Strategy Pattern

Strategy Pattern: Introduction

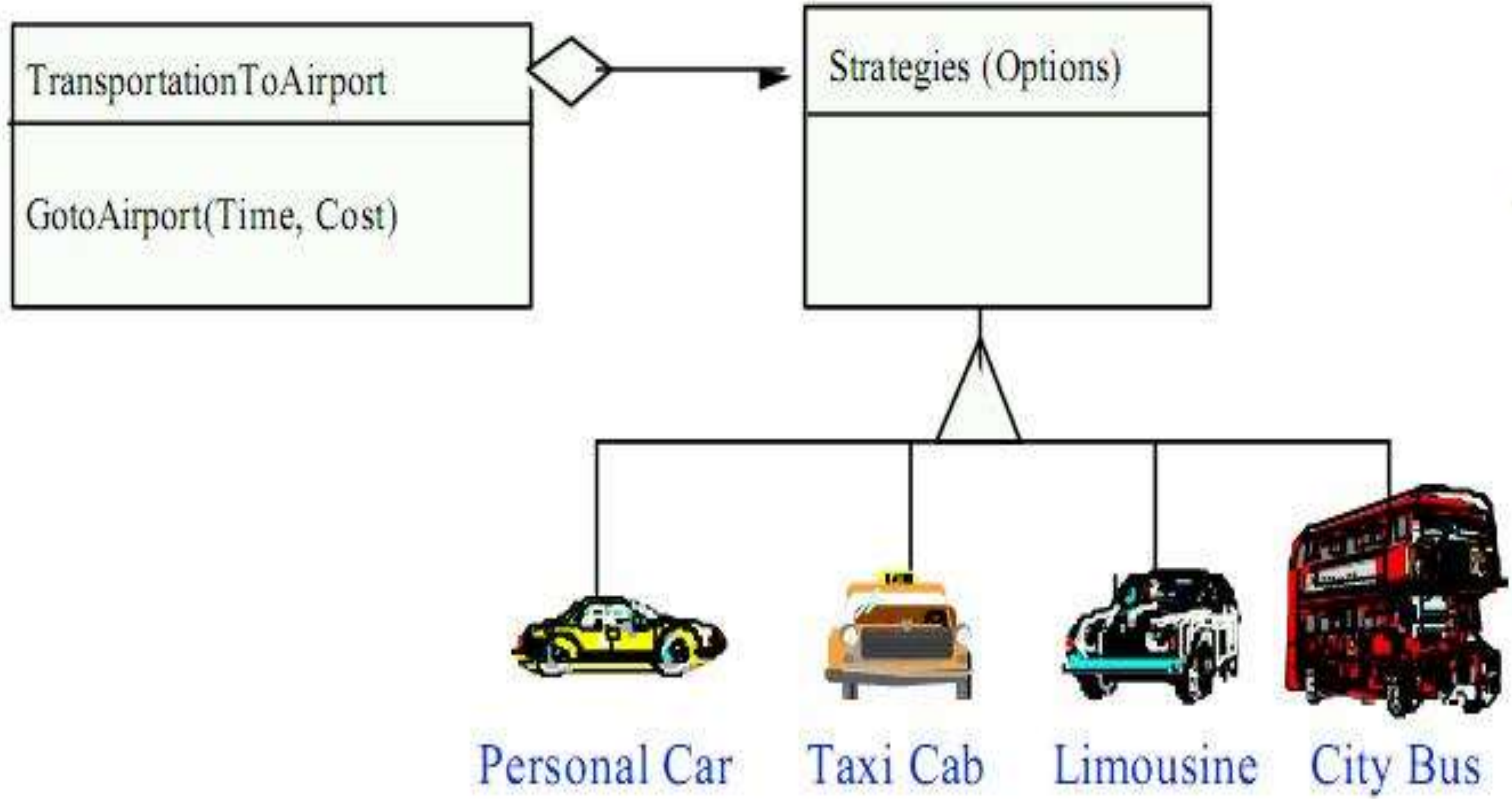
- Lets a family of algorithms to be used interchangeably by a client.
- **Non-software example: Transportation to airport.**
- Several options exist:
  - Drive own car, take a taxi, or an airport shuttle.
  - New modes such as subways and helicopters can become available later.
  - A traveler can chose a Strategy based on tradeoffs between cost, convenience, and time.

Strategy Pattern

- Helps manage several different implementations:
  - Of what is, conceptually, the same functionality.

- **A strategy is an algorithm represented as an object.**

Non software example



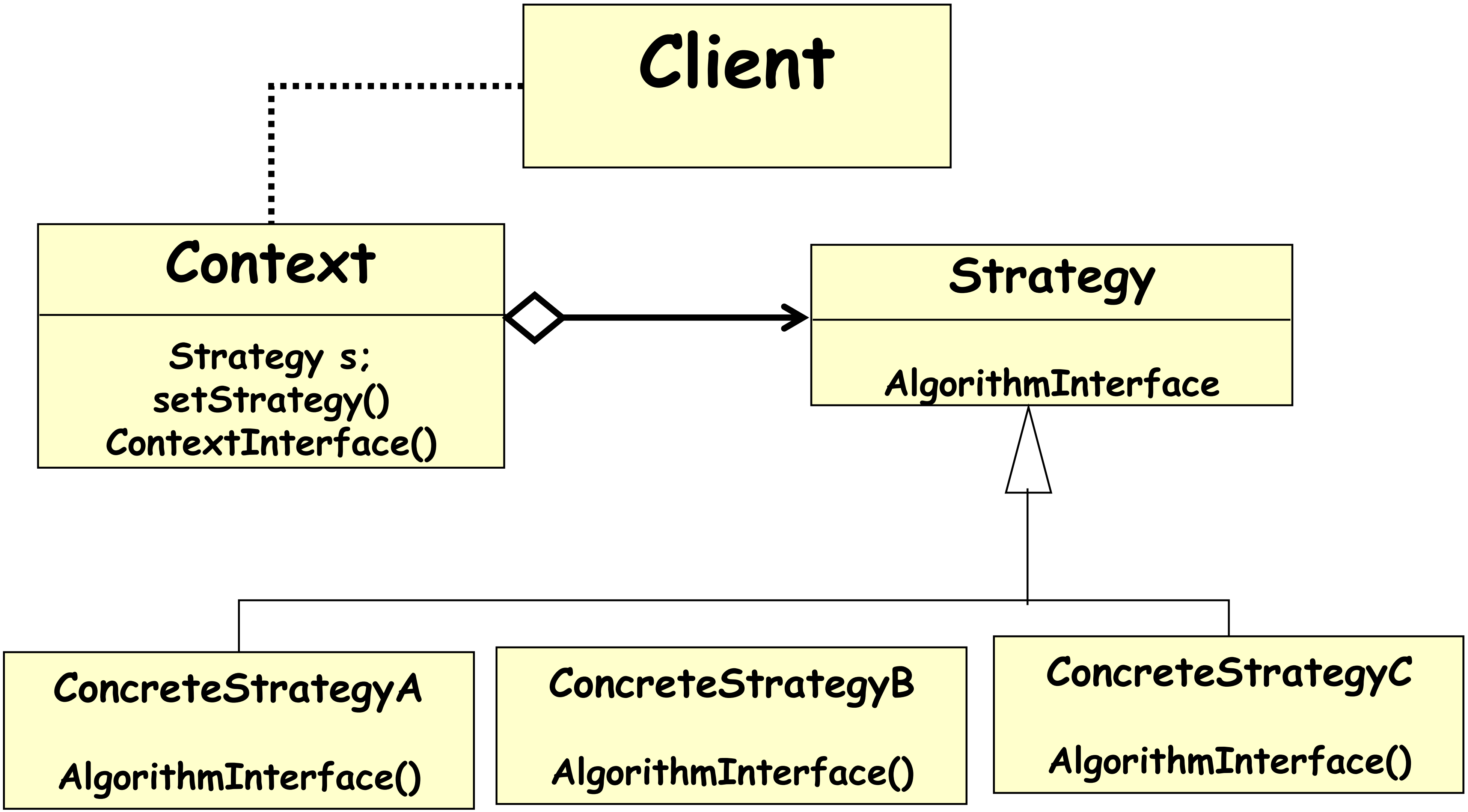
Strategy defines a set of algorithms that can be used interchangeably.

The Strategy Pattern: Intent

- Define a family of algorithms:
  - **Encapsulate each one, and make them interchangeable.**

- Strategy lets the algorithm vary independently from clients that use it.

Strategy Pattern: Structure



Policy decides which Strategy is best given the current Context

Context Class

- Context:
  - Clients interact with the Context, not Strategy
  - Context uses strategy

Advantages of Strategy

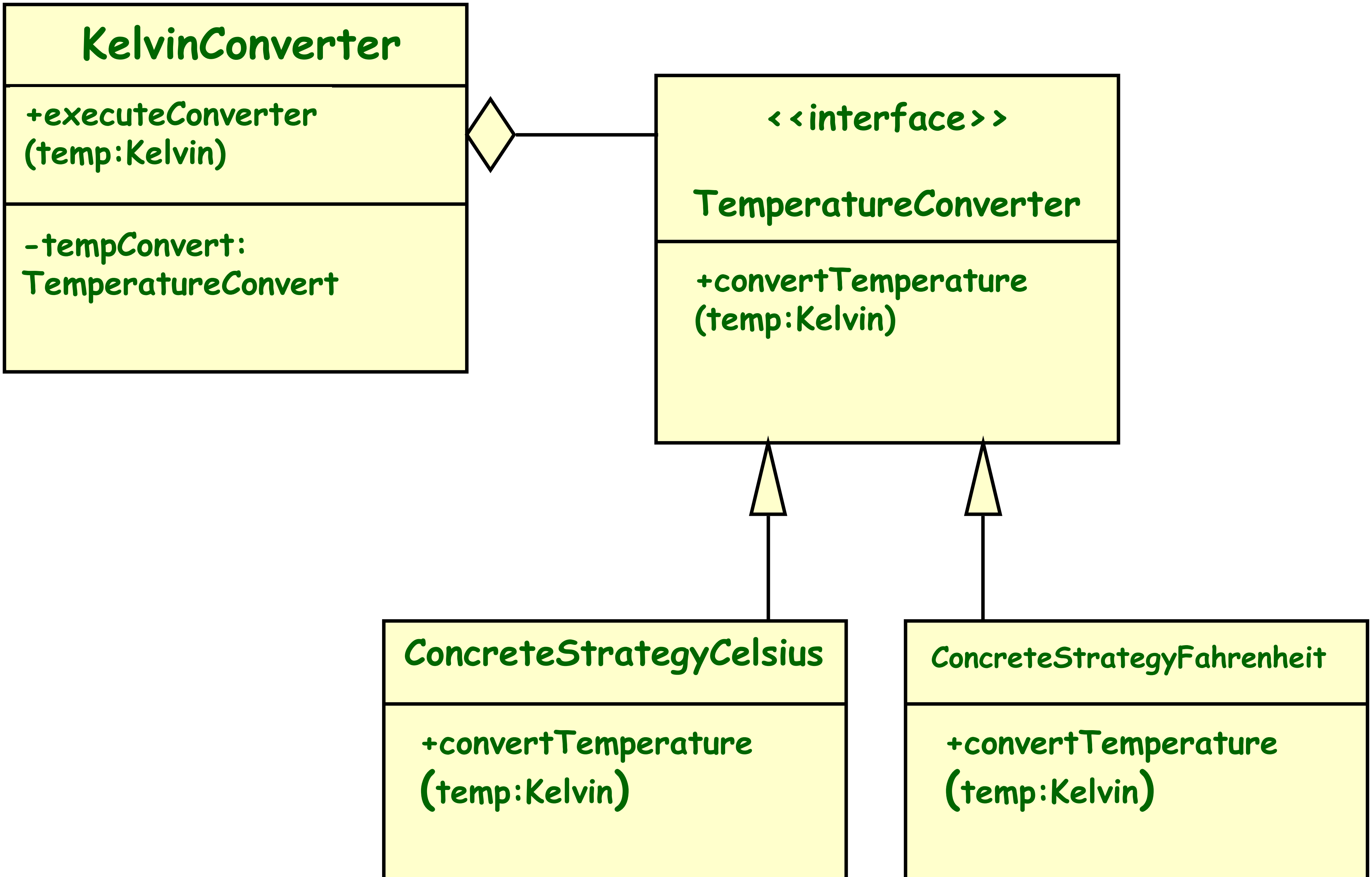
- Can define a family of algorithm in one place
- Can make a class hierarchy of algorithms
- Easy to replace one algorithm with another
- Can change dynamically
- Can encapsulate private data of algorithm

Simple Example: Temperature Converter

- A temperature sensor generates temperature reading in Kelvin.

- Two displays are required
  - One in Celsius
  - Other in Fahrenheit

Simple Example: Solution



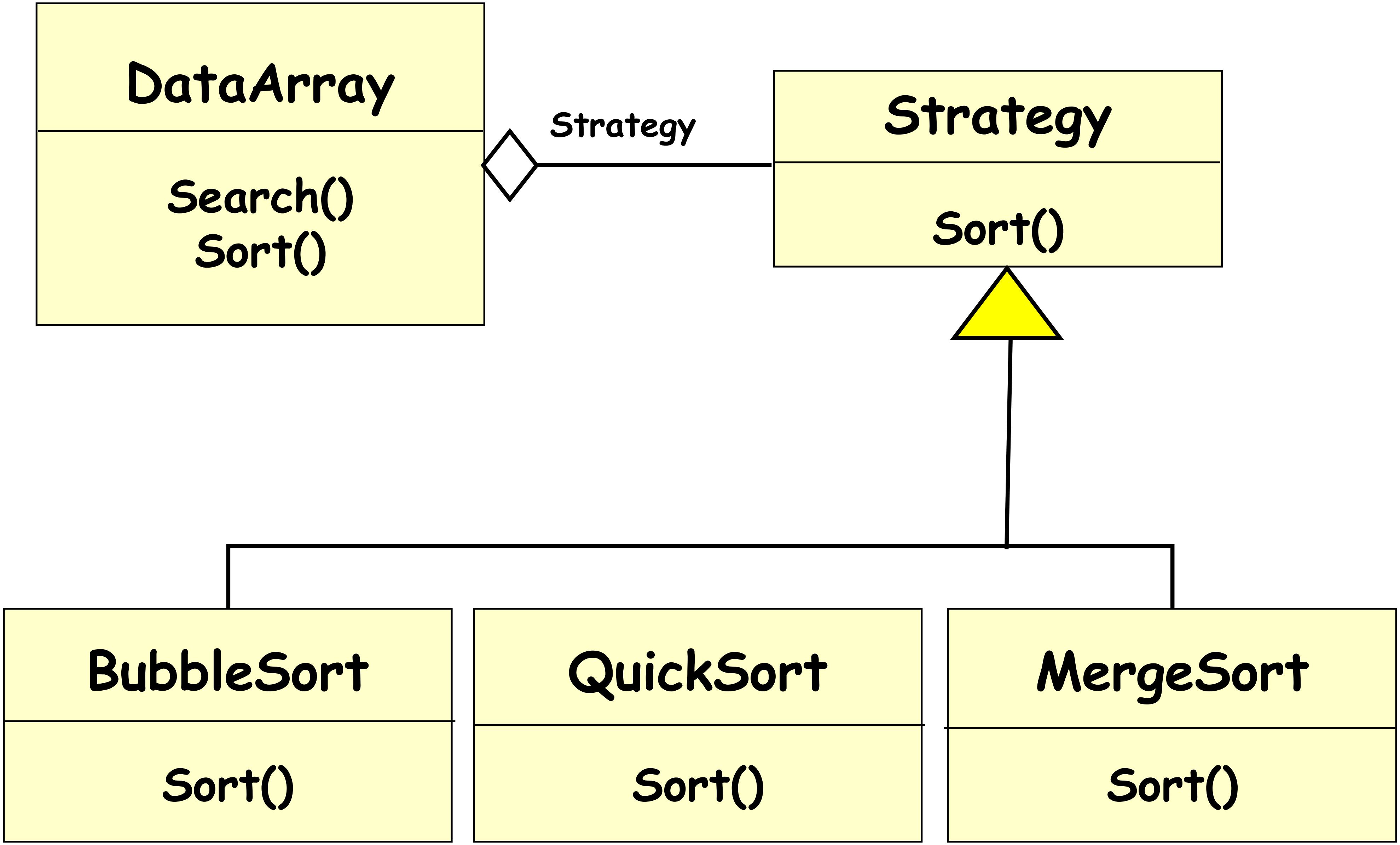
```
class KelvinConverter {
      private TemperatureConverter tempConverter;
      public KelvinConverter(
           TemperatureConverter tempConverter) {
              this.tempConverter = tempConverter;
      }
      public double executeConverter(double temp) {
    return tempConverter.convertTemperature(temp);
      }
}

  interface TemperatureConverter {
        double ConvertTemperature(double temp);
  }
```

```
class ConcreteStrategyCelsius implements
                        TemperatureConverter {

        public double convertTemperature(double temp) {
                return temp - 273.15;
        }
}

class ConcreteStrategyFahrenheit
                    implements TemperatureConverter {

        public double convertTemperature(double temp) {
                return ((temp - 273) * 1.8 ) + 32;
        }
}
```

```
public StrategyTest(){
        KelvinConverter kelConvert;
        double testTemp = 273.00;
        kelConvert = new KelvinConverter(new
ConcreteStrategyCelsius());
        double celsiusResult =
kelConvert.executeConverter(testTemp);
        System.out.println(celsiusResult);
        kelConvert = new KelvinConverter(new
ConcreteStrategyFahrenheit());
        double fahrenheitResult =
kelConvert.executeConverter(testTemp);
        System.out.println(fahrenheitResult);

   }

   public static void main(String[] args) {
        new StrategyTest();
   }
```

Strategy Pattern: Exercise 1

- You have an array of items:
  - At run-time you want to decide which sorting algorithm to use.
  - Bubble sort, quick sort, or merge sort
- **Solution:**
  - Encapsulate each different sorting algorithm using the strategy pattern.

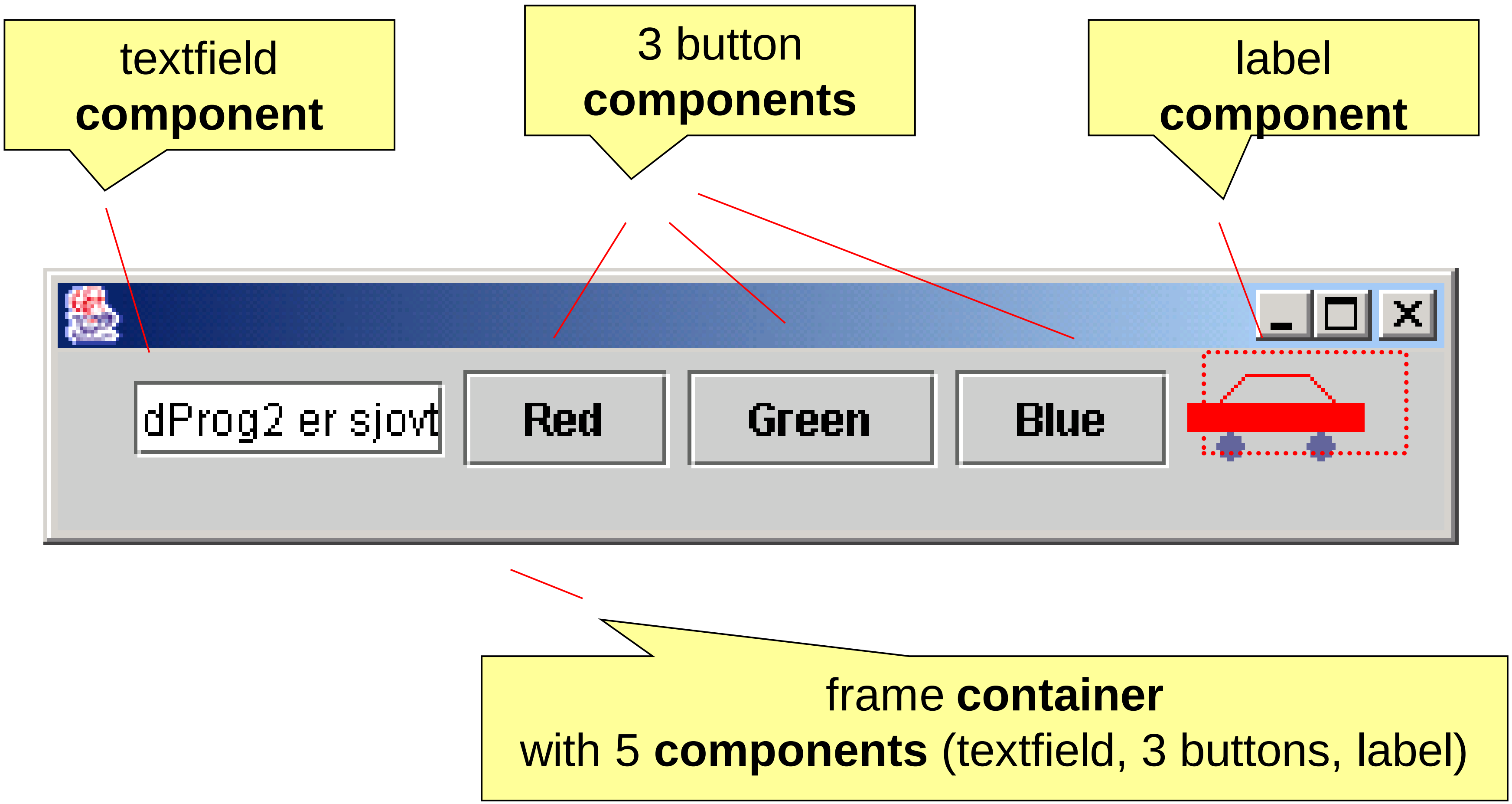

DataArray
Search()
Sort()
Strategy
Strategy
Sort()
BubbleSort
Sort()
QuickSort
Sort()
MergeSort
Sort()

Exercise 2

- Many different layout strategies exist.
  - Flow layout, Border layout, card layout
  - A GUI container wants to decide at run-time which layout to use

- Encapsulate each different layout algorithm using the strategy pattern.

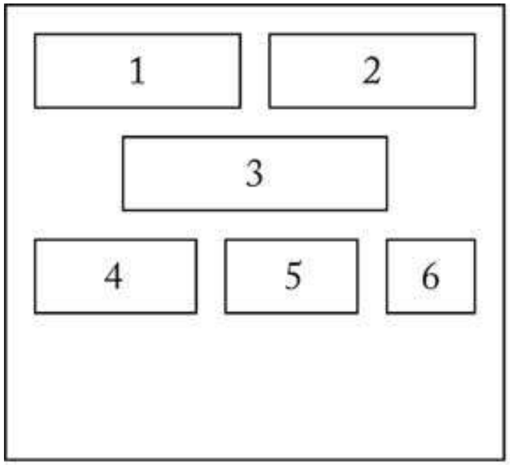
Explanation: GUI components and containers



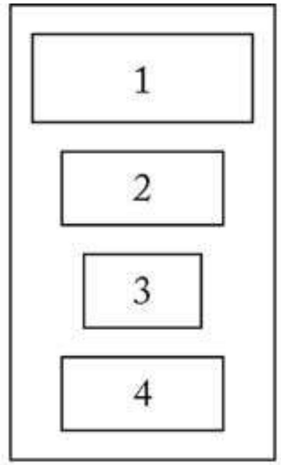
Layout Managers

- User interfaces made up of *components*
- Components placed in *containers*
- Container needs to arrange components
- Swing doesn't use hard-coded pixel coordinates
- Advantages:
  - Can switch "look and feel"
  - Can internationalize strings
- Layout manager controls arrangement

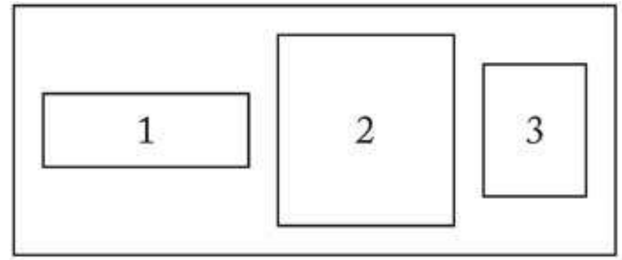
Layout Manager: Options



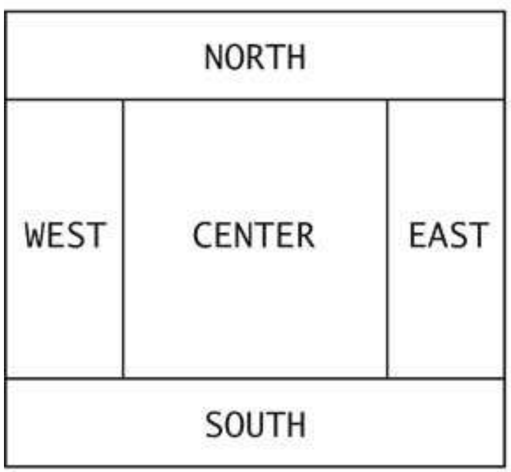
FlowLayout



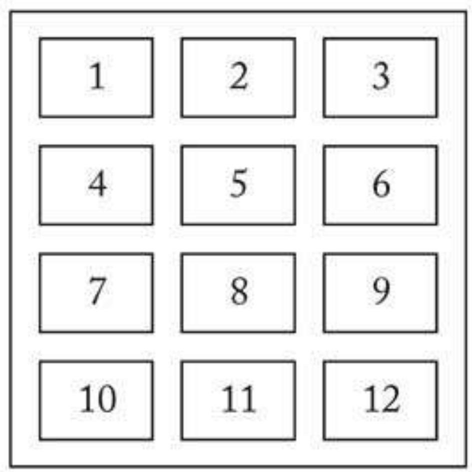
BoxLayout (vertical)



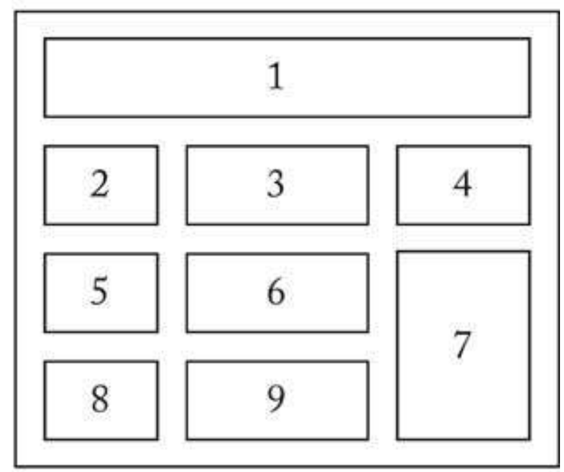
BoxLayout (horizontal)



BorderLayout

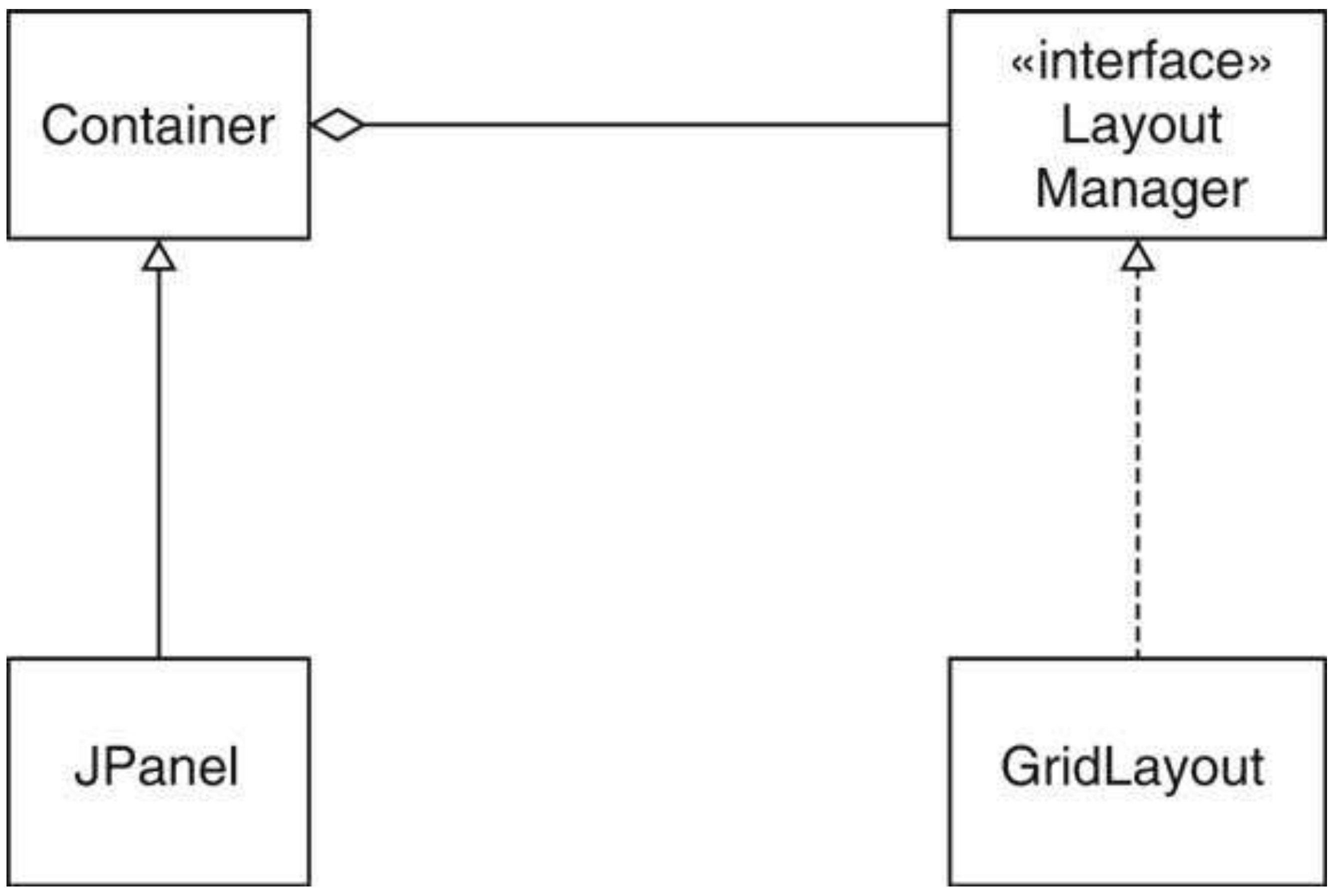


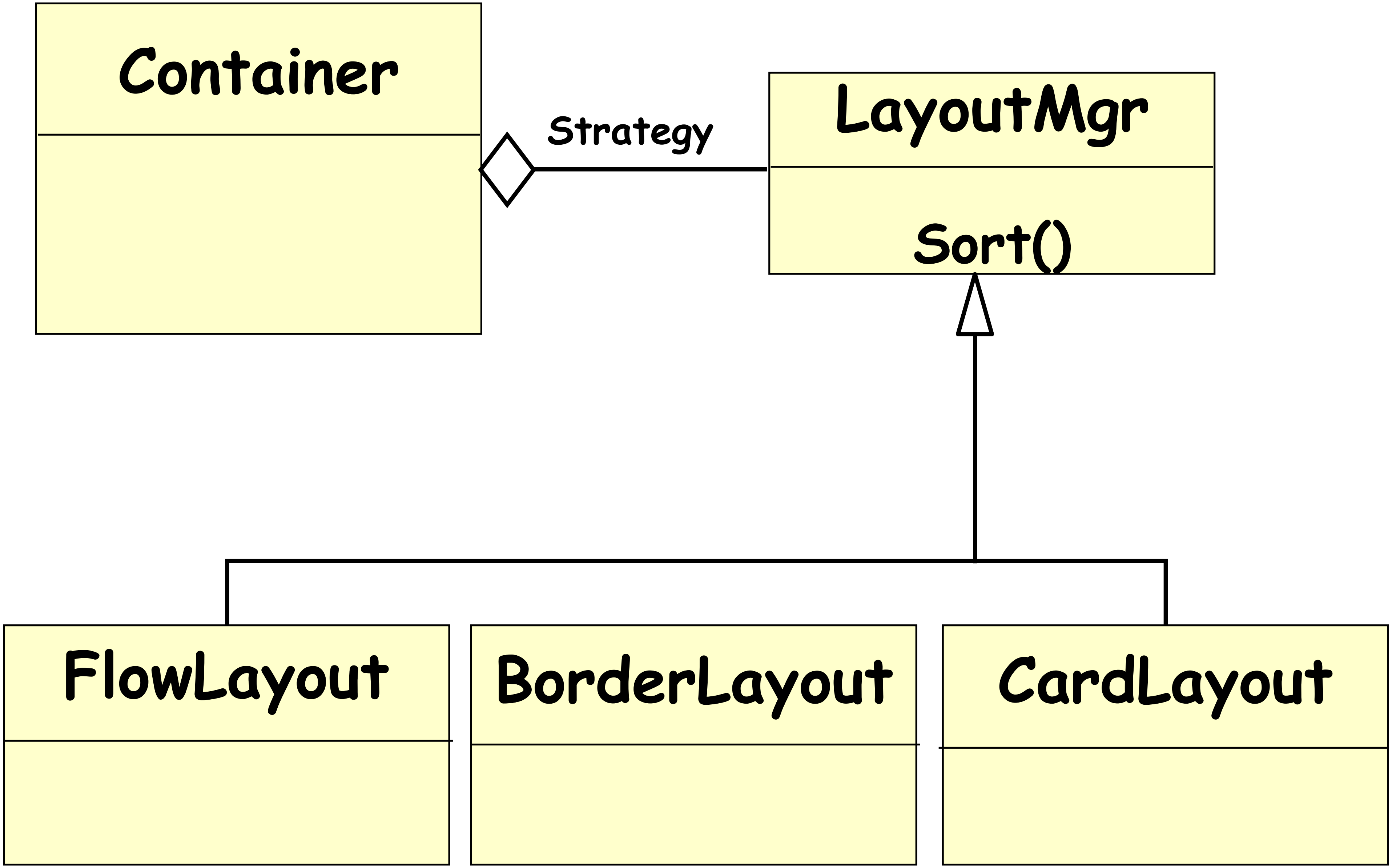
GridLayout

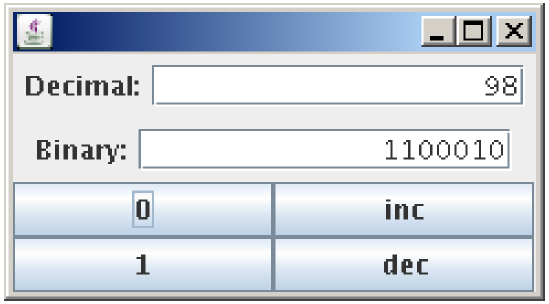


GridBagLayout

Layout Managers




Container
Strategy
LayoutMgr
Sort()
FlowLayout
BorderLayout
CardLayout

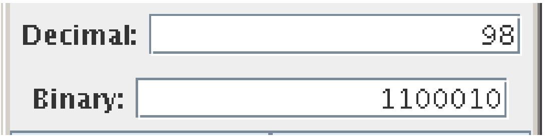


Example

```
JPanel decDisplay = new JPanel();
final JTextField digits = new JTextField(“98",15);
decDisplay.add(new JLabel("Decimal:"));
decDisplay.add(digits);
```

`JPanel` Default : `FlowLayout`

```
JPanel display = new JPanel();
display.setLayout(new BorderLayout());
display.add(decDisplay, BorderLayout.NORTH);
display.add(binDisplay, BorderLayout.SOUTH);
```

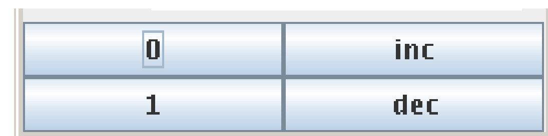


`BorderLayout`

Example

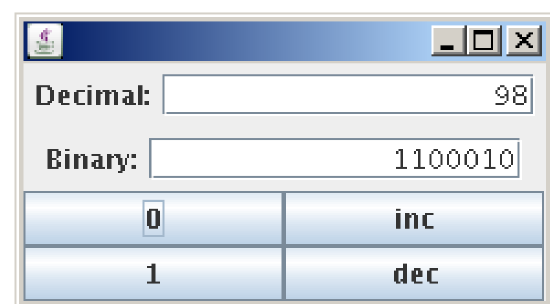
```
JPanel keyboard = new JPanel();
keyboard.setLayout(new GridLayout(2,2));
keyboard.add(new JButton("0"));
keyboard.add(new JButton("inc"));
keyboard.add(new JButton("1"));
keyboard.add(new JButton("dec"));
```

GridLayout



JFrame Default : BorderLayout

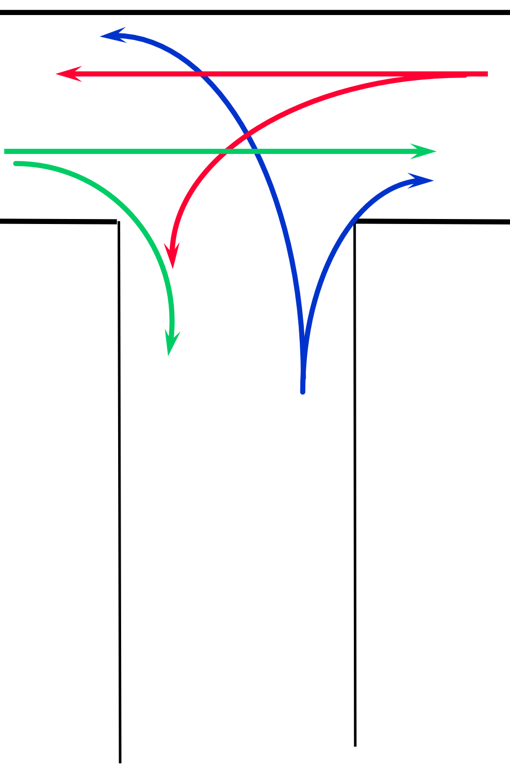
```
JFrame f = new JFrame();
f.add(display,BorderLayout.NORTH);
f.add(keyboard,BorderLayout.CENTER);
```



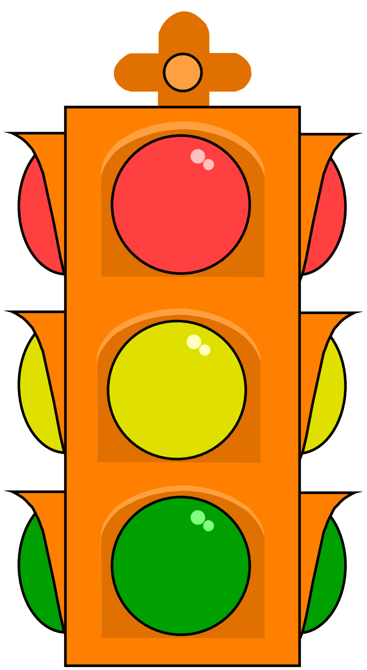
Applicability

- Use when many different algorithms exist for essentially the same task.
- Some Examples:
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- Different algorithms will be appropriate in different situations
  - We don't want to support all the algorithms if we don't need them
- If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm

Exercise 3:
Intersection Traffic Lights Control



The light-switching policy changes by the hour



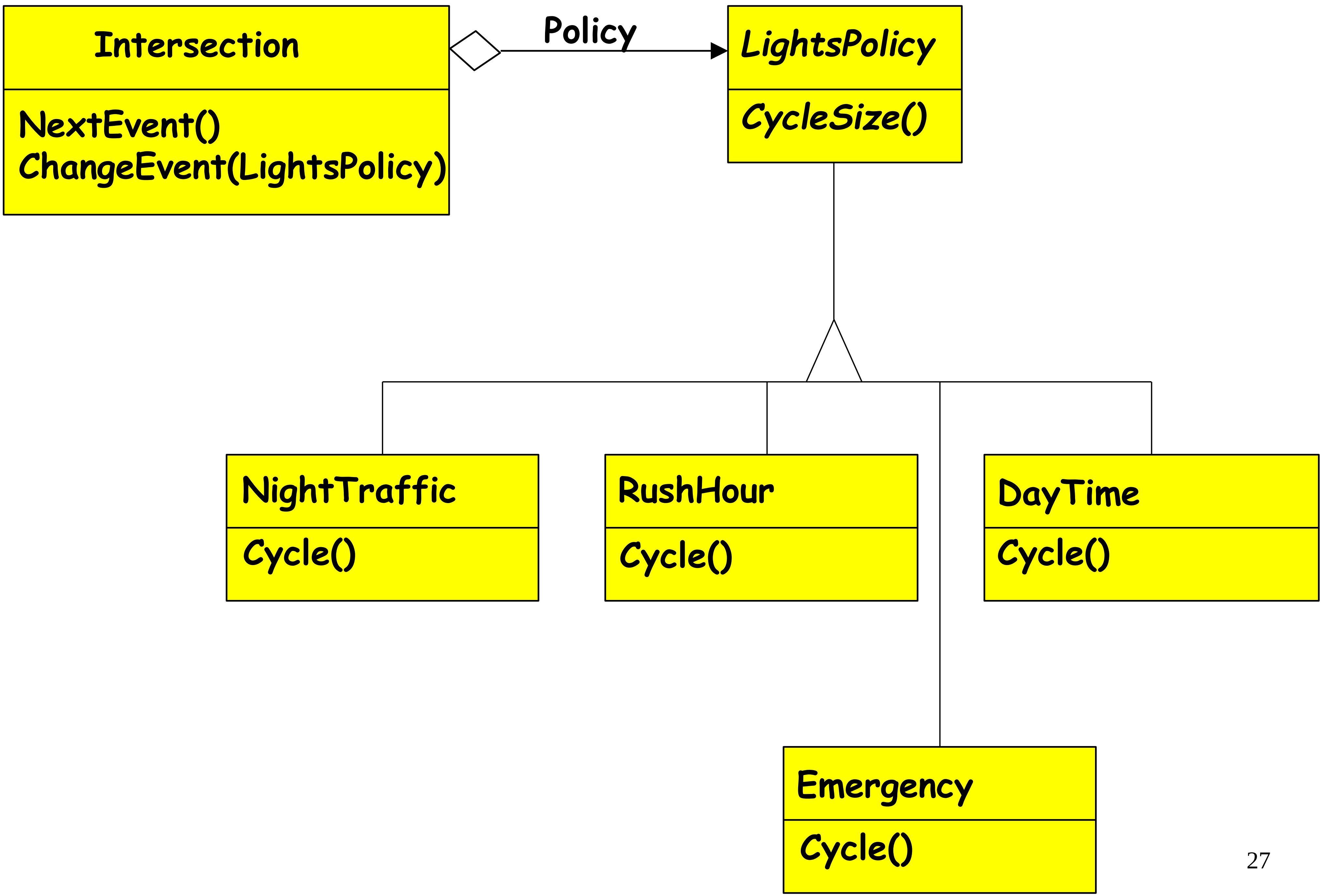
25

Traffic Light Behavior Varies

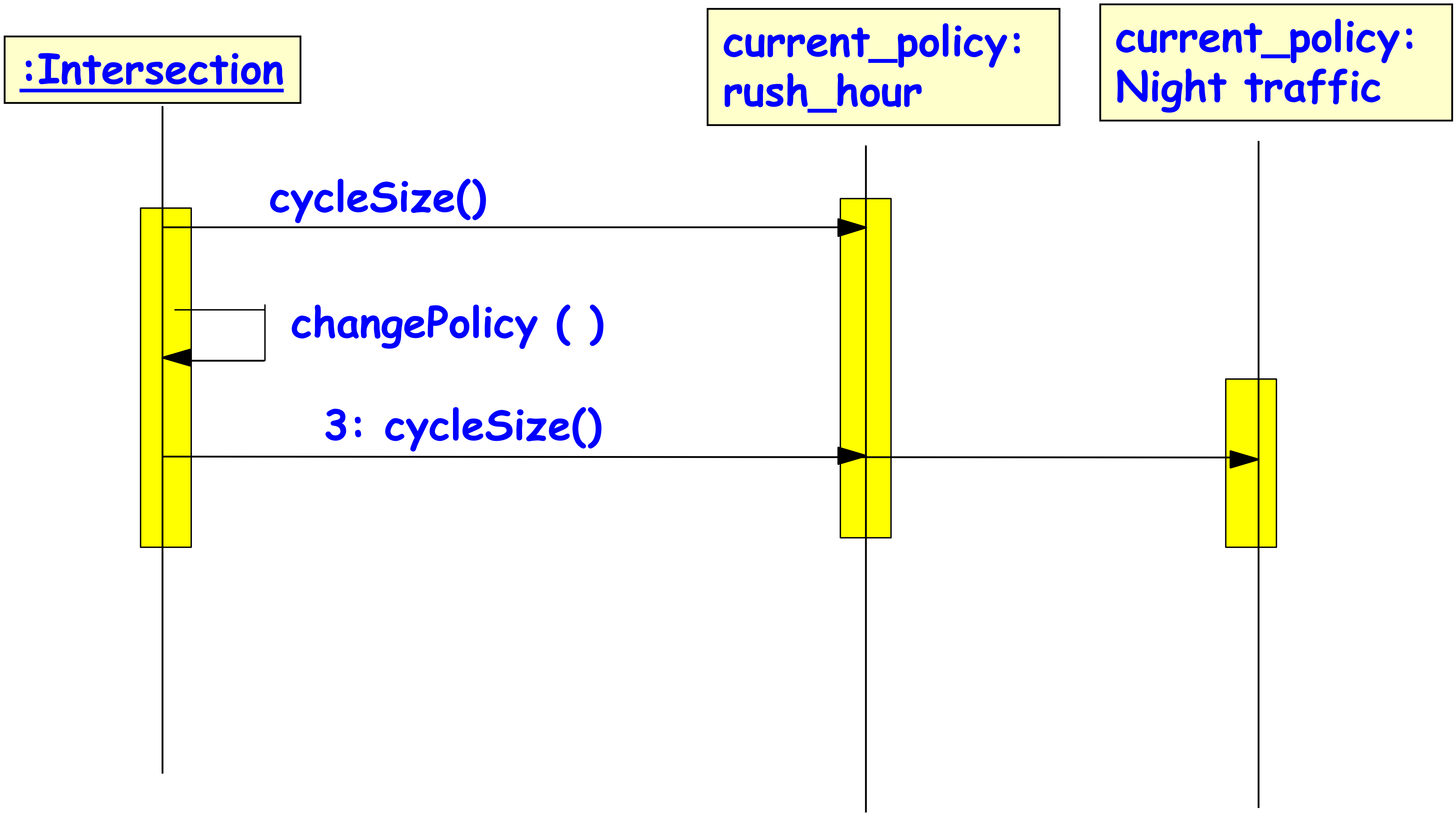
- The "dumb" policy:
  - Change the green route every 5 seconds
- Midnight policy:
  - Change to green whenever a "sensor" detects a vehicle
- Rush hour policy:
  - Double the "green time" in the busy route

26

Traffic Lights Management



Sequence Diagram

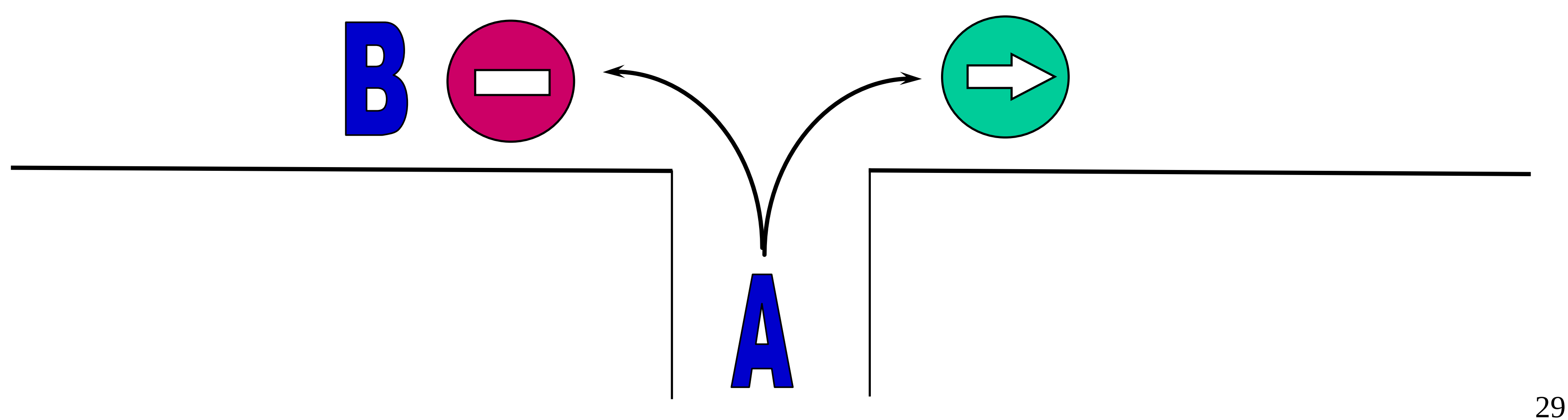


Strategy: Consequences

- Easy to add new strategy or remove an existing one,

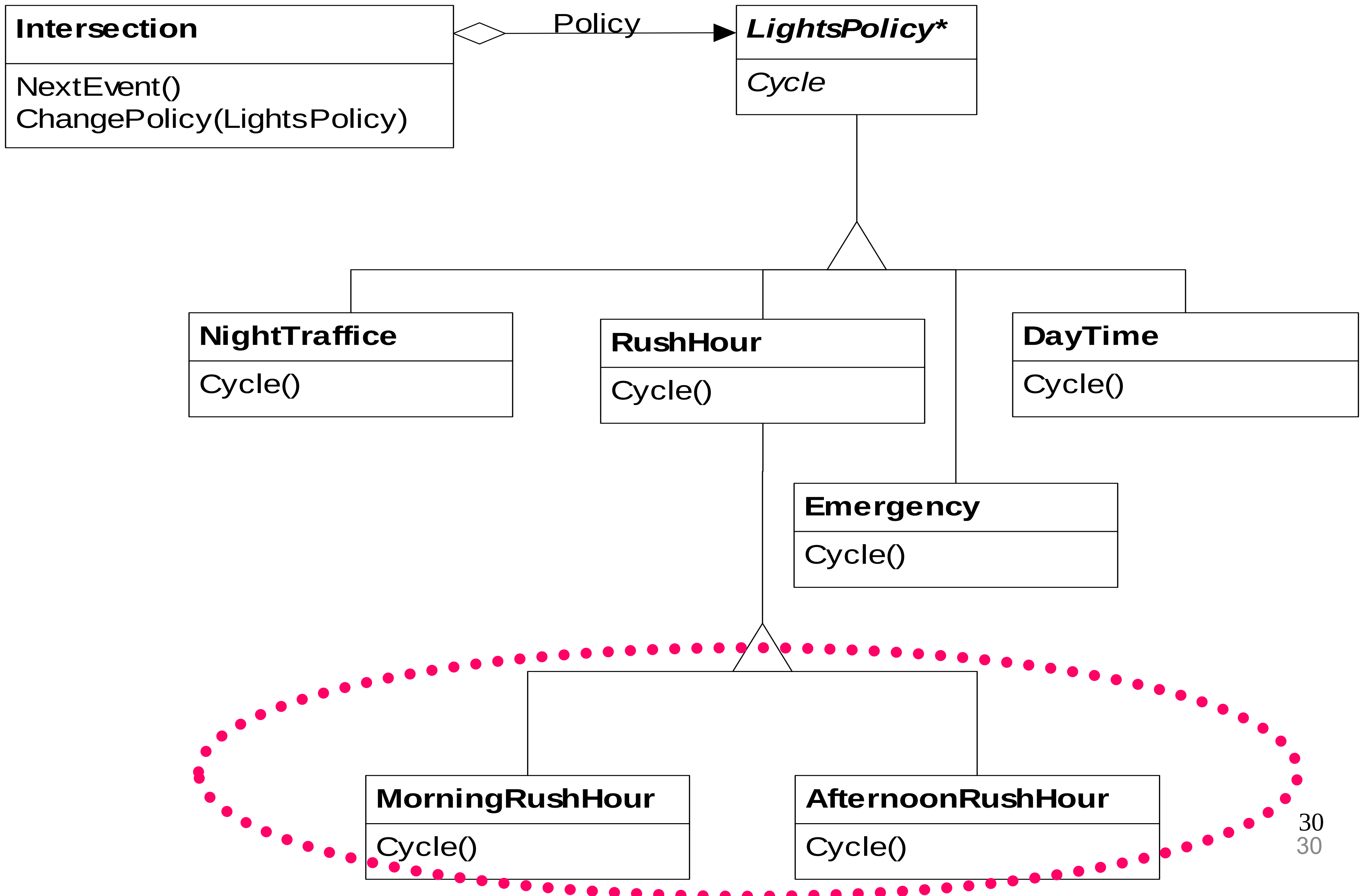
For instance: New policy

From 8 a.m. to 10 a.m. there is no turn left from A onto B



Strategy: Consequences

- Easy to factor out similar strategies using inheritance



30

Inheritance vs. Strategy Solution

- **Subclassing:**
  - Mixes the algorithm implementation with context's, making Context harder to understand, maintain, and extend. Can not vary the algorithm dynamically.
- **Strategy:**
  - Vary the algorithm independently of its context, making it easier to switch, understand, and extend.

Trade-offs

Advantages

- Eliminates large conditional statements.
- Easier to keep track of different behavior because they are in different classes.
- A variety of implementations for the same behavior.

Disadvantages

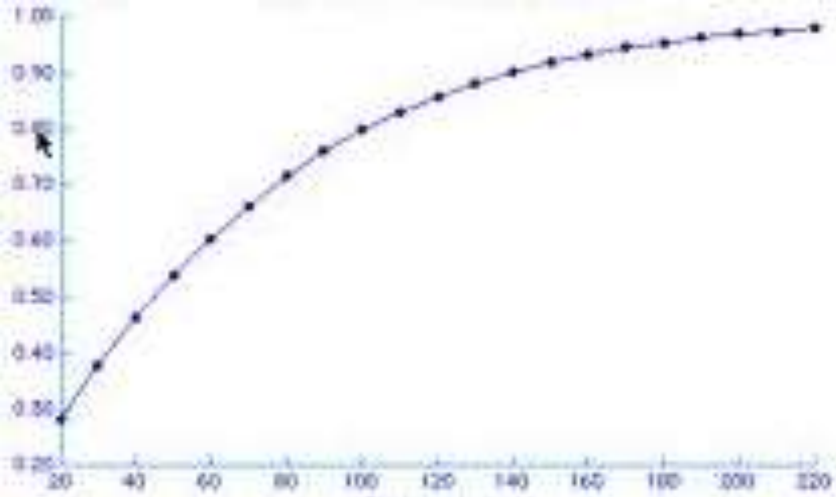
- Increases the number of objects.
- All algorithms use the same interface.

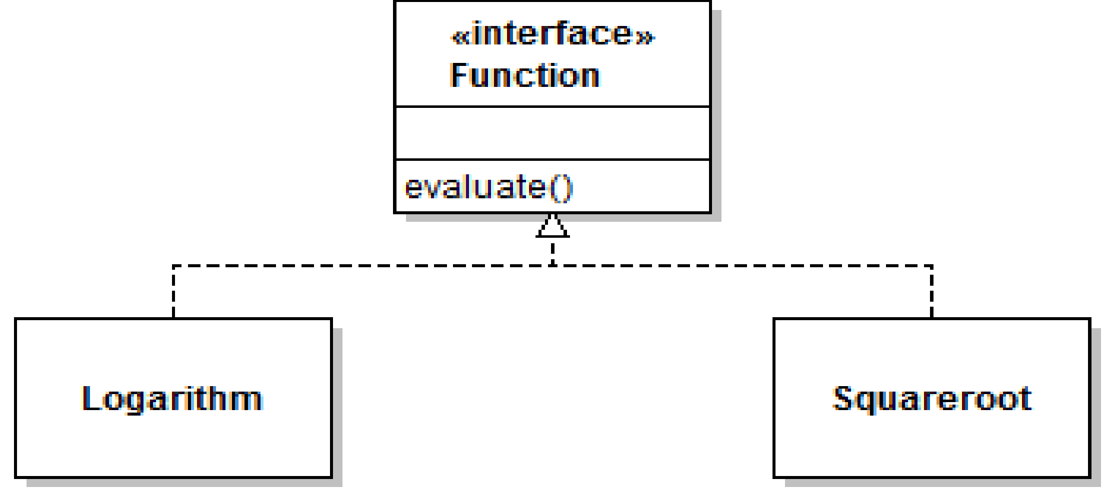
Summary of Strategy Pattern

- Many related classes differ in behavior
- Need to use the same algorithm with a slight variation
- Hides complex, algorithm-specific data structures from the client
- Eliminate conditional branches and put them in their own separate strategy class

Home Work

- There are a lot of similarities between state and strategy patterns.
  - What is the difference between the two?





QUIZ

Which versions of Graph class use **strategy** pattern?

1. None
2. A
3. B
4. A,B
5. I don't know

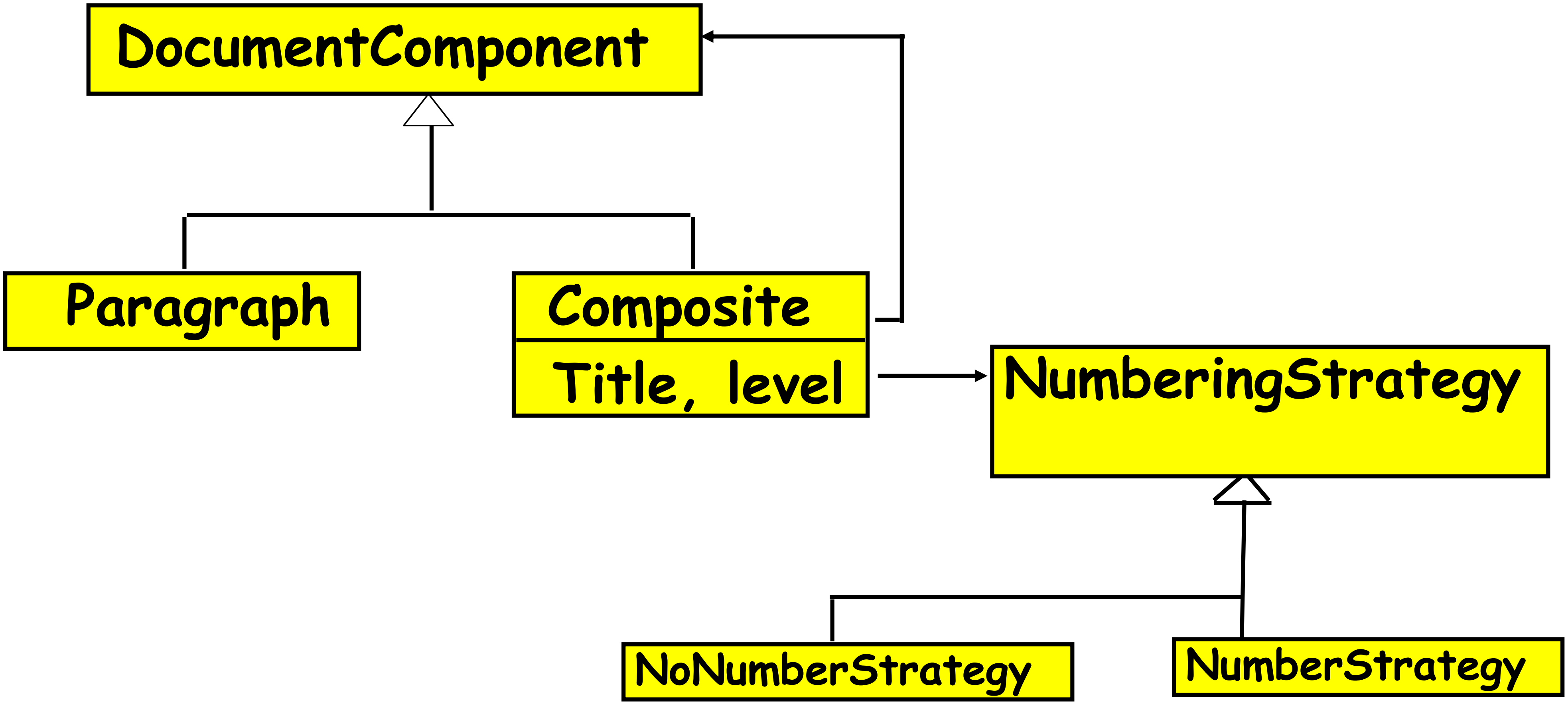
A

```
public class Graph {
  private Function f = new Logarithm();
  public draw()
    { plot(1,f.evaluate(1)); … }
}
```

B

```
public class Graph {
  private Function f;
  public Graph(Function fun) { f=fun; }
  public draw()
    { plot(1,f.evaluate(1)); … }
}
```

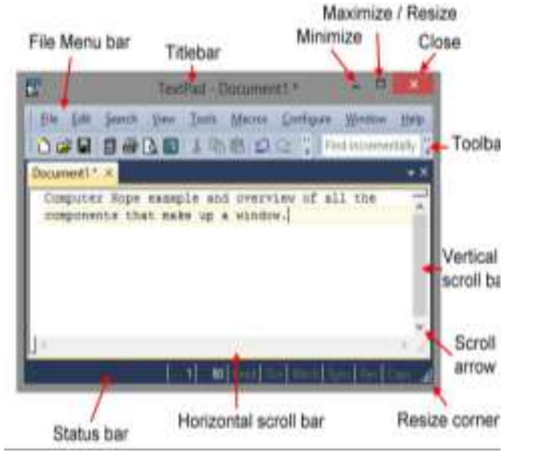
Final Example: Design with Strategy



Command Pattern

Introduction

- **Sometimes a class needs to perform actions without knowing what the actions are…**



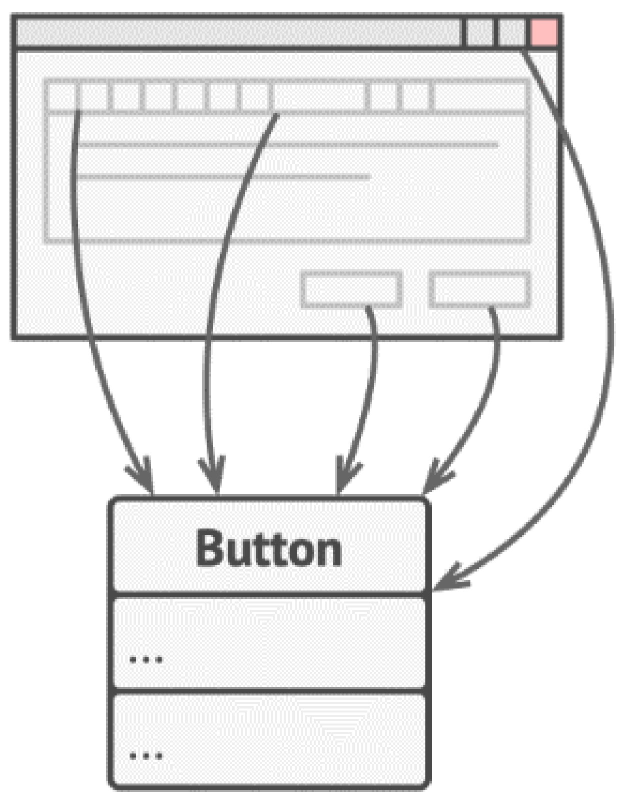
- **Example:**
  - A GUI toolkit provides several components: Buttons, scroll bars, text boxes, menus, etc.

- Toolkit components only know apriori know how to draw themselves on the screen:
  - But they don't know how to perform application logic

- Application developers need a way to associate required application logic with GUI components
  - What should happen when a button is pressed?
  - What should happen when a menu item is selected?

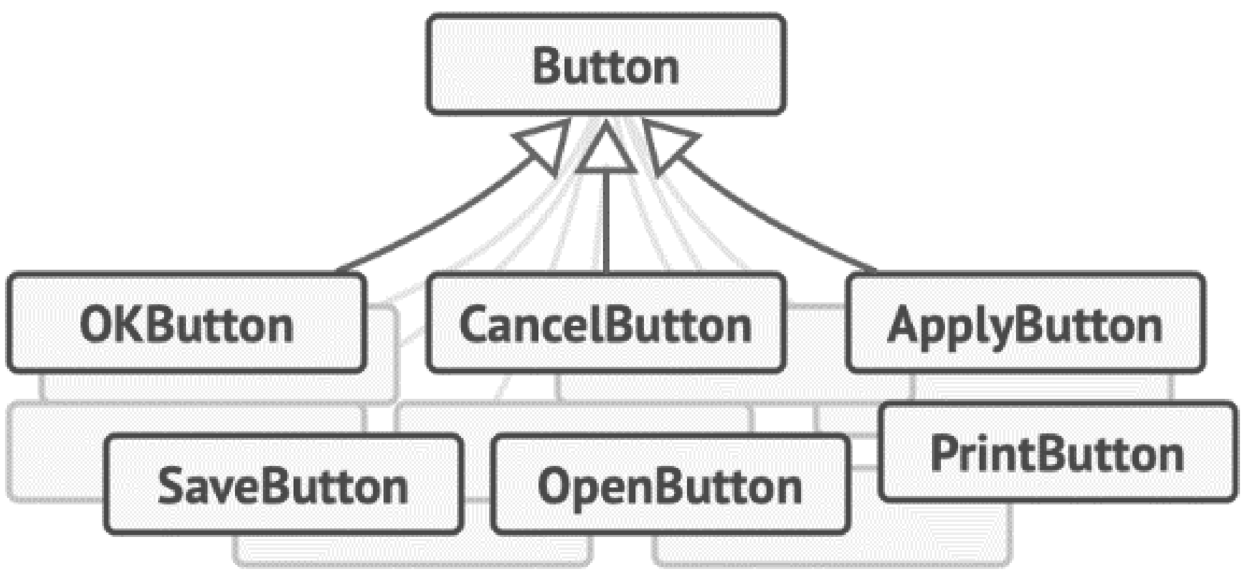
Motivation

- Suppose you're working on a new text-editor app.
- You created a very neat Button class:
  - To be used as generic buttons in various dialogs.



First-Cut Design

- But, generic buttons won't do much:
  - But, each button needs to carry out separate actions.
- You toyed with the idea of creating hundreds of sub-classes of the button class.



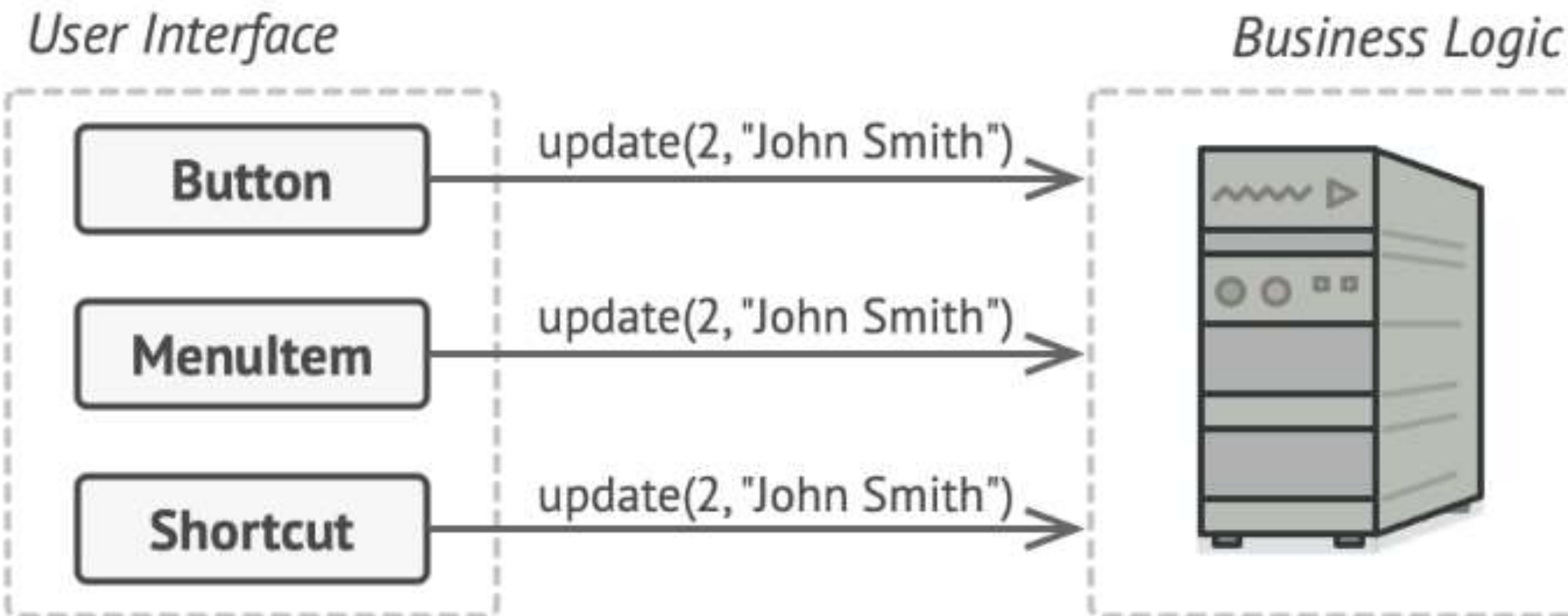
First-Cut Design --- Cons

- You have an enormous number of subclasses --- Poses Maintenance issues

- You break the design each time you modify a base class

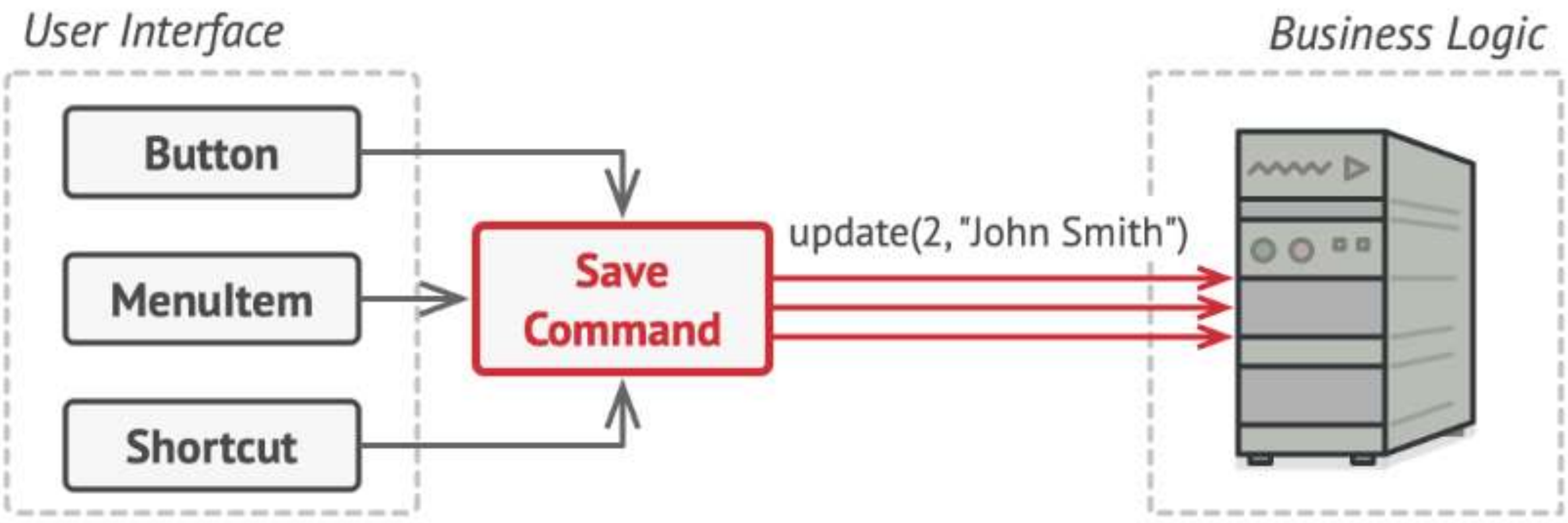
- You may have different methods for invoking the same command, e.g hot keys
  - Either duplicate code or make hot-keys dependent on buttons

Refined Design



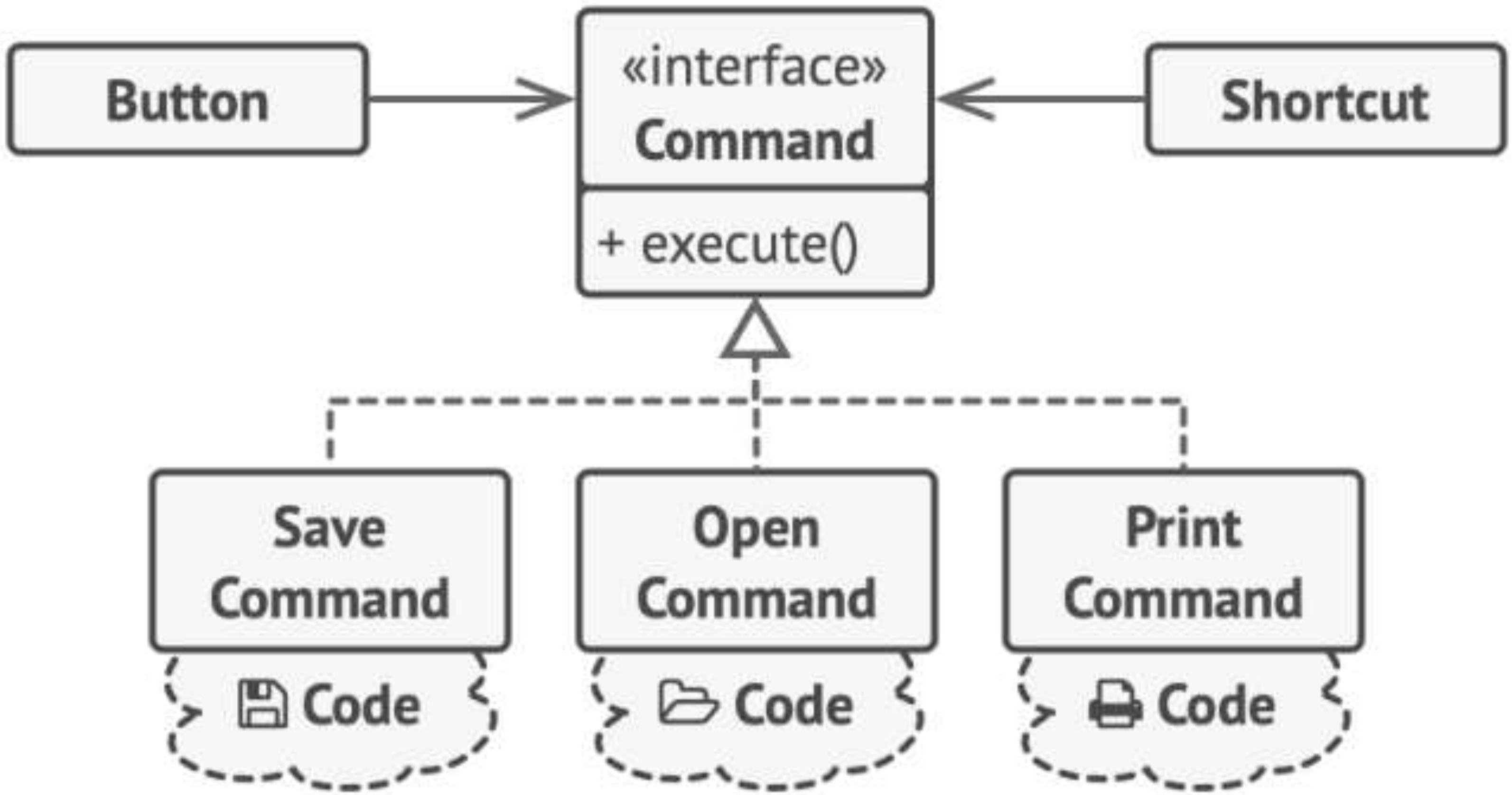
- GUI object calls a method of a business logic object, passing it some arguments.

Next Refinement

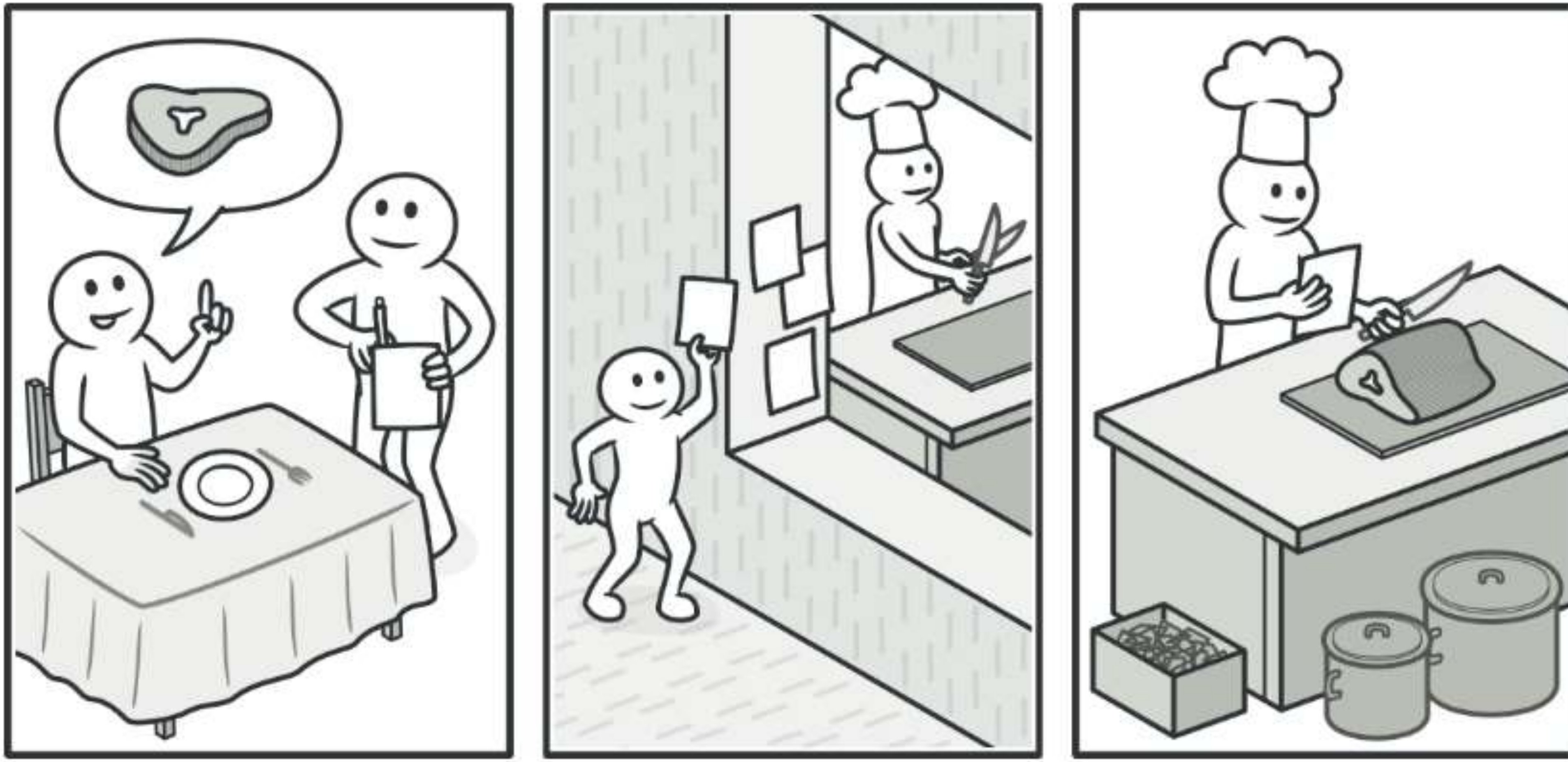


- But, you next recognize that all commands need to implement the same interface…

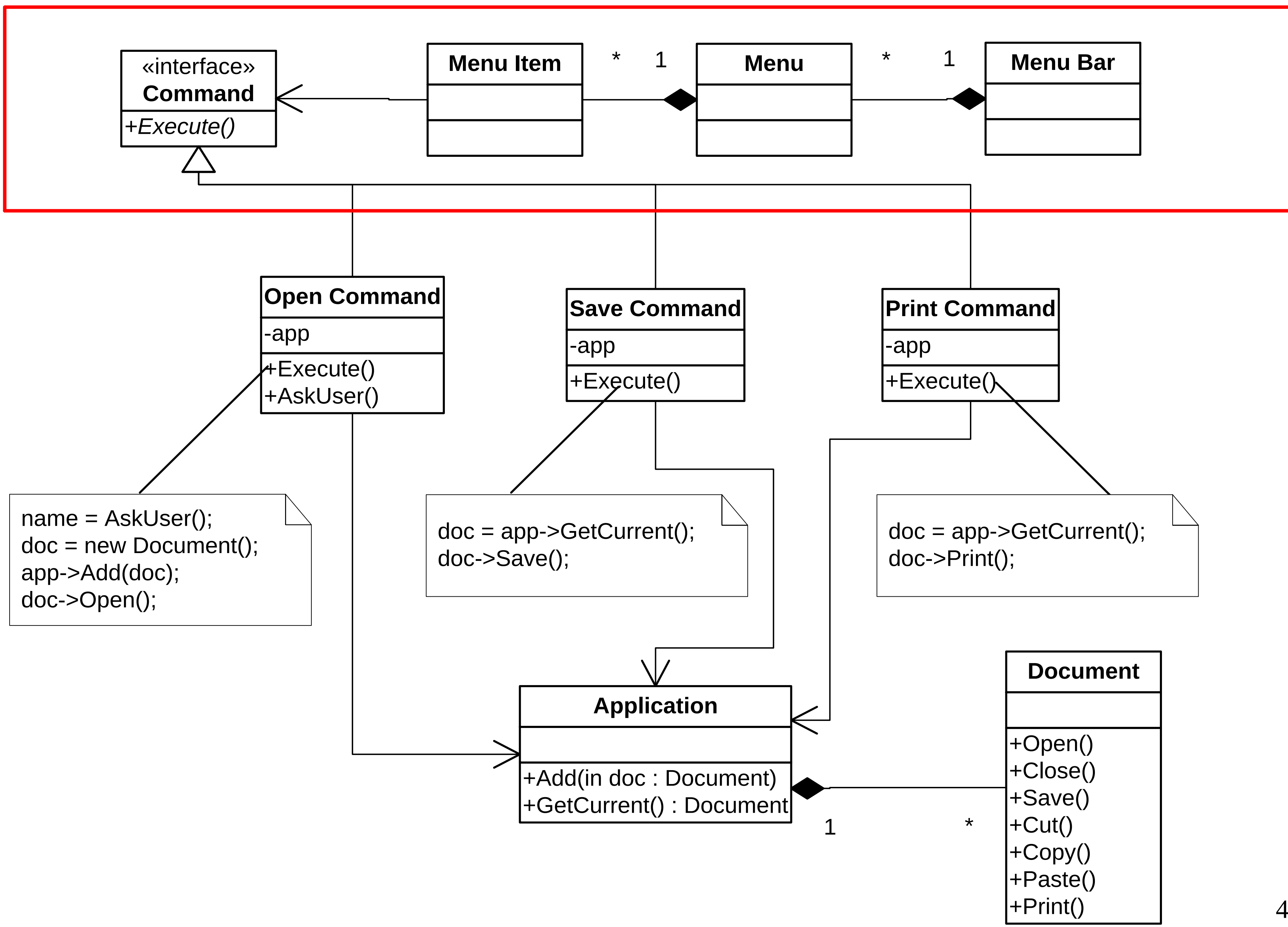
Command Pattern: GUI Objects Delegate Work to Command Objects



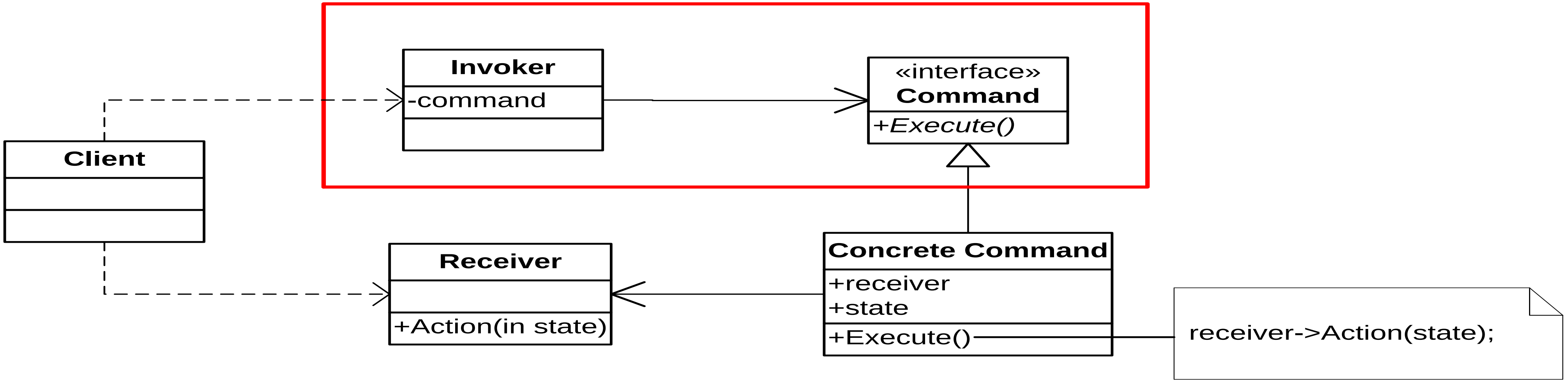
Real World Example

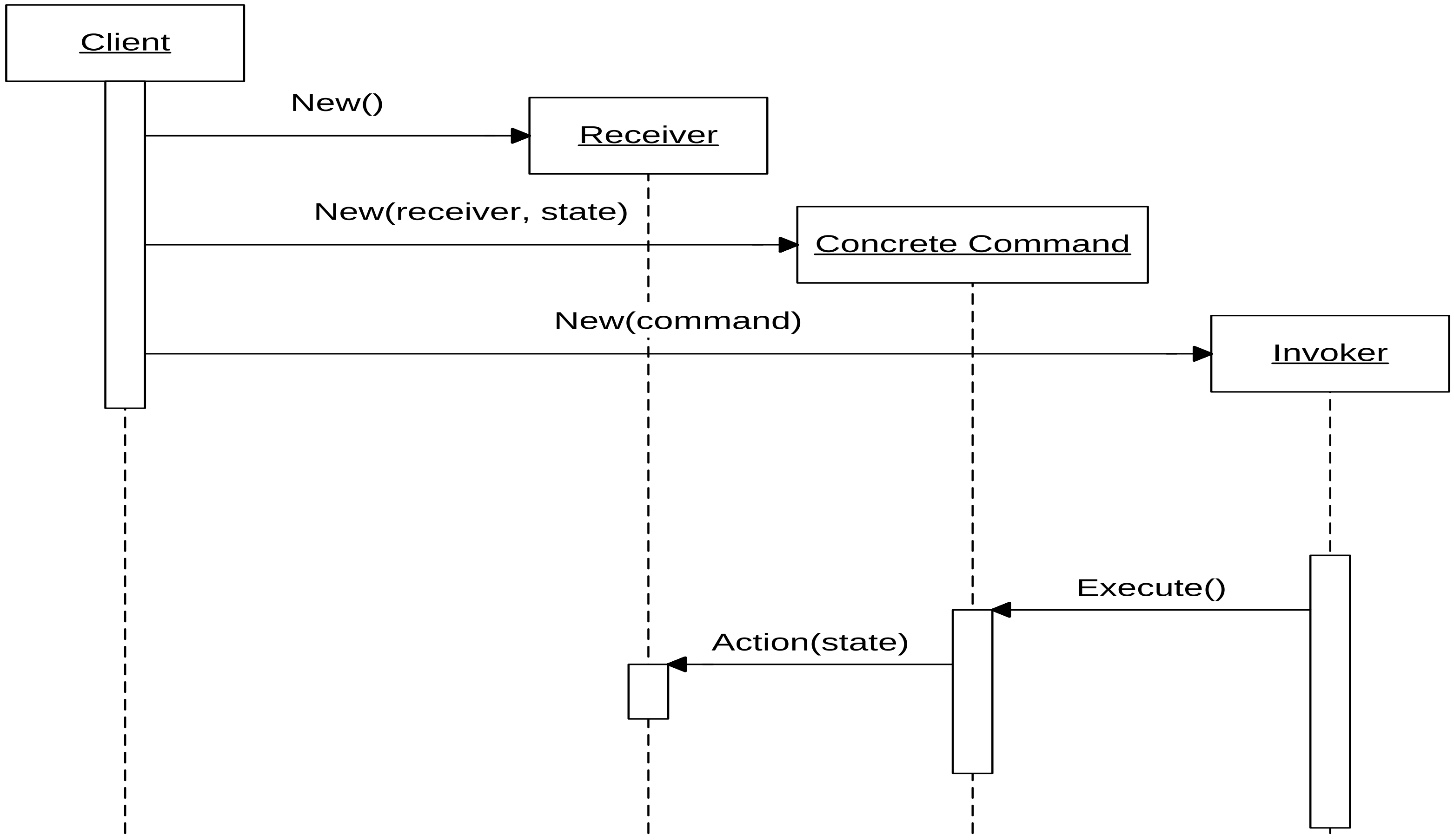


Example: GUI Toolkit



Example: GUI Toolkit





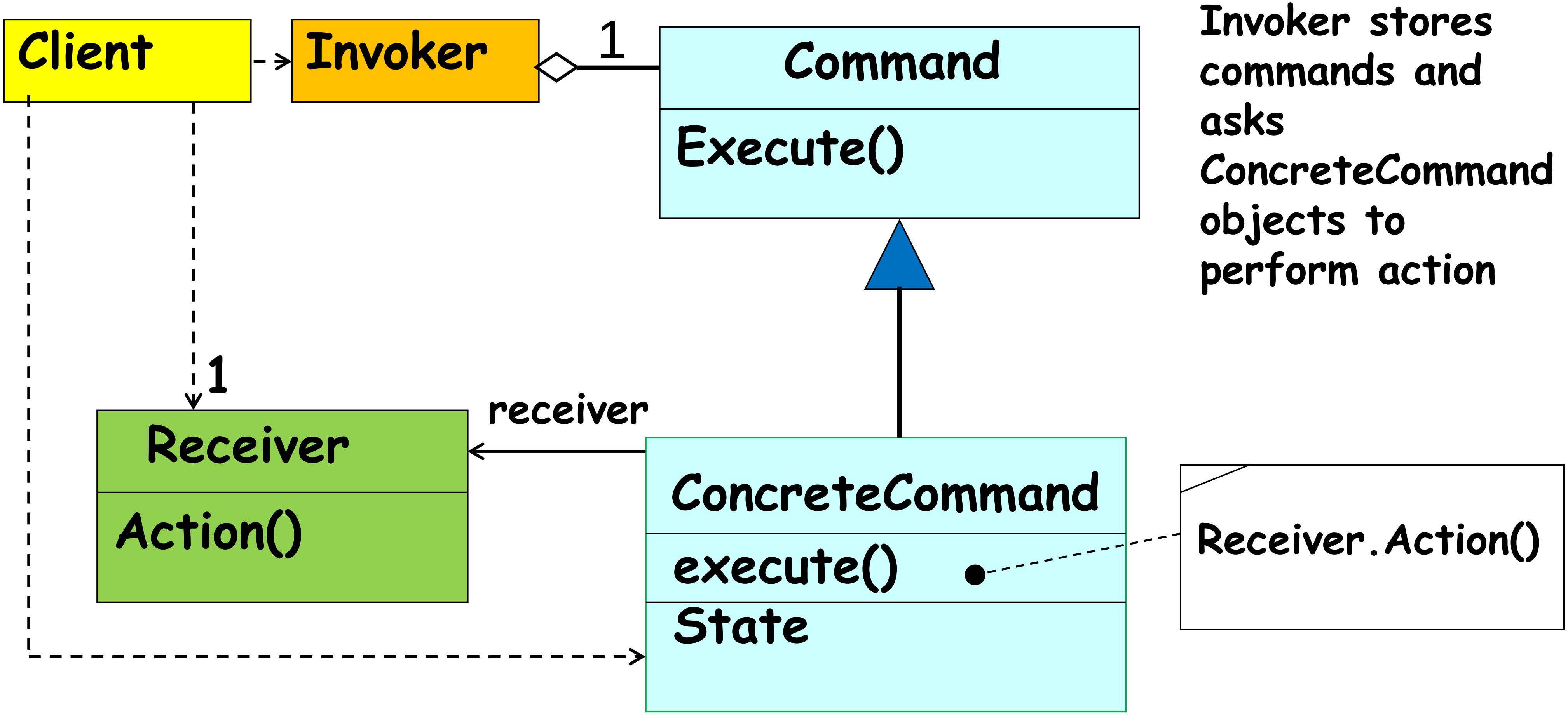
Other Applications of Command

- **Support Undo:**
  - It's difficult to undo effects of an arbitrary method as Methods vary over time.

- What are some applications that need to support undo?
  - Editor, calculator, database with transactions. etc

- **Support Redo:**
  - Similar complex issues

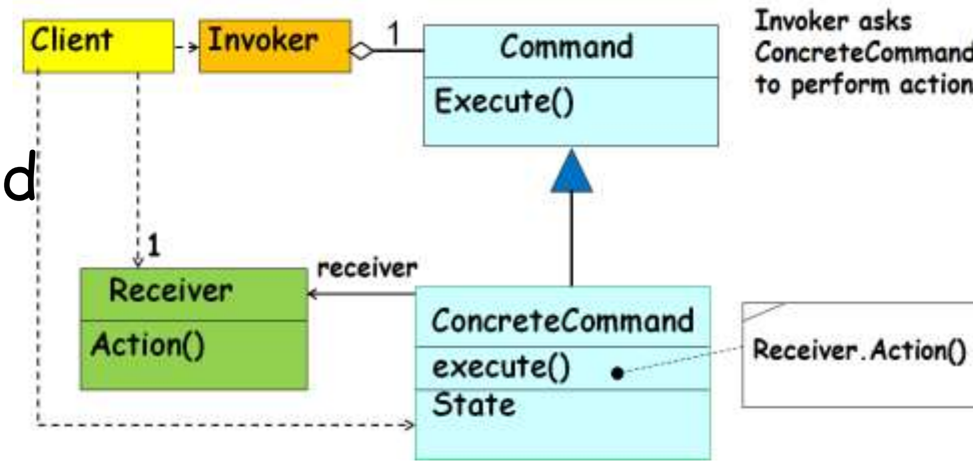
Structure of command pattern



Invoker stores commands and asks ConcreteCommand objects to perform action

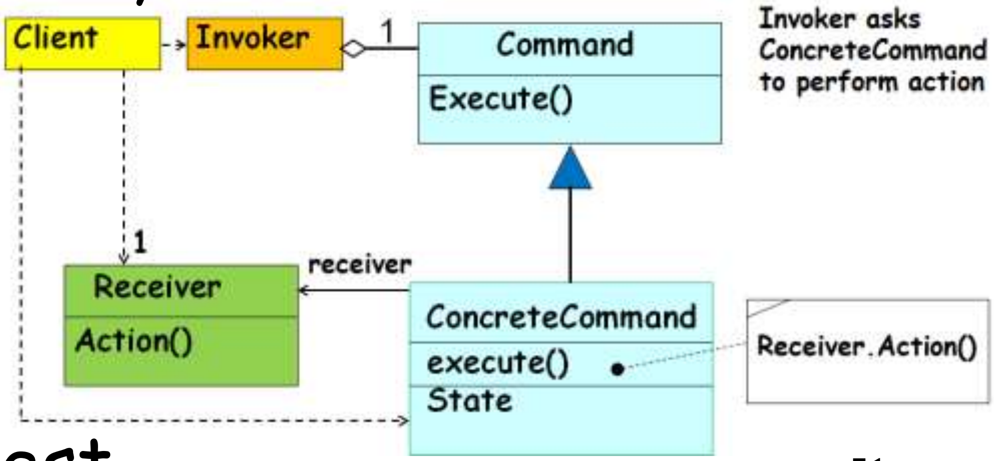
Command pattern: Participants

- **Command (Command)** declares an interface for executing an operation
- **ConcreteCommand** defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
- **Invoker** asks the command to carry out the request
- **Receiver** knows how to perform operations associated with carrying out the request
- **Client** creates a ConcreteCommand object and sets its receiver

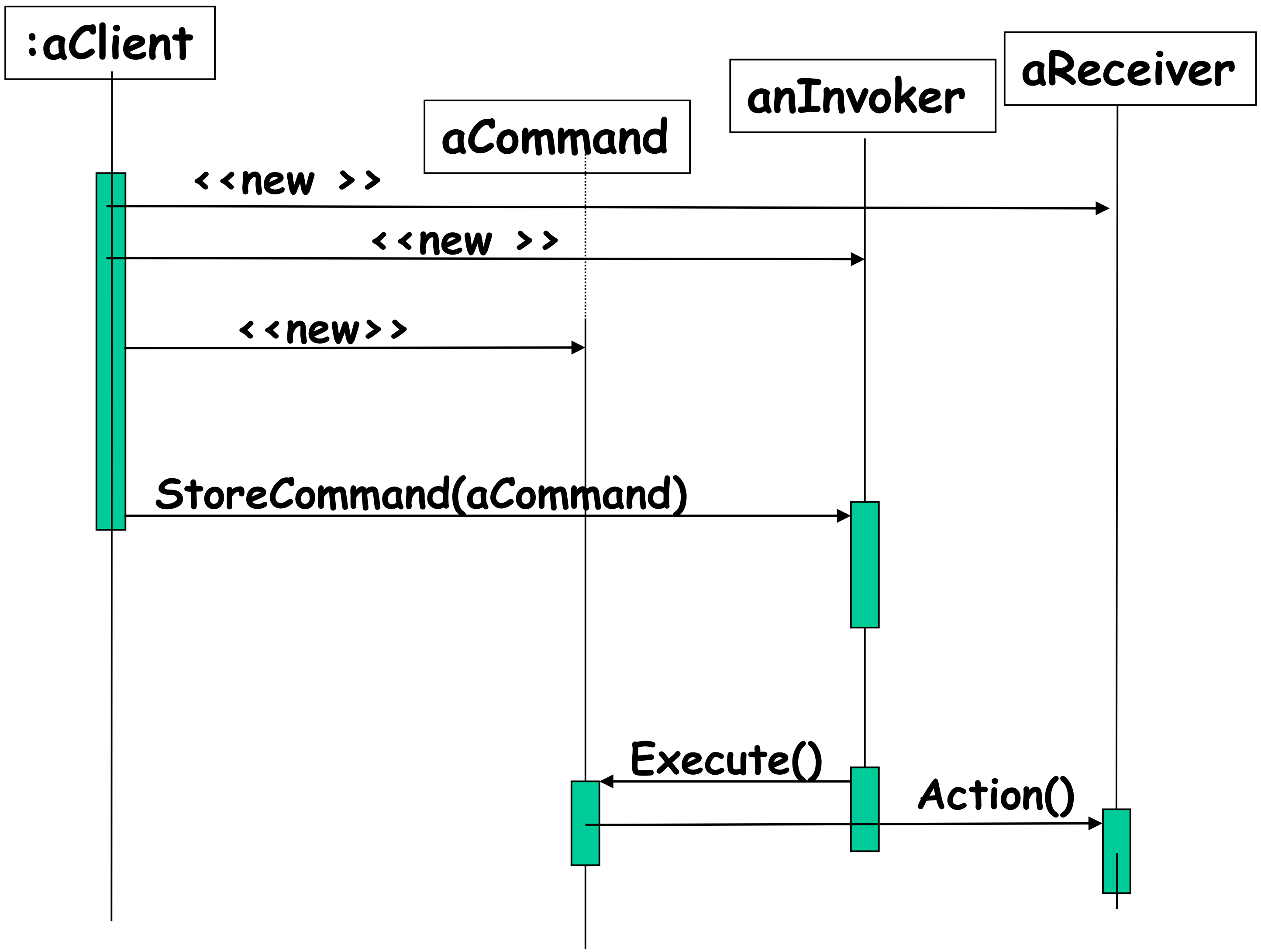


Command pattern: Operation

- Client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object
- The invoker issues a request by calling Execute on the command.
- When commands are undoable, ConcreteCommand stores state for undoing before invoking Execute
- ConcreteCommand object invokes operations on its receiver to carry out request



Sequence Diagram



Consequences

- Completely decouples objects from the actions they execute
- Objects can be parameterized with arbitrary actions
- Adding new kinds of actions is easy
  - Just create a new class that implements the Command interface

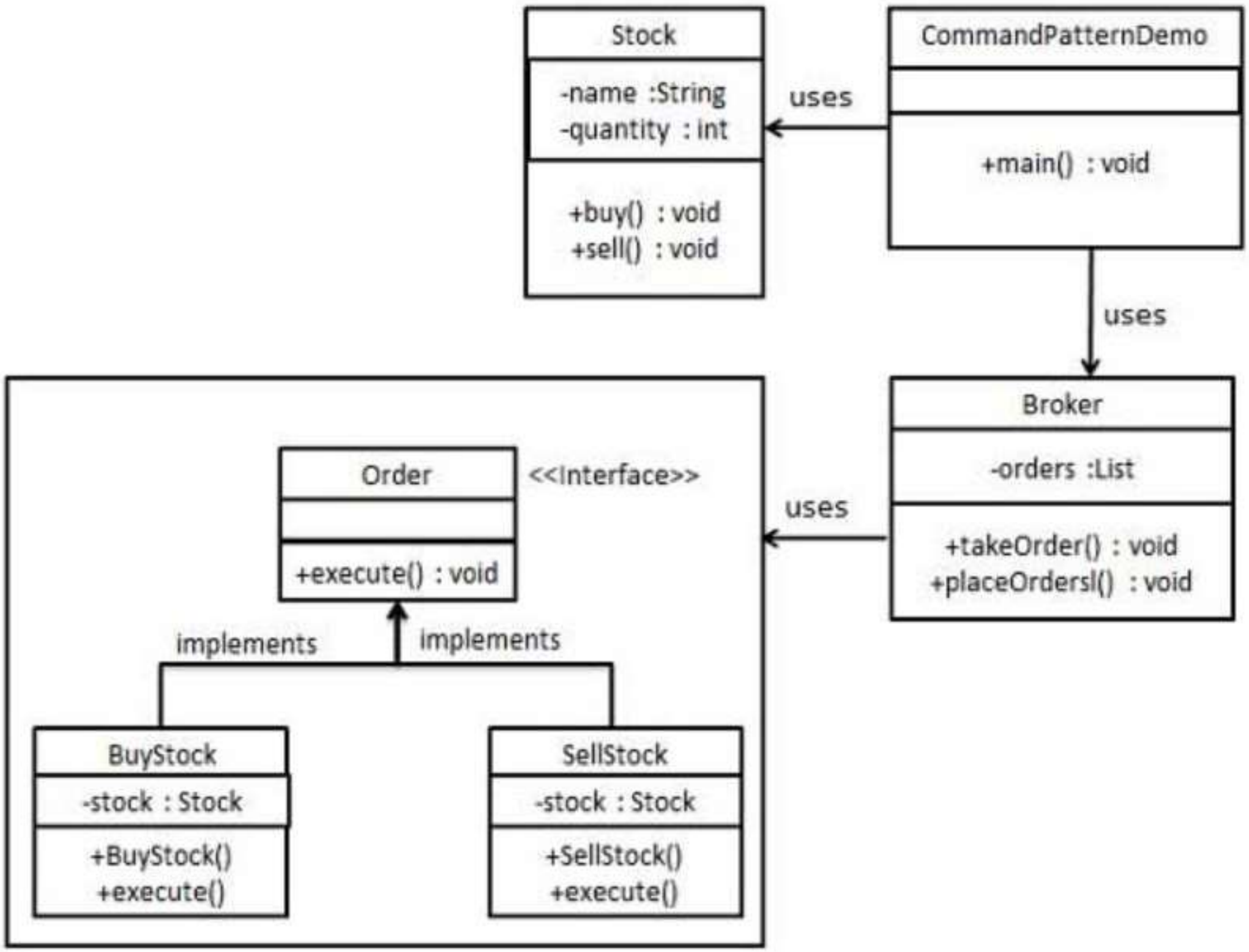
Known Uses: Undo/Redo

- Store a list of actions performed by the user
- Each action has
  - A “do” method that knows how to perform the action
  - An “undo” method that knows how to reverse the action
- Store a pointer to the most recent action performed by the user
- Undo – “undo” the current action and back up the pointer
- Redo – move the pointer forward and “redo” the current action

Exercise 1

- Assume that you can place on a trading platform (also called a broker) a set of buy ands sell order on specific stocks.
- The broker deals with a large number of stocks.
- It will place either buy or sell order for specific stocks as you might have specified.

Exercise 1: Solution



Players in the Design

- interface *Order* which is acting as a command.
- *Stock* class acts as a request.
- Concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing.
- *A* class *Broker* is created which acts as an invoker object.
  - It can take and place orders.

Implementation

```
public interface Order {
   void execute();
}
```

```
public class Stock {  //Receiver

   private String name = "ABC";
   private int quantity = 10;

   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```

```java
public class BuyStock implements Order {
   private Stock abcStock;
   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }
   public void execute() {
      abcStock.buy();
   }
}
```

```java
Public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

```java
import java.util.ArrayList;
import java.util.List;

  public class Broker {
  private List<Order> orderList = new ArrayList<Order>();

  public void takeOrder(Order order){
    orderList.add(order);
  }

  public void placeOrders(){

    for (Order order : orderList) {
      order.execute();
    }
    orderList.clear();
  }
}
```

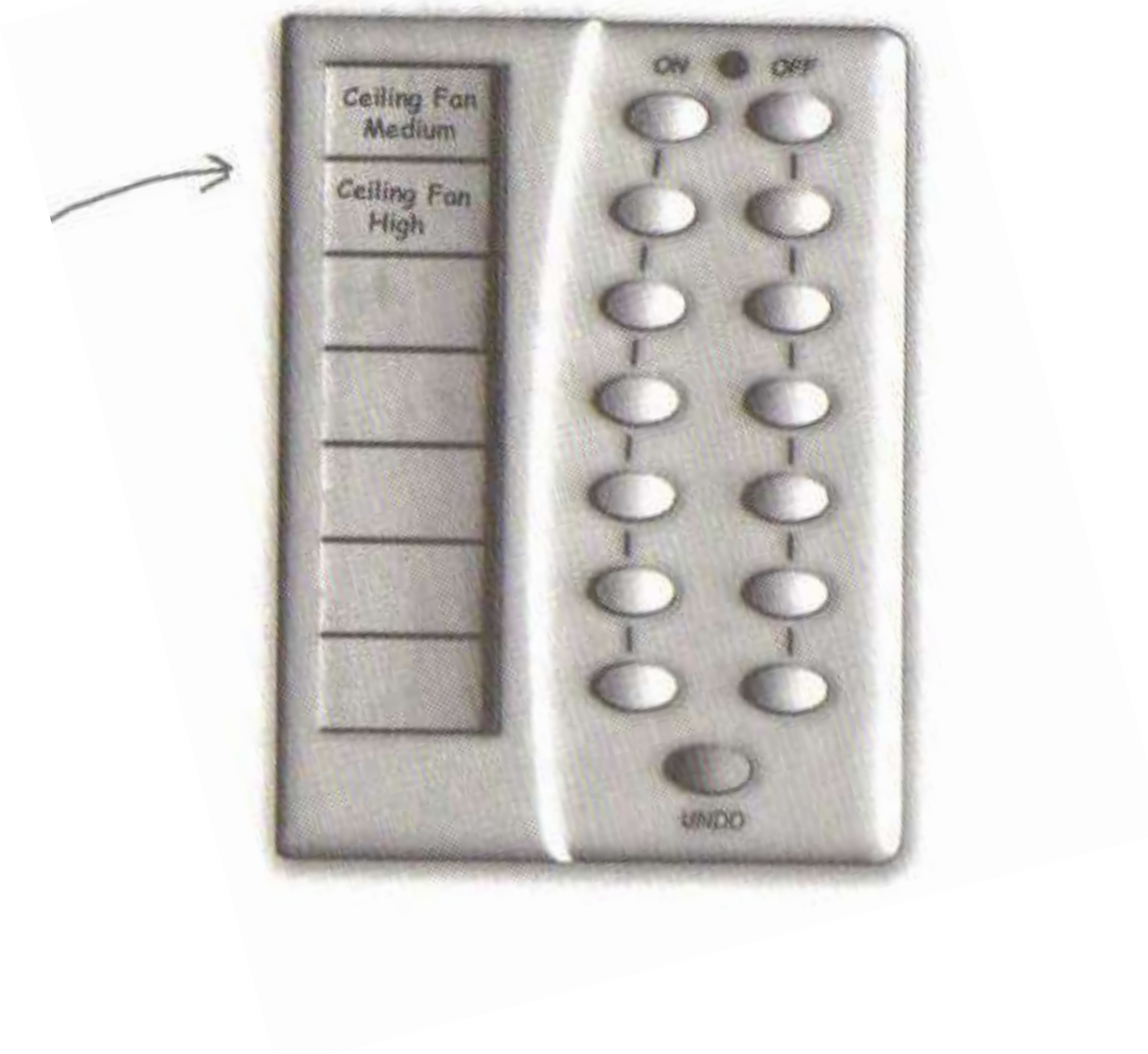
```java
public class CommandPatternDemo {
  public static void main(String[] args) {
    Stock abcStock = new Stock();

    BuyStock buyStockOrder = new BuyStock(abcStock);
    SellStock sellStockOrder = new SellStock(abcStock);

    Broker broker = new Broker();
    broker.takeOrder(buyStockOrder);
    broker.takeOrder(sellStockOrder);

    broker.placeOrders();
  }
}
```

Exercise

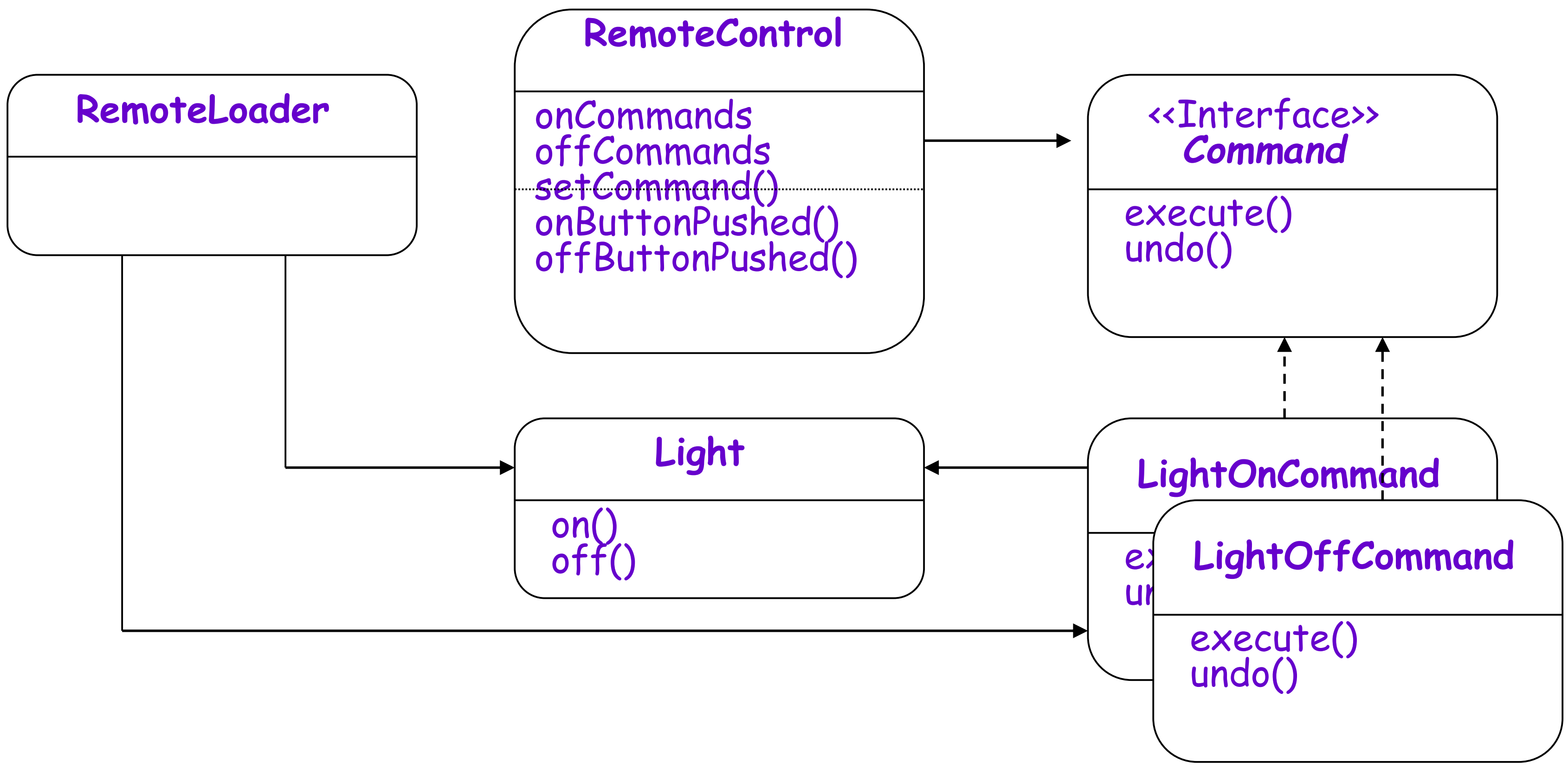
- Design and Build a remote that will control variety of home devices
  - Add an “undo” button to support one undo operation

- Sample devices: lights, stereo, TV, ceiling light, thermostat, sprinkler, hot tub, garden light, ceiling fan, garage door

Command pattern – Undo operation



Command Pattern Class Diagram for Home automation



```
Public  interface Command{
        Public  void  execute();
}
```

```
Public class SwitchOnCommand implements Command{
    Switch  switch;
     public LightOnCommand(Switch switch){
           this.switch = switch;  }
      public void execute(){ switch.on();  }
}
```

```
Public class RemoteControlTest{
  Public  static void  main(String[] args){
    RemoteControl remote=new RemoteControl();
    GarageDoor garageDoor = new GarageDoor();
    GarageDoorOpenCommand = new
    GarageDoorOpenCommand(garageDoor);
    remote.setCommand(garageOpen);
    remote.buttonPressed();
}
}
```

```
Public class  RemoteControl{
   Command   slot;
   Public  RemoteControl(){}
  Public void setCommand(Command command){
     slot  = command;
  }

  public  void  buttonPressed(){
     slot.execute();
 }
}
```

Exercise

- **Macros:**
  - Record a sequence of user actions so they can be turned into a macro
  - Macro can be re-executed on demand by the user

```
public class MacroCommand implements Command {
   Command[] commands;
   public MacroCommand(Command[] commands) {        this.commands =
   commands;
   }

   public void execute() {
    for (int i = 0; i < commands.length; i++) {
        commands[i].execute();
    }
   }

   public void undo() {
    for (int i = 0; i < commands.length; i++) {
    commands[i].undo();
    }
   }
}
```

Macro Commands

Command pattern: Final Analysis

- It is easy to add new Commands, because you do not have to change existing classes
  - Command is an abstract class, from which you derive new classes
  - execute(), undo() and redo() are polymorphic functions
- You can undo/redo any Command
  - Each Command stores what it needs to restore state
- You can store Commands in a stack or queue
  - Command processor pattern maintains a history