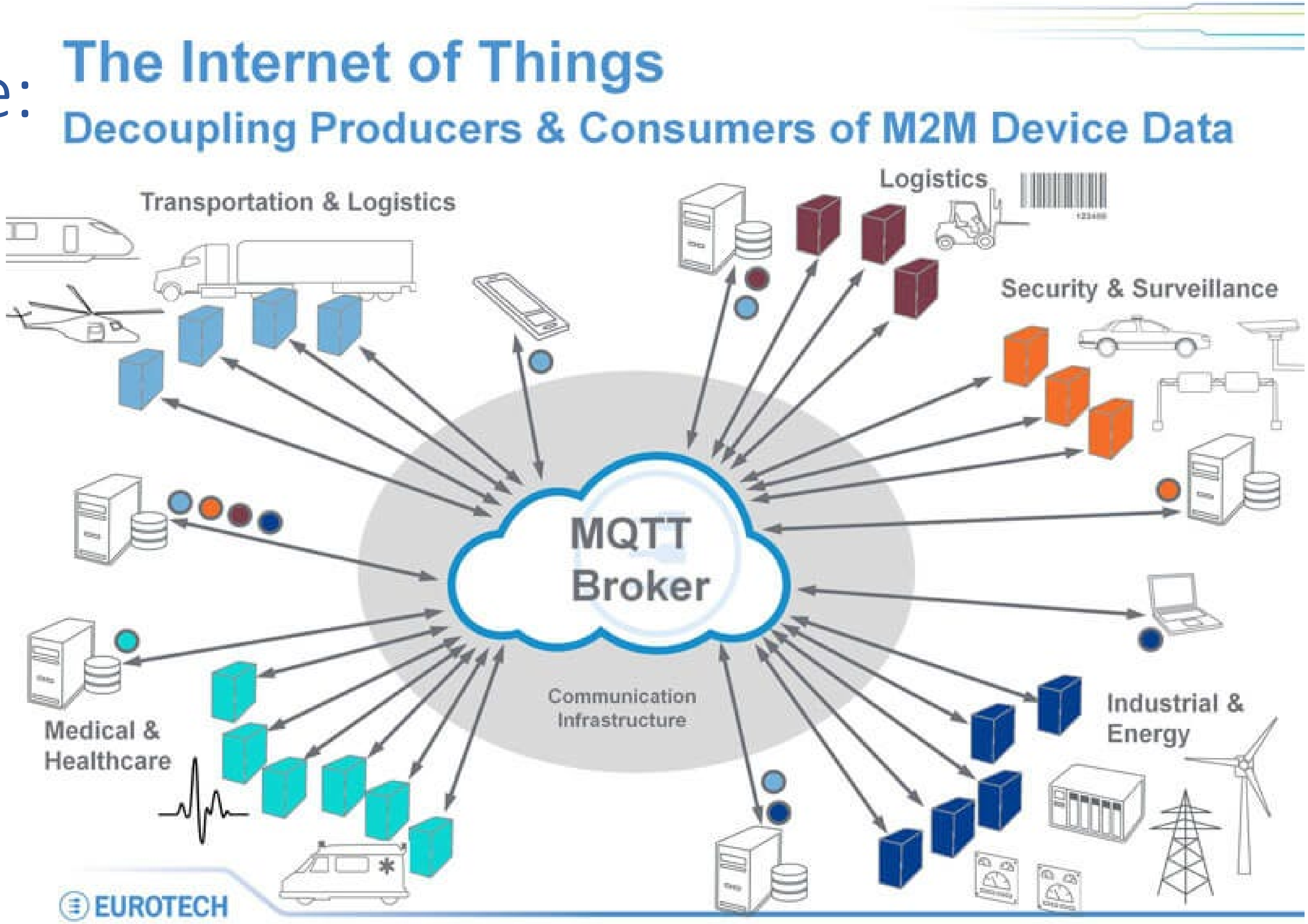
ActionListener is Observer Pattern

```
public class CalculatorForm {
    private JTextField displayField;
    private JPanel CalcPanel;
    private JButton buttonCE;
    private JButton button0;
    ……
    ……
    public CalculatorForm() {
        button0.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {

            }
        });
     ……
```

Enter Your Own Code Here

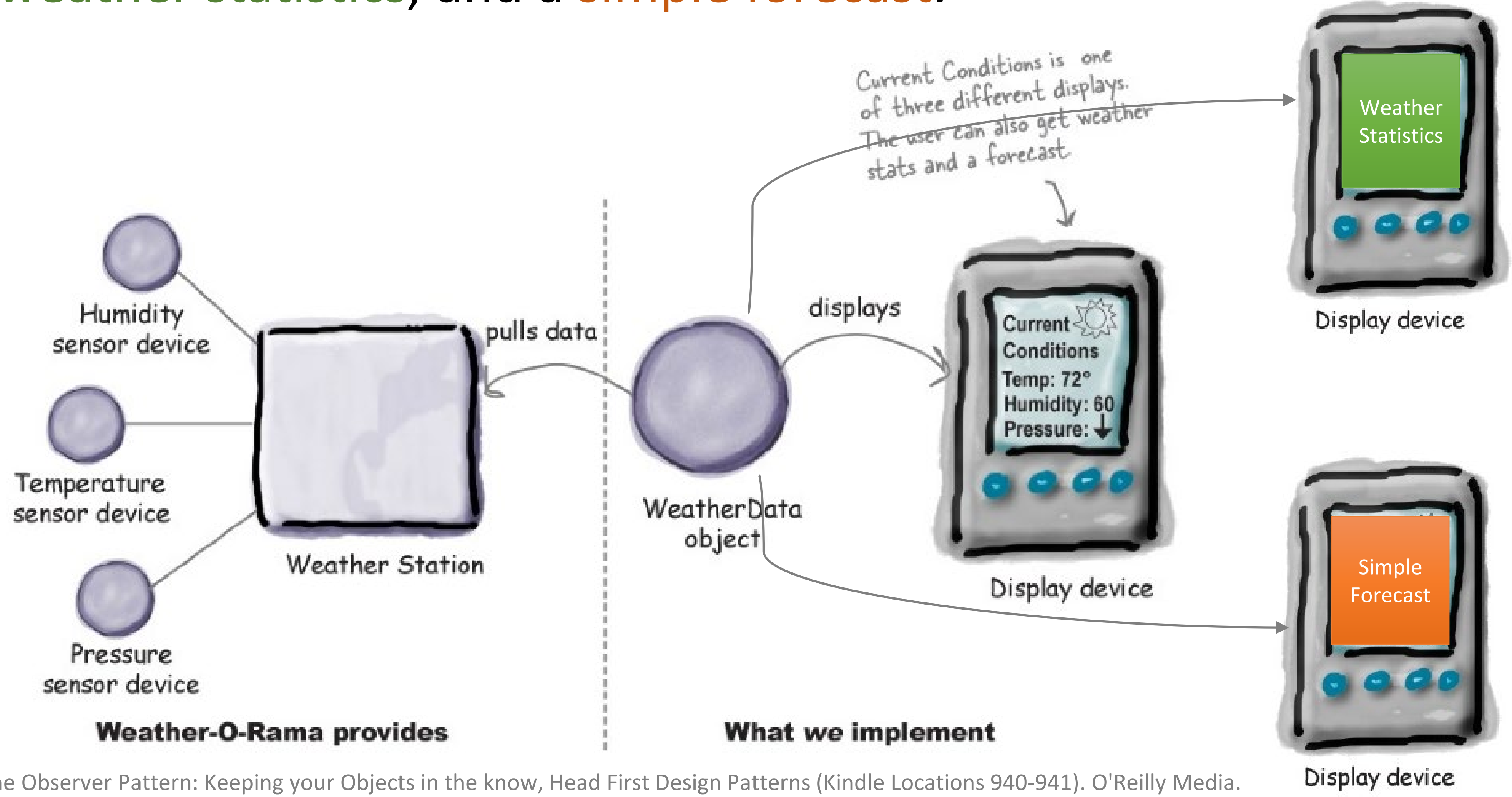

1
Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
2
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
3
«interface»
Subscriber
+ update()
4
Concrete
Subscribers
...
+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

Case Study: A Weather Monitoring

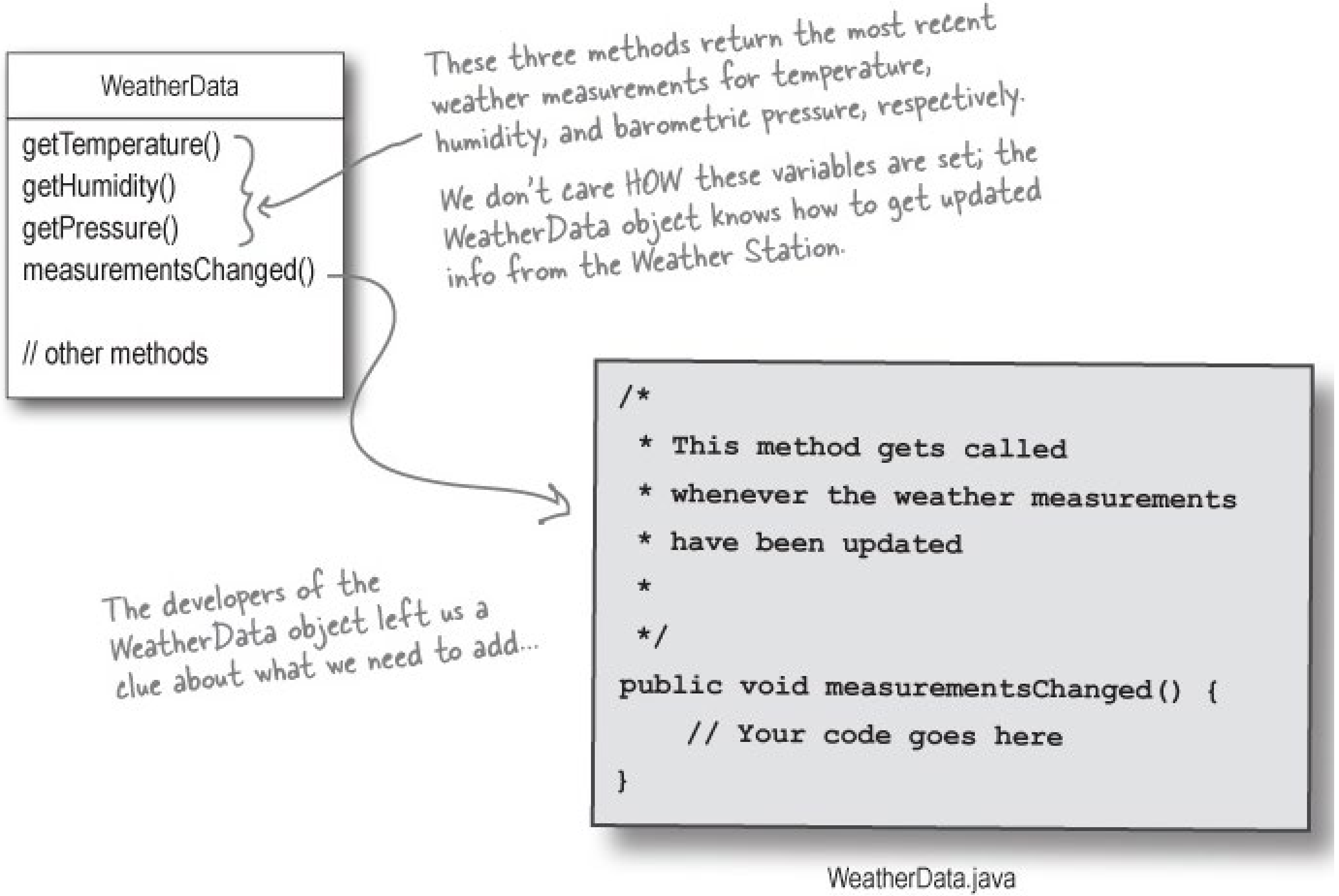
- A weather company provides APIs to provide weather information.
- We need to read the information and show on 3 displays: current conditions, weather statistics, and a simple forecast.



Chapter 2. The Observer Pattern: Keeping your Objects in the know, Head First Design Patterns (Kindle Locations 940-941). O'Reilly Media.

The API class: WeatherData

- The 3 APIs are packed in class WeatherData



1st Implementation of measurementsChanged()

- But it's hard to add new display in the future!

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

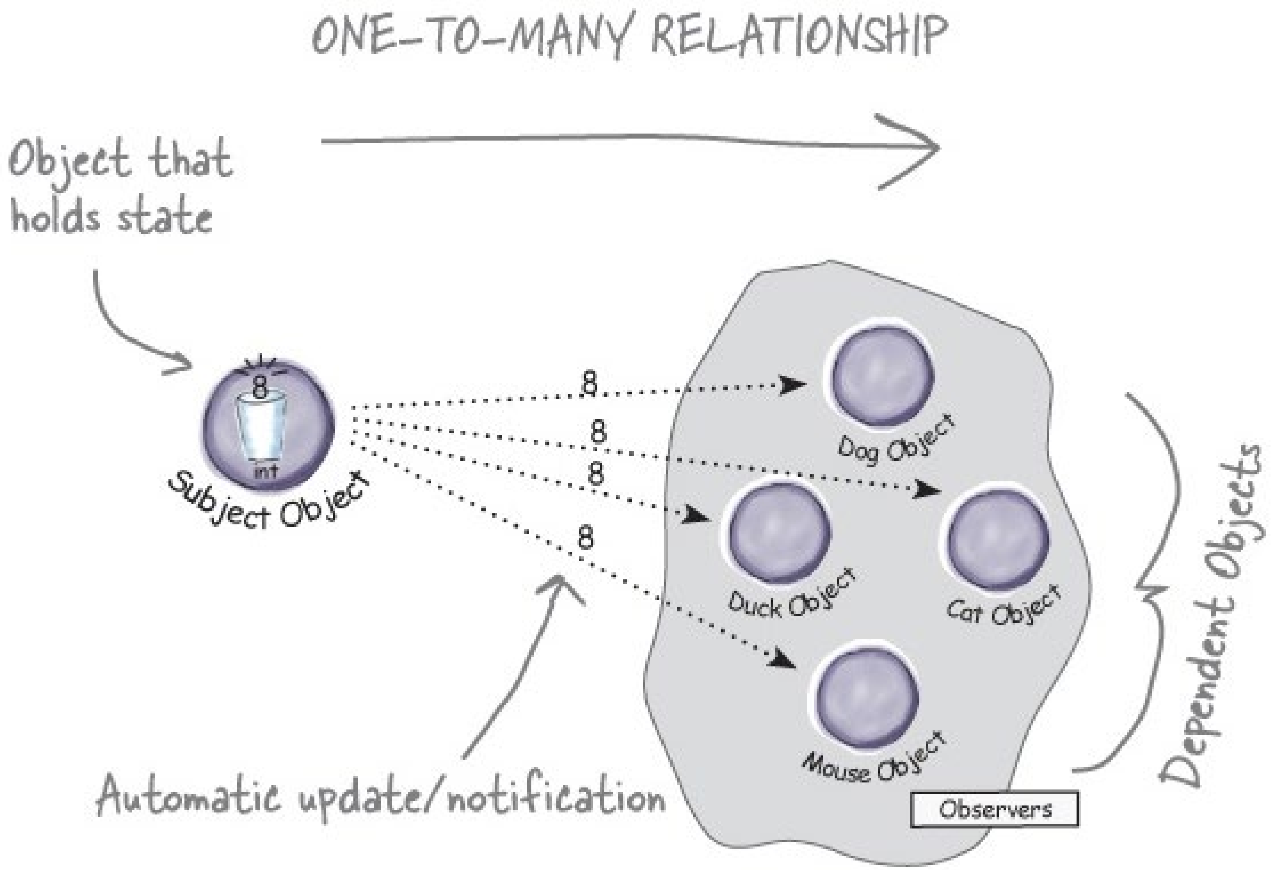
Grab the most recent measurements by calling the WeatherData's getter methods (already implemented).

Now update the displays...

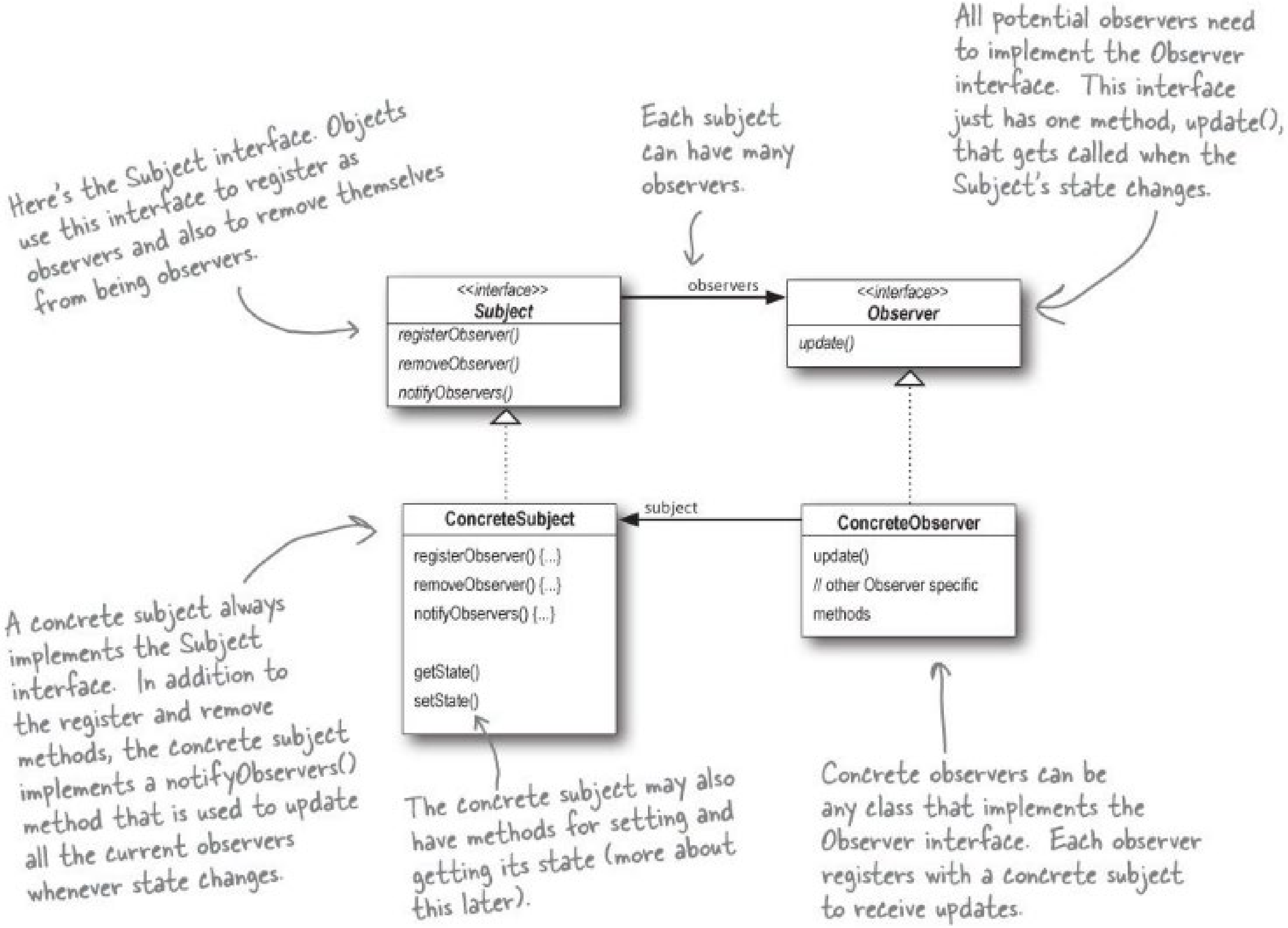
Call each display element to update its display, passing it the most recent measurements.

Publishers + Subscribers = Observer Pattern

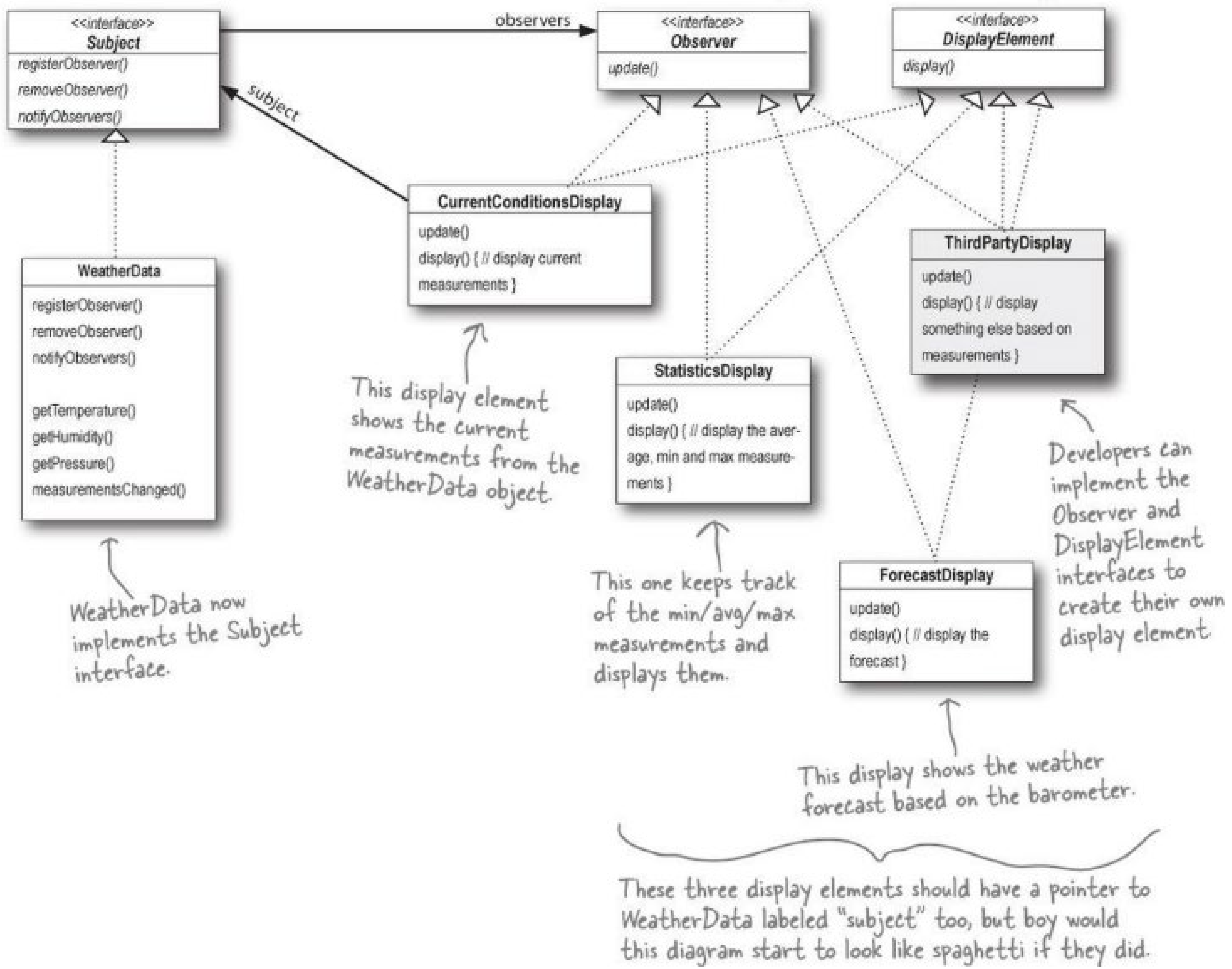
- One-to-many relationship



Observer Pattern for Weather Station



Design the Weather Station



Create Subject interface

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

Implement Subject interface

- Use ArrayList to save all observers
- Notify observers in the function notifyObservers()

```java
public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

Here we implement the Subject interface.

When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the Web.

Build Display Element

This display implements Observer so it can get changes from the WeatherData object

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.

Test our Weather Station

```
File  Edit  Window  Help  StormyWeather
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

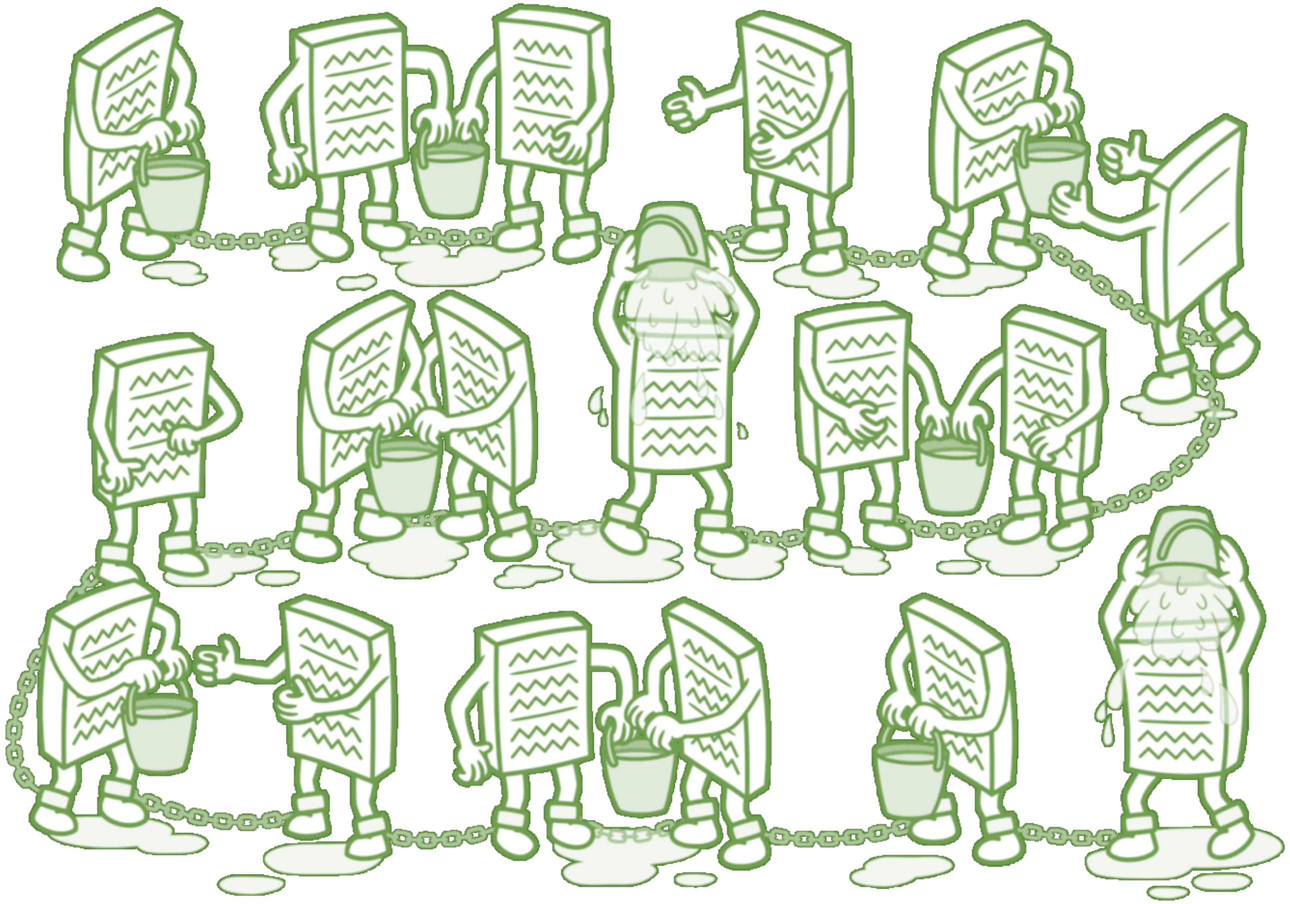
If you don't want to download the code, you can comment out these two lines and run it

Create the three displays and pass them the WeatherData object

Simulate new weather measurements.

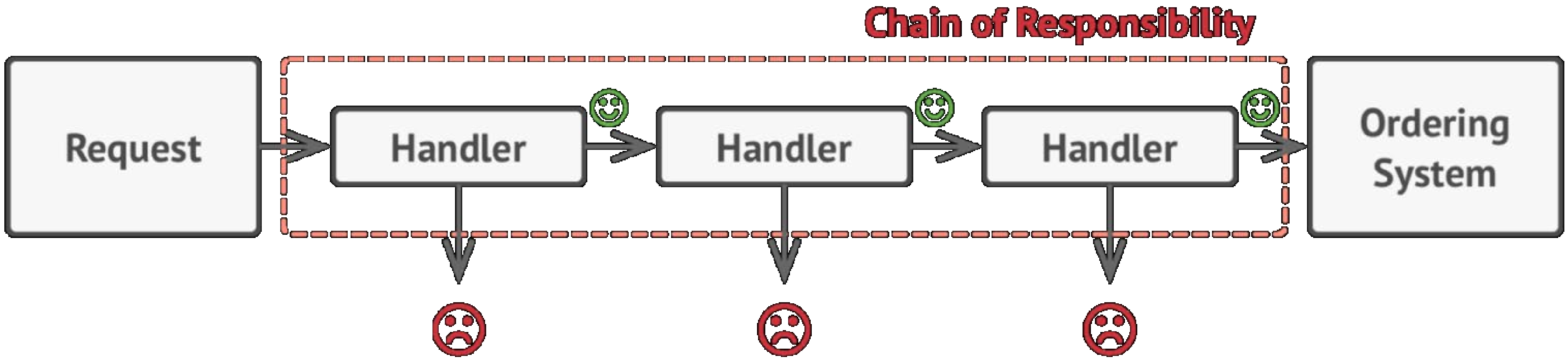
Chain of Responsibility

- Pass requests to the chain of handlers



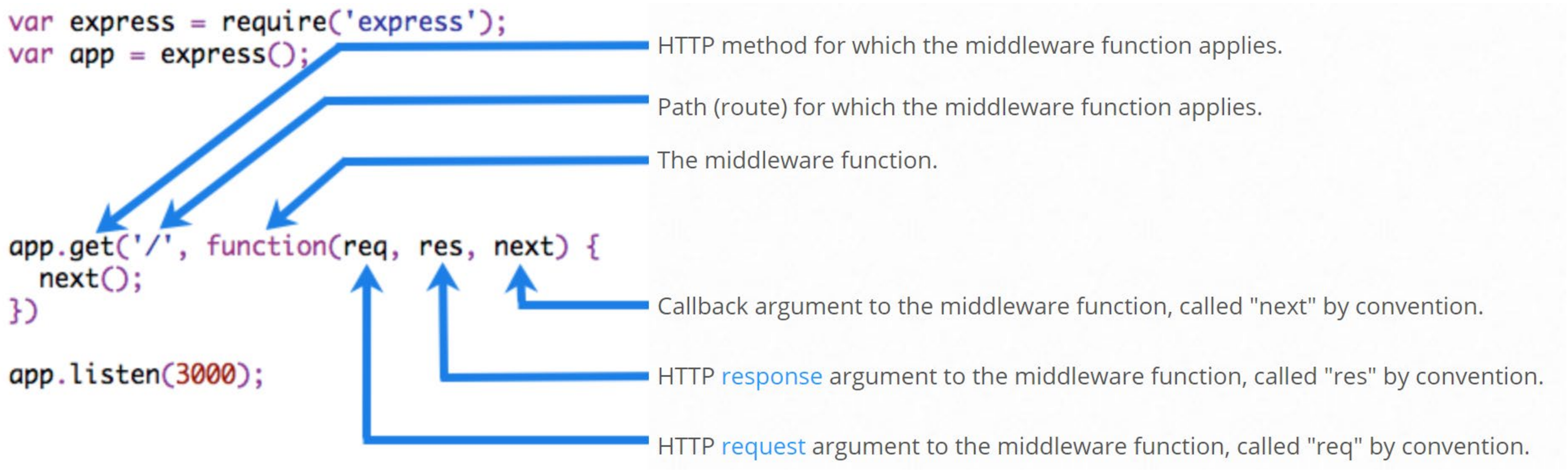
Transform Behavior into "handlers"

- Example: node.js



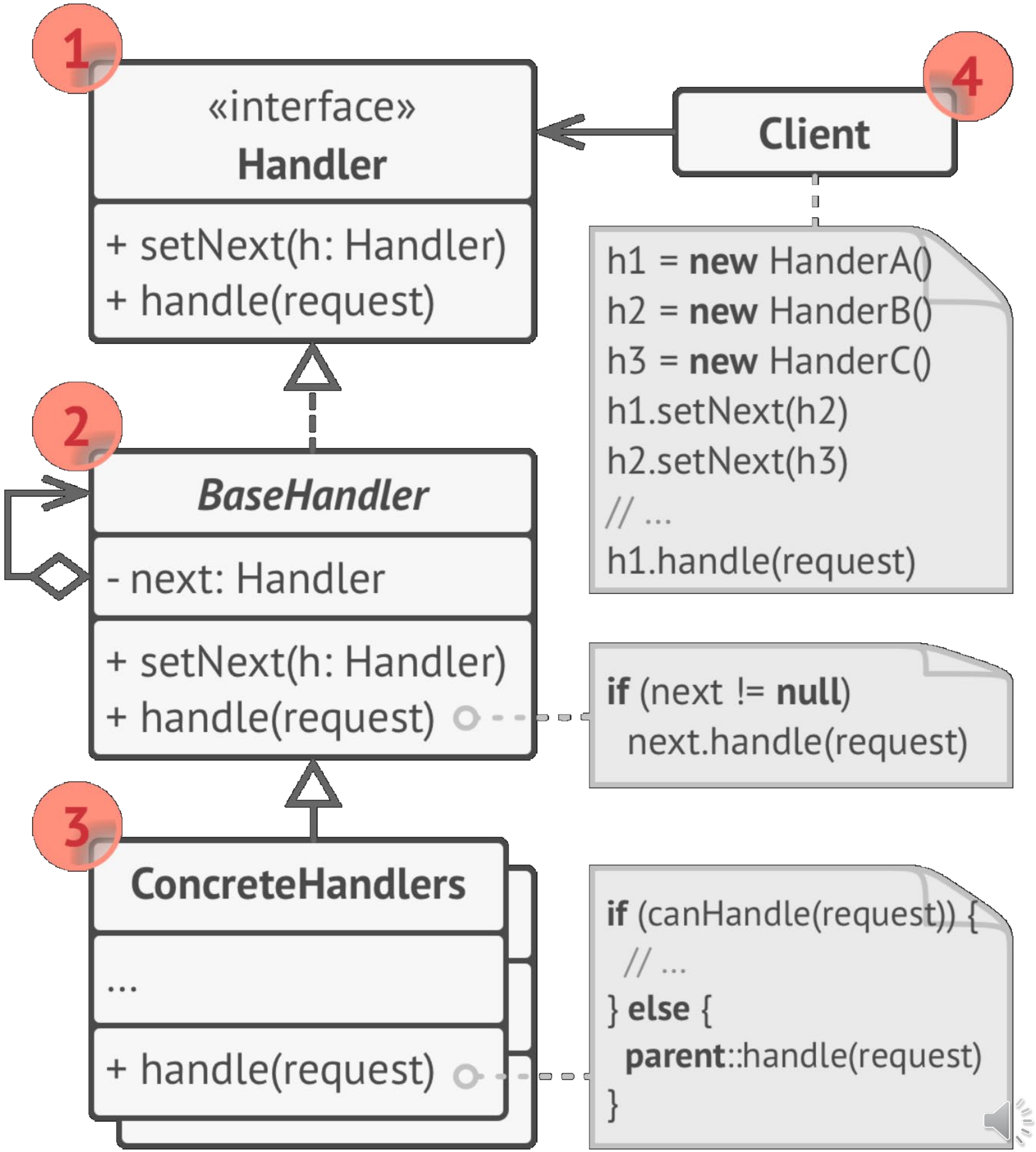
Example: node.js

- Callback function: next()

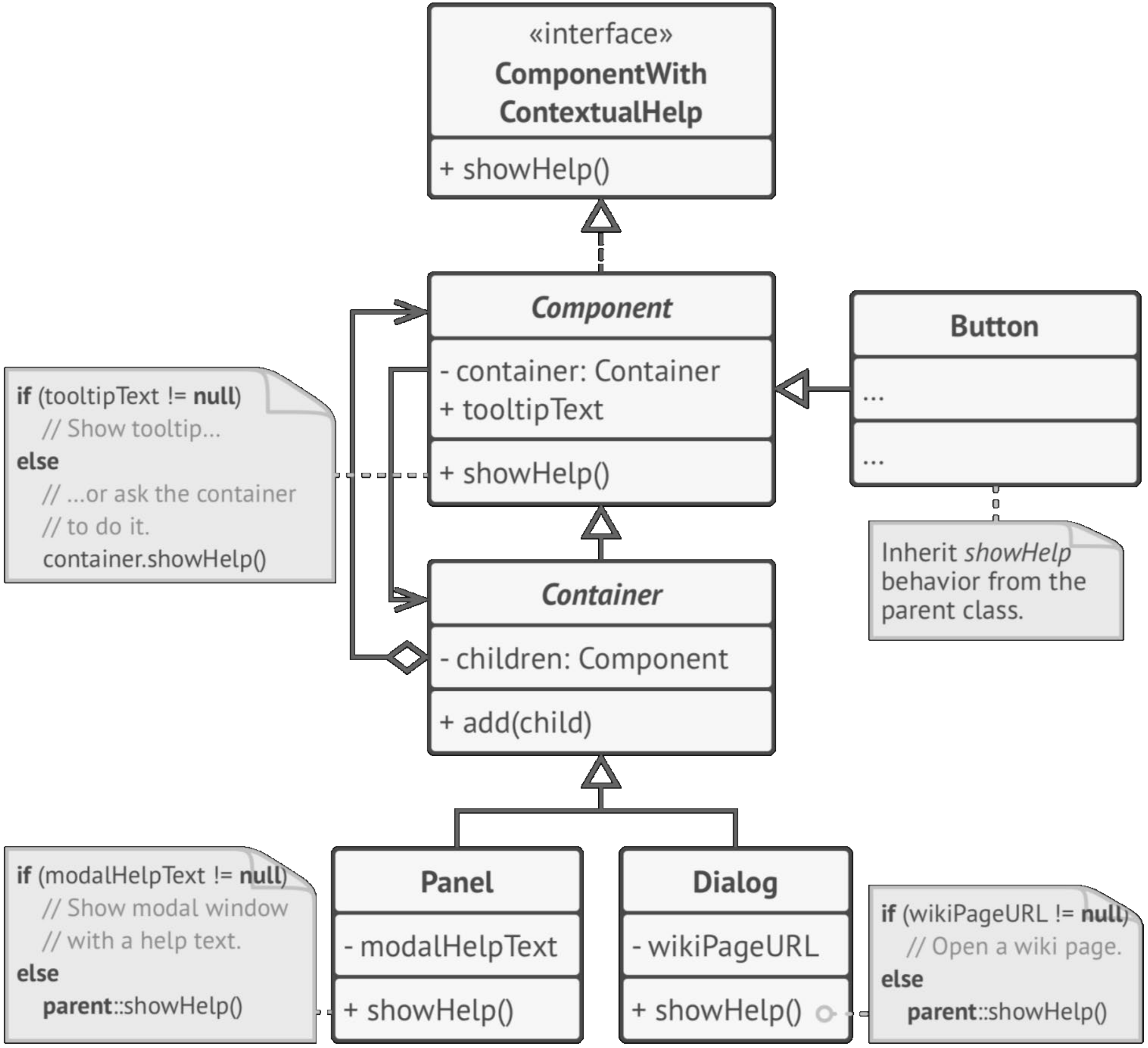


Chain of Responsibility Structure

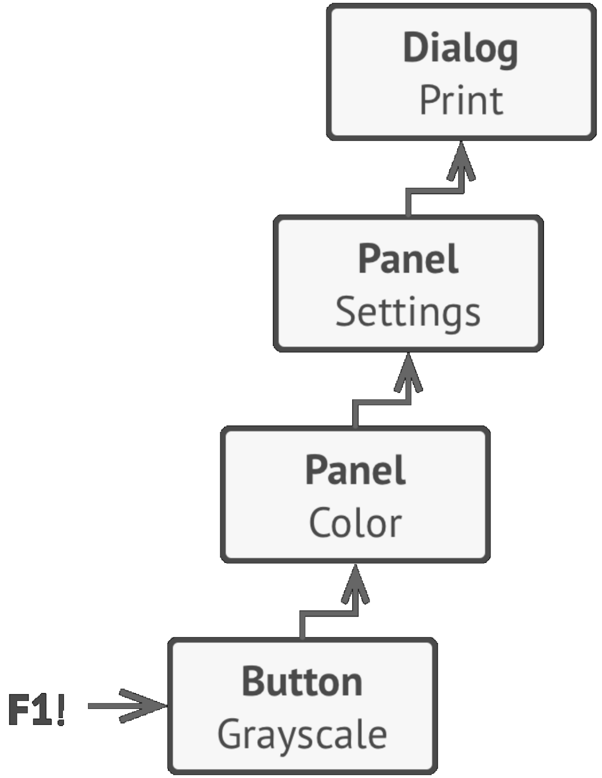
- **Handler** declares the interface, common for all concrete handlers
- **Base Handler** is an optional class where you can put the boilerplate code
- **Concrete Handlers** contain the actual code for processing requests



Working with Composite Pattern

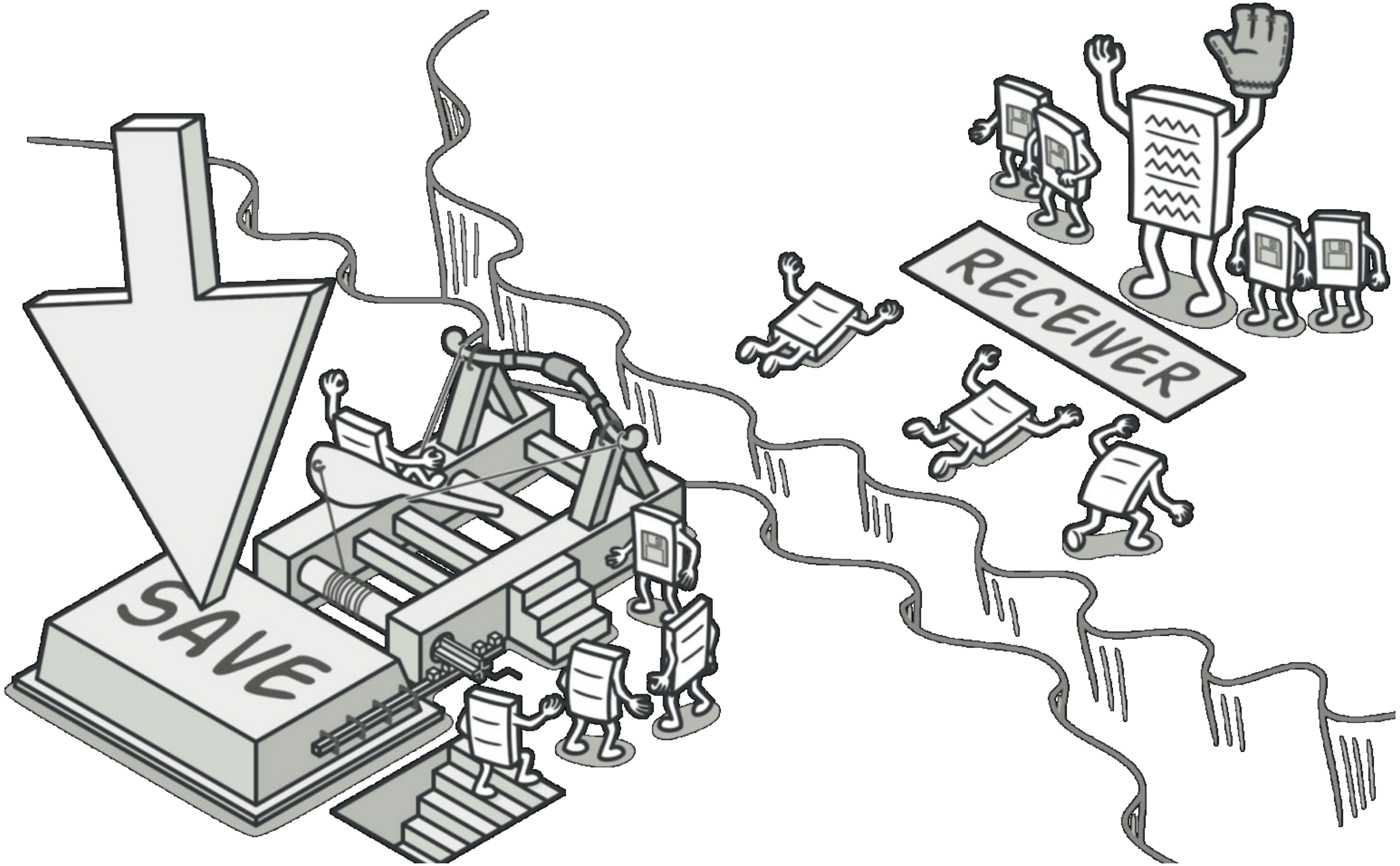


- Find the right class to do showHelp()

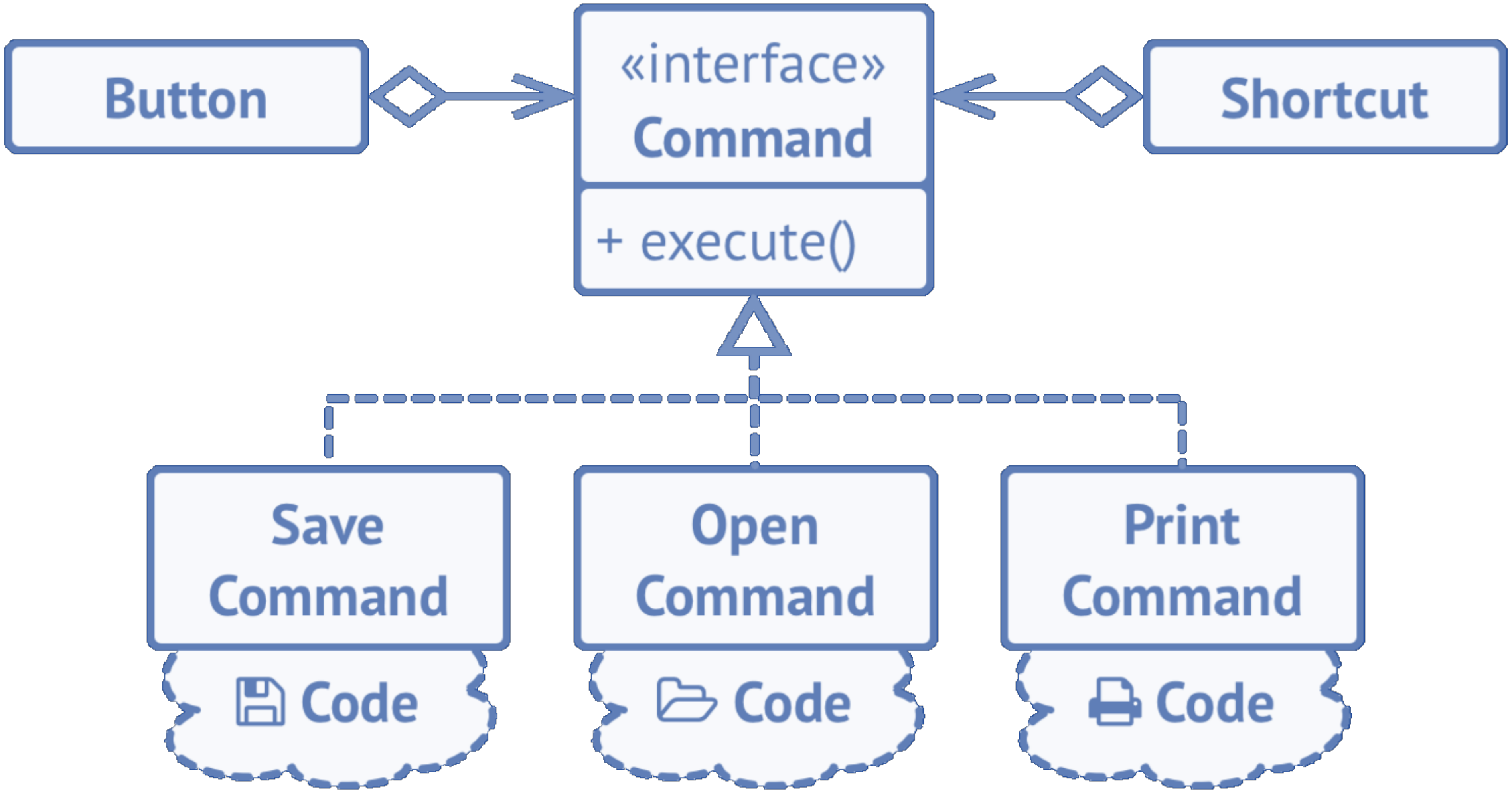


Command Pattern

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

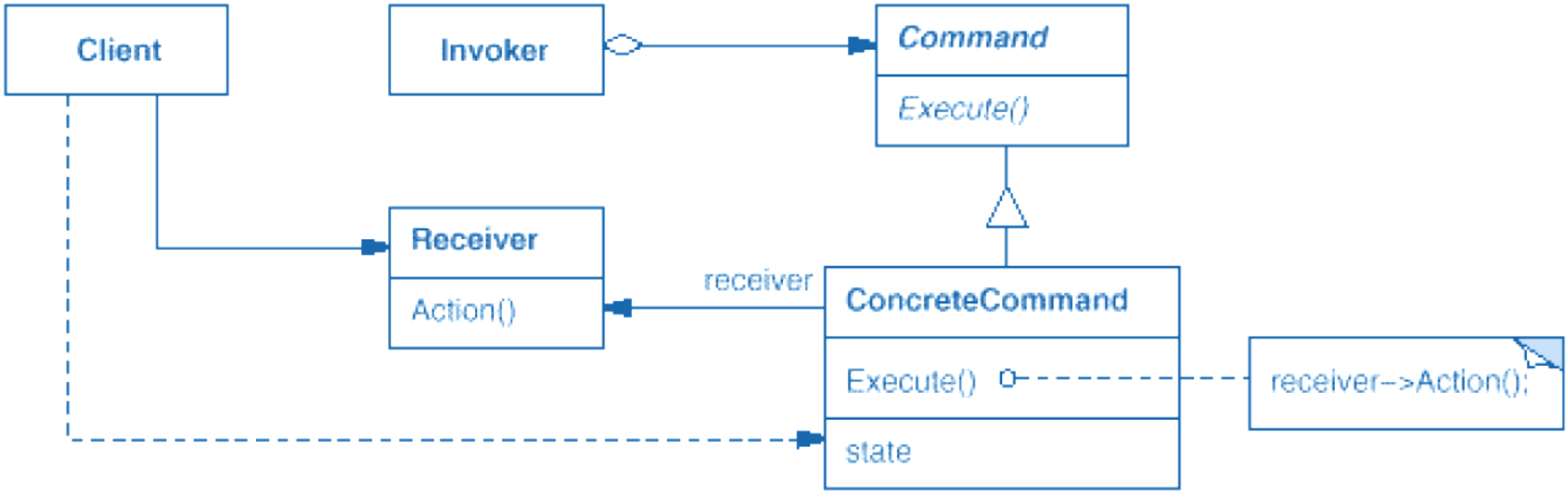


COMMAND for a Editor

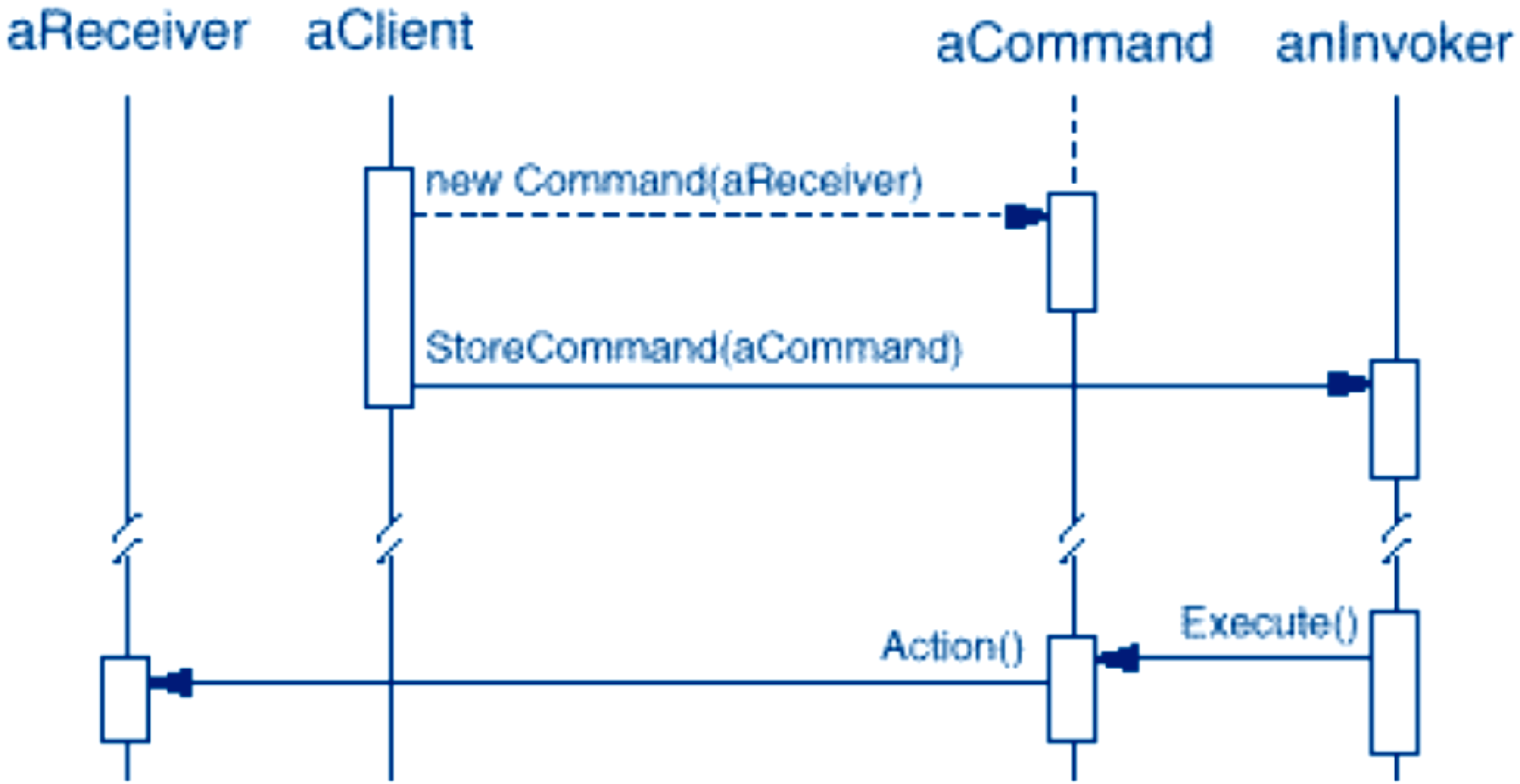


Command Structure

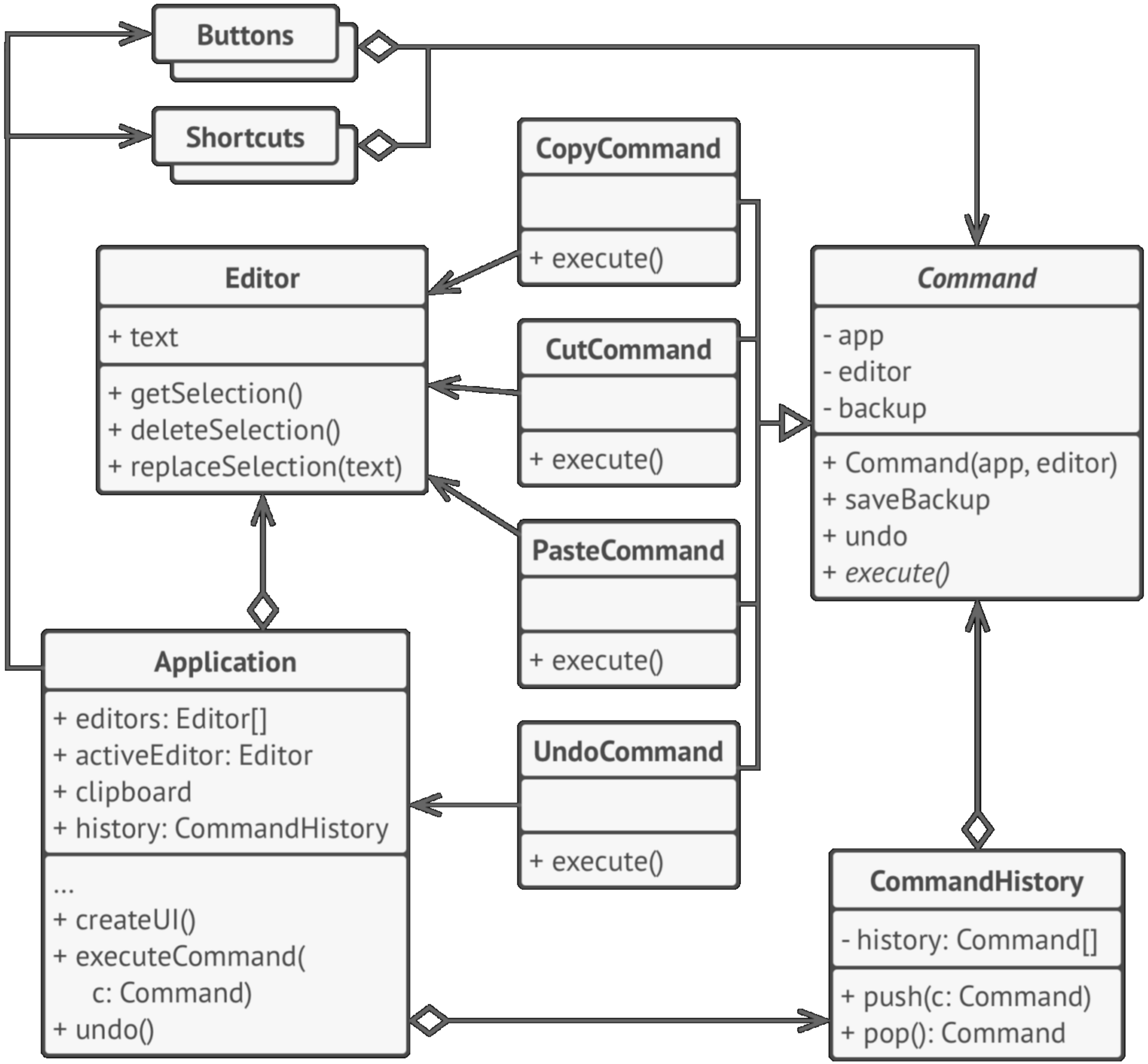
- ConcreteCommand
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.



Collaboration



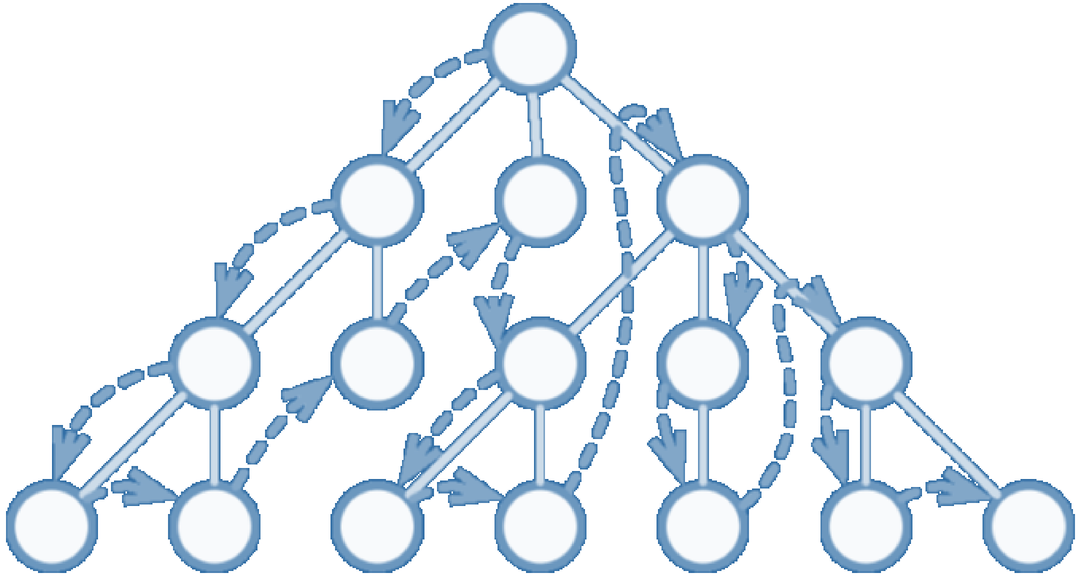
Undo

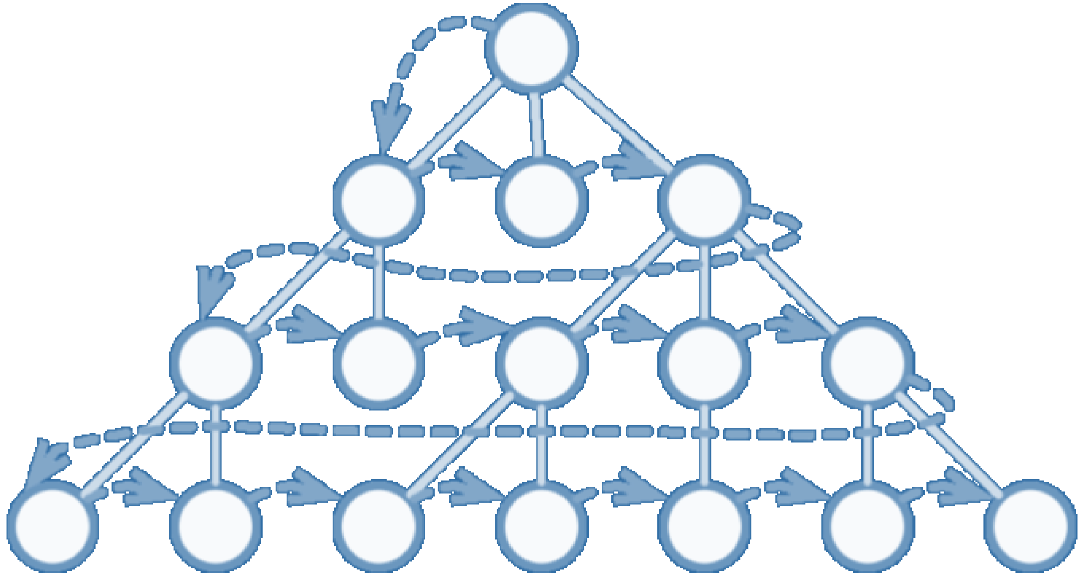


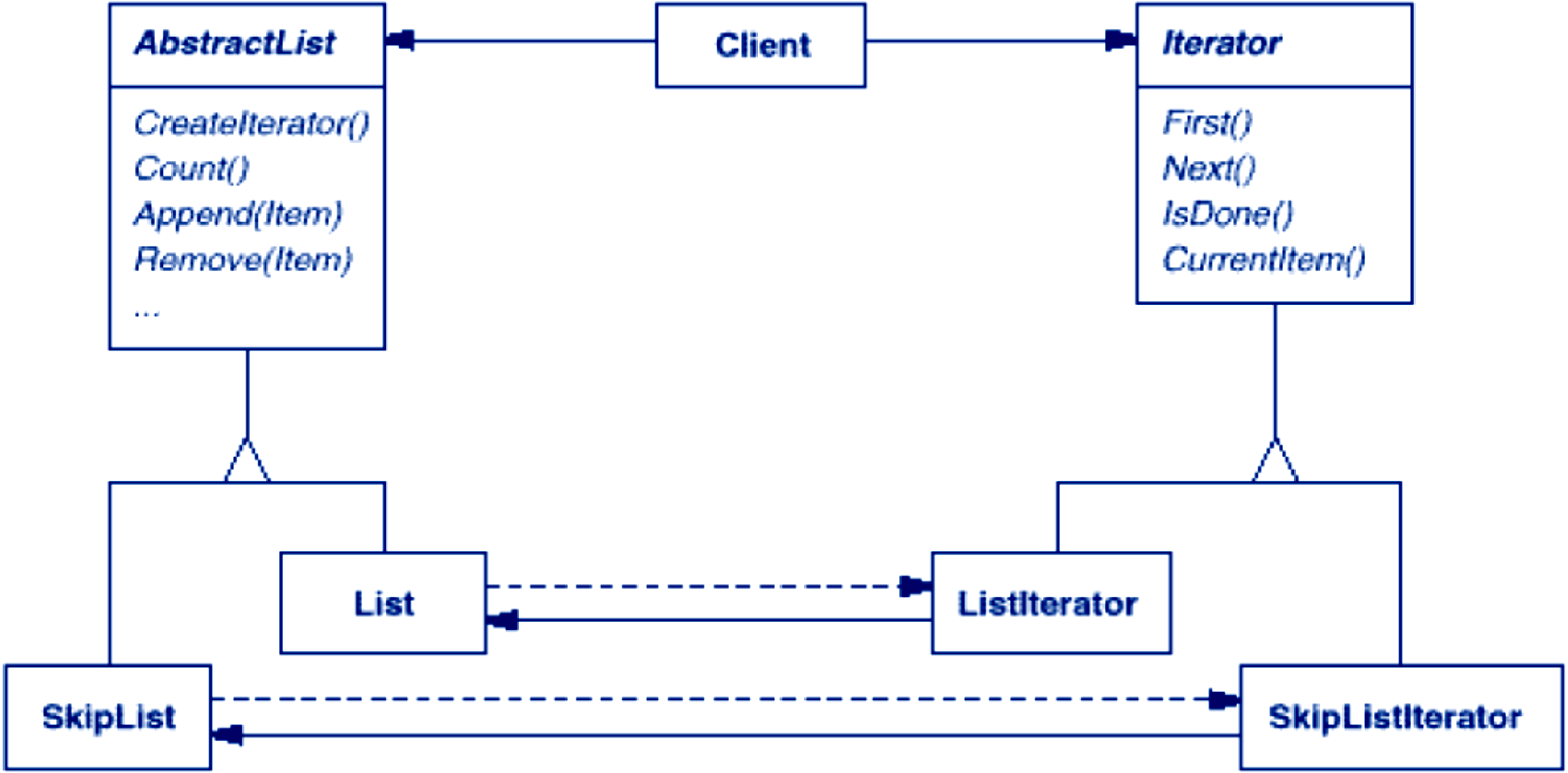
Buttons
Shortcuts
CopyCommand
+ execute()
Editor
+ text
+ getSelection()
+ deleteSelection()
+ replaceSelection(text)
CutCommand
+ execute()
Command
- app
- editor
- backup
+ Command(app, editor)
+ saveBackup
+ undo
+ execute()
PasteCommand
+ execute()
Application
+ editors: Editor[]
+ activeEditor: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

Iterator

- A pattern that traverses elements of a collection





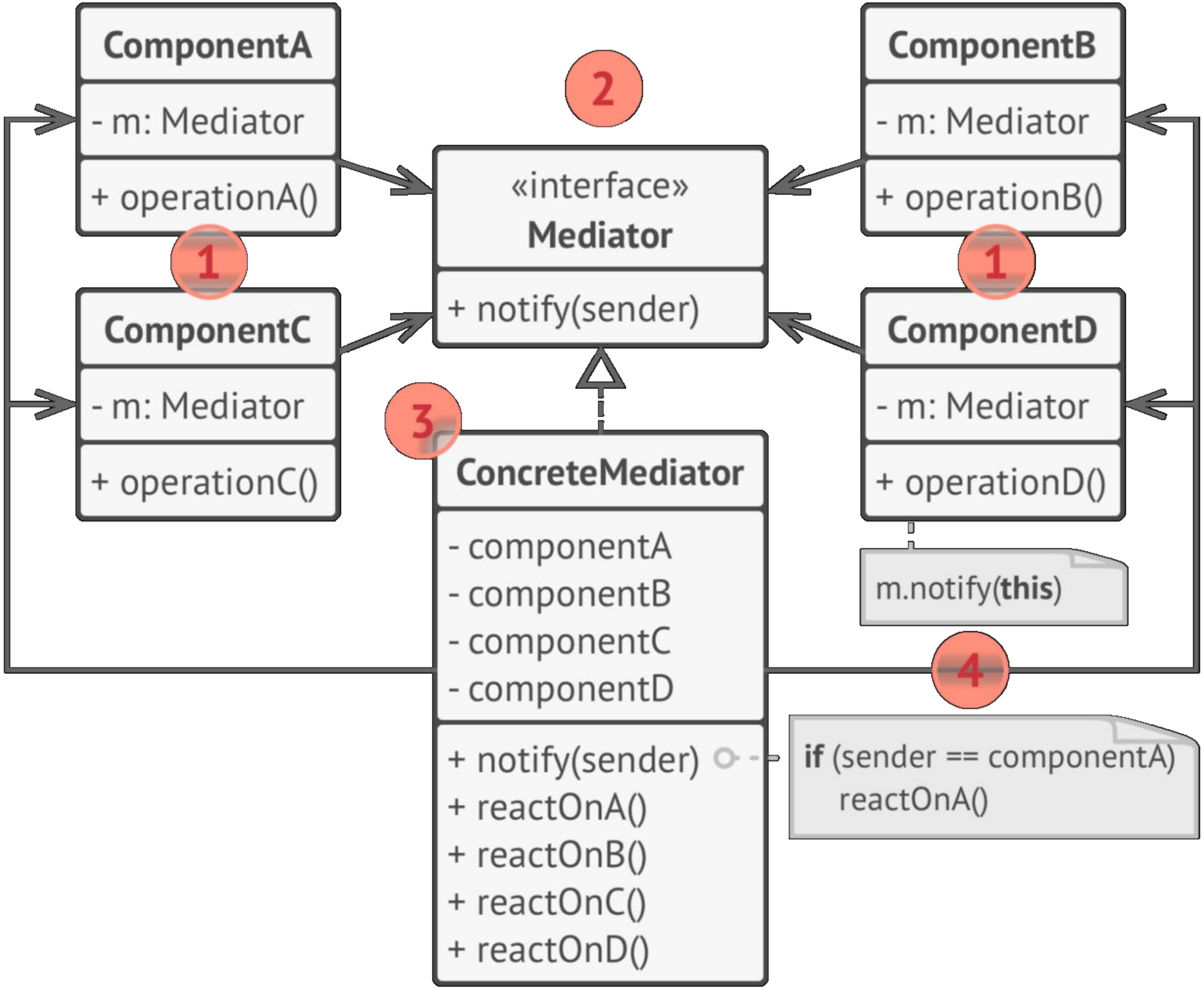

AbstractList
CreateIterator()
Count()
Append(Item)
Remove(Item)
...
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator

```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;
}
```

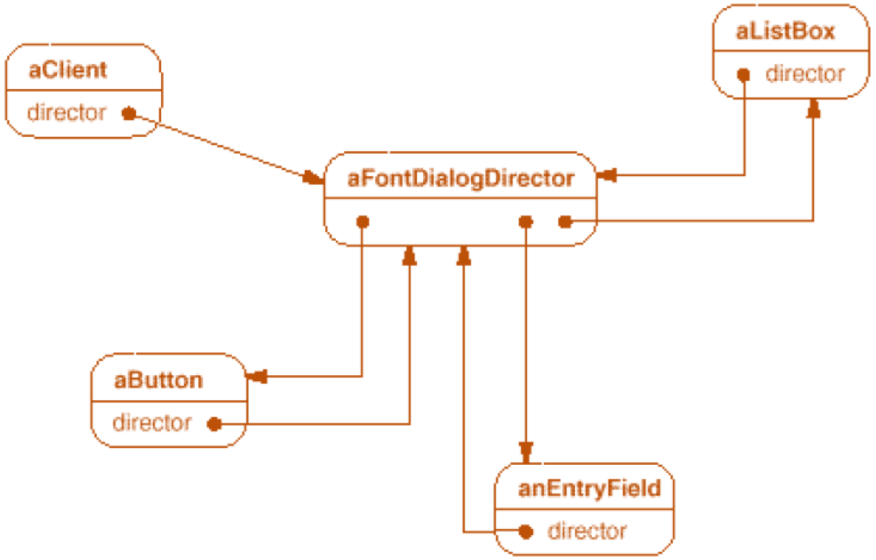
Or
it != from.end()

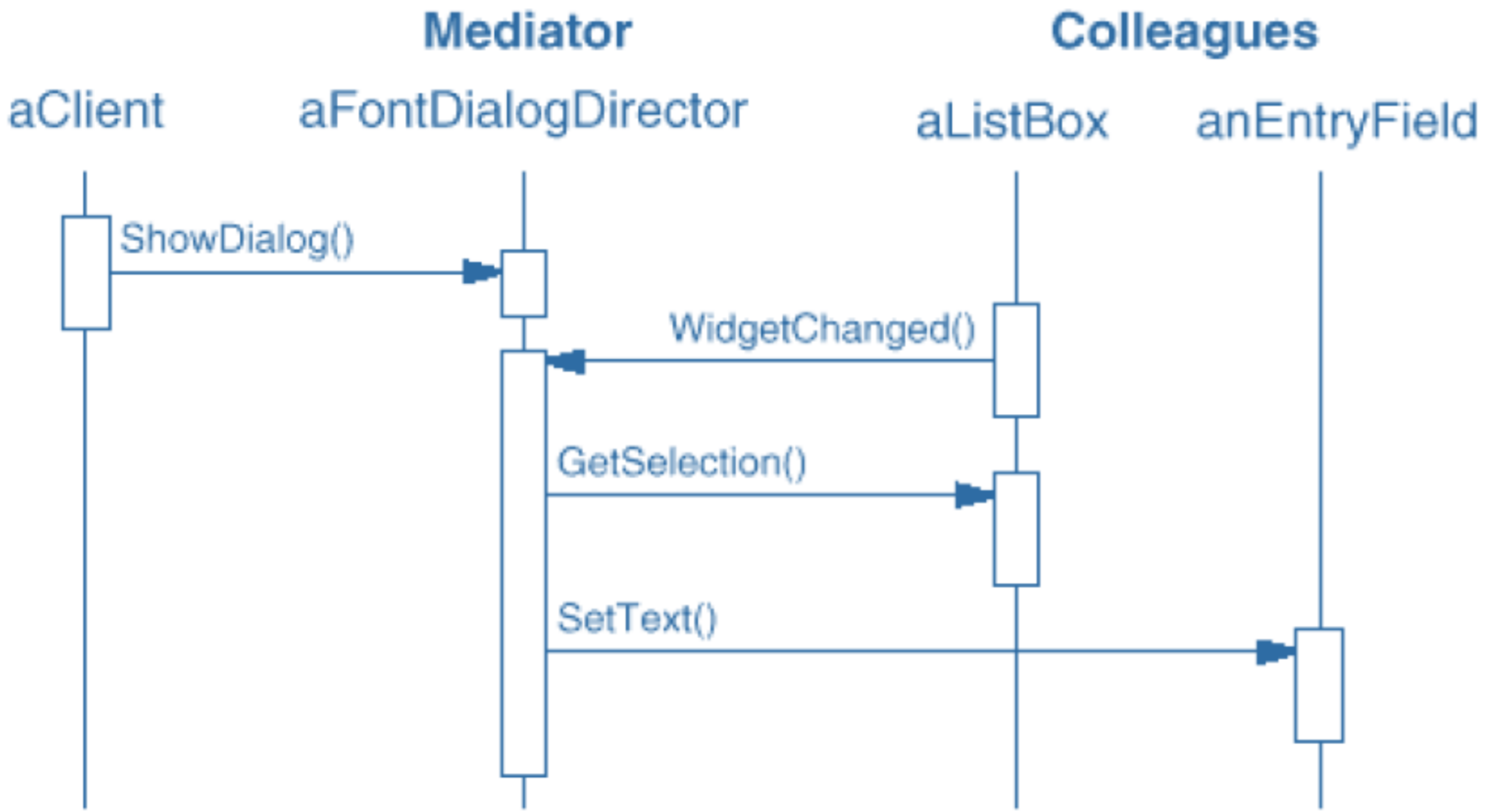
Mediator (a.k.a. Intermediary, Controller)

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

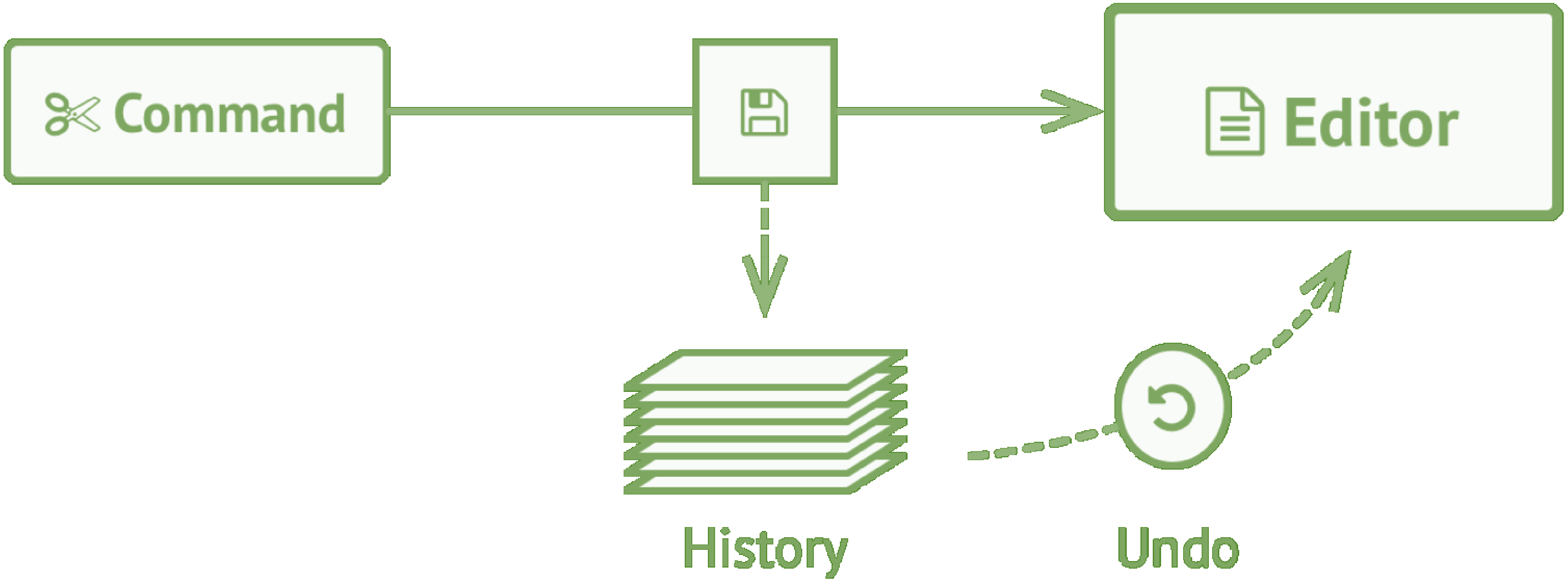
Example: Font Dialog





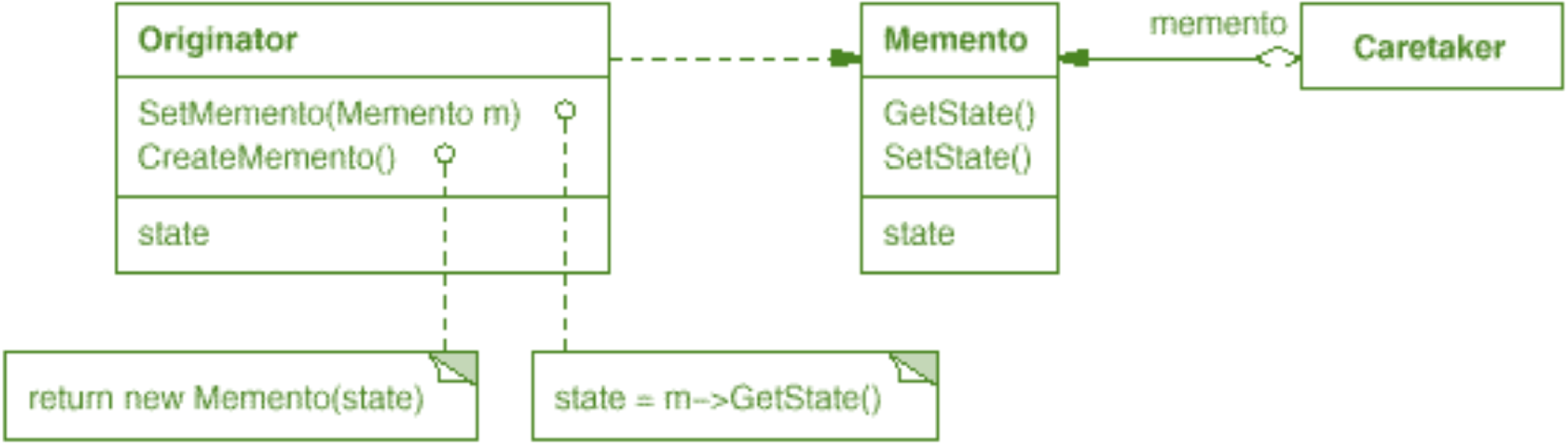
Memento

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

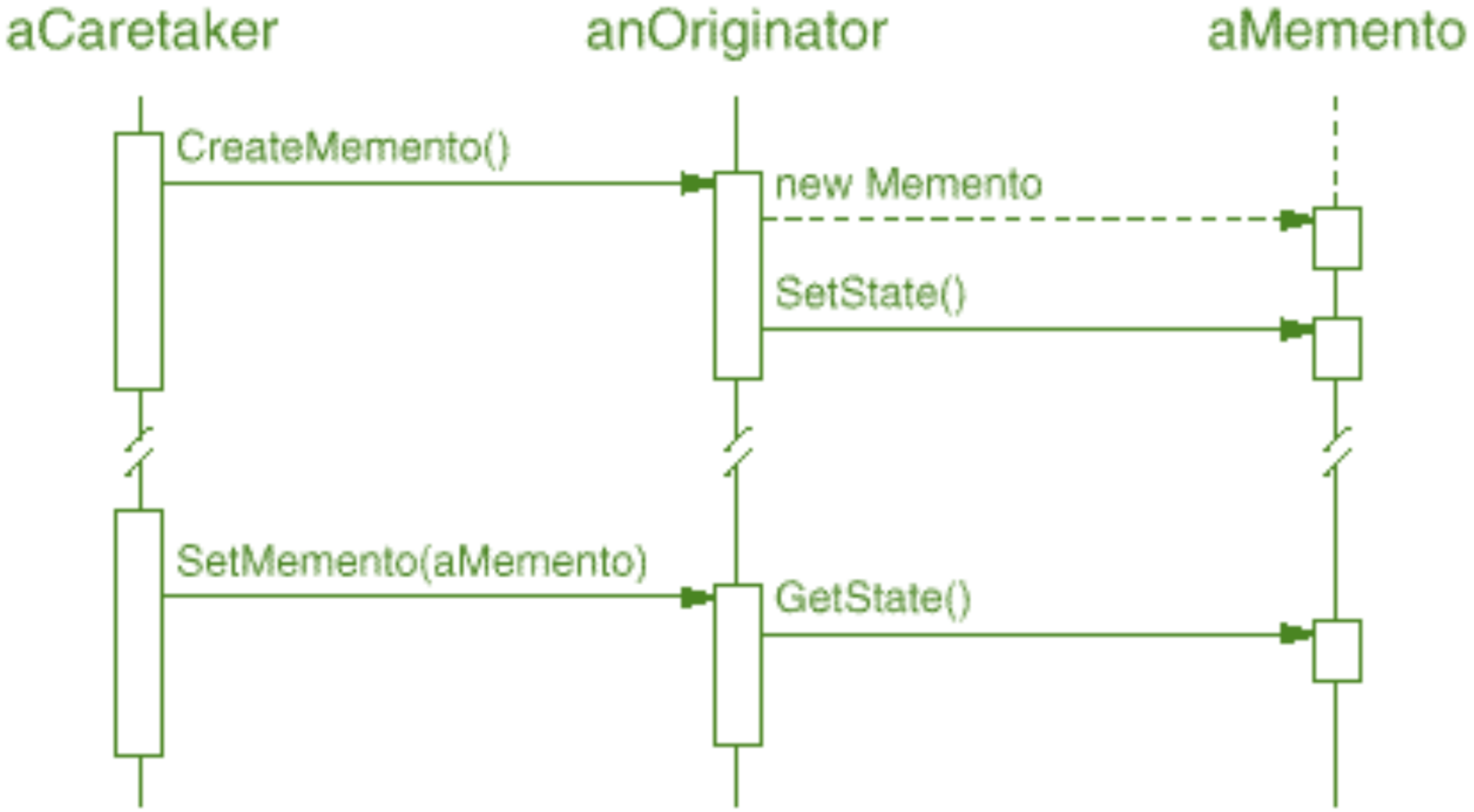


Memento Structure

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

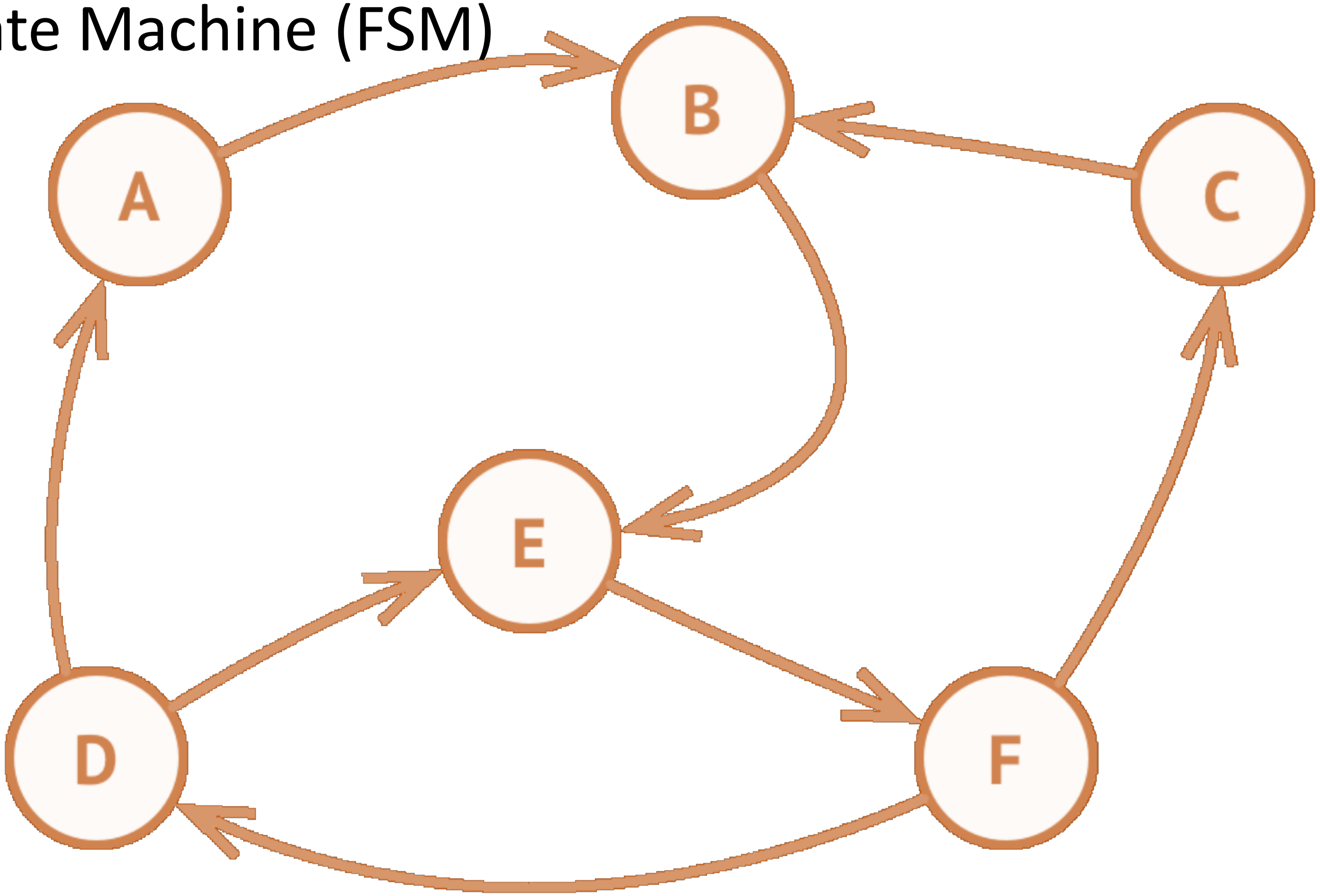


Memento Collaborations

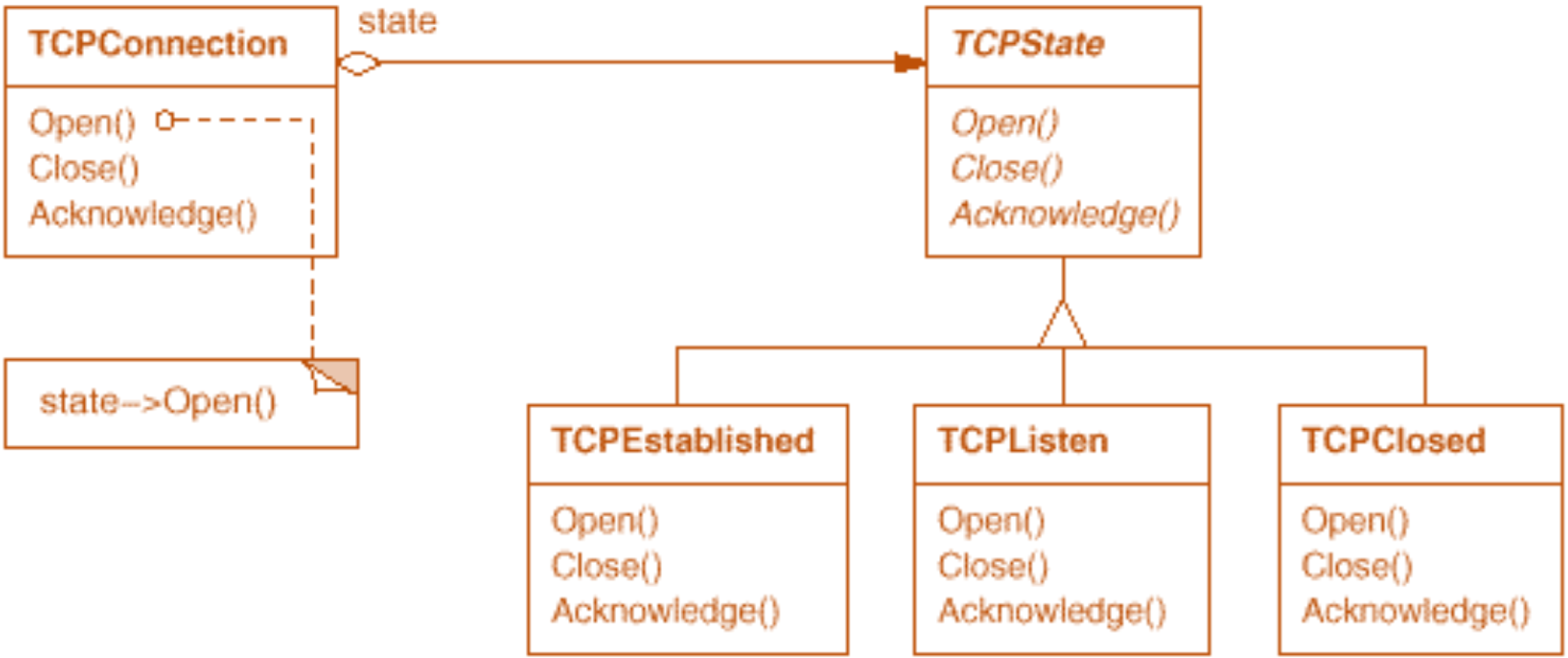


8. STATE

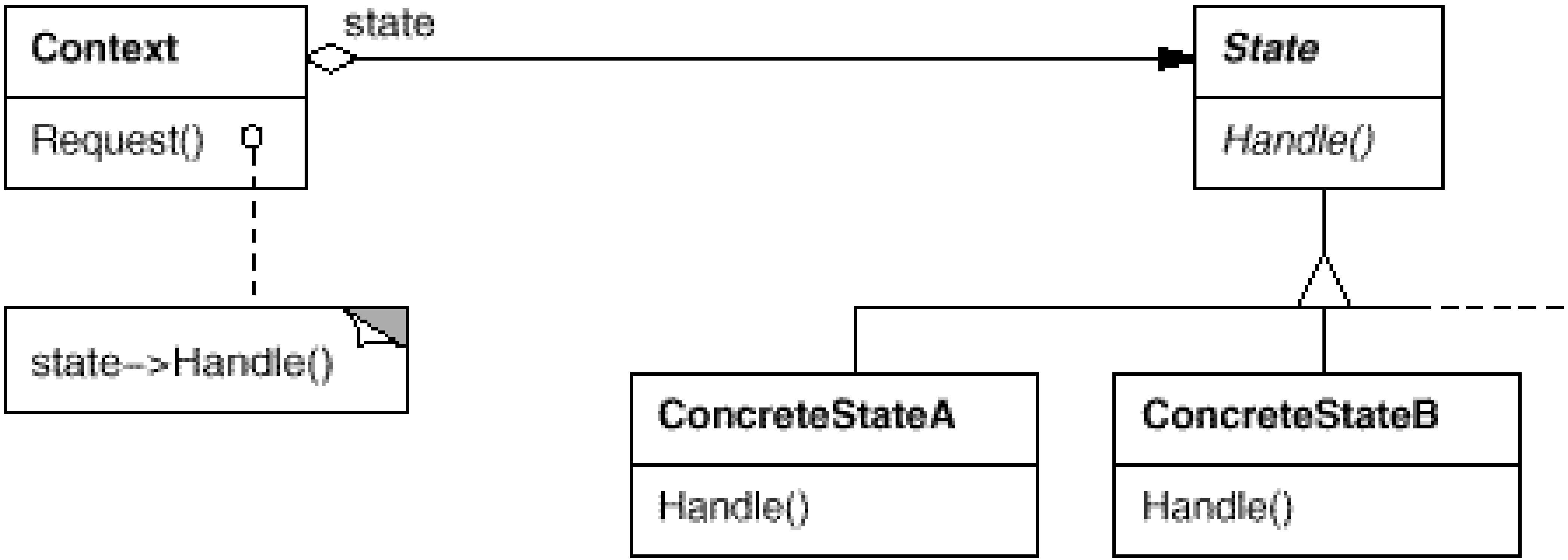
- Let an object alter its behavior when its internal state changes
- Ex: Finite State Machine (FSM)



Example: TCP Connection

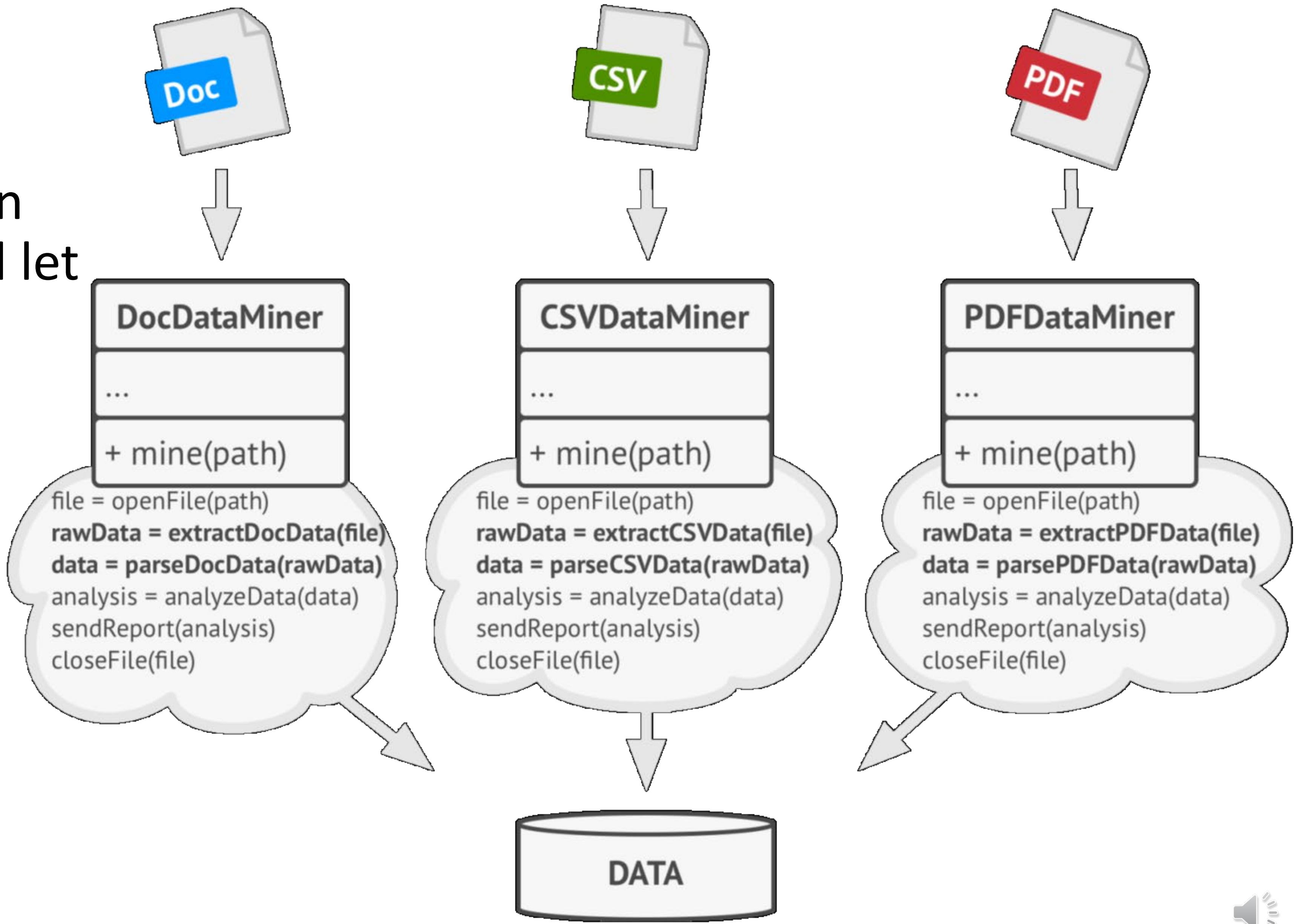


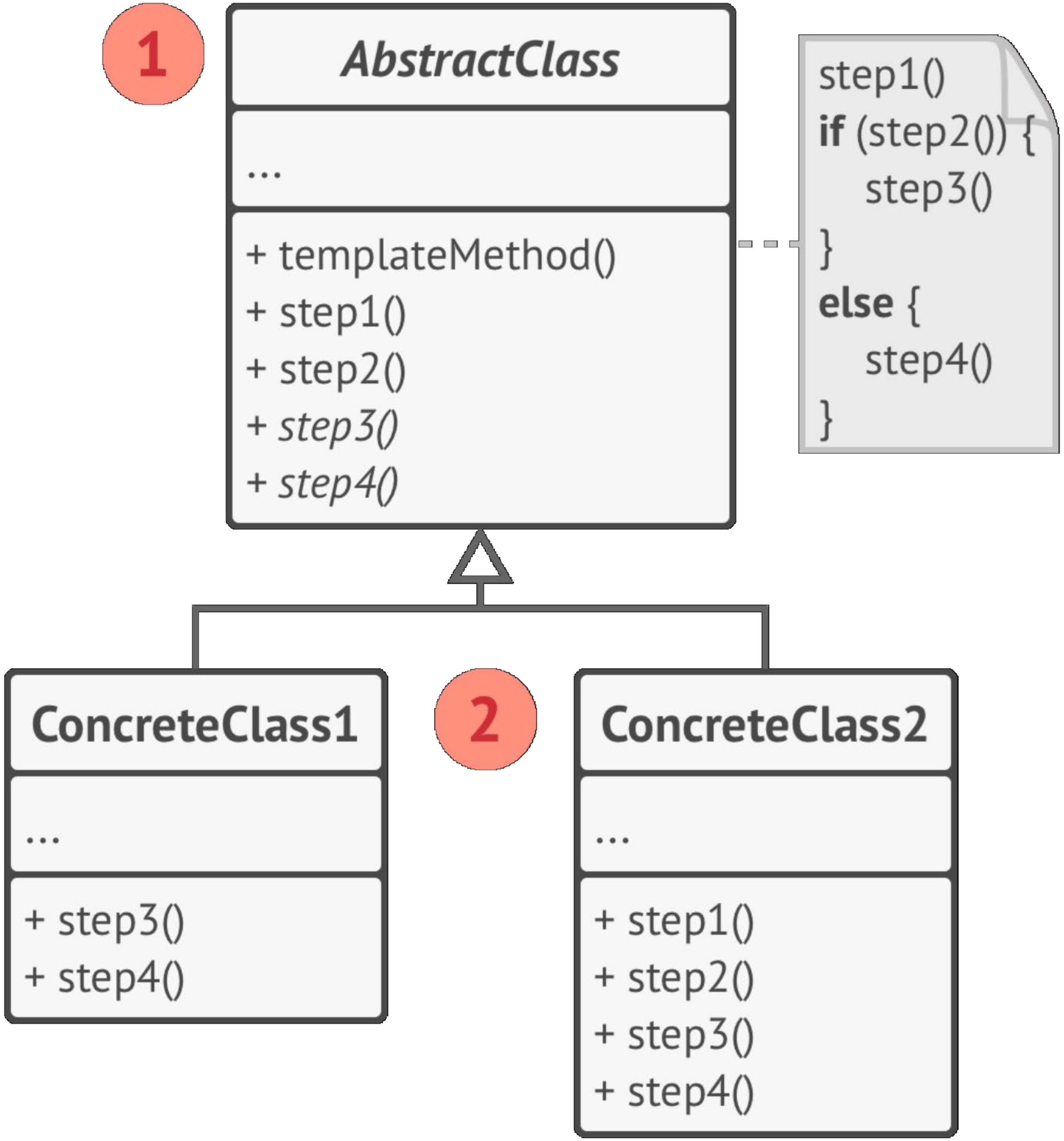
State Structure



10. Template

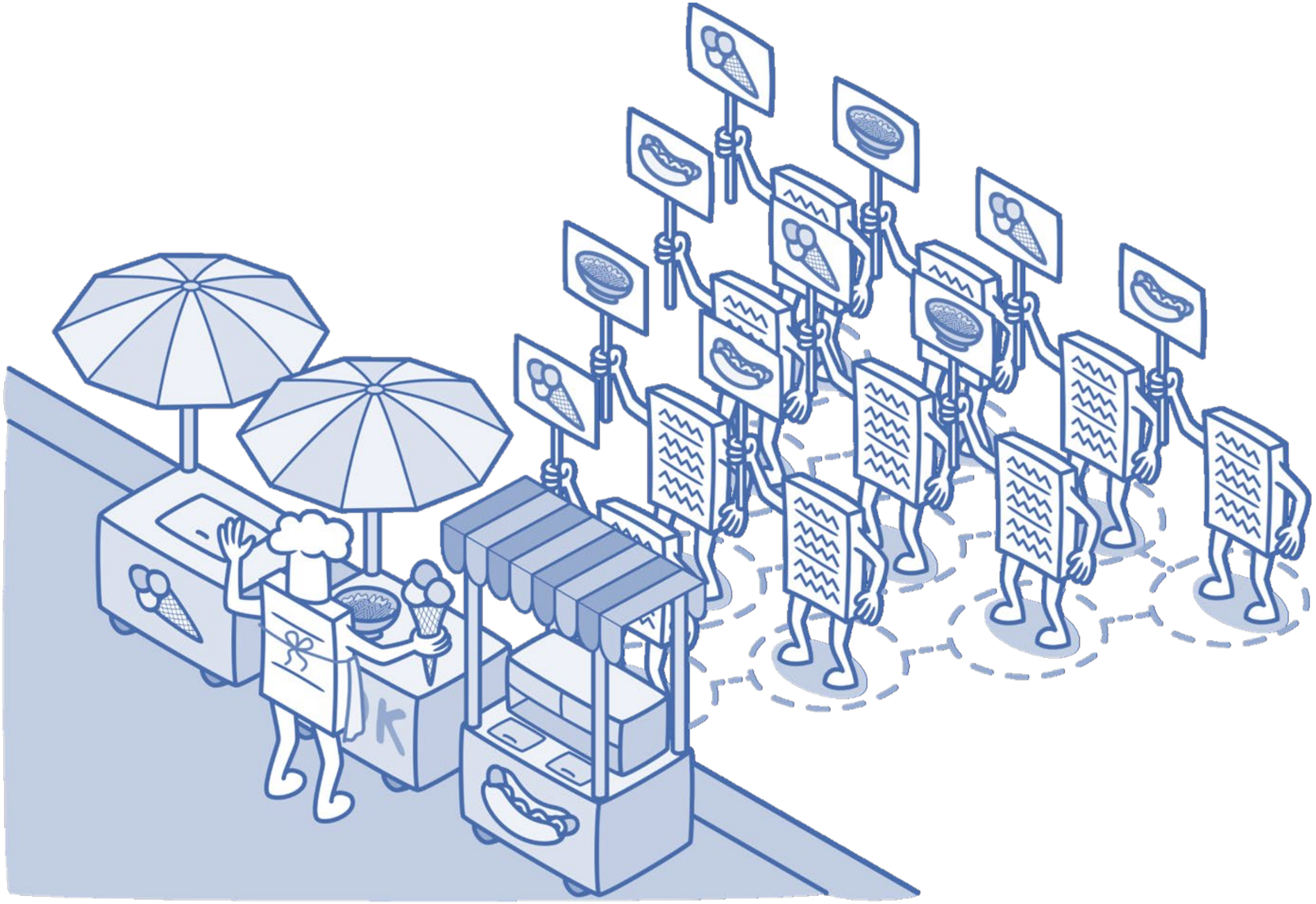
- Defines the skeleton of an algorithm and let subclasses override specific steps




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

11. Visitor

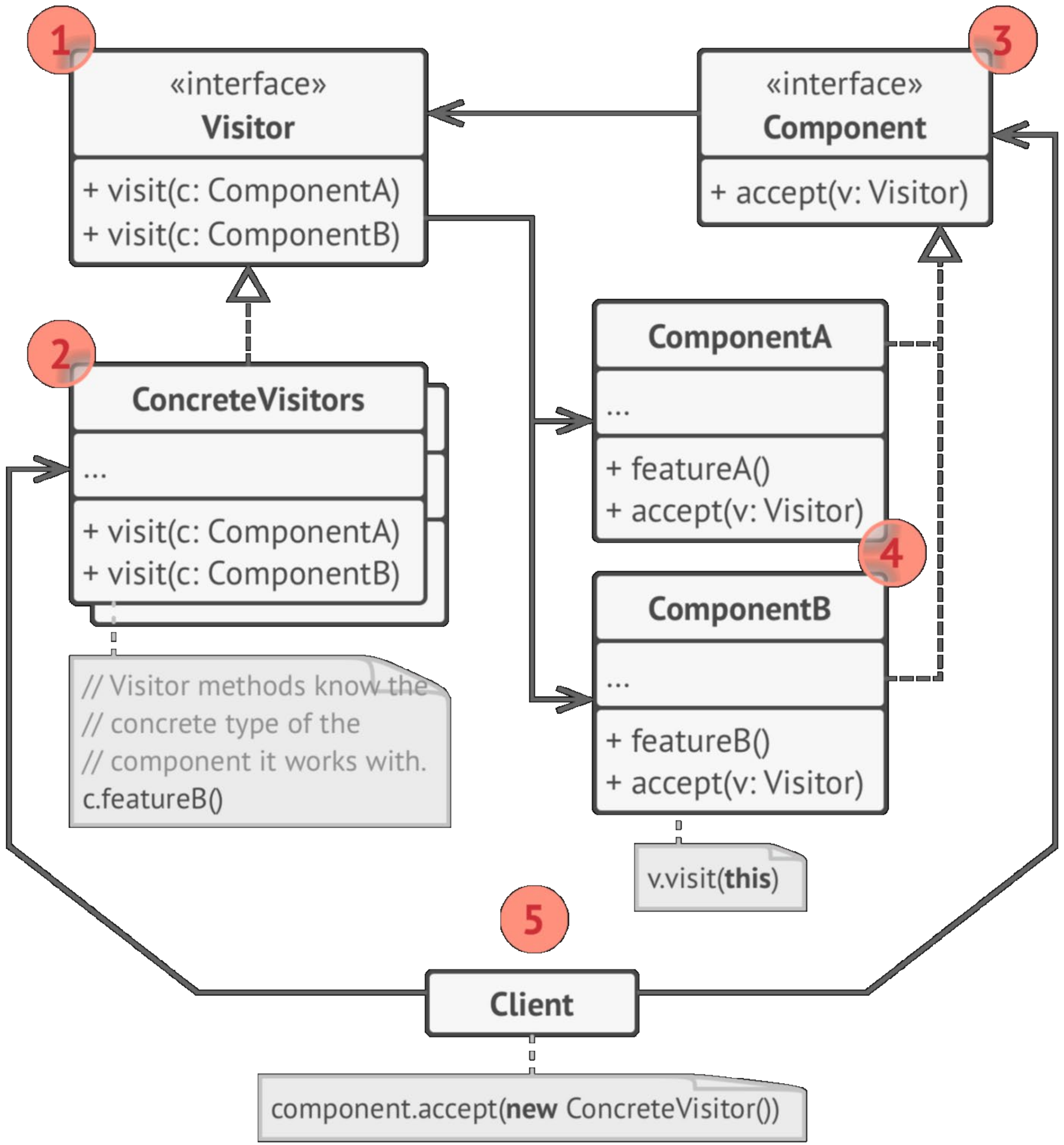
- Separate algorithms from the objects on which they operate



Visitor vs. Iterator

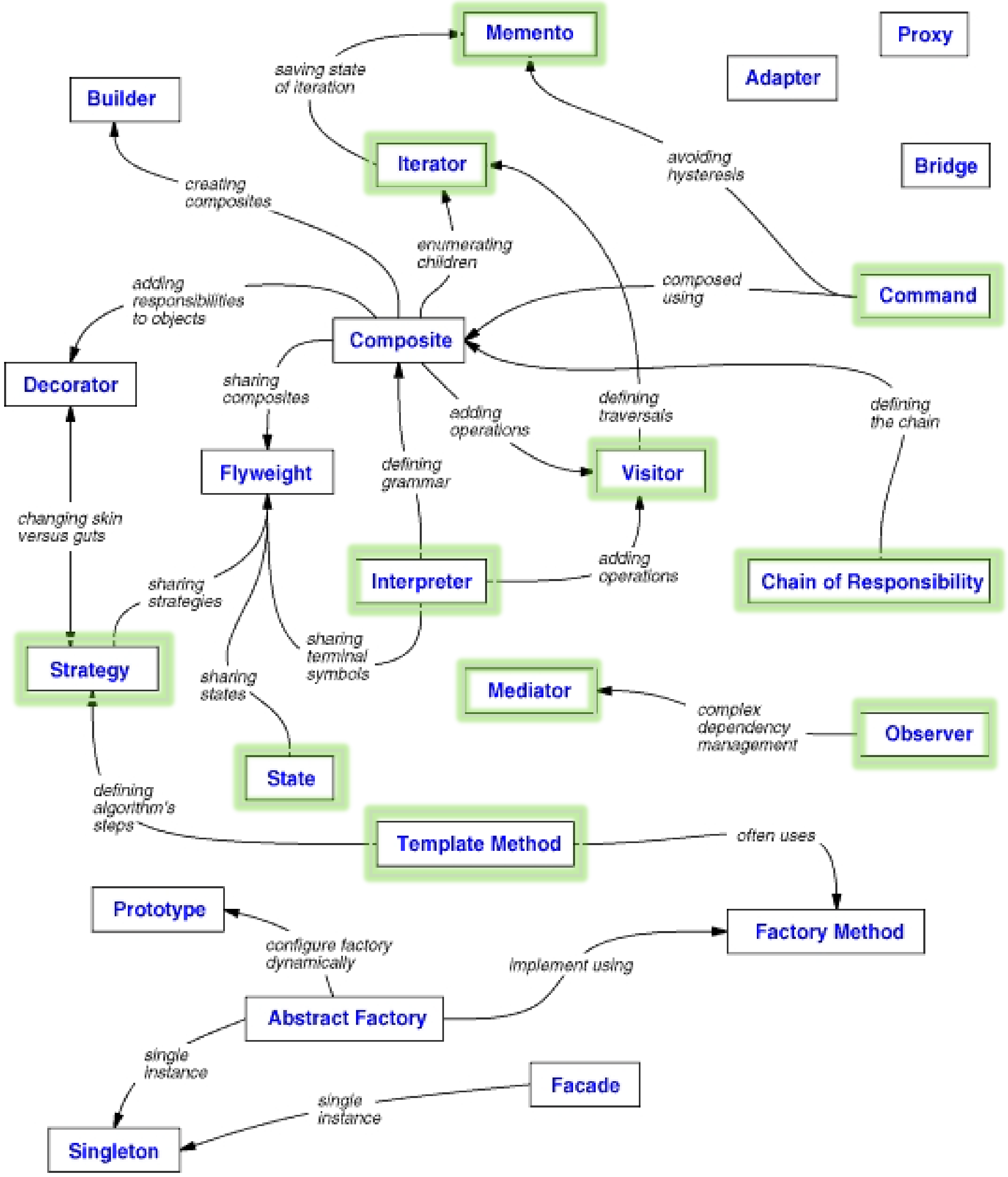
- Visitor Pattern is used to perform an action on a structure of elements

```
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())

Origin Behavioral Design Patterns

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



Memento
Proxy
Adapter
Builder
saving state of iteration
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
adding operations
Interpreter
Chain of Responsibility
sharing strategies
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
configure factory dynamically
Factory Method
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

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