Software Design 3

Software Engineering
Prof. Maged Elaasar

Learning Objectives

- GoF Behavioral Patterns
    - Strategy pattern
    - Observer pattern
    - Mediator pattern
    - Command pattern
    - State pattern

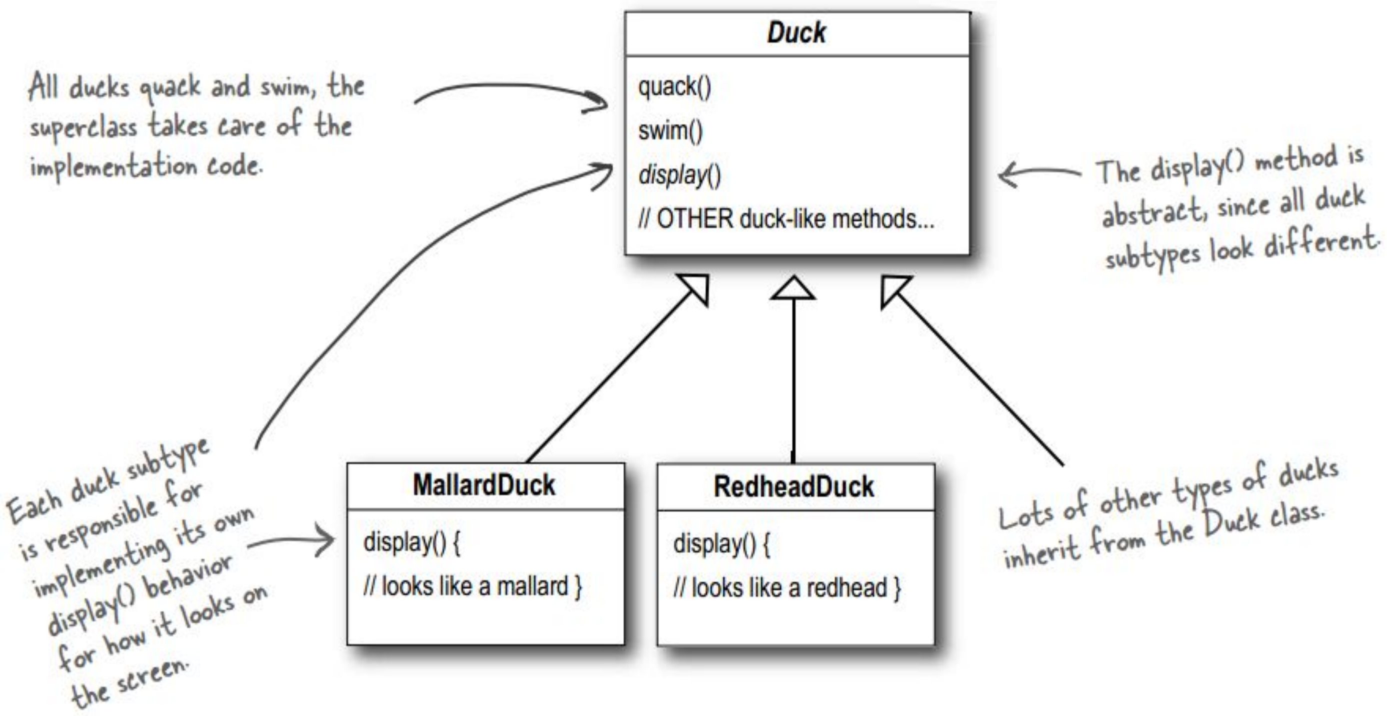
Strategy Pattern

Strategy Pattern

Problem

When a class can do a task in a lot of different ways at possibly different times to the point that it is becomes bloated and fragile.

Motivation: Duck



Duck

quack()

swim()

display()

// OTHER duck-like methods...

MallardDuck

display() {

// looks like a mallard }

RedheadDuck

display() {

// looks like a redhead }

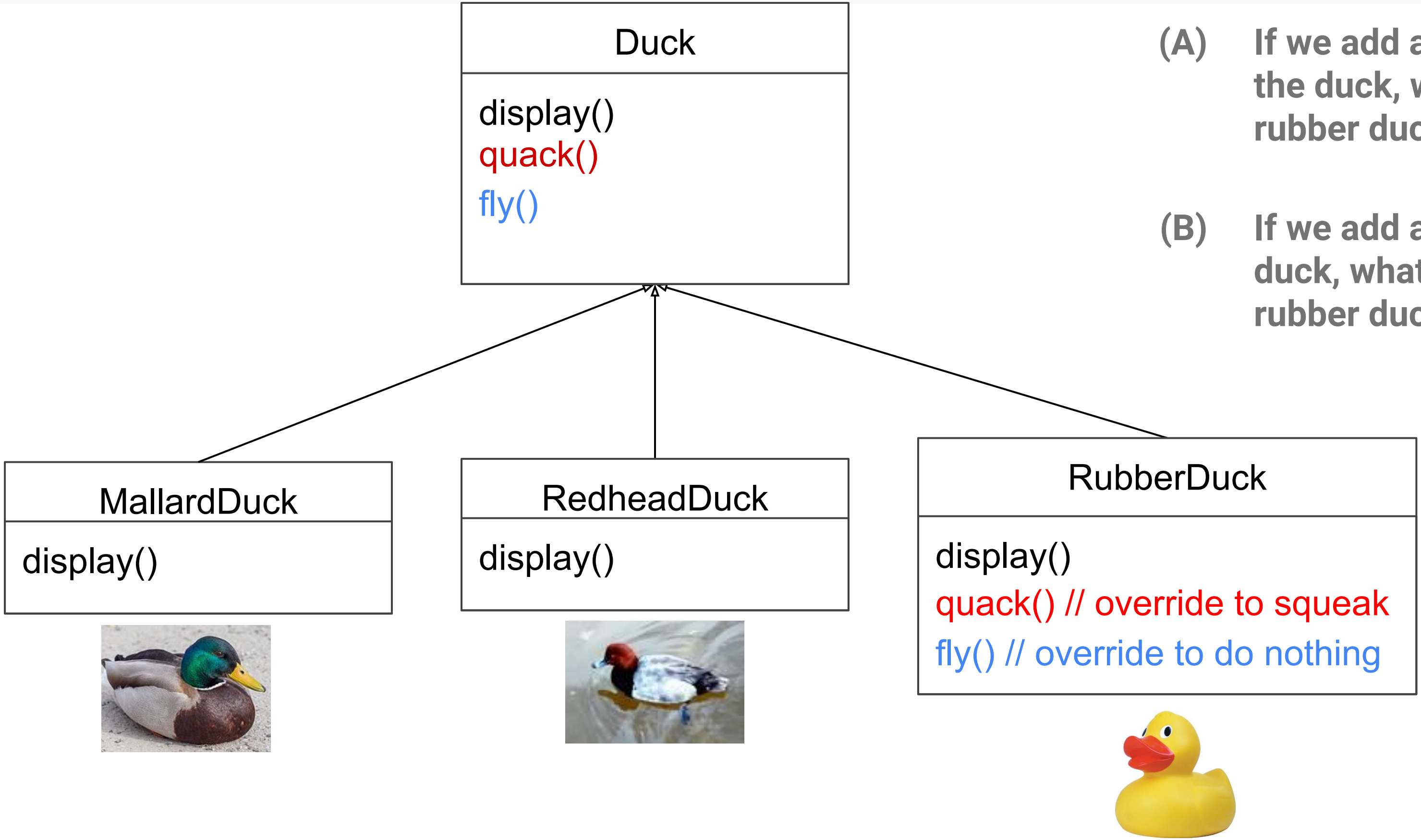
All ducks quack and swim, the superclass takes care of the implementation code.

The display() method is abstract, since all duck subtypes look different.

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

Lots of other types of ducks inherit from the Duck class.

Motivation: Consider These Changes



Duck

display()
quack()
fly()

MallardDuck

display()

RedheadDuck

display()

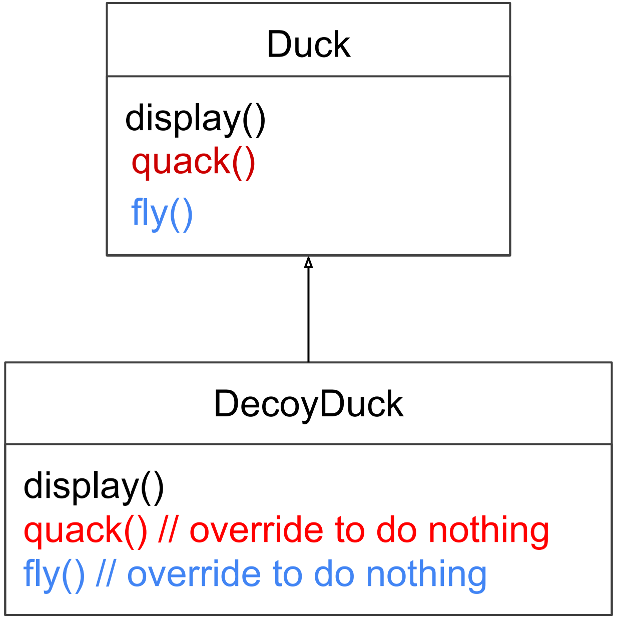
RubberDuck

display()
quack() // override to squeak
fly() // override to do nothing

(A) If we add a “quack” method to the duck, what will we do for a rubber duck that squeaks?

(B) If we add a “fly” method to the duck, what will we do for a rubber duck that does not fly?

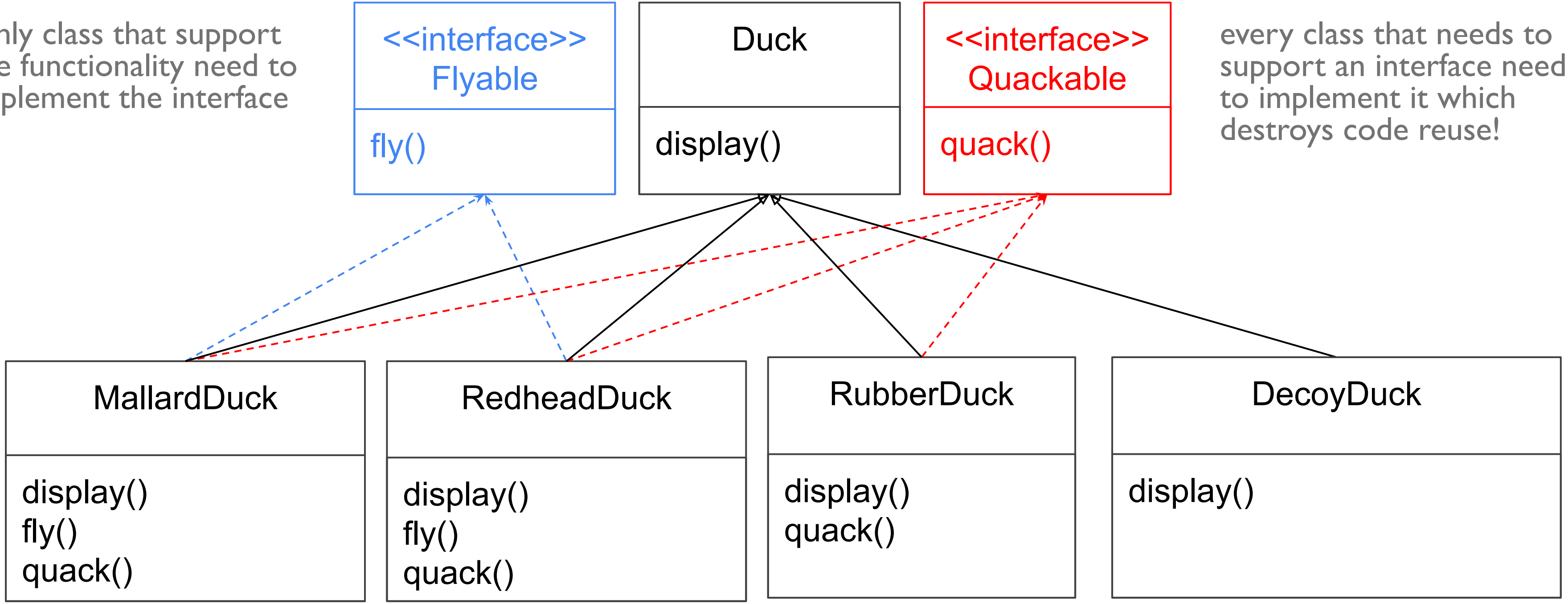
Motivation: Add a Wooden Decoy Duck


Duck
display()
quack()
fly()
DecoyDuck
display()
quack() // override to do nothing
fly() // override to do nothing

- Every new class that inherits unwanted behavior needs to be overridden
- Hence, inheritance is not always the right answer
- How about using **interfaces** instead?

Motivation: Recast Using Interfaces

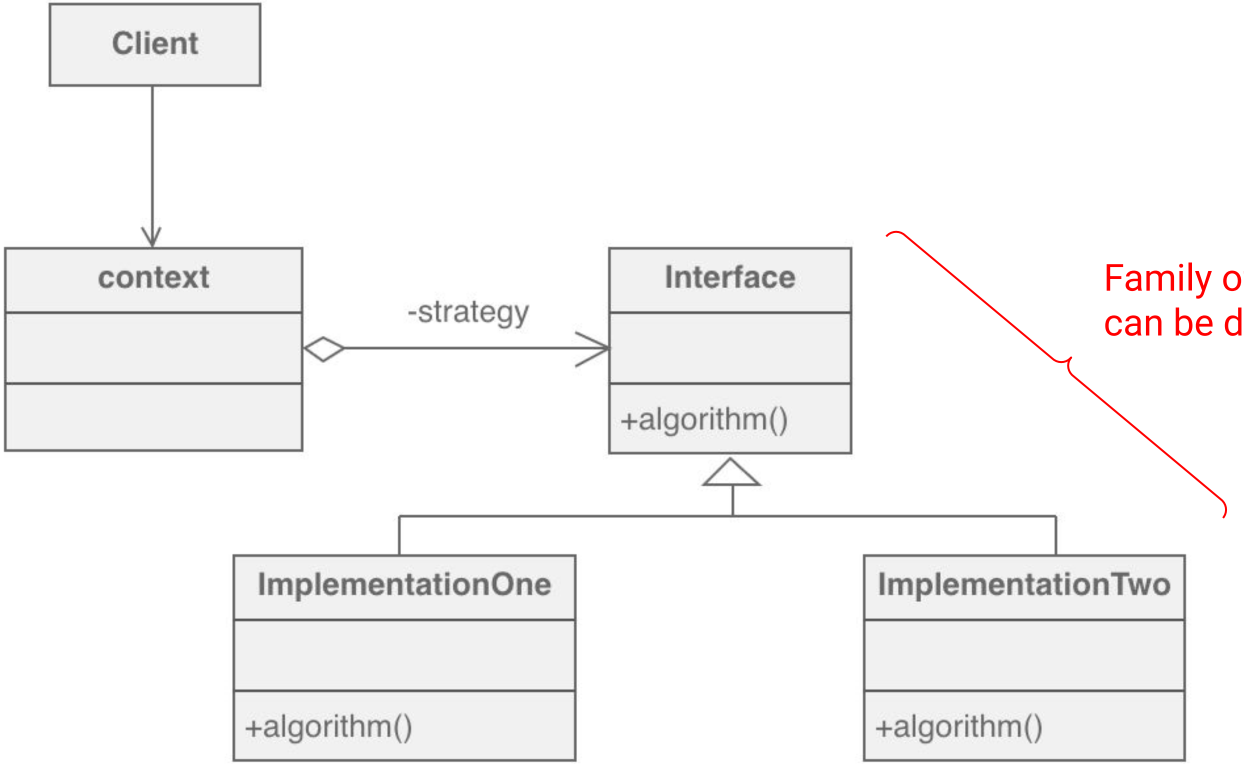
Only class that support the functionality need to implement the interface



every class that needs to support an interface needs to implement it which destroys code reuse!

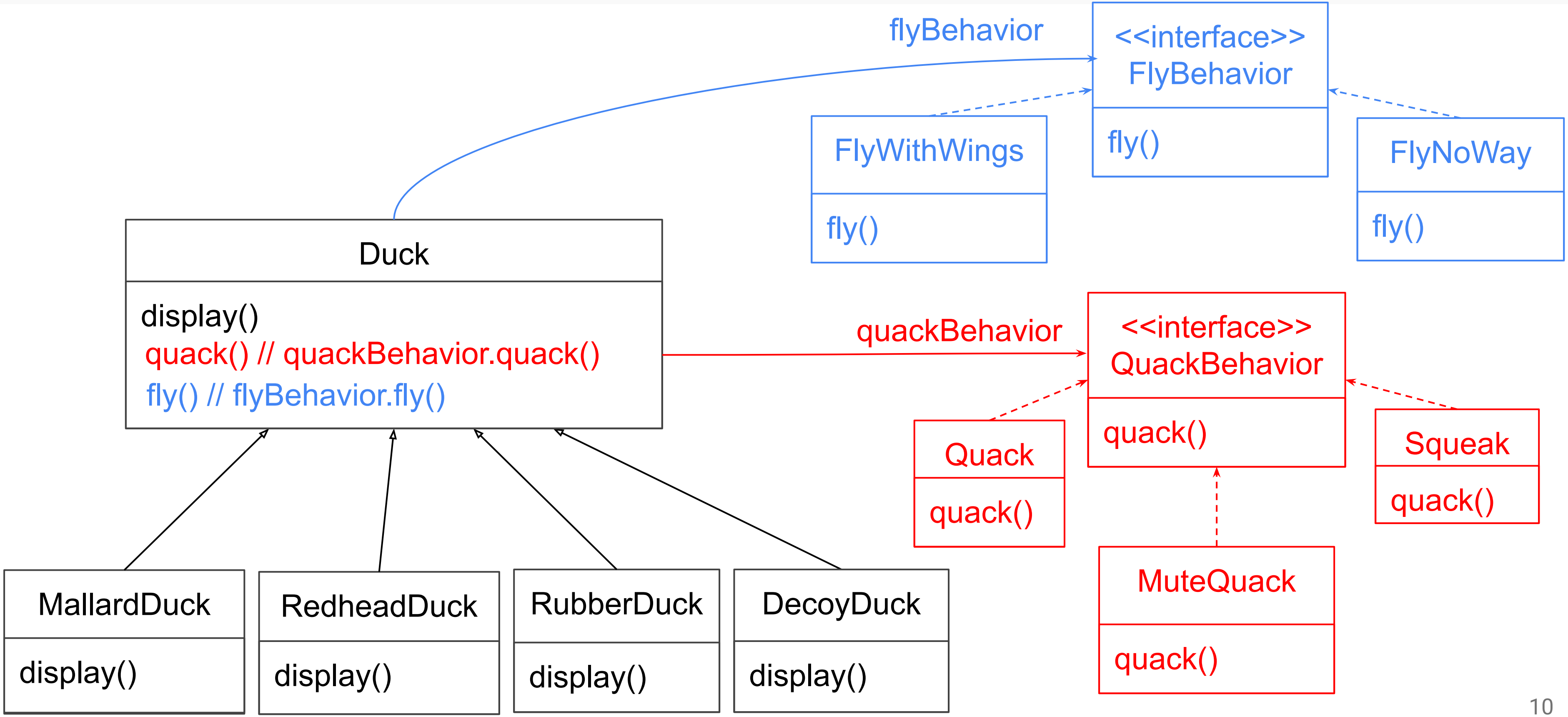
What are pros and cons of this design?

Strategy Pattern



Family of strategies that can be dynamically set

Example: SimUDuck with Strategies



Equivalent Code

```
abstract class Duck {
        FlyBehavior flyBehavior;
        QuackBehavior quackBehavior;

        public Duck() {        }
        public void setFlyBehavior (FlyBehavior fb) {
                flyBehavior = fb;
        }
        public void setQuackBehavior(QuackBehavior qb) {
                quackBehavior = qb;
        }
        public void performFly() {
                 flyBehavior.fly();
        }
        public void performQuack() {
                quackBehavior.quack();
        }
        abstract void display();
}
```

```
class RubberDuck extends Duck {
        public RubberDuck() {
                setQuackBehavior(new Squeek());
                setFlyBehavior(new FlyNoWay());
        }
        public void display() {
                System.out.println("I'm a rubber duck");
        }
}

class DecoyDuck extends Duck {
        public DecoyDuck() {
                setQuackBehavior(new MuteQuack());
                setFlyBehavior(new FlyNoWay());
        }
        public void display() {
                System.out.println("I'm a decoy duck");
        }
}
```

SimUDuck

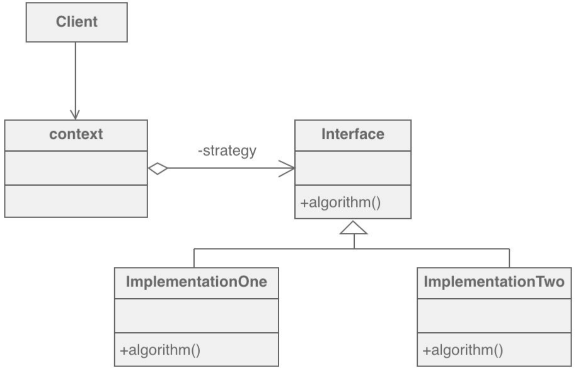
```
public class SimUDuck {

        public static void main(String[] args) {
                MallardDuck mallard = new MallardDuck();
                mallard.performQuack();

                RubberDuck rubberDuckie = new RubberDuck();
                rubberDuckie.performQuack();

                // change behavior at runtime
                rubberDuckie.setFlyBehavior(new FlyBatteryPowered());
                rubberDuckie.performFly();
        }
}
```

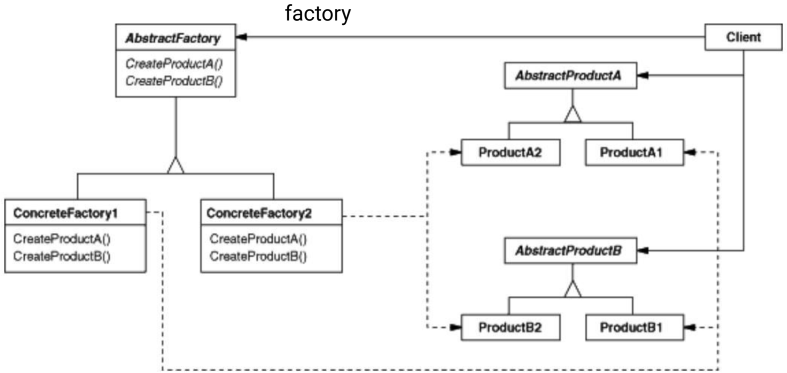
Strategy vs. Abstract Factory

- Both Strategy and Abstract Factory patterns involve delegation

```
// Strategy
Duck duck = new MallardDuck();
duck.quack();
// calls duck.strategy.quack()
```

```
// Abstract Factory
PizzaStore store = new NYPizzaStore();
Pizza p = store.createPizza("cheese");
// calls store.factory.createCheese()
// calls store.factory.createSauce()
```



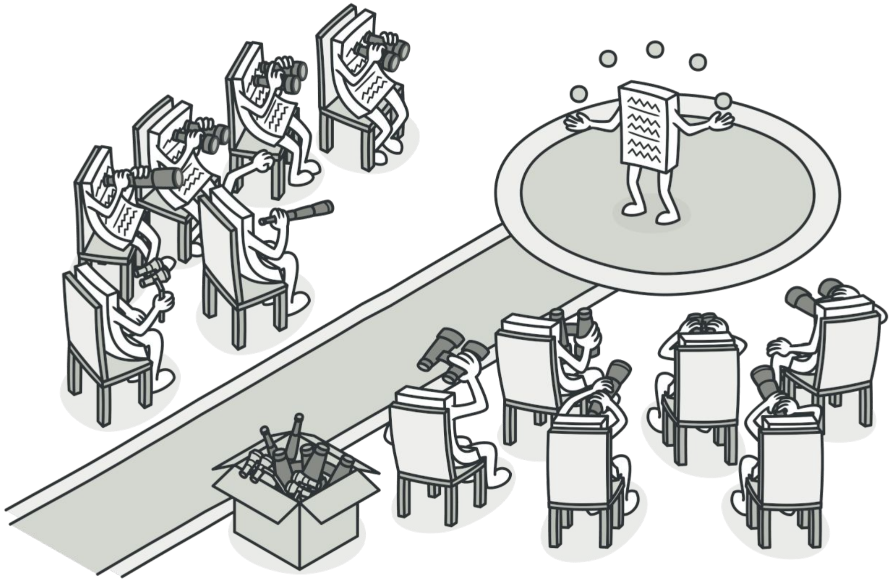


Observer Pattern

Observer Pattern

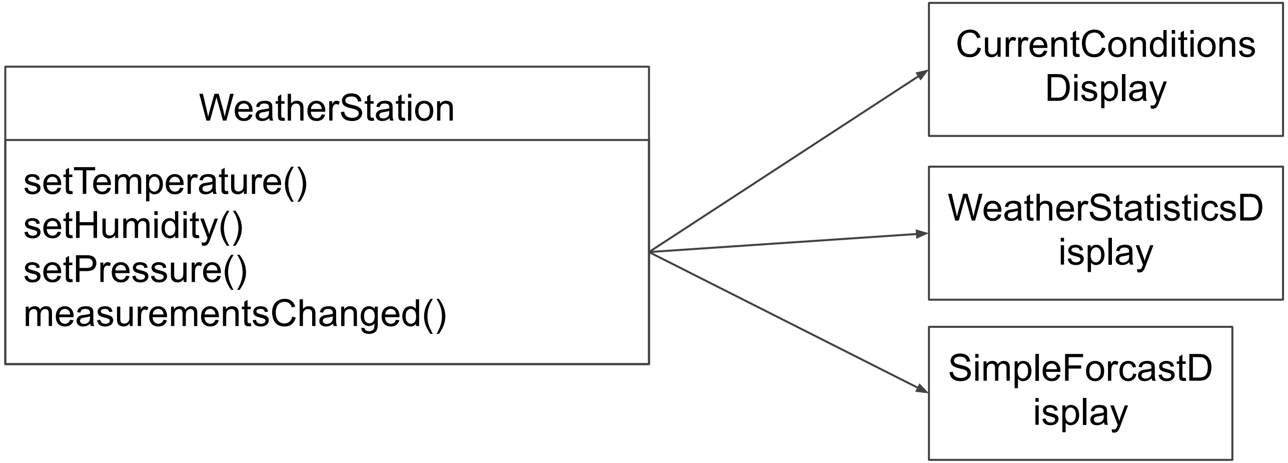
Problem

When we need to ensure that when one object changes state, all of its dependent objects are updated.



Motivation: Weather Station

- WeatherStation class has a setter method for each measurement state
- measurementsChanged() method called whenever there is a change in state
- Three displays needs to be updated as a result:
  - current conditions,
  - weather statistics
  - simple forecast
- System should be expandable to other types of displays in the future



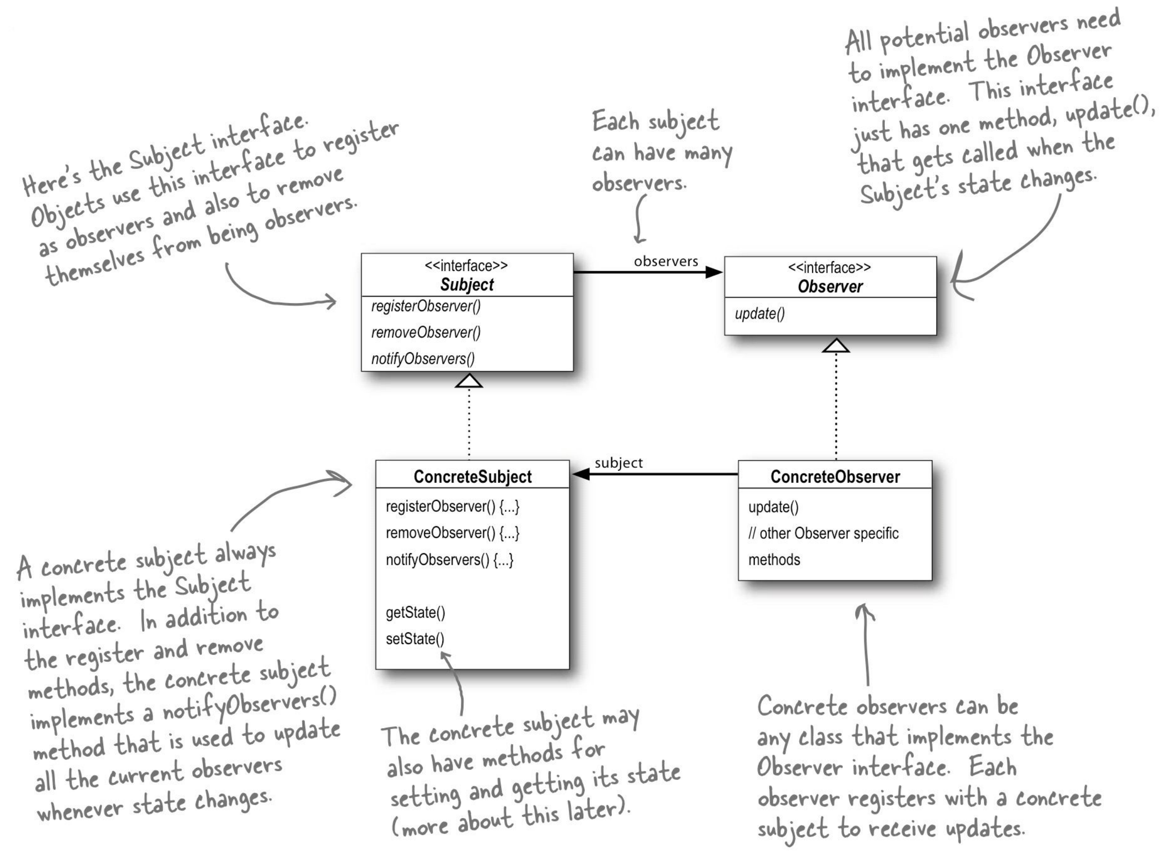
Motivation: First cut at implementation

```
class WeatherStation {
  private float temp, humidity, pressure;
  private CurrentConditionsDisplay currentConditionsDisplay;
  private WeatherStatisticsDisplay weatherStatisticsDisplay;
  private SimpleForecastDisplay simpleForecastDisplay;
  public void measurementsChanged(){
        currentConditionsDisplay.update (temp, humidity, pressure);
        weatherStatisticsDisplay.update (temp, humidity, pressure);
        simpleForecastDisplay.update (temp, humidity, pressure);
  }
  // other methods
}
```

By hard coding the displays, there is no way to add additional display elements without making code change

How do we support another type of displays beyond those three?

Observer Pattern



Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.

<<interface>>
Subject

registerObserver()
removeObserver()
notifyObservers()

observers

<<interface>>
Observer

update()

subject

ConcreteSubject

registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}

getState()
setState()

ConcreteObserver

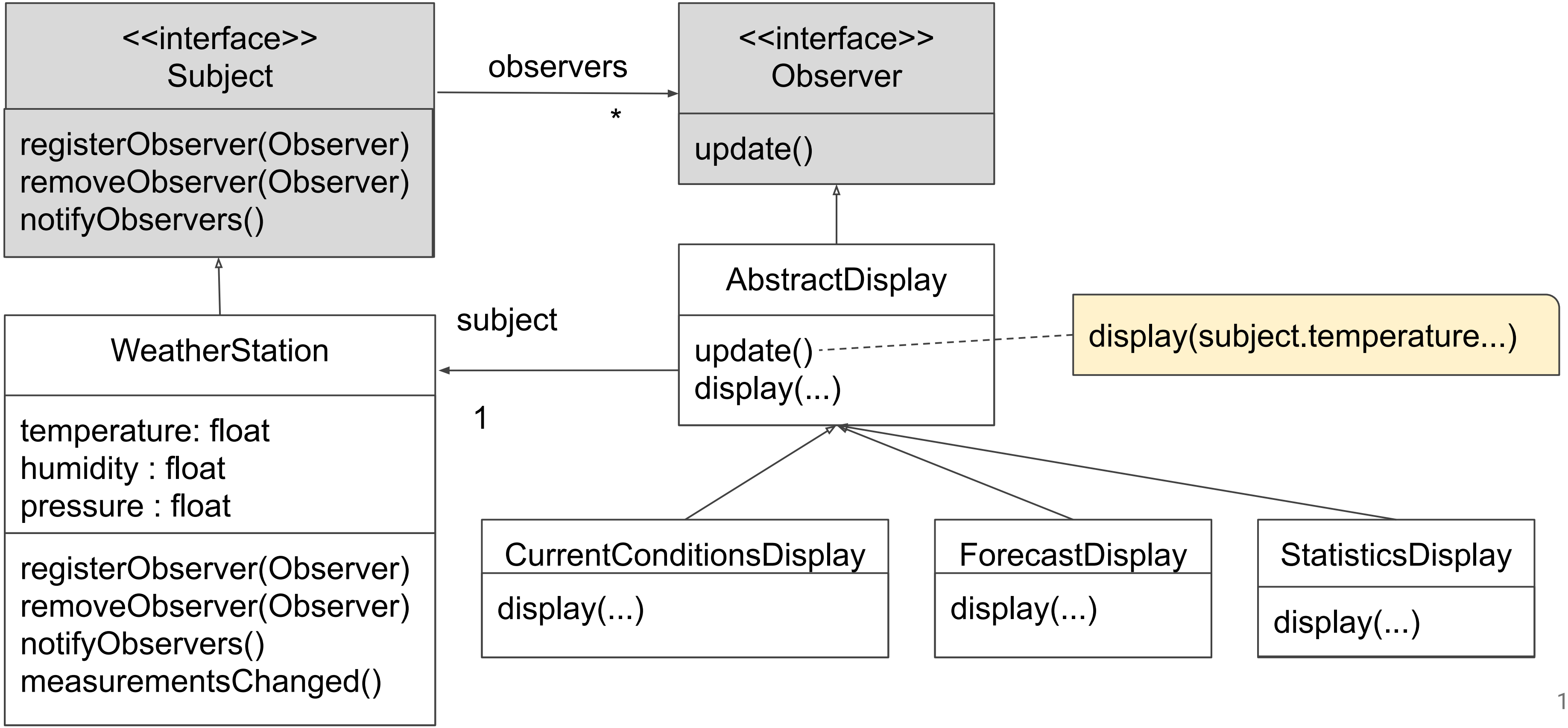
update()
// other Observer specific methods

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Example: Weather Station



Example: Subject and Observer Interfaces

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

Example: Concrete Subject

```
class WeatherStation implements Subject {
        private List<Observer> observers;

        // the state (with getters and setters)
        private float temperature;
        private float humidity;
        private float pressure;

        public WeatherStation() {
                observers = new ArrayList();
        }
        public void registerObserver(Observer o) {
                observers.add(o);
        }
        public void removeObserver(Observer o) {
                observers.remove(i);
        }

        private void notifyObservers() {
                for (int i = 0; i < observers.size(); i++) {
                        Observer observer = observers.get(i);
                        observer.update();
                }
        }
        // called from state setters
        protected void measurementsChanged() {
                notifyObservers();
        }
}
```

Example: Concrete Observer

```
abstract class AbstractDisplay  implements Observer{
        private WeatherStation subject;
        public AbstractDisplay(WeatherStation subject) {
                this.subject = subject;
                subject.registerObserver(this);
        }
        @Override
        public update() {
                float temperature = subject.getTemperature();
                …
                display(temperature, …);
        }
        protected abstract void display(float temperature, …);
}
```

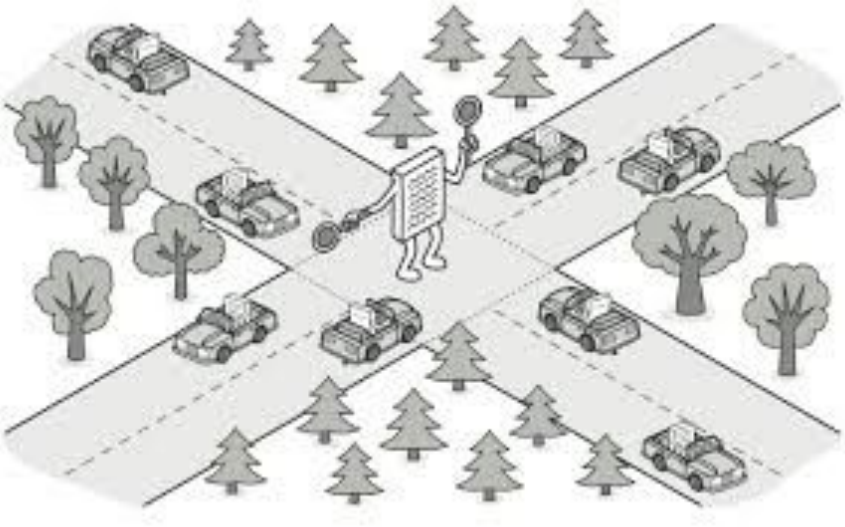
```
class ForcastDisplay  extends AbstractDisplay {
        public ForcastDisplay(WeatherStation subject) {
                super(subject);
        }
        @Override
        protected void display(float temperature, …) {
                System.out.println(“forecast is …”);
        }
}
```

Mediator Pattern

Mediator Pattern

Problem

When there should not be tight coupling between a set of interacting objects and it should be possible to change the interaction independent of the objects.

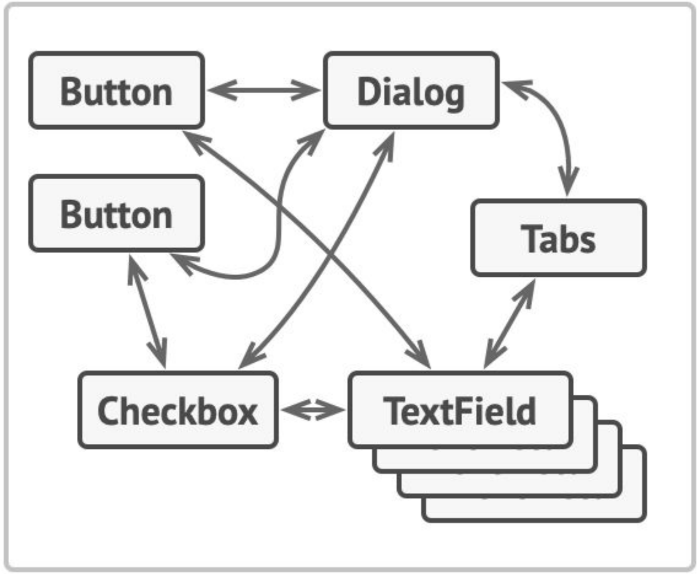


Motivation

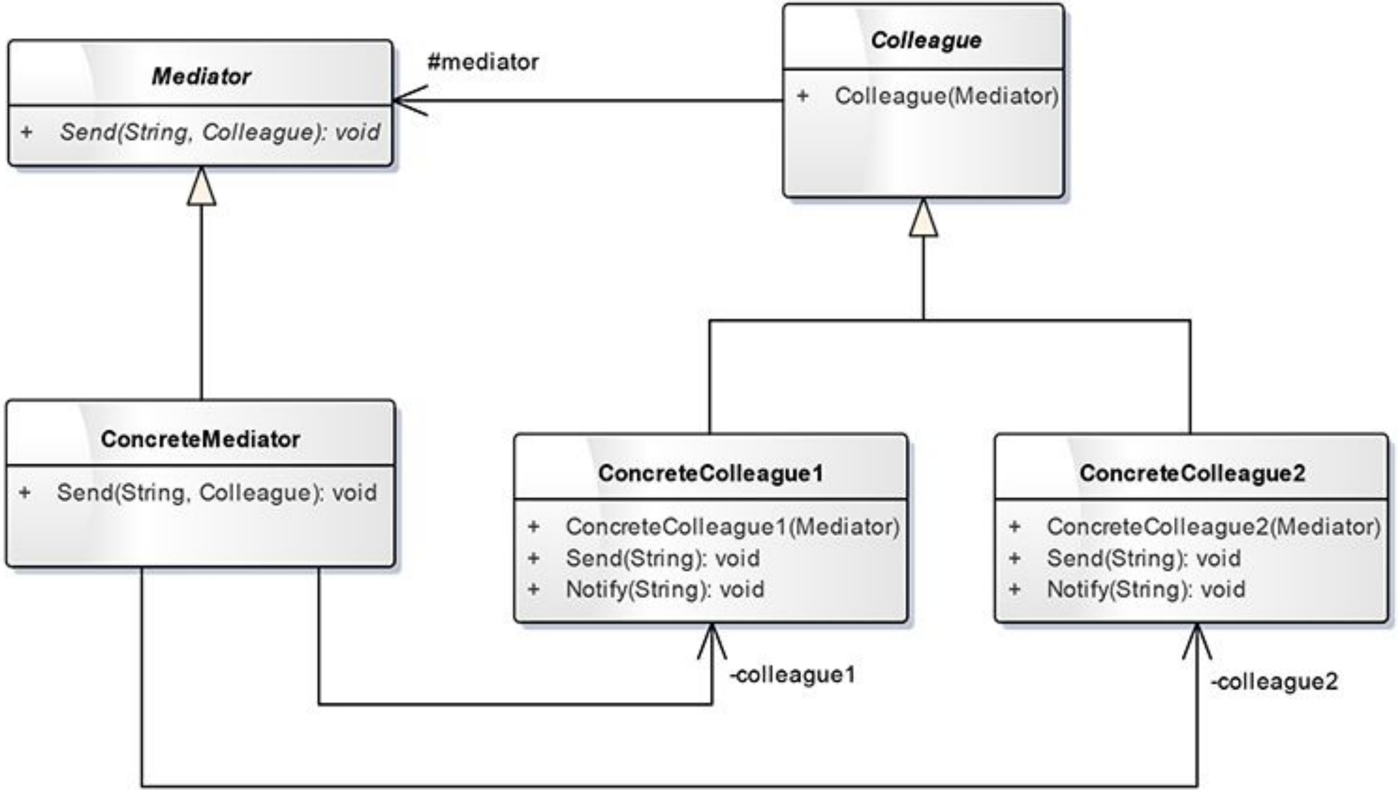
- You have a dialog that consists of various controls (e.g., text fields, checkboxes, buttons, etc).

- Some of the controls interact with others (e.g., selecting a checkbox may reveal a hidden text field)

- By having this logic implemented directly inside the code of the control you make these classes much harder to reuse in other forms of the app.



Mediator Pattern



Mediator Benefits and Drawbacks

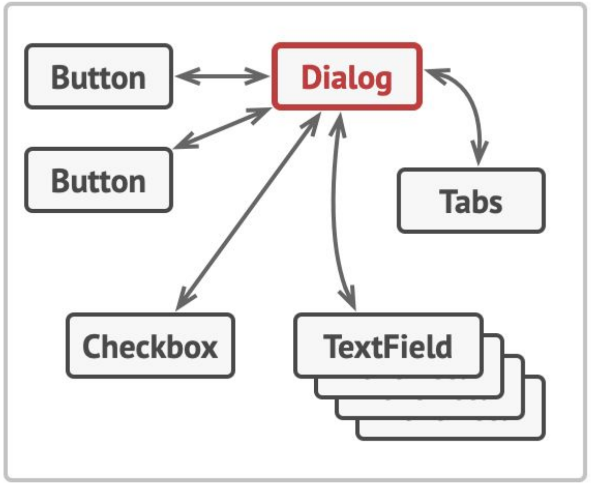
- Benefits
  - Increase the reusability of the objects supported by the Mediator by decoupling them
  - Simplifies maintenance of the system by centralizing control logic
  - Simplifies and reduces the variety of messages sent between objects in the system
- Drawbacks
  - Without proper design, the mediator object itself can become overly complex
  - Mediator also exposes a single point of vulnerability and complexity

Example: Dialog as Mediator

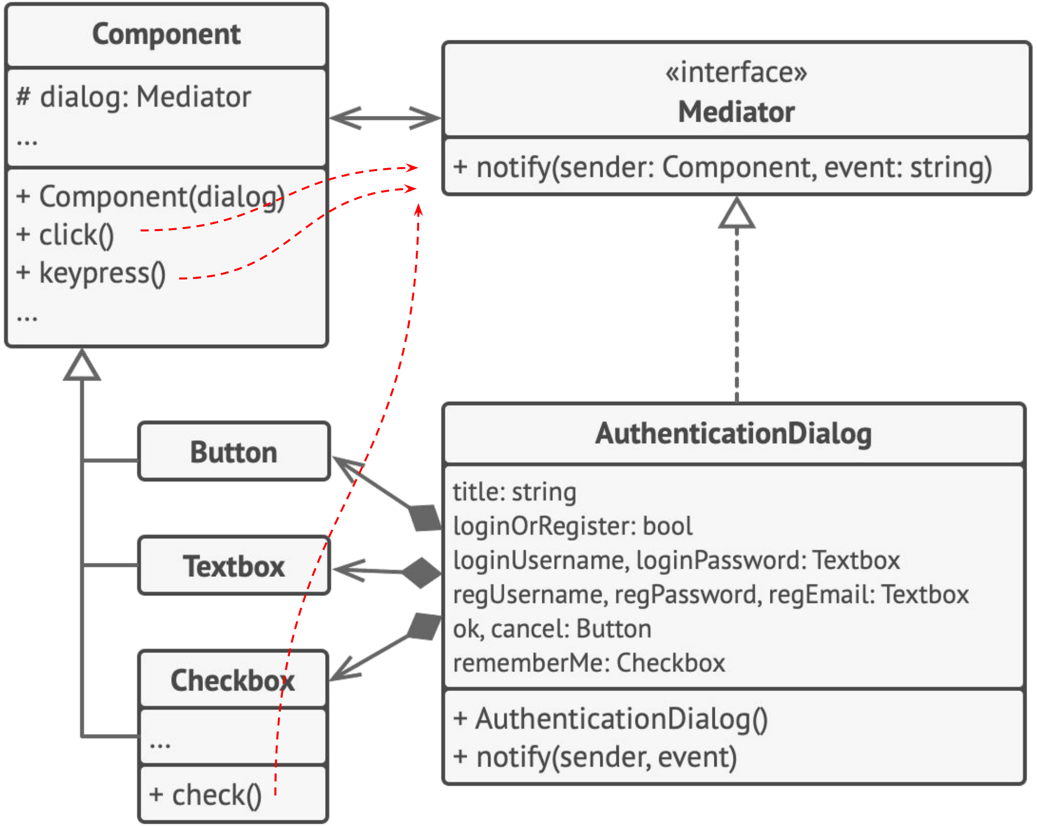
- With a mediator in place, all control objects can be decoupled
    - They tell the Mediator when their state changes
    - They respond to requests from the Mediator

- Mediator contains all control logic for the entire system.

- When an existing control has a new logic or a new control is added to the dialog, you will know that all logic will be added to the mediator



Example: Dialog as Mediator



Example: Dialog as Mediator

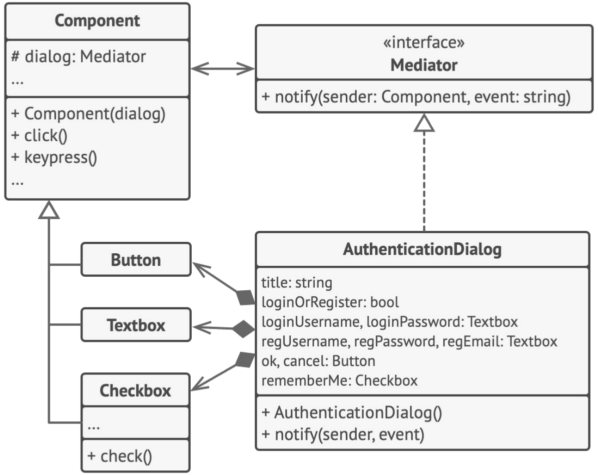
```
interface Mediator {
        public void notify(Component sender, String event);
}

class AuthenticationDialog implements Mediator {
        private Checkbox loginOrRegisterChkBx;
        private Textbox usernameText, passwordText;
        private Button okBtn, cancelBtn;

        public void notify(Component sender, String event) {
           if (sender == loginOrRegisterChkBx && event.equals(“check”)) {
              if (loginOrRegisterChkBx.isChecked)
                 title = "Log in";
              else
                 title = "Register";
           } else if (sender == usernameText) {
              ok.serEnabled(!usernameText.isEmpty()
                   &&  !passwordText.isEmpty());
           }
          ...
        }
}
```

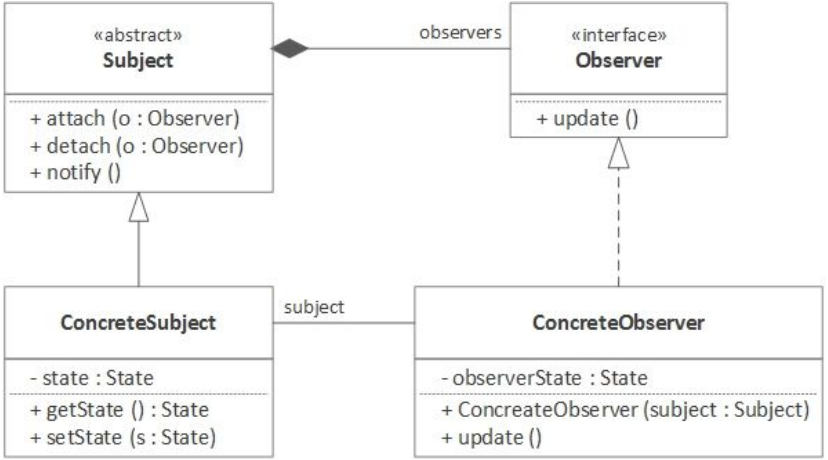
```
abstract class Component {
        protected Mediator dialog;
        public Component(Mediator dialog){
                this.dialog=dialog;
        }
        public void click() {
                dialog.notify(this, “click”);
        }
        public void keyPress() {
                dialog.notify(this, “keyPress”);
        }
}

class Checkbox extends Component {
        public void check() {
                dialog.notify(this, “check”);
        }
}
```

Mediator vs. Observer

- **Mediator** encapsulates many to many communication dependencies.

- **Observer** encapsulates one to many communication dependencies



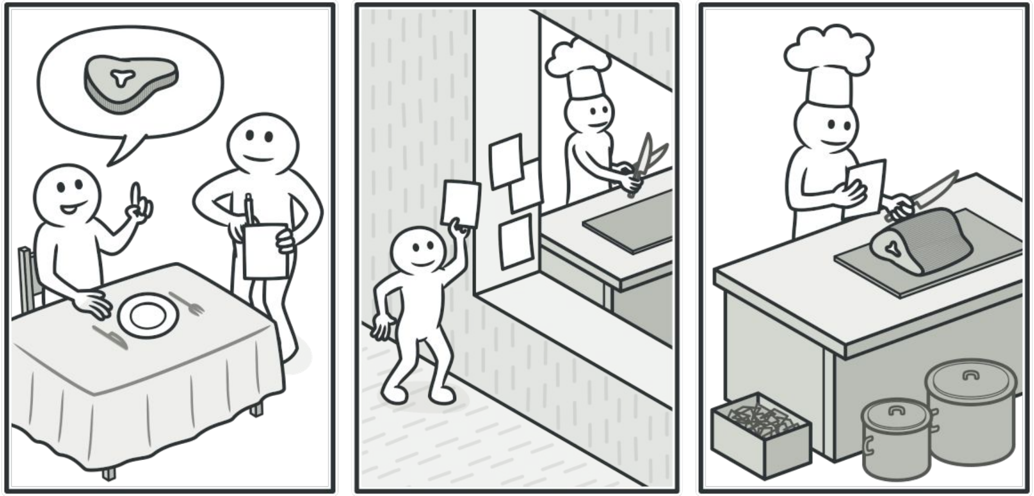


Command Pattern

Command Pattern

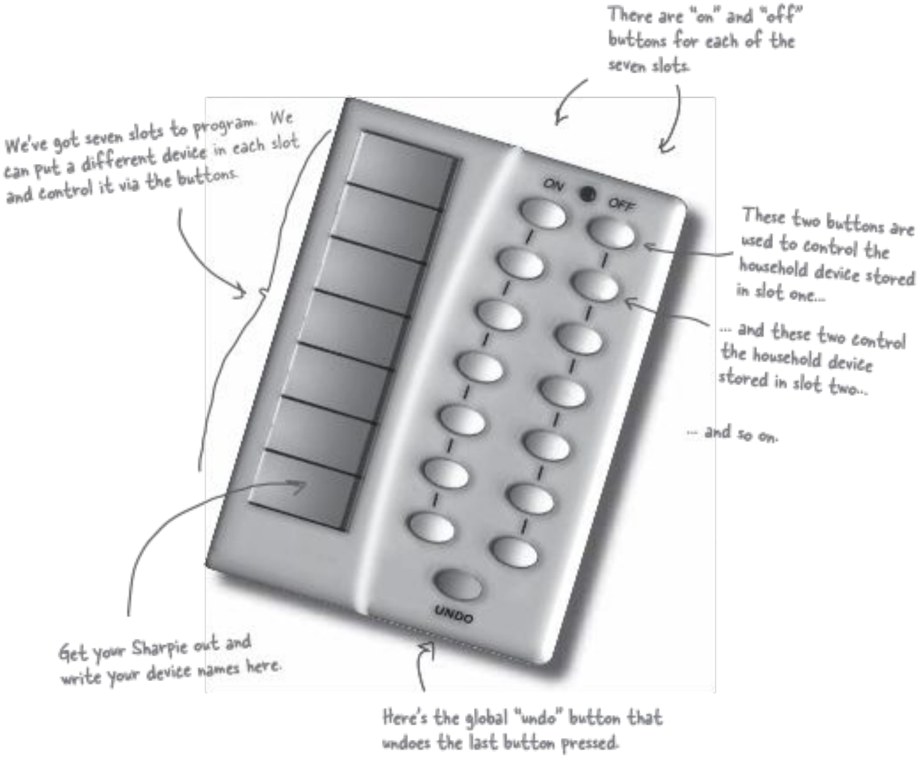
Problem

When we need to issue requests to objects without knowing the receiver of the request nor how the request will be handled.



Making an order at a restaurant

Motivation: A Remote Control



There are "on" and "off" buttons for each of the seven slots.

We've got seven slots to program. We can put a different device in each slot and control it via the buttons.

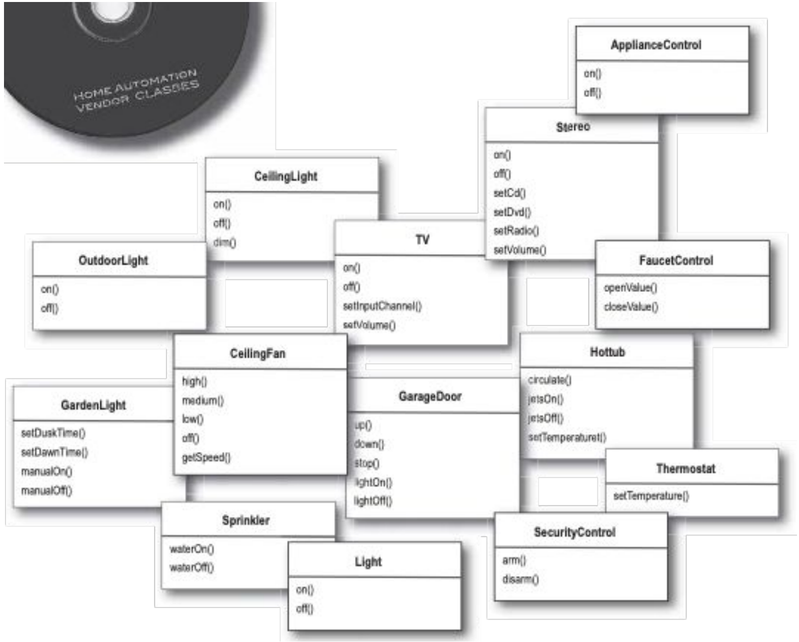
These two buttons are used to control the household device stored in slot one...

... and these two control the household device stored in slot two...

... and so on.

Get your Sharpie out and write your device names here.

Here's the global "undo" button that undoes the last button pressed.



ApplianceControl
on()
off()

Stereo
on()
off()
setCd()
setDvd()
setRadio()
setVolume()

CeilingLight
on()
off()
dim()

TV
on()
off()
setInputChannel()
setVolume()

OutdoorLight
on()
off()

FaucetControl
openValue()
closeValue()

CeilingFan
high()
medium()
low()
off()
getSpeed()

Hottub
circulate()
jetsOn()
jetsOff()
setTemperature()

GardenLight
setDuskTime()
setDawnTime()
manualOn()
manualOff()

GarageDoor
up()
down()
stop()
lightOn()
lightOff()

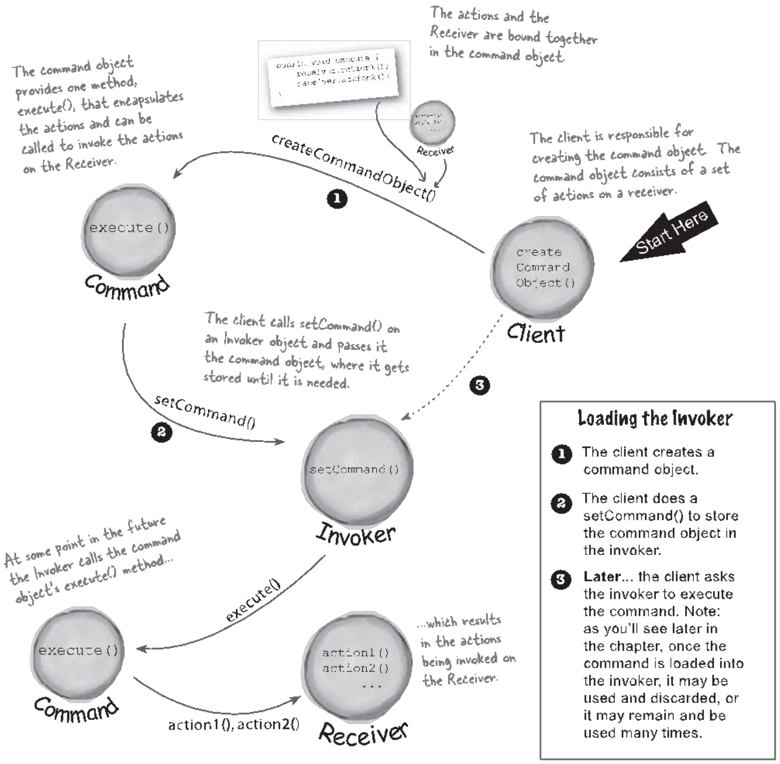
Thermostat
setTemperature()

Sprinkler
waterOn()
waterOff()

Light
on()
off()

SecurityControl
arm()
disarm()

Command Pattern



The command object provides one method, execute(), that encapsulates the actions and can be called to invoke the actions on the Receiver.

The actions and the Receiver are bound together in the command object.

The client is responsible for creating the command object. The command object consists of a set of actions on a receiver.

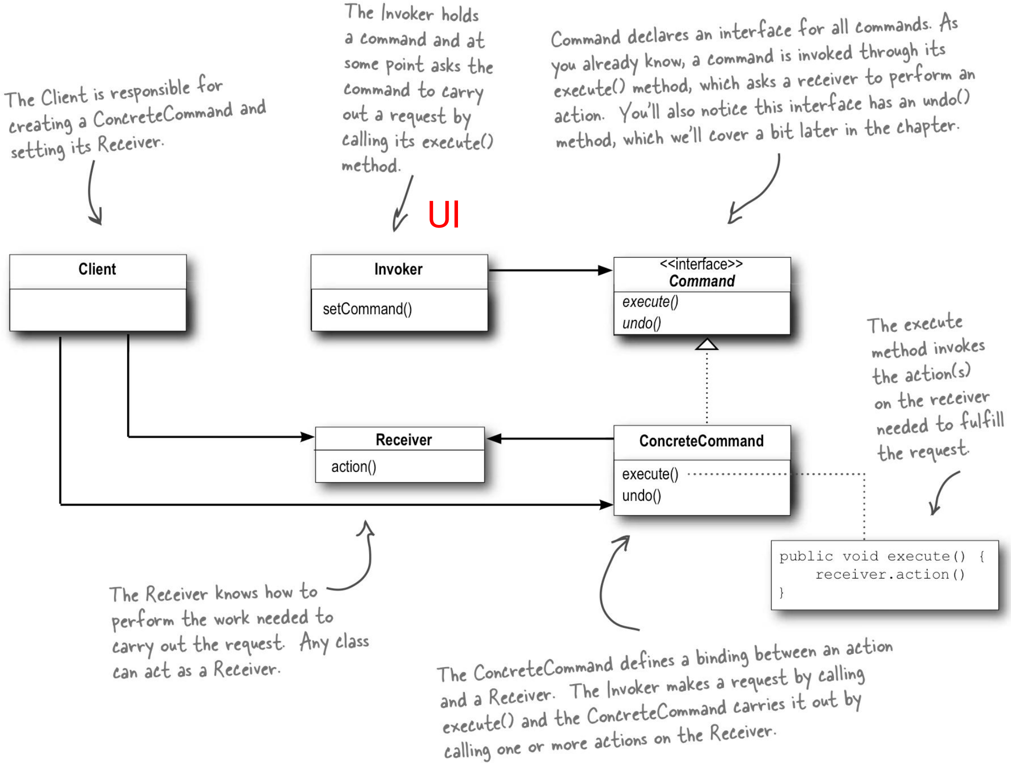
The client calls setCommand() on an Invoker object and passes it the command object, where it gets stored until it is needed.

At some point in the future the Invoker calls the command object's execute() method...

...which results in the actions being invoked on the Receiver.

Loading the Invoker

1. The client creates a command object.
2. The client does a setCommand() to store the command object in the invoker.
3. **Later**... the client asks the invoker to execute the command. Note: as you'll see later in the chapter, once the command is loaded into the invoker, it may be used and discarded, or it may remain and be used many times.



```
public void execute() {
    receiver.action()
}
```

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.

The execute method invokes the action(s) on the receiver needed to fulfill the request.

The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

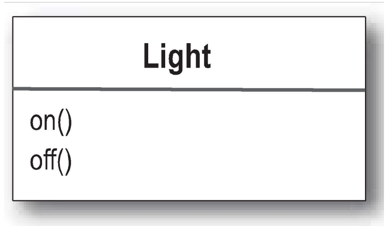
The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

Workflow

Pattern

Example: Remote Control

```
public interface Command {
  public void execute ();
  public void undo ();
}

public class LightOnCommand implements Command {
  Light light; // the receiver
  public LightOnCommand (Light light) {
    this.light = light;
  }
  public void execute () {
    light.on();
  }
  public void undo () {
    light.off();
  }
}
```



Light

on()
off()

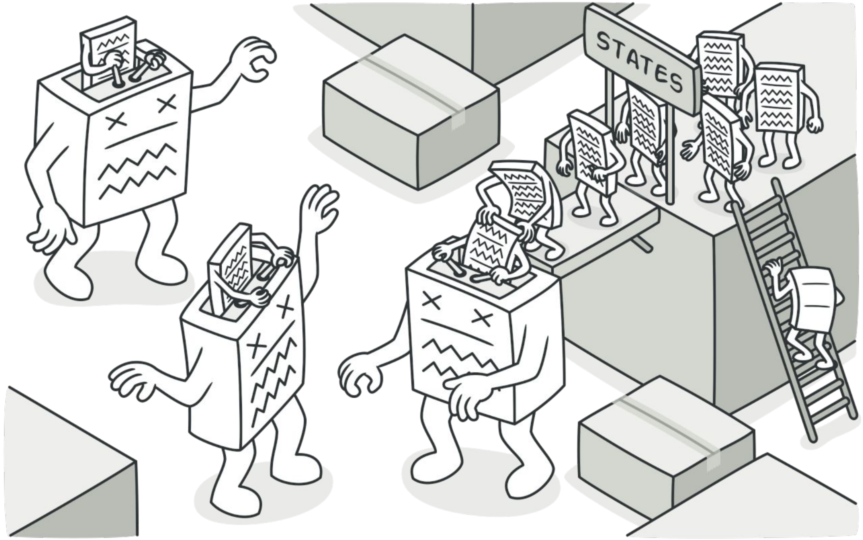
```
public class SimpleRemoteControl {//invoker
  Command slot;
  public void setCommand(Command command) {
    slot = command;
  }
  public void buttonWasPressed() {
    slot.execute();
  }
}
public class RemoteControlTest {//Client
  public static void main(String[] args) {
    SimpleRemoteControl remote =
        new SimpleRemoteControl();
    Light light = new Light();
    LightOnCommand lightOn = new LightOnCommand(light);
    remote.setCommand(lightOn);
    remote.buttonWasPressed();
  }
}
```

State Pattern

State Pattern

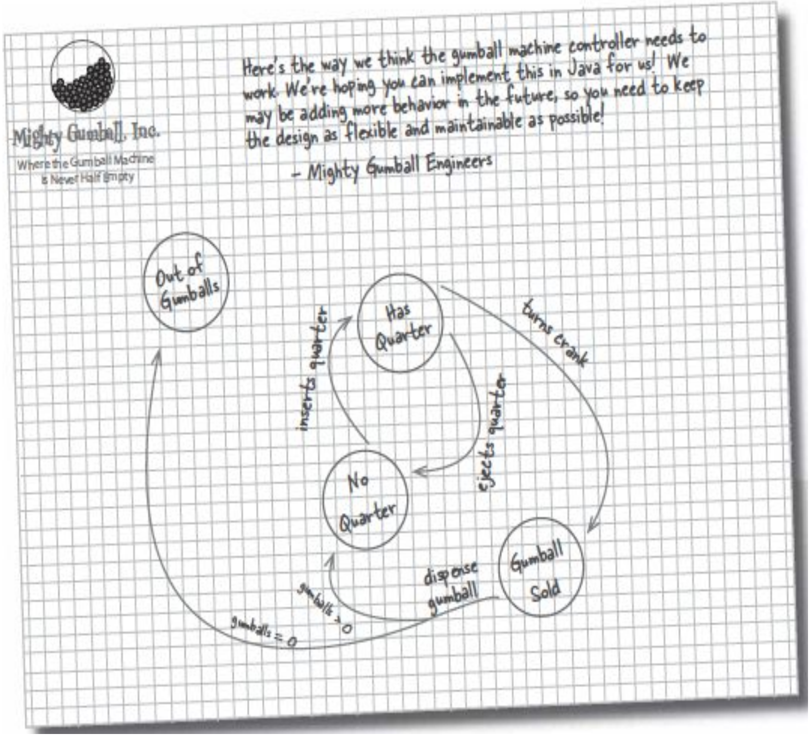
Problem

When a monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state.



Motivation: Gumball machine

- The Gumball Machine requirements are represented as a State Machine diagram



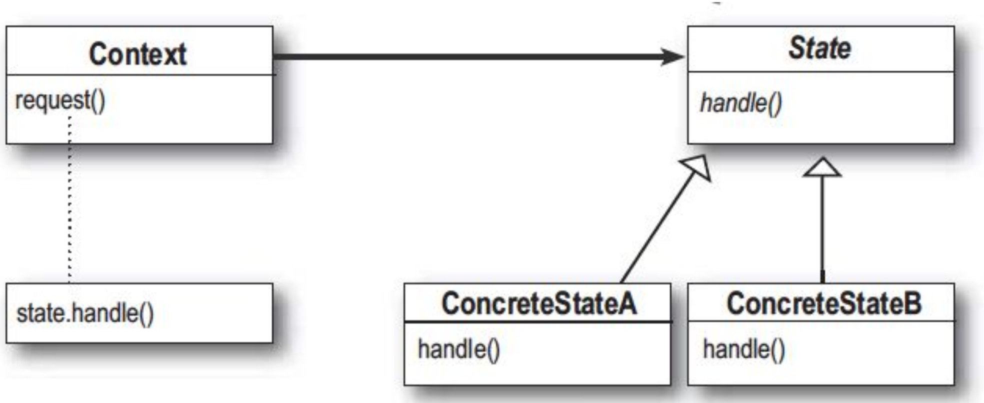
Mighty Gumball, Inc.
Where the Gumball Machine is Never Half Empty

Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this in Java for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

– Mighty Gumball Engineers

State Pattern

- Define a State interface
- Implement the interface for every state of the machine
- Localize the behavior for each state in the implementation class
- Eliminate the conditional code by delegating to the current state class
- Adding new State => adding a new class
- Handling new requirements is a matter of overriding some method



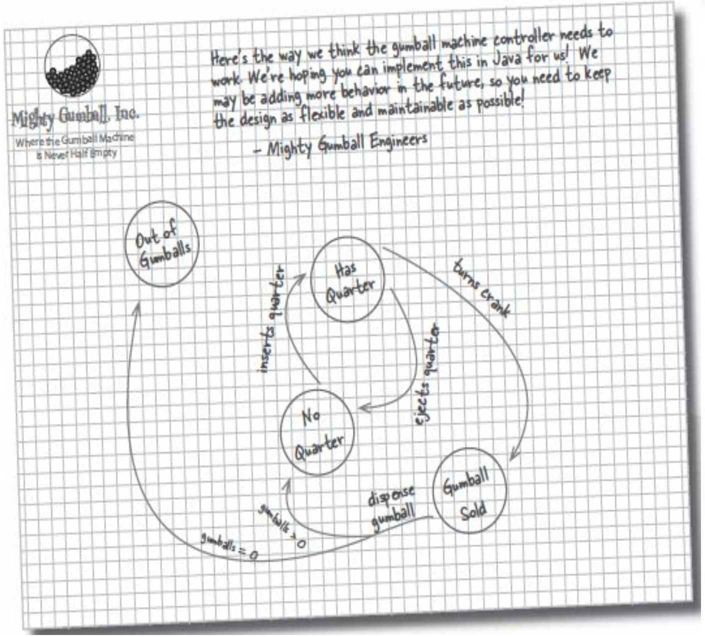
Example: Gumball State Machine

```
interface State {
   public void insertQuarter();
   public void ejectQuarter();
   public void turnCrank();
   public void dispenseGumball();
}
abstract class StateImpl implements State {
   void enterState() {}
   void exitState() {}
   public void insertQuarter() {}
   public void ejectQuarter() {}
   public void turnCrank() {}
   public void dispenseGumball() {}
}
public class GumballMachine implements State {
   StateImpl soldOutState = new SoldOutState(this);
   StateImpl noQuarterState = new NoQuarterState(this);
   StateImpl hasQuarterState  = new HasQuarterState(this);
   StateImpl soldState = new SoldState(this);

   private StateImpl state;
   private int balls = 0;
   // … continue on next column ....
```

```
   public GumballMachine(int numberGumballs) {
          this.balls = numberGumballs;
          if (balls > 0) {
               setState(noQuarterState);
          } else {
               setState(soldOutState);
          }
   }
   void setState(StateImpl nextState) {
          If (state != null)
               state.exitState();
          state = nextState;
          state.enterState();
    }

   int getBalls() { return balls; }
   void releaseBall() { if (balls > 0) balls--; }

    public void insertQuarter() { state.insertQuarter(); }
    public void ejectQuarter() { state.ejectQuarter(); }
    public void turnCrank() { state.turnCrank(); }
    public void dispenseGumball() { state.dispenseGumball(); }
}
```

Example: Gumball State Machine

```
class SoldState extends StateImpl {
        GumballMachine gumballMachine;
        public SoldState(GumballMachine gumballMachine) {
                this.gumballMachine = gumballMachine;
        }
        public void insertQuarter() {
                System.out.println("Please wait, we're already giving you a gumball");
        }
        public void ejectQuarter() {
                System.out.println("Sorry, you already turned the crank");
        }
        public void turnCrank() {
                System.out.println("Turning twice doesn't get you another gumball!");
        }
        public void dispenseGumball() {
                gumballMachine.releaseBall();
                if (gumballMachine.getCount() > 0) {
                        gumballMachine.setState(gumballMachine.noQuarterState);
                } else {
                        System.out.println("Oops, out of gumballs!");
                        gumballMachine.setState(gumballMachine.soldOutState);
                }
        }
}
```



Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this in Java for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

– Mighty Gumball Engineers

Benefits and Drawbacks of State Pattern

- Benefits
  - Encapsulates all the behavior of a state in one object
  - Helps avoiding inconsistent states changes

- Drawback
  - Make the code bulky and increases the number of objects
  - State Interface is brittle

Behavioral Patterns Quiz

References

- Freeman, E., Freenman, E., “Head First Design Pattern.” O’Rielly, 2004.
- Software Design Patterns https://bibekg.github.io/software-design-patterns/