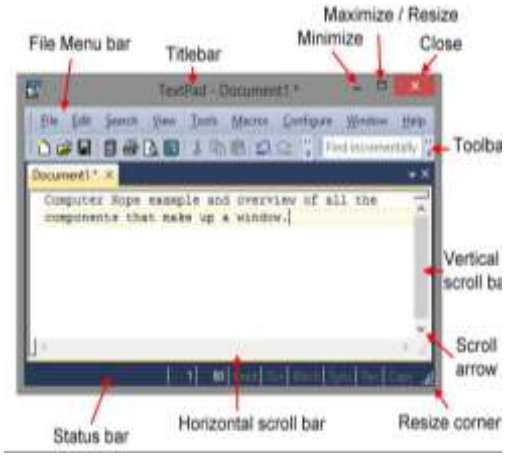
# Command Pattern

# Command Pattern

- Command is a behavioral design pattern:
  - **Turns a request into a stand-alone object that contains all information about the request.**
- This transformation lets you:
  - parameterize different sets of requests,
  - delay or queue a request's execution, and
  - support undoable operations.

# Introduction

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



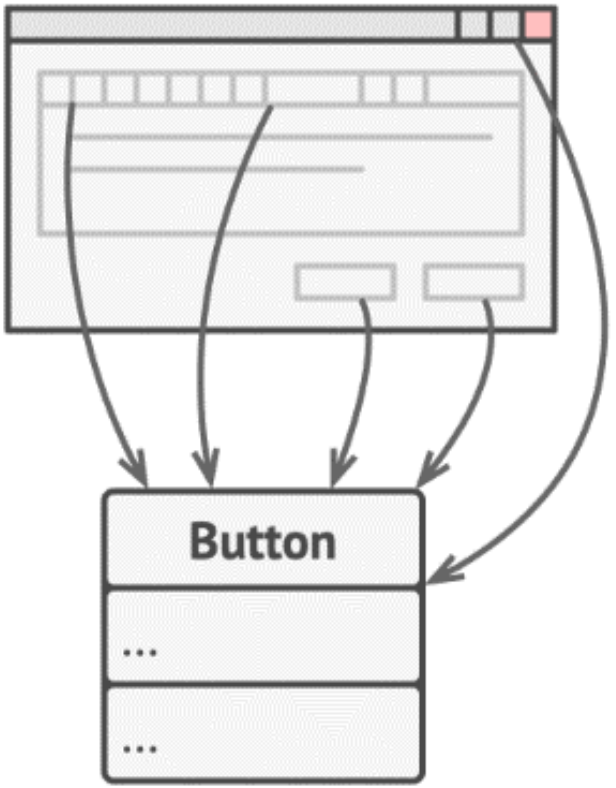

- **Example:**
  - A GUI toolkit provides several components: Buttons, scroll bars, text boxes, menus, etc.

- 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?

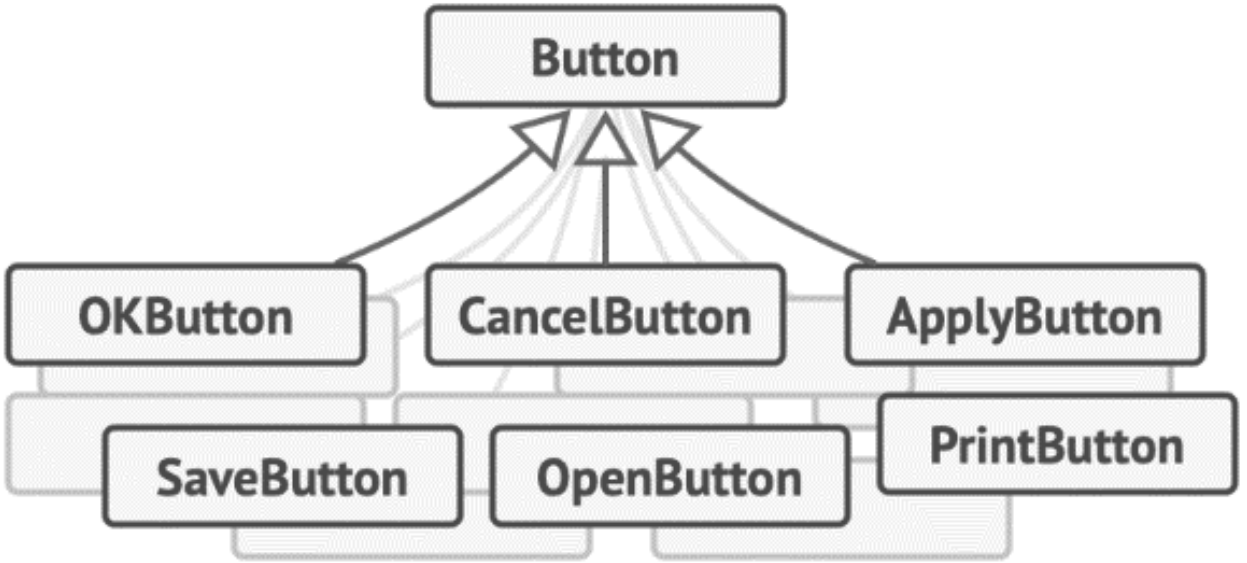
# Motivation

- 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

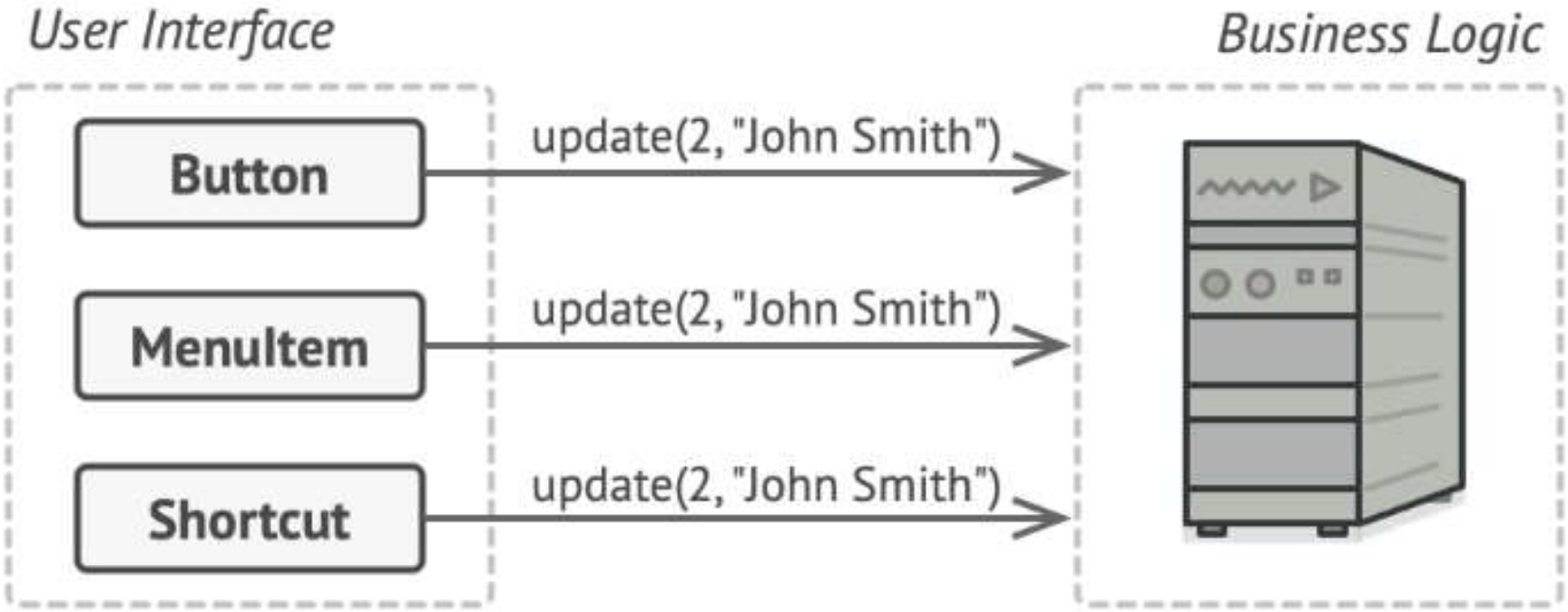
- Generic buttons won't do much:
  - But, each button needs to carry out separate actions.
- You toyed with the idea of creating hundreds of sub-classes of the button class.



# 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

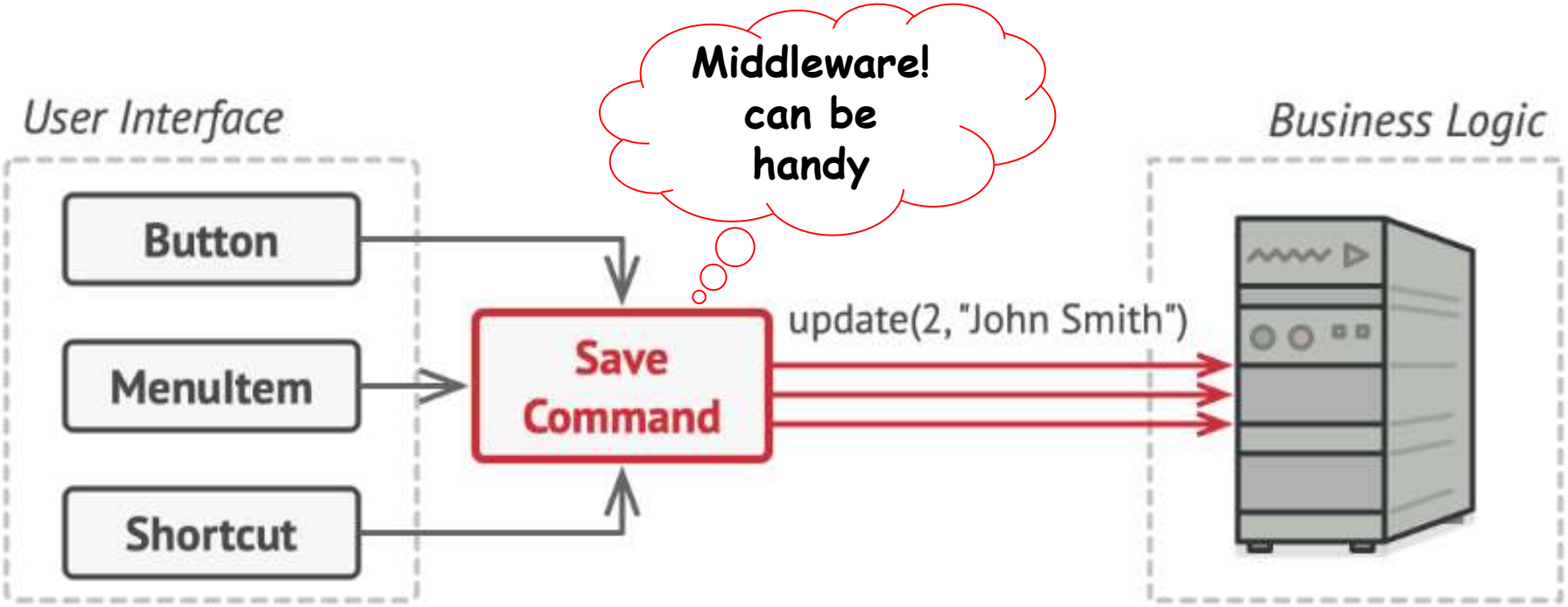



- GUI object calls a method of a business logic object, passing it some arguments.

# Analysis of the Design

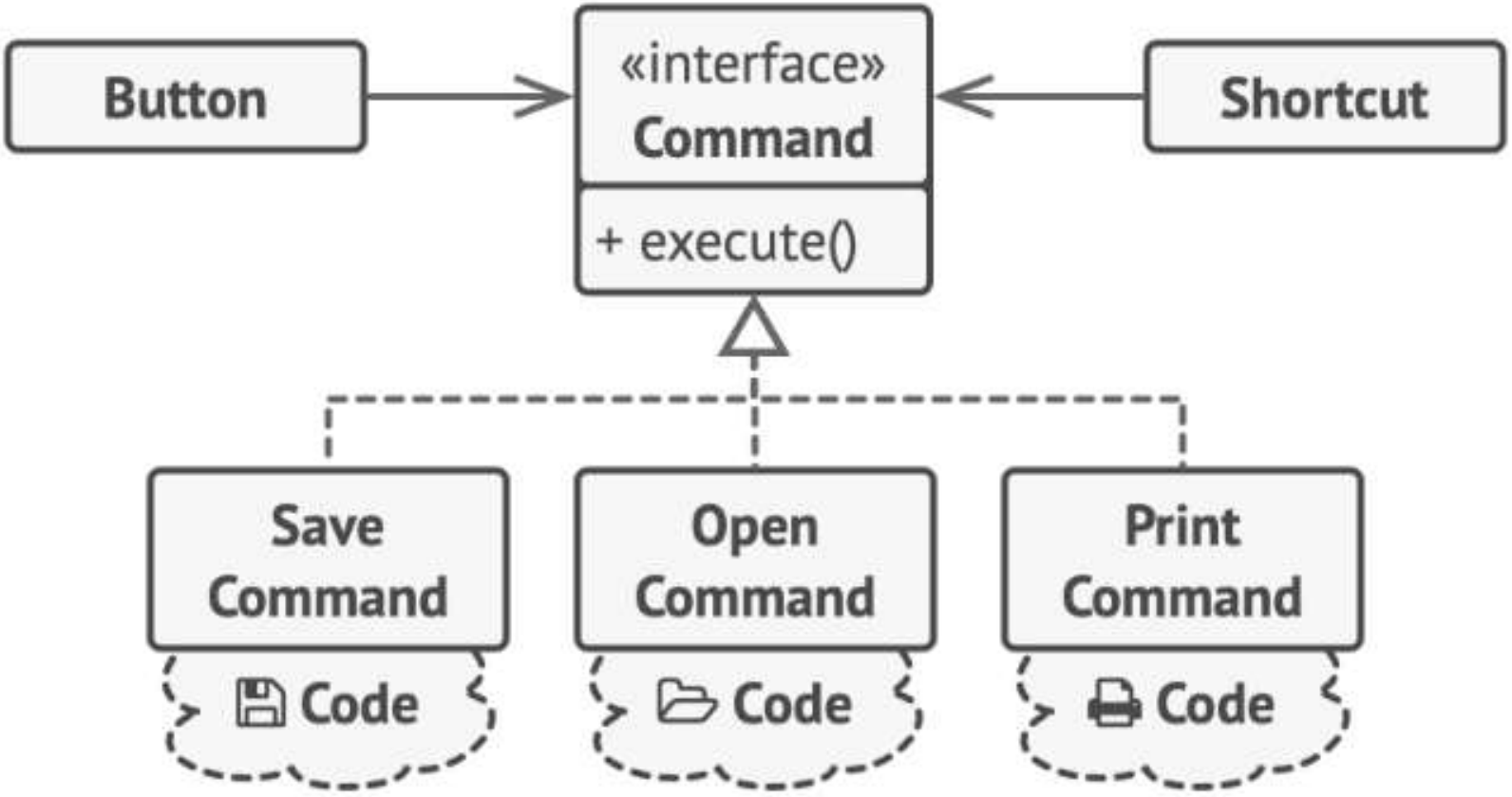
- Unnecessary Duplication of Code
- GUI directly interacts with the Entity classes --- Need a middleware
- Command pattern solution:
  - **Put name of the method name and arguments into a separate Command class,**

# Next Refinement




- We now can recognize that all commands need to implement the same interface…

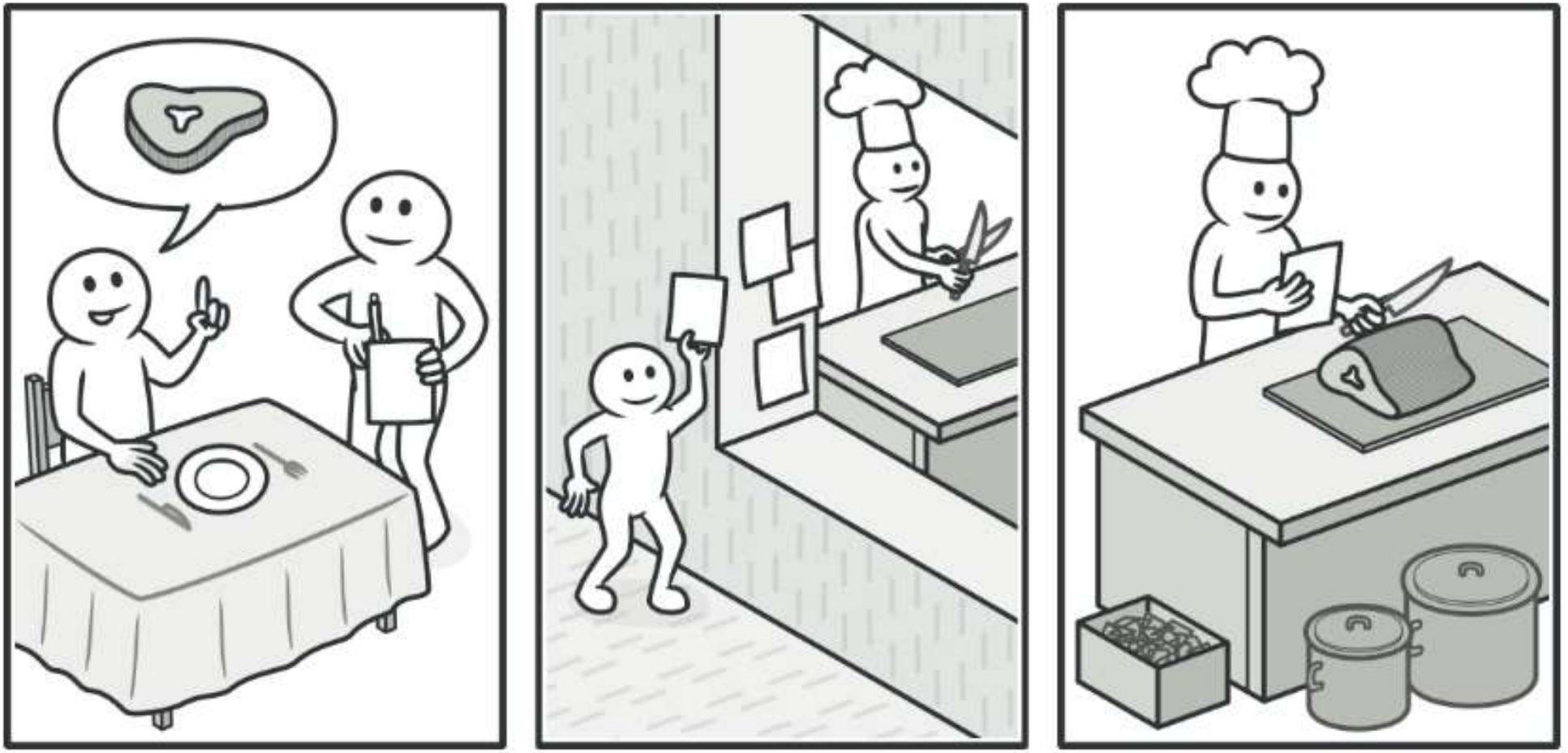
# Command Pattern: GUI Objects Delegate Work to Command Objects



# Button Example: Solution

- Implement a bunch of command classes for every possible operation:
  - Link them with particular buttons, depending on the buttons' intended behavior
- Commands become a convenient middle layer:
  - Reduces coupling between the GUI and business logic layers.
  - And that's only a fraction of the benefits that the Command pattern can offer!

# Real World Example



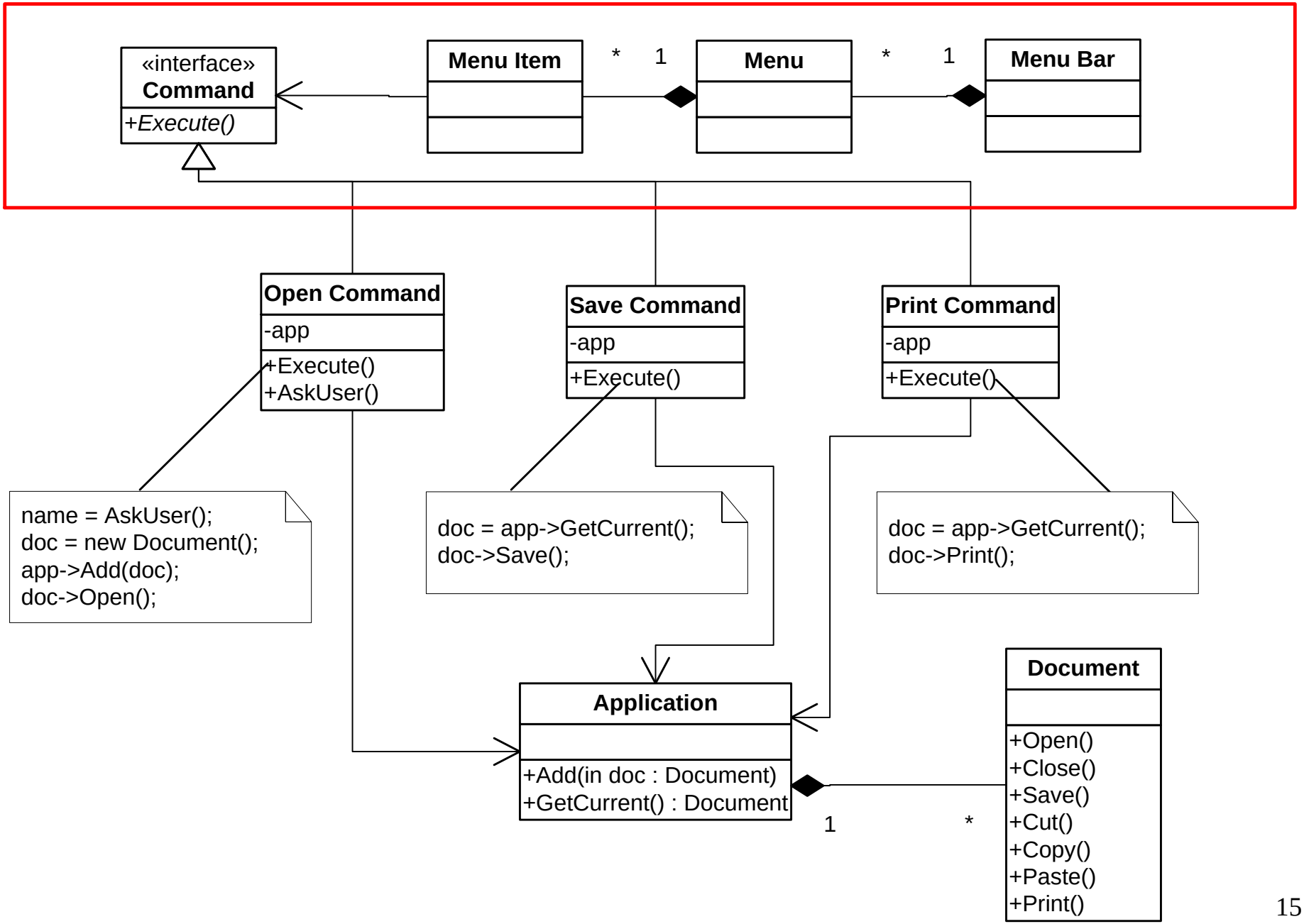
# Command Pattern: Intent

- Encapsulate a request from the client as an object, thereby
  - Parameterize requests,
  - Queue or log requests, and
  - Support undoable operations

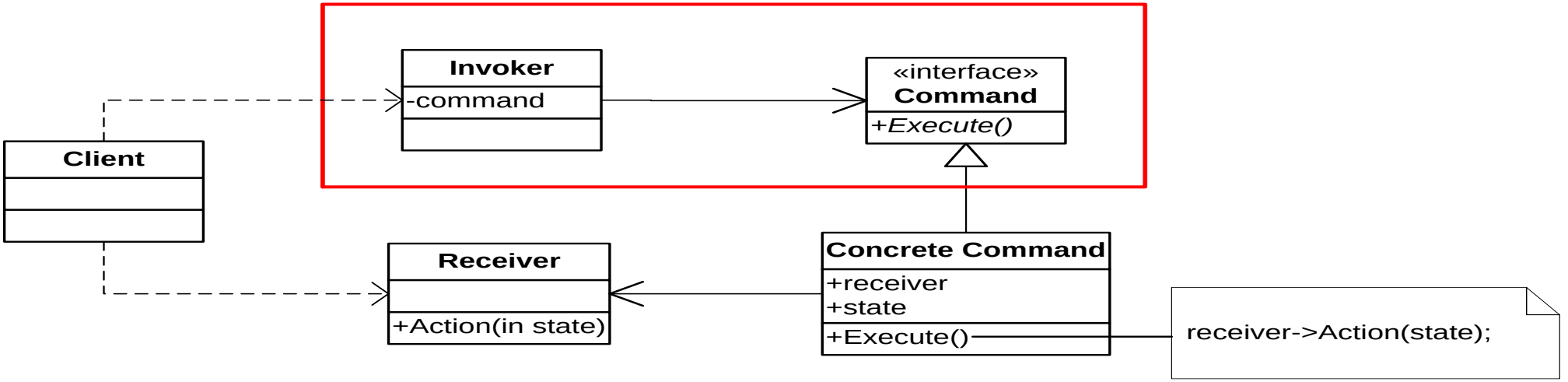
# When to use Command?

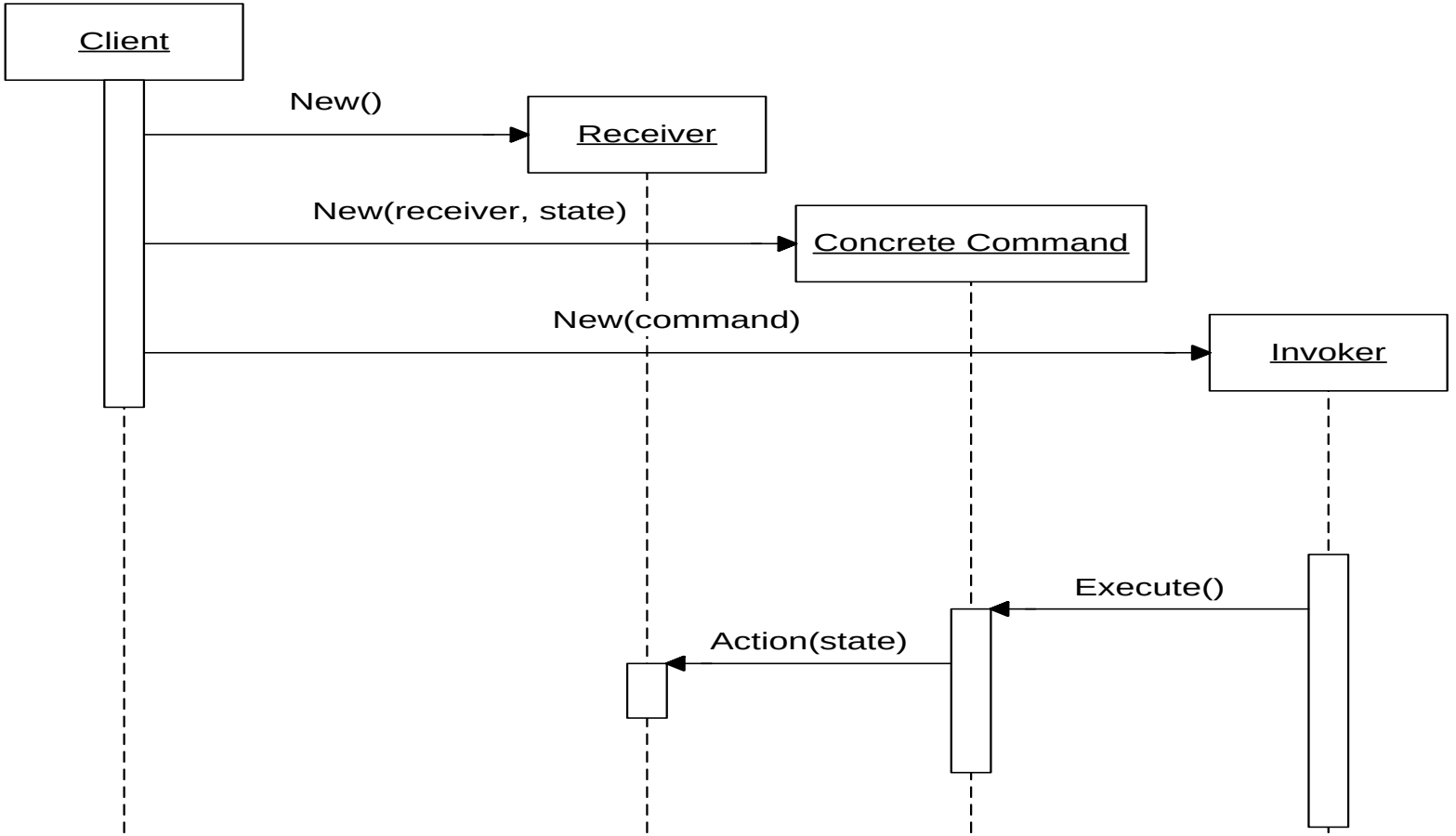
- Command Pattern is useful when:
  - A history of requests is needed
  - You need callback functionality
  - Requests need to be handled at variant times or in variant orders
  - The invoker should be decoupled from the object handling the invocation

# Example: GUI Toolkit



# Example: GUI Toolkit



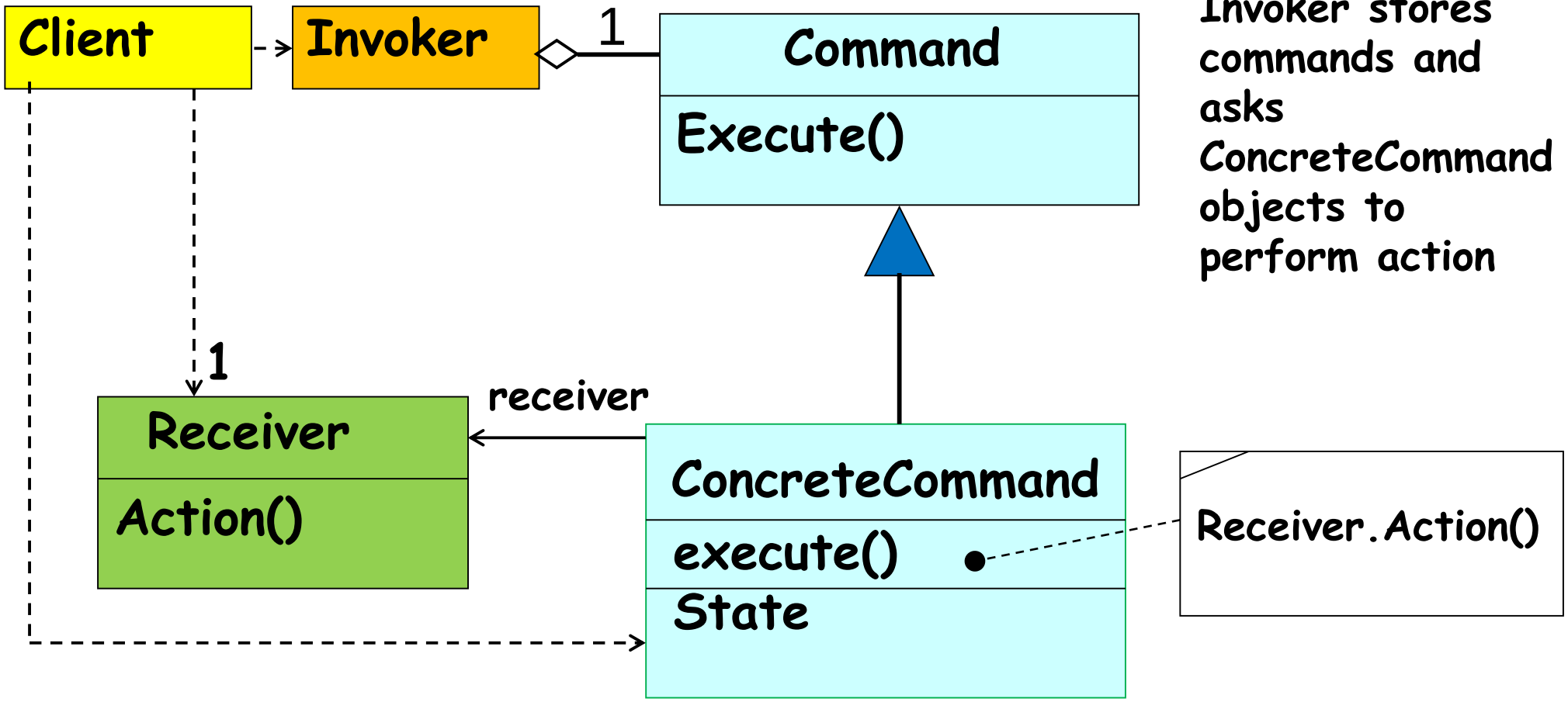
# Other Applications of Command

- **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

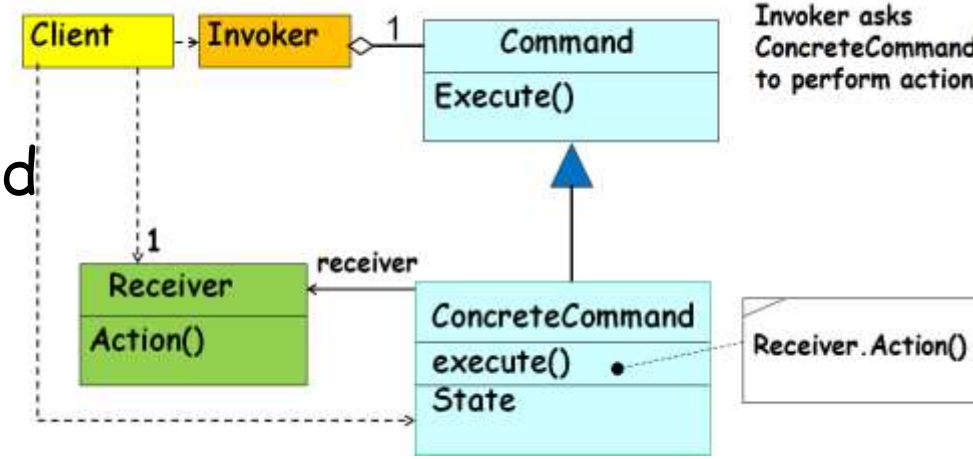
# Structure of command pattern




Invoker stores commands and asks ConcreteCommand objects to perform action

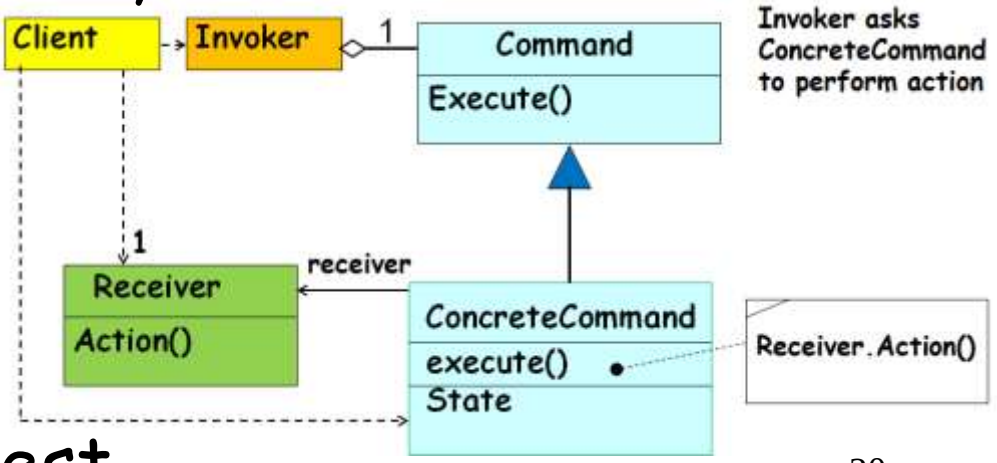
# Command pattern: Participants

- **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

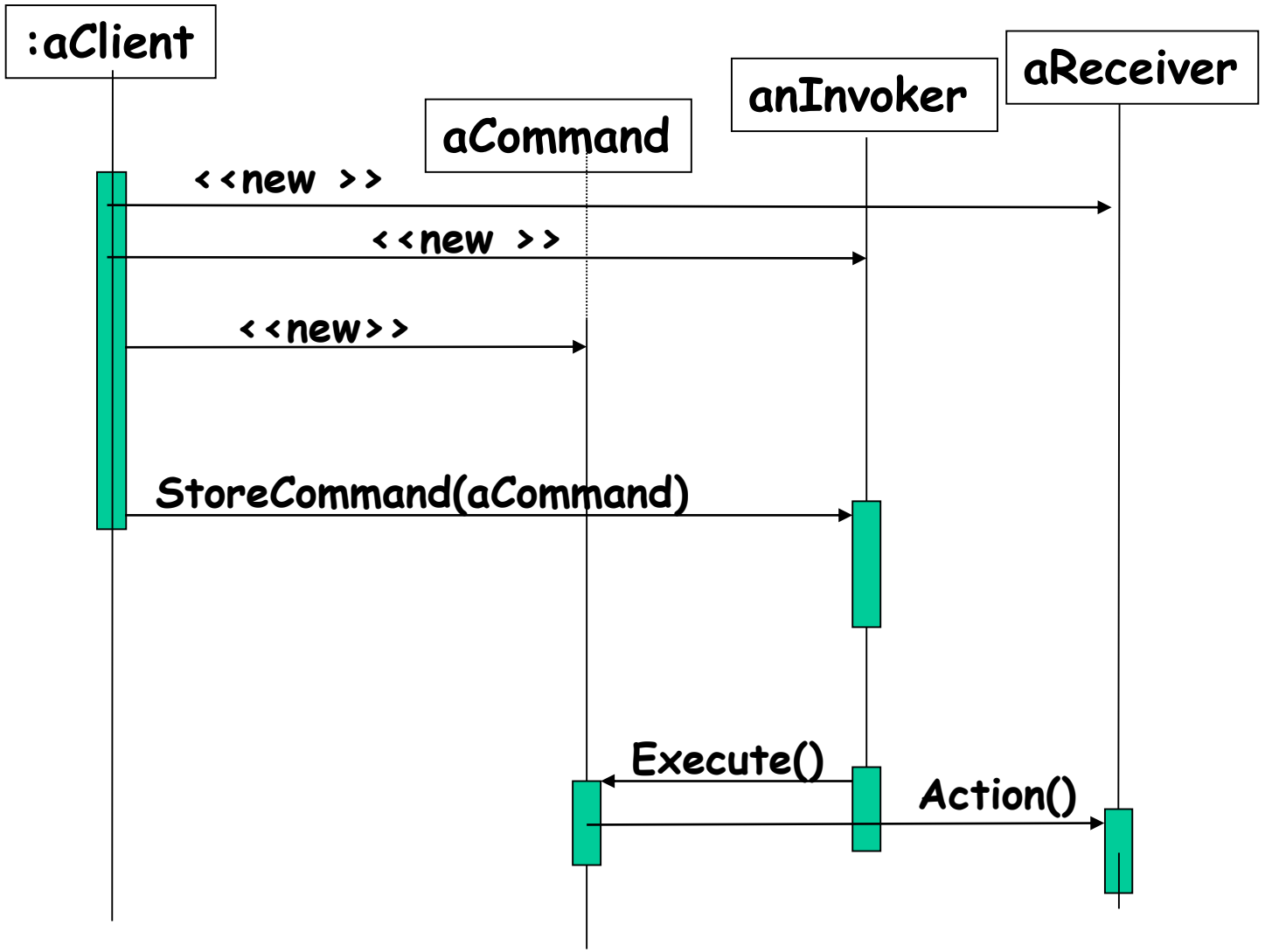


# Command pattern: Operation

- 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

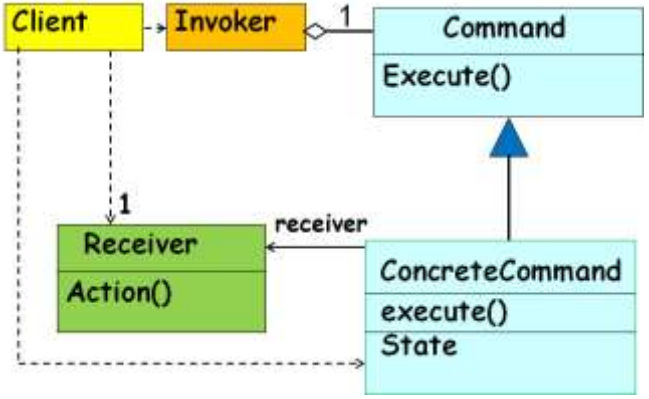
# Known Uses: Undo/Redo

- 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

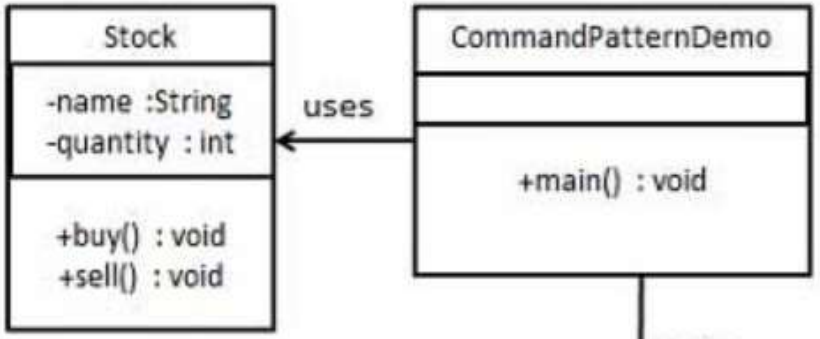
# Exercise 1

- 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.

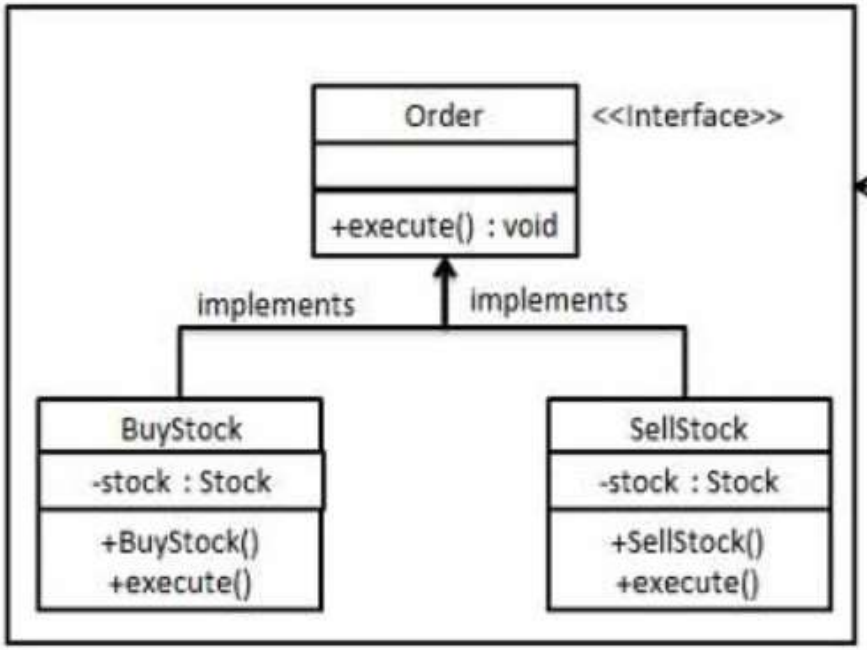
# Exercise 1: Solution

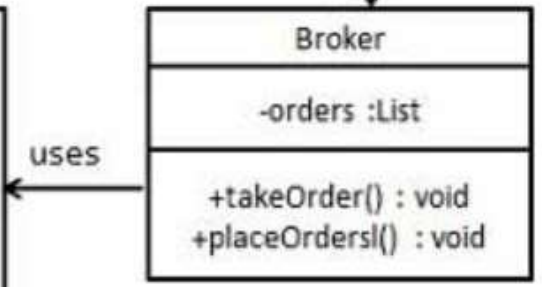



Receiver




Command




Invoker

# 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 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");
   }
}
```

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

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

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

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

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

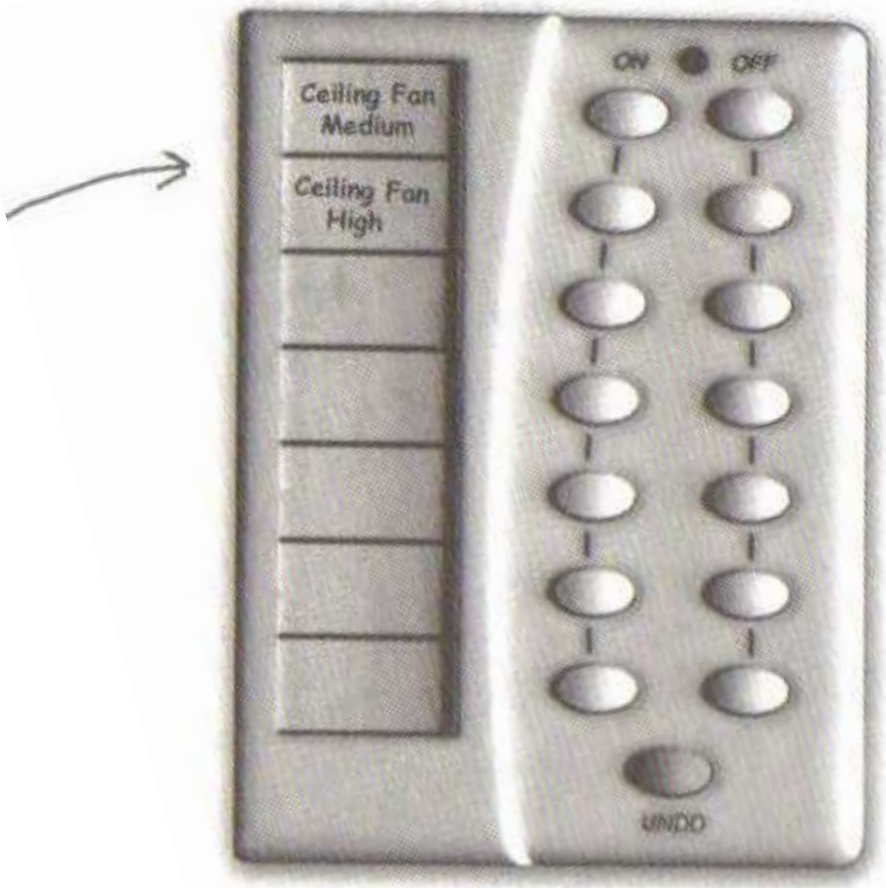
```
import java.util.ArrayList;
import java.util.List;

   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();
   }
}
```

```
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();
   }
}
```

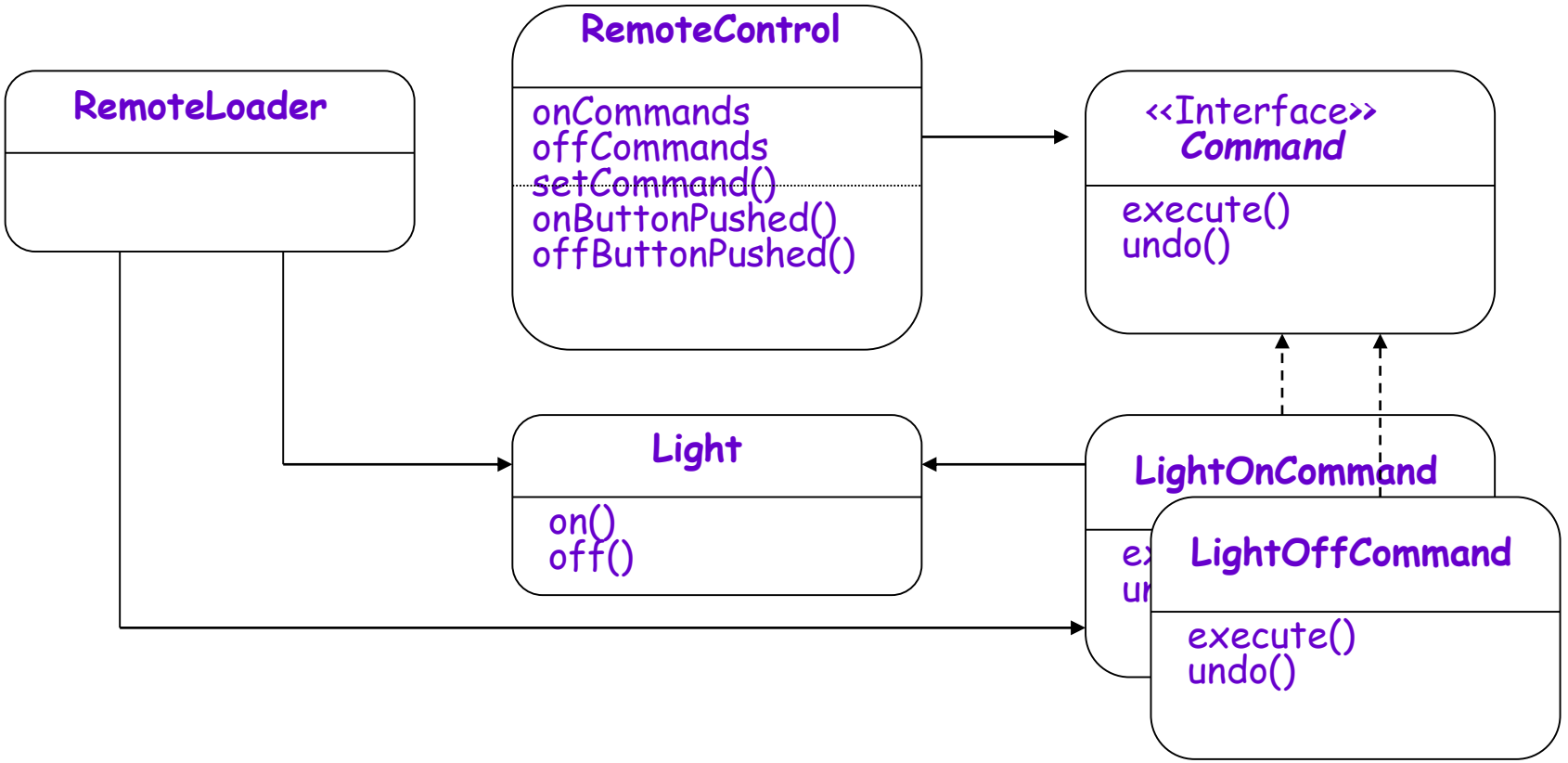
# Exercise

- 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();
  }
}
```

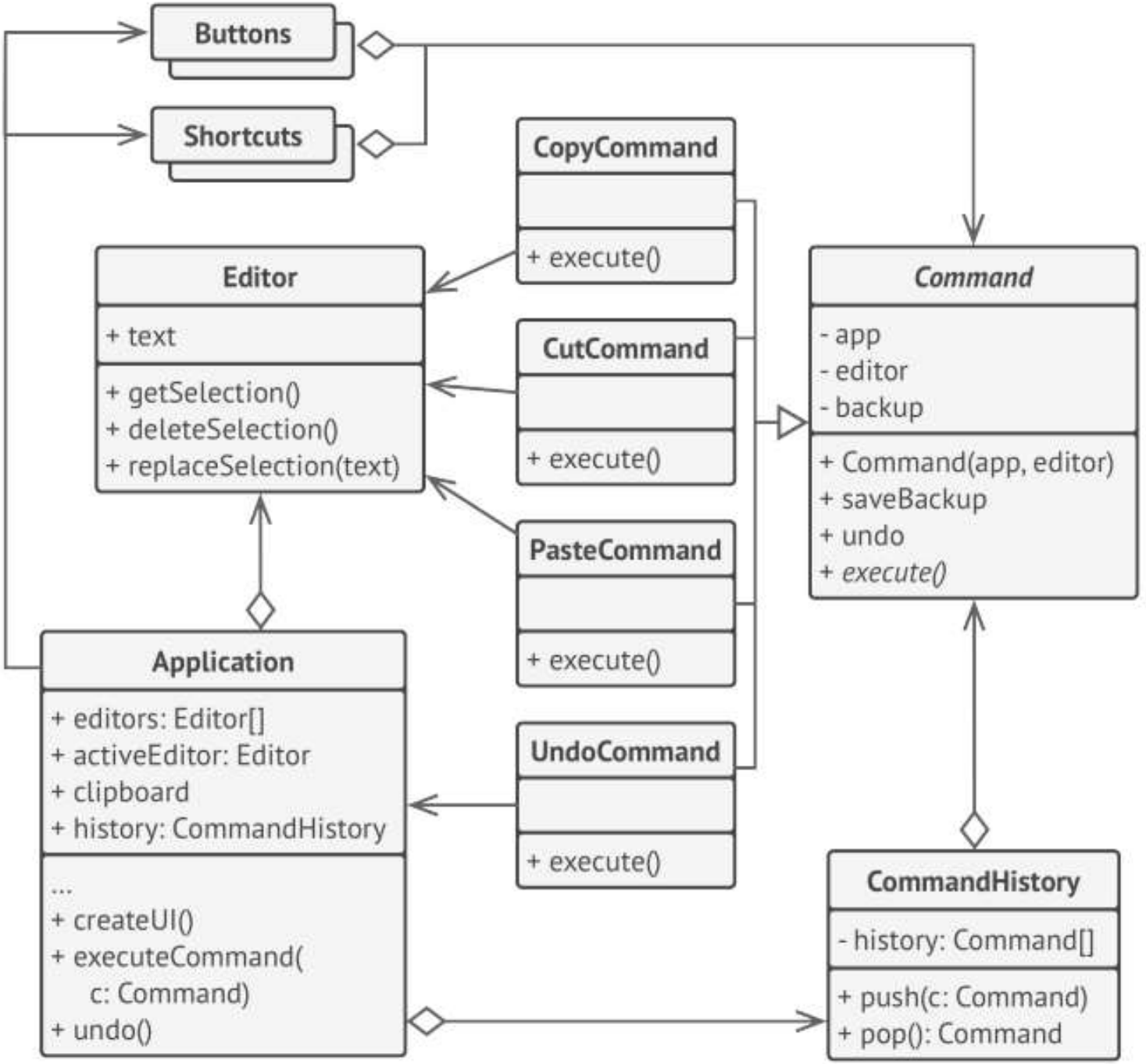
# Exercise

- **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

# Text Editor Example




*Undoable operations in a text editor.*

# 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

# FlyWeight Pattern

# Flyweight Pattern

## Intent

- Use sharing to support a large number of fine-grained objects efficiently.
- Factors the common properties of multiple instances of a class into a single object:
  - Saves space and maintenance of duplicate instances.

**Flyweight(fli-wat) (Dictionary meaning)**
Something that is particularly small, light, or inconsequential.

# Problem of redundant objects

- Problem: existence of redundant objects can bog down any system.
  - many objects have same state
  - intrinsic vs. extrinsic state

- Example: File objects that represent the same file on disk
  - new File("mobydick.txt")
  - new File("mobydick.txt")
  - new File("mobydick.txt")

    ...
  - new File("notes.txt")
  - new File("notes.txt")



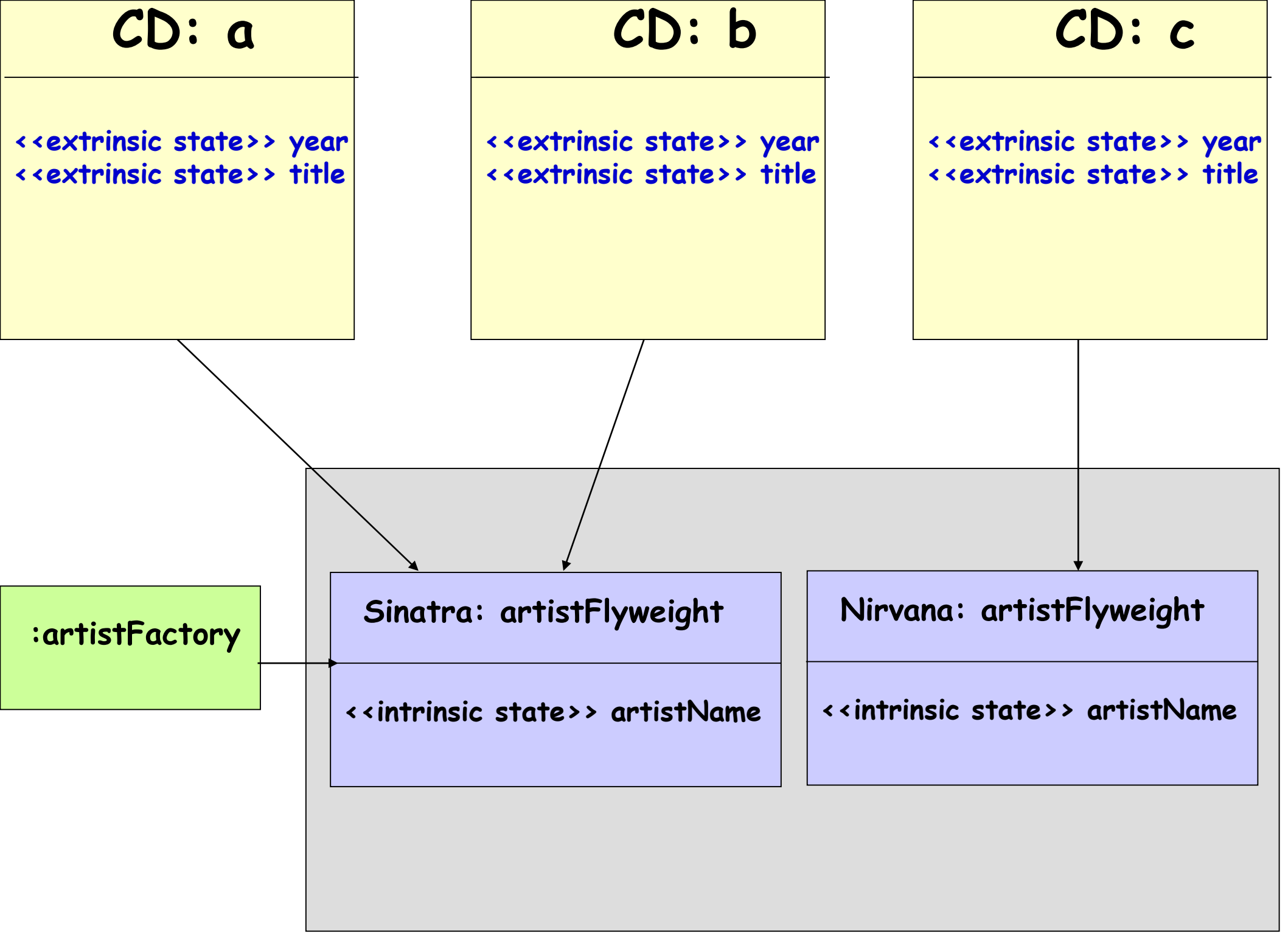
43

# Intrinsic vs. Extrinsic State

- Intrinsic state is stored in the Flyweight:
  - It is independent of the context.
  - Example: Character code, and its graphical style.
- Extrinsic state depends on the context and thus it cannot be shared
  - Coordinate position, character's typographic style.

# Intrinsic vs. Extrinsic State

- **Create "Flyweight" Objects for Intrinsic State**
- **Intrinsic State:**
  - Information independent of the object's context, sharable
  - Instead of storing the same information n times for n objects, it is only stored once in Flyweight
- **Extrinsic State:**
  - Information dependent on the object's context, unsharable,
  - This is excluded from the Flyweight


CD: a
<<extrinsic state>> year
<<extrinsic state>> title
CD: b
<<extrinsic state>> year
<<extrinsic state>> title
CD: c
<<extrinsic state>> year
<<extrinsic state>> title
:artistFactory
Sinatra: artistFlyweight
<<intrinsic state>> artistName
Nirvana: artistFlyweight
<<intrinsic state>> artistName

# Flyweight pattern

- **flyweight**: Ensures that no more than one instance of a class will have identical state
  - Similar to singleton, but has many instances, one for each unique-state object (object pool)
  - Useful for cases when there are many instances of a type but many are the same
  - Used in conjunction with Factory pattern to create a very efficient object-builder
  - **Examples in Java: String**

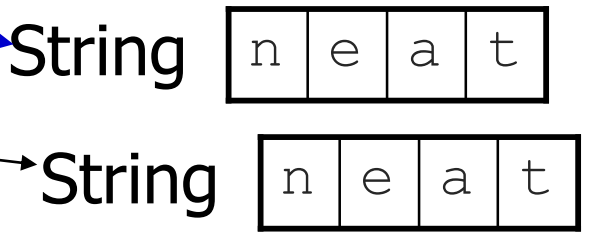


47

# String flyweighting

- **interning**: Synonym for flyweighting; sharing identical instances.
  - Java ***String*** objects are automatically interned (flyweighted) by the compiler whenever possible.
  - If you declare two string variables that point to the same literal.
  - If you concatenate two string literals to match another literal.

```
String a = "neat";
String b = "neat";
String c = "n" + "eat";
```



# Limits of String flyweight

```
String a = "neat";
Scanner console = new Scanner(System.in);
String b = console.next();  // user types "neat"
if (a == b) { ...           // false
```

- There are many cases where the compiler can't flyweight:
  - When you build a string later out of arbitrary variables
  - When you read a string from a file or stream (e.g. `Scanner`)
  - When you build a new string from a `StringBuilder`
  - When you explicitly ask for a `new String` (bypasses flyweighting)

- **Java Strings are flyweighted by the compiler wherever possible**
- can be flyweighted at runtime with the intern method

```
public class StringTest {
  public static void main(String[] args) {
    String fly  = "fly", weight  = "weight";
    String fly2 = "fly", weight2 = "weight";

    System.out.println(fly == fly2);                    // true
    System.out.println(weight == weight2);              // true

    String distinctString = fly + weight;
    String flyweight = (fly + weight);
    System.out.println(distinctString == flyweight); // false

    String flyweight = (fly + weight).intern();
    System.out.println(flyweight == "flyweight");       // true
  }
}
```

# Implementing a Flyweight

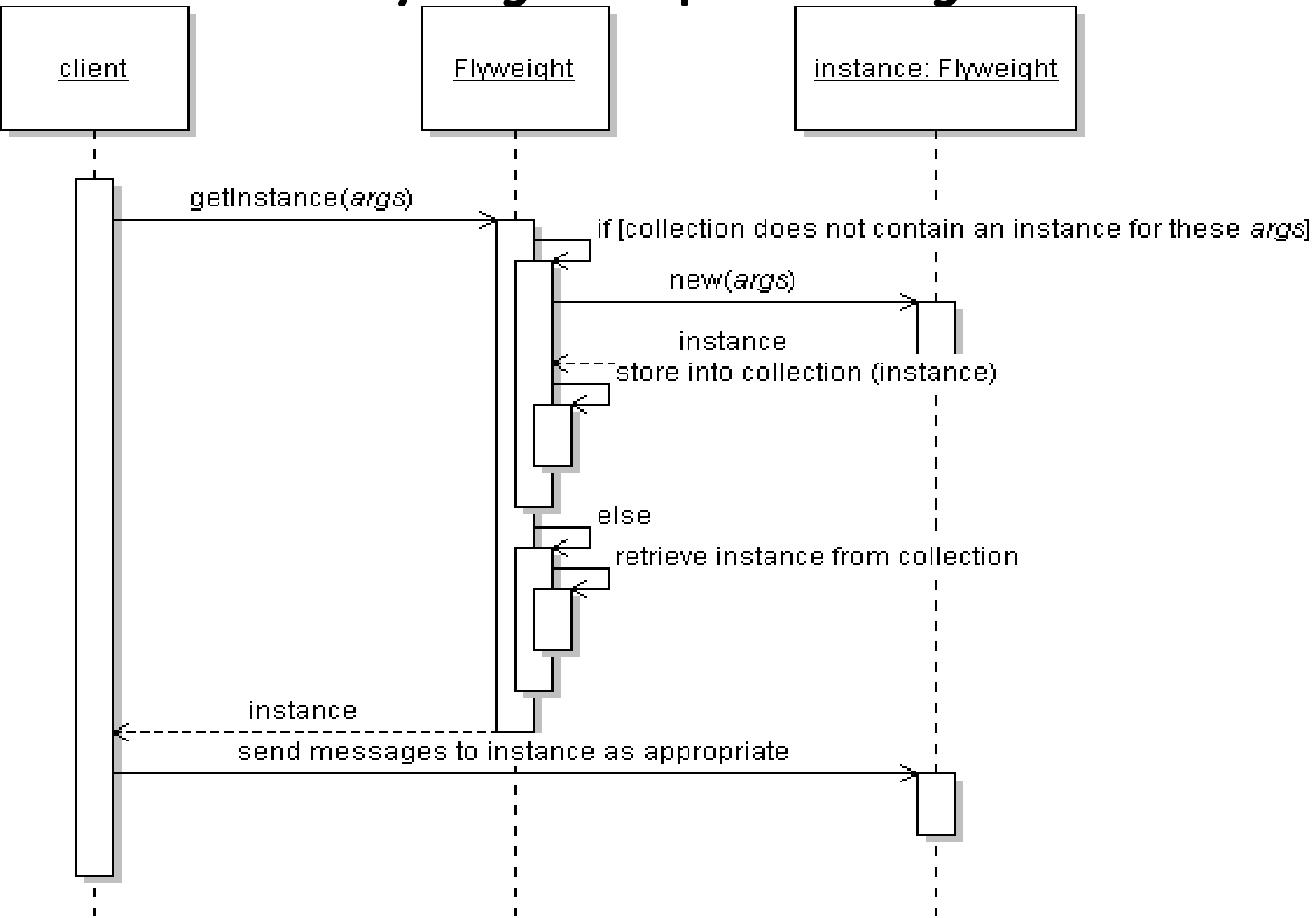
- Flyweighting works best on immutable objects
  - **immutable**: cannot be changed once constructed

```
public class Flyweighted {
    static map or table of instances
    private constructor
    static method:  get-an-instance(){
        if we have created this type of instance before,
          get it from map and return it
        otherwise, make the new instance, store and
          return it}
}
```



51

# Flyweight sequence diagram



# Implementing a Flyweight

```
public class Flyweighted {
  Map or table of instances
  private Flyweighted() {}
   public static synchronized Flyweighted getInstance(Object key) {
    if (!myInstances.contains(key)) {
      Flyweighted fw = new Flyweighted(key);
      myInstances.put(key, fw);
      return fw;
    } else
      return (Flyweighted)myInstances.get(key);  }
}
```

# Class before flyweighting

```
public class Point {
  private int x, y;
  public Point(int x, int y) {
    this.x = x;    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public String  toString() {
    return "(" + this.x + ", " + this.y + ")";  }
}
```

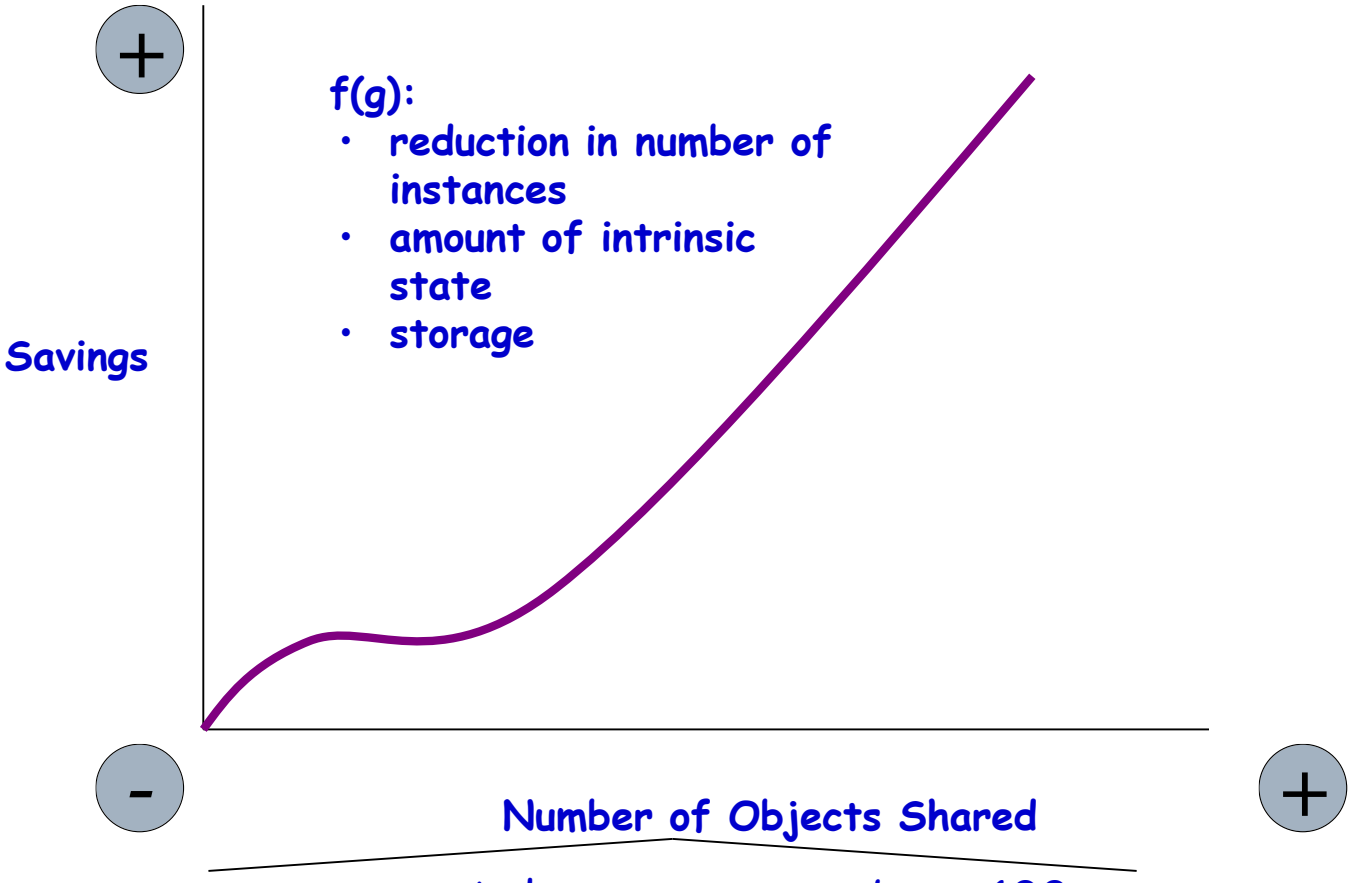


54

# Class after flyweighting

```
public class Point {
  private static Map instances = new HashMap();
  public static Point getInstance(int x, int y) {
    String key = x + ", " + y;
    if (instances.containsKey(key))  //reuse existing pt
      return (Point)instances.get(key);
    else{
    Point p = new Point(x, y);
    instances.put(key, p);
    return p;}
  }
  private final int x, y;  // immutable
  private Point(int x, int y) {
    ...
```

# Efficiency

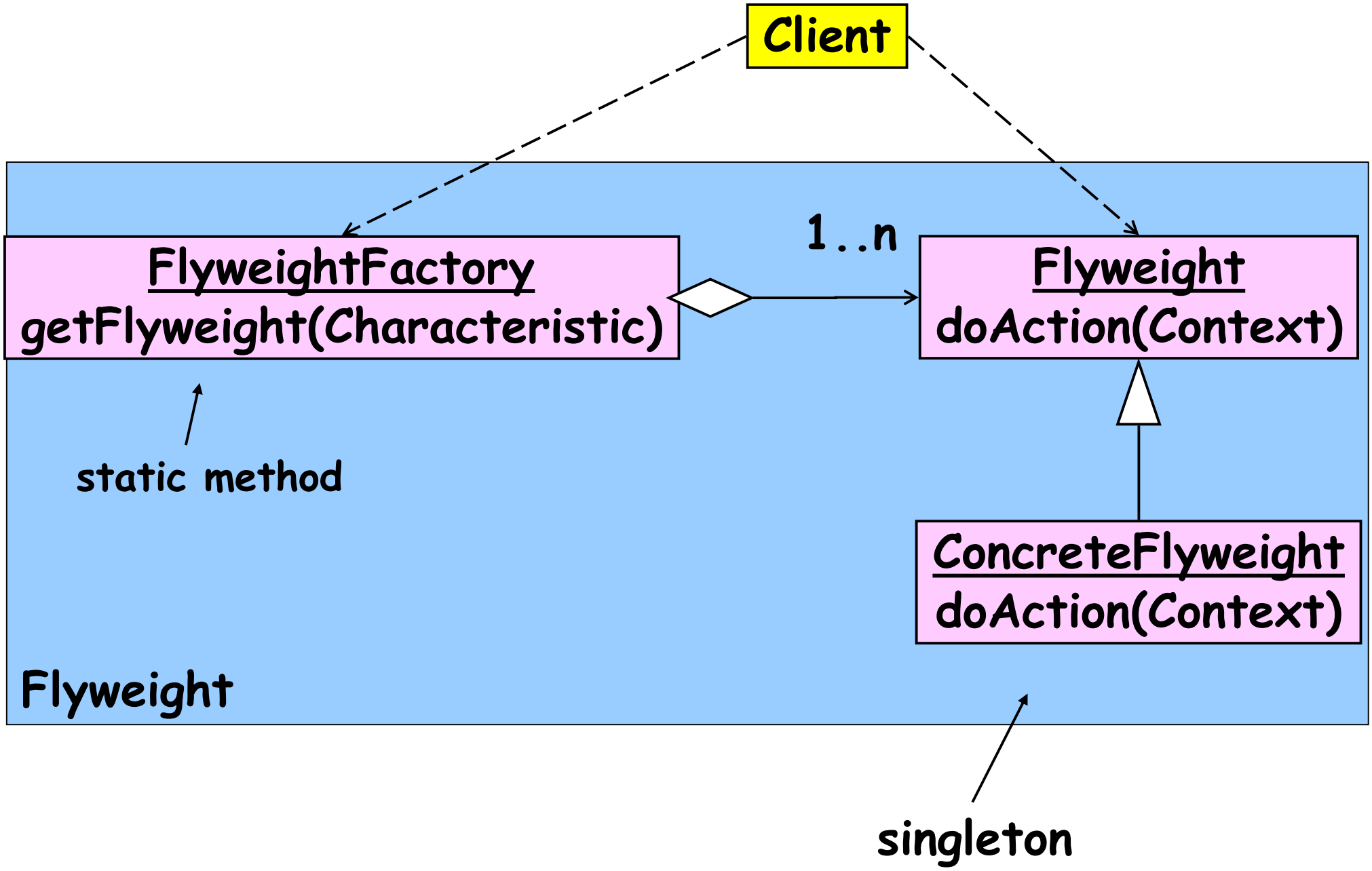



- A document uses about 100 character objects (the ASII set)

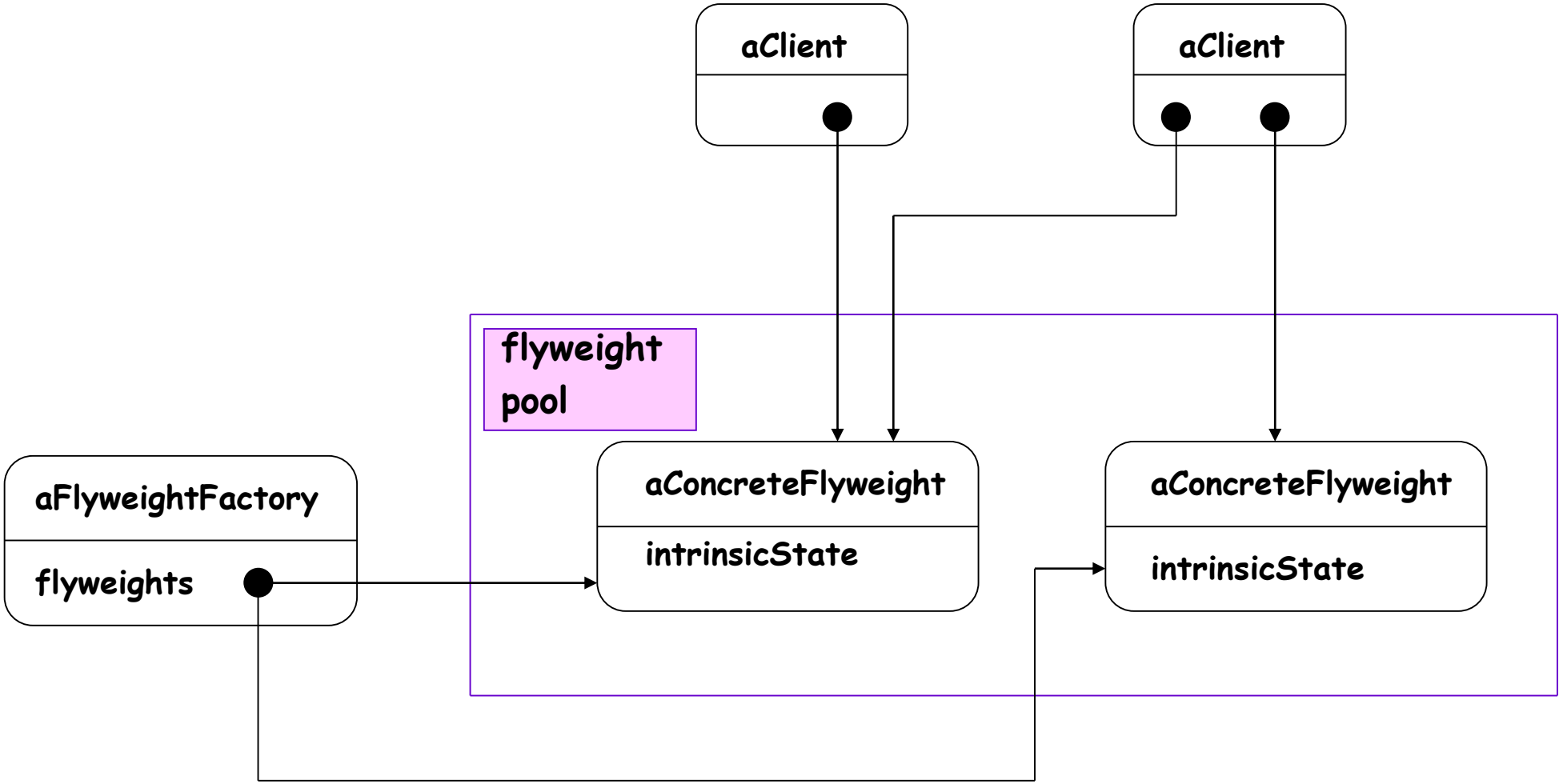
100 Chars shared in 8,000 char document

100 Chars shared in 200,000 char document

# Flyweight Class Model



# Flyweight Pattern



# Flyweight Pattern

## Participants

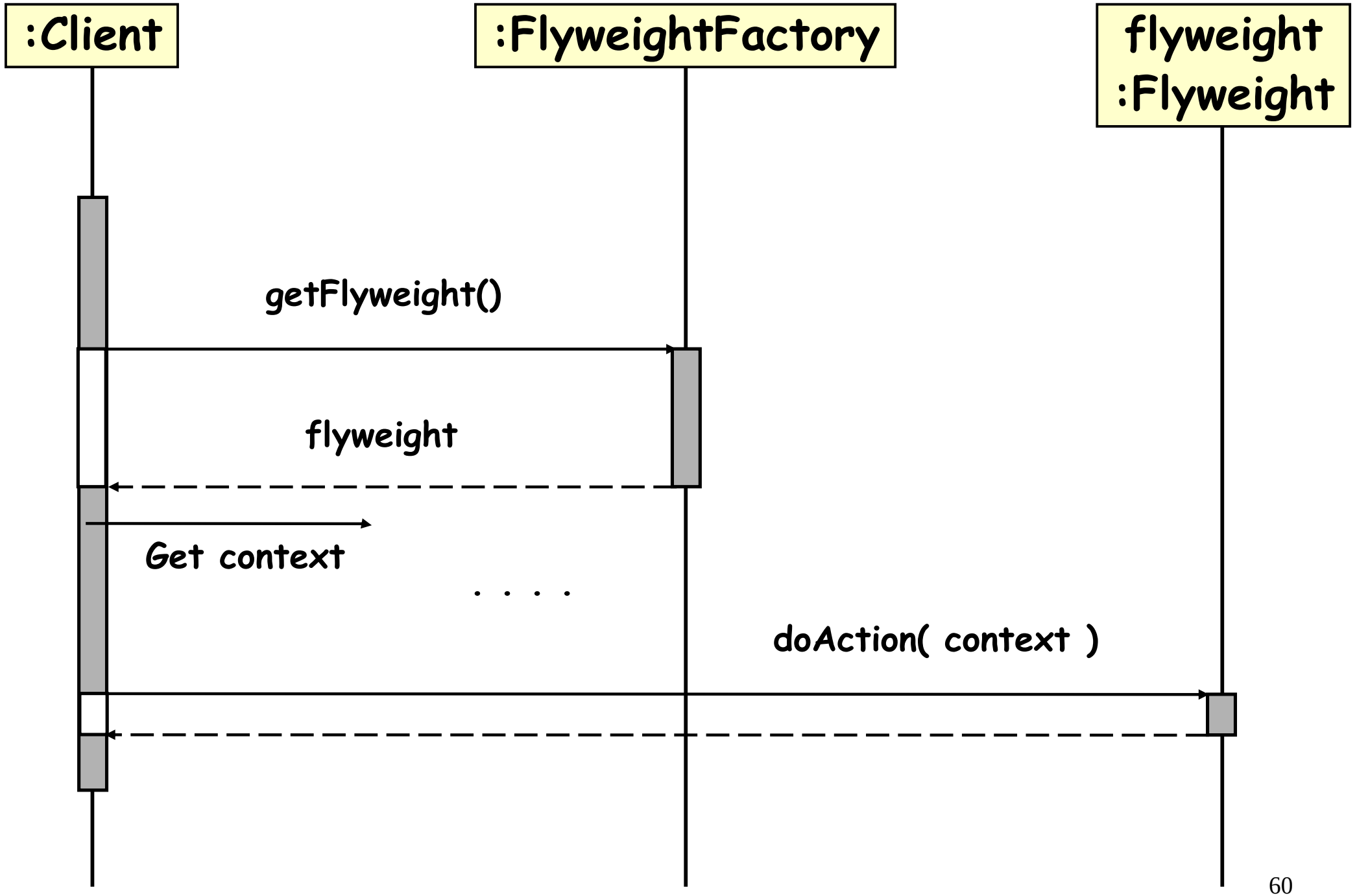
- **Flyweight** (Glyph) – declares an interface through which flyweights can receive and act on extrinsic state.

- **ConcreteFlyweight** (Character) – implements the Flyweight interface and adds storage for intrinsic state. It must be sharable and any stored state must be independent of the flyweight's context.

- **UnsharedConcreteFlyweight** (Row, Column) – not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level of the flyweight object structure.

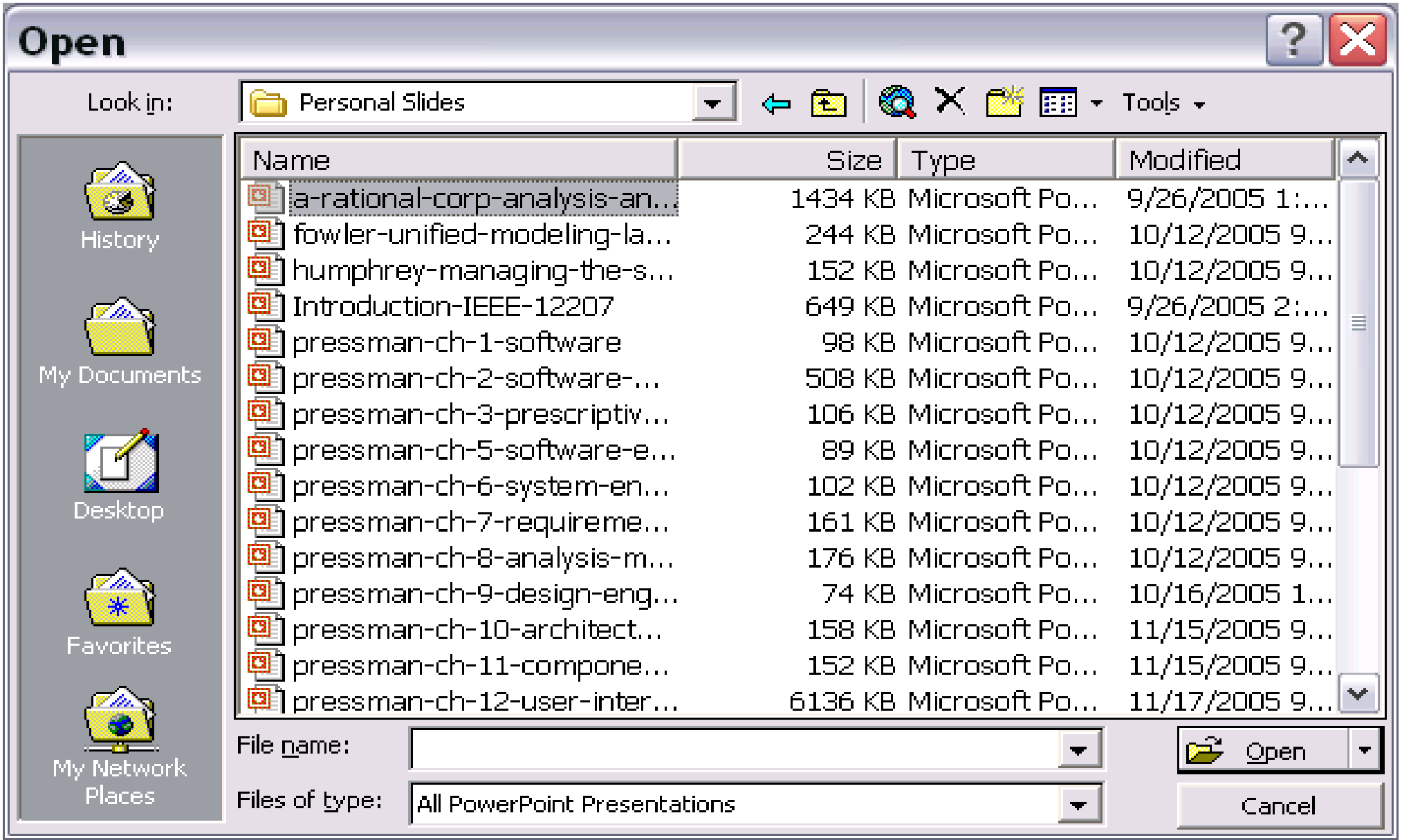
- **FlyweightFactory** – creates and manages flyweight objects. When a client requests a flyweight object, the factory supplies an existing flyweight object if one exists and creates one if it doesn't.

- **Client** – maintains a reference to flyweights and computes or stores their extrinsic state.


:Client
:FlyweightFactory
flyweight
:Flyweight
getFlyweight()
flyweight
Get context
. . . .
doAction( context )

**Example:** Window size of 15; folder contains over 500 items
Each line item had a drawing window associated with it

**We want to avoid proliferating an object for every item to be displayed.**



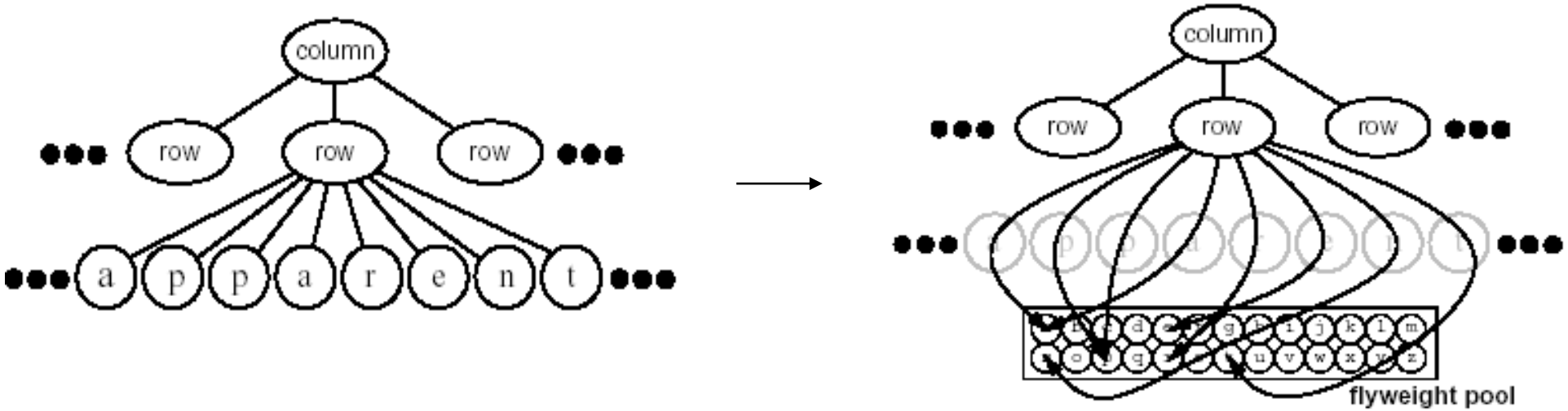
# Flyweight Example: Document Editor

- An Object Oriented Document editor implementation:
  - Usually objects representing each character.
- Even a moderate sized Document may need lots of character objects.
- **The Flyweight pattern helps to share objects and yet allow their use at fine granularities.**

# Why are Characters Objects?

- Using an object for each character in the document:
  - Gives flexibility
  - Characters and embedded elements could be treated uniformly with respect to how they are drawn and formatted
  - The application could be extended to support new character sets without disturbing other functionality
  - The application's object structure could mimic the document's physical structure

**logically:** **physically:**




- Creating a flyweight for each letter of the alphabet:
  - Intrinsic state: a character code
  - Extrinsic state: coordinate position in the document
    - typographic style (font, color)
    - Determined from the text layout algorithms and formatting commands in effect wherever the character appears

# Consequences

- **"Flyweights" Must Separate Intrinsic/Extrinsic states:**
  - "Flyweight" Representing a Node in a Graph Cannot Point to Parent (Parent must be Passed)
- **"Flyweights" Save Space**
  - More Intrinsic State Yields More Savings
- **"Flyweights" May Introduce CPU Costs**
  - Finding "Flyweights" (Large Pool Increases cost)
  - Computing Extrinsic State (Previously Stored)
  - Transferring the Extrinsic information

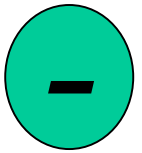
# Flyweight – Implementation

- **Removing Extrinsic State**
  - The pattern's applicability is determined largely by how easy it is to identify the extrinsic state and remove it from shared objects.

- **Managing Shared Objects**
  - Clients shouldn't instantiate the objects directly. Flyweight factory uses an associative store to locate existing objects.
  - Reference counting and Garbage Collection may be needed if an object is no longer in use. This is not necessary if the number of objects is fixed and small.

# Benefits and Consequences

- If the size of the set of objects used is substantially smaller than the number of times the object is logically used
  - There may be an opportunity for a considerable cost benefit



Overhead

- computation

When To Use Flyweight:

- There is a need for many objects to exist that share some intrinsic, unchanging information

When Not To Use Flyweight:

- If the extrinsic properties have a large amount of state information that would need passed to the flyweight (overhead)