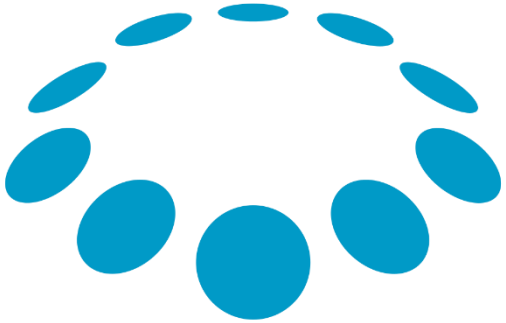
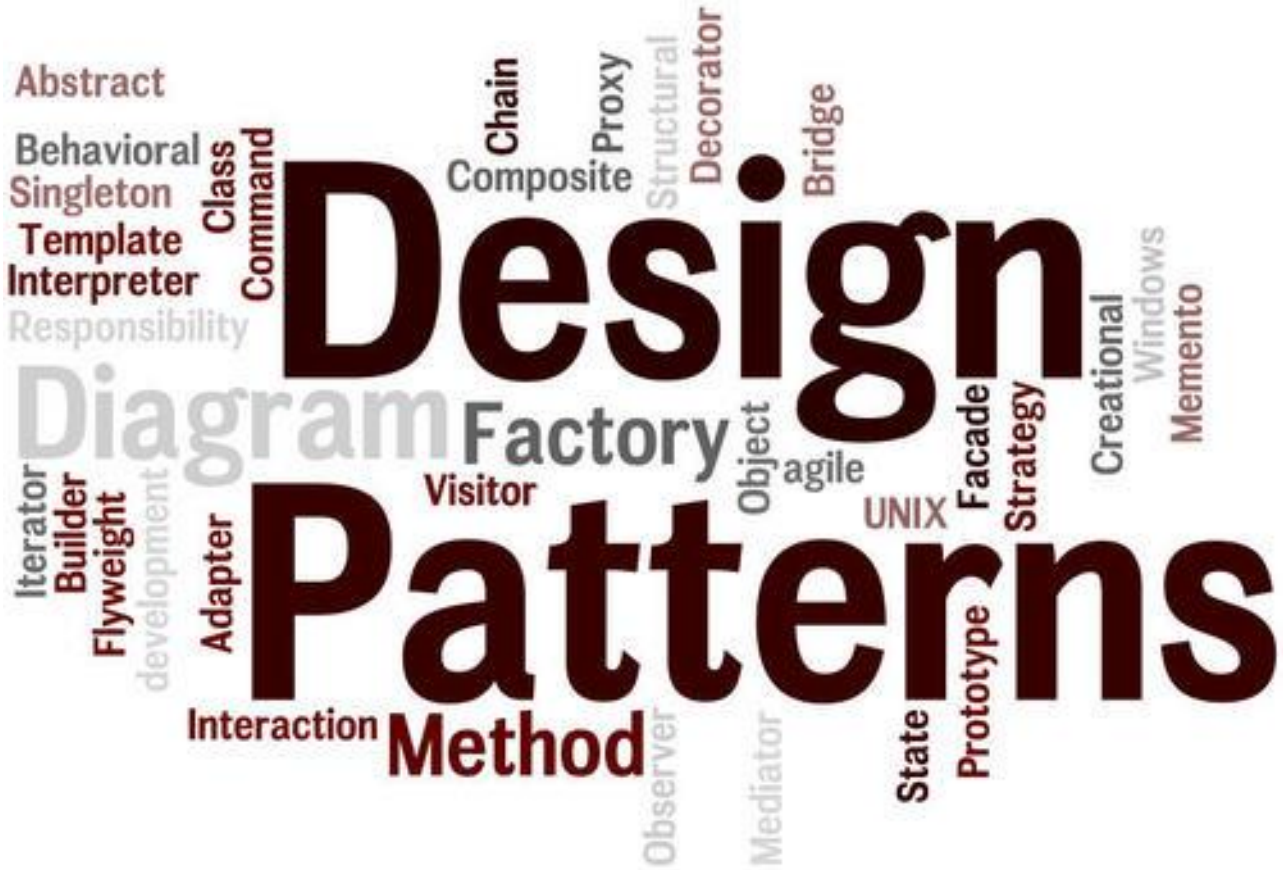
Advanced Software Development 1 – Design Patterns

Dr. Eliahu Khalastchi

2017



המסלול האקדמי
המכללה למינהל

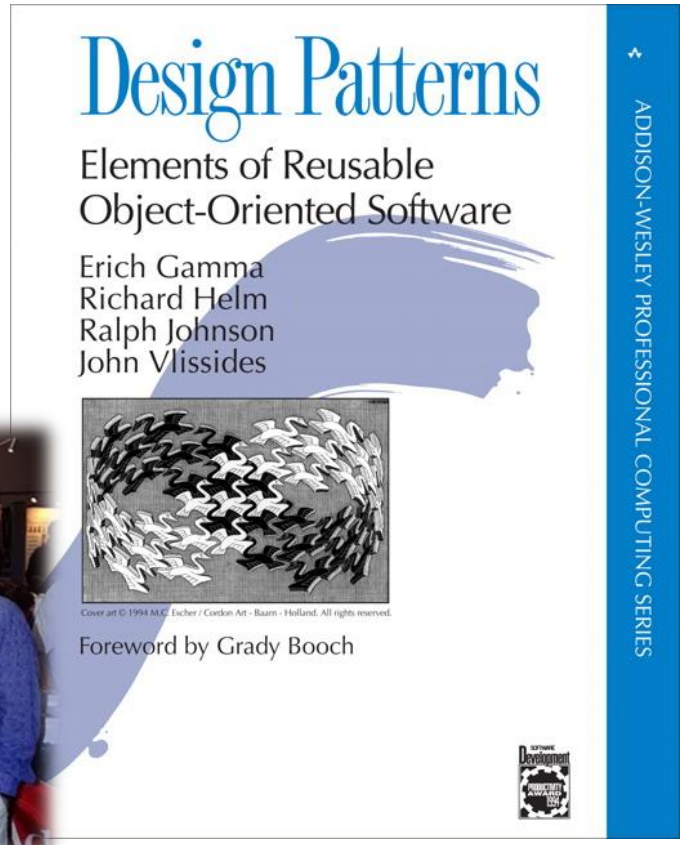

Abstract
Behavioral
Singleton
Template
Interpreter
Responsibility
Class
Command
Chain
Composite
Proxy
Structural
Decorator
Bridge
Design
Diagram
Factory
Visitor
Object
agile
UNIX
Facade
Strategy
Creational
Windows
Memento
Iterator
Builder
Flyweight
development
Adapter
Patterns
Interaction
Method
Observer
Mediator
State
Prototype

Introduction

- A design pattern is a general reusable solution
  - to a commonly occurring problem
  - within a given context in software design
- It is a description or template for how to solve different design problems
- Object-oriented design patterns
  - typically show relationships and interactions between classes or objects

Introduction

- Design patterns were learned from good and bad experience
  - Similar design problems tend to have similar solutions
- Design Patterns help to
  - Reuse successful designs
  - Separate things that change from things that do not change
  - Maintain the code
  - Achieve useful designs quickly



Introduction

- Design Patterns are divided to four main groups:

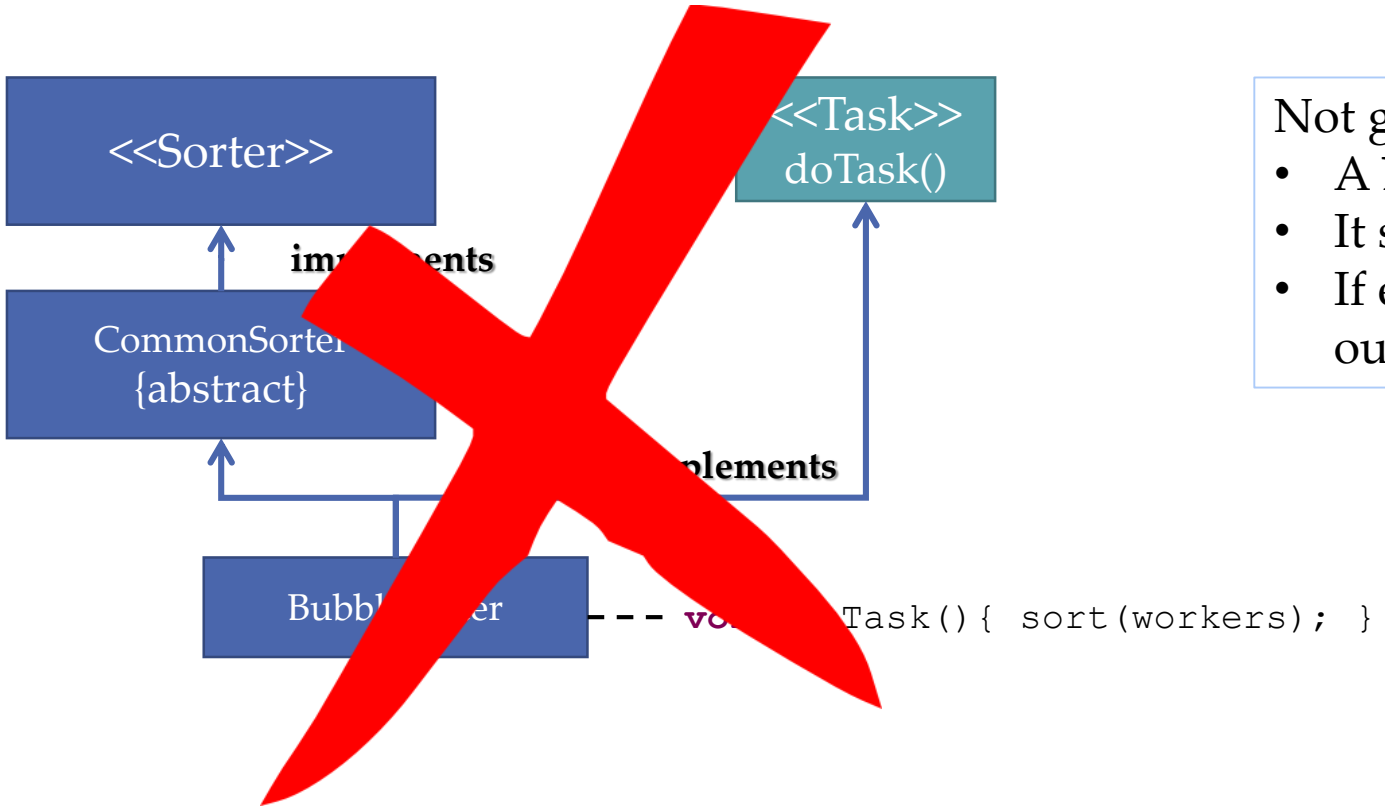
- **Creational Patterns**
  - deal with object creation mechanisms
- **Structural Patterns**
  - realize relationships between entities
- **Behavioral Patterns**
  - identify common communication patterns between objects

- **Concurrency Patterns**
  - deal with the multi-threaded programming paradigm

Structural Design Patterns

Adapter Design Pattern

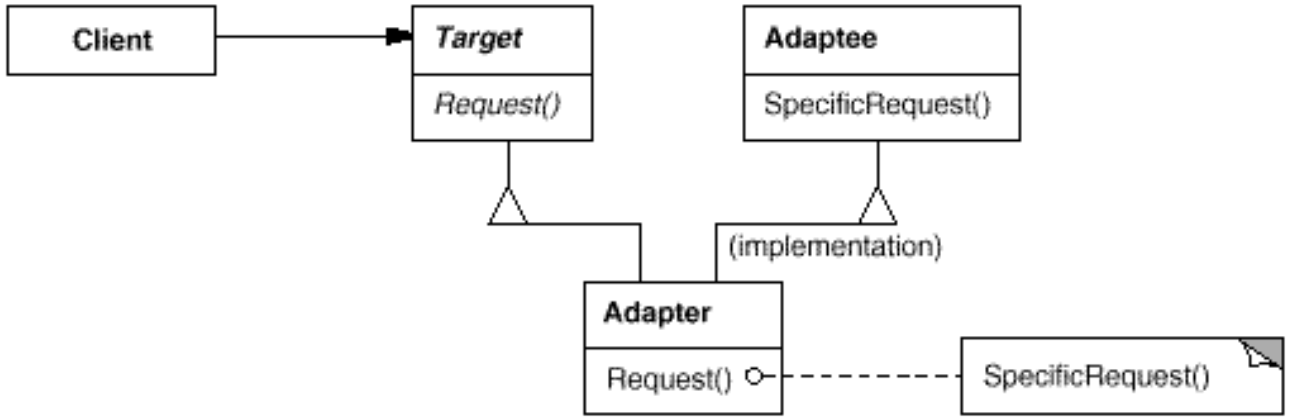
Problem: how to adapt a Sorter to a Task



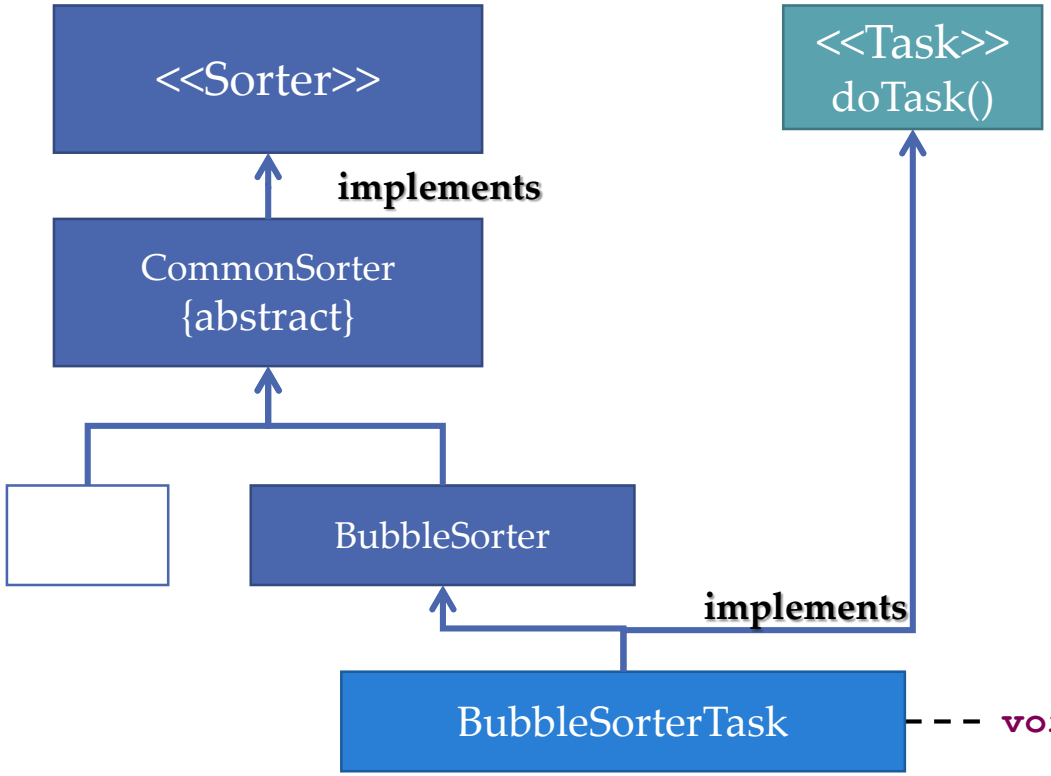
Not good!

- A **BubbleSorter** is not a **Task**!
- It should stay a pure sorter…
- If each interface has 10 methods then our class will have 20!

Class Adapter Pattern



Class adapter

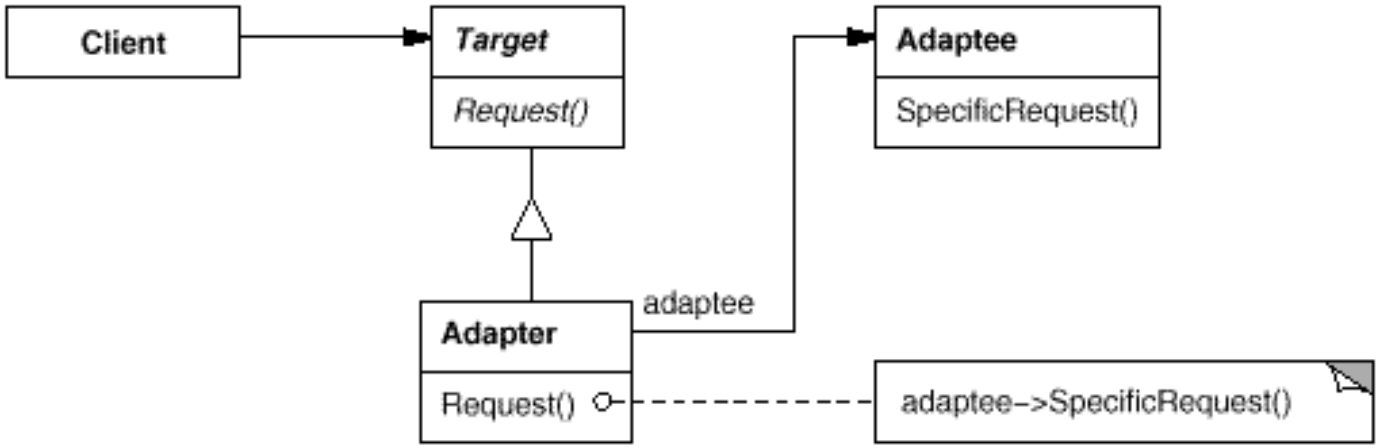


- The **BubbleSorter** stays a pure sorter…
- **BubbleSorterTask** is a **Task**
- **doTask** calls the **inherited sort** method

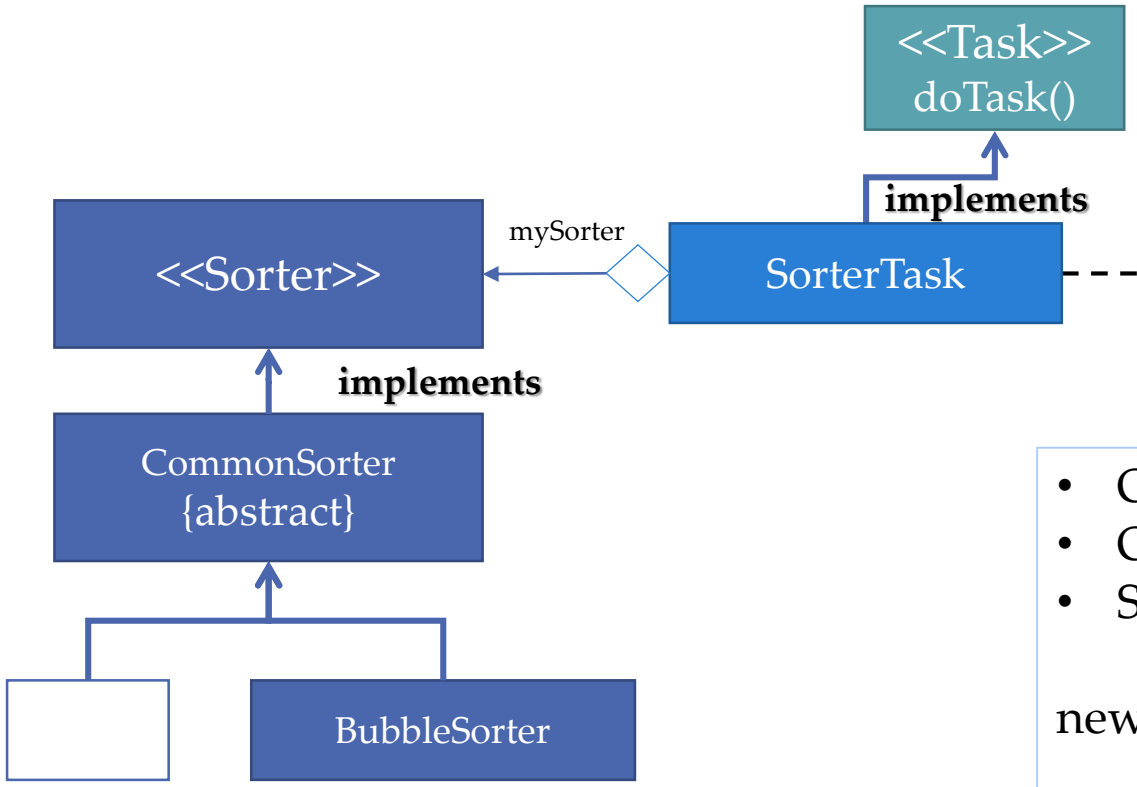
```
new TaskManager().runInBackground(
        new BubbleSorterTask());
```

- Our adapter works only for bubble sort
- We should apply the same solution to each sorter…

Object Adapter Pattern



Solution 3: object adapter

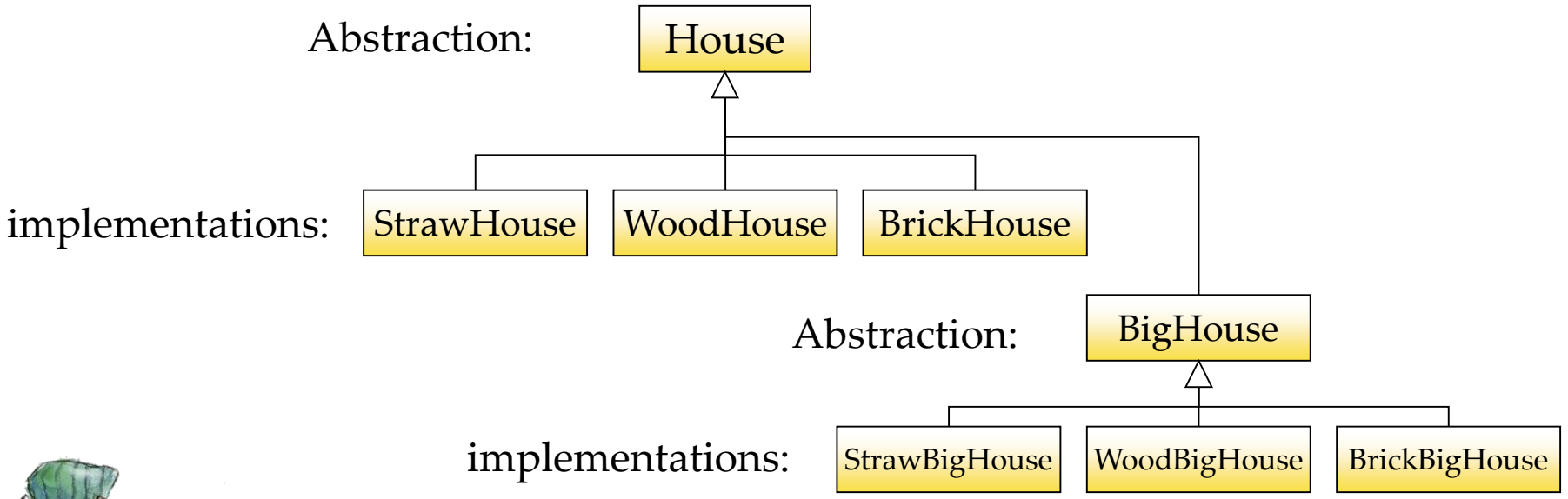


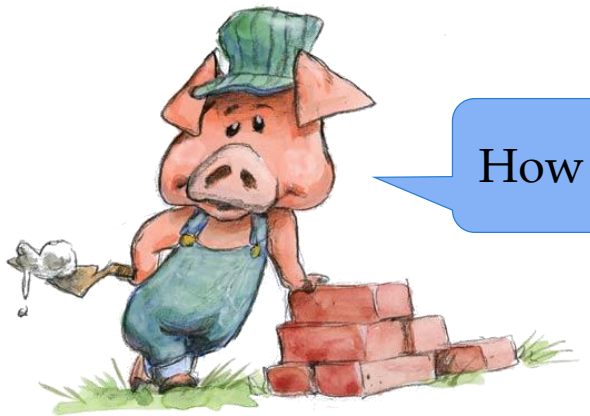
- One adapter to every type of Sorter!
- Our solution is in a higher level
- So it applies to all lower levels! ☺

```
new TaskManager().runInBackground(
        new SorterTask(
                new BubbleSorter()));
```

Bridge Design Pattern

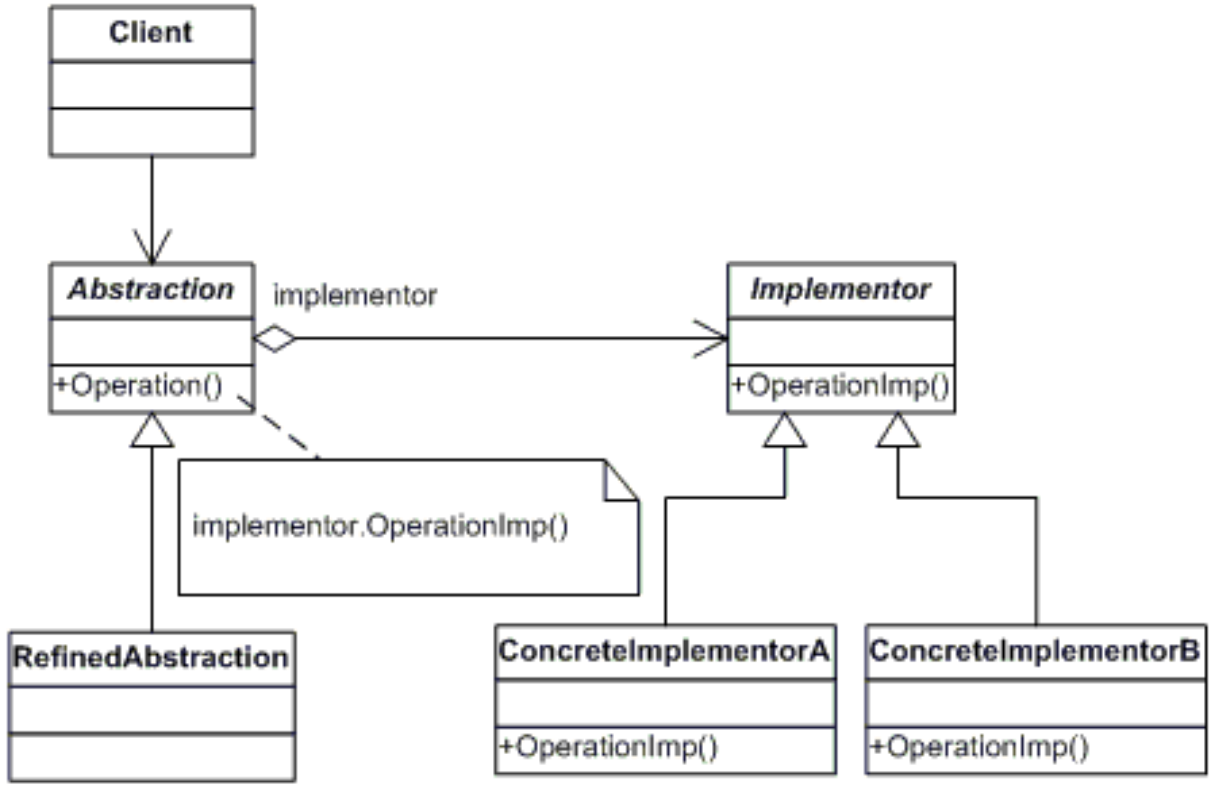
Problem:



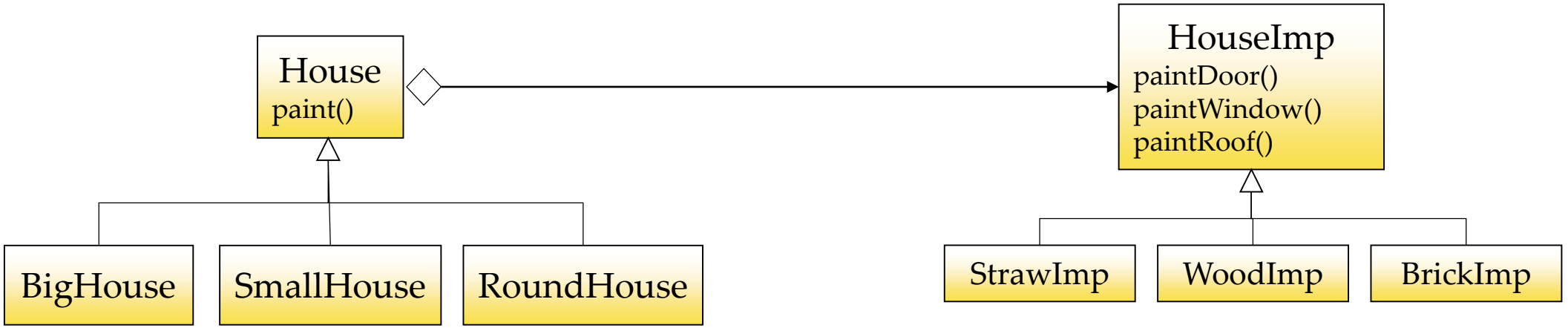


How can we extend abstractions independently of extending implementations?

The Bridge Pattern



The Bridge Pattern



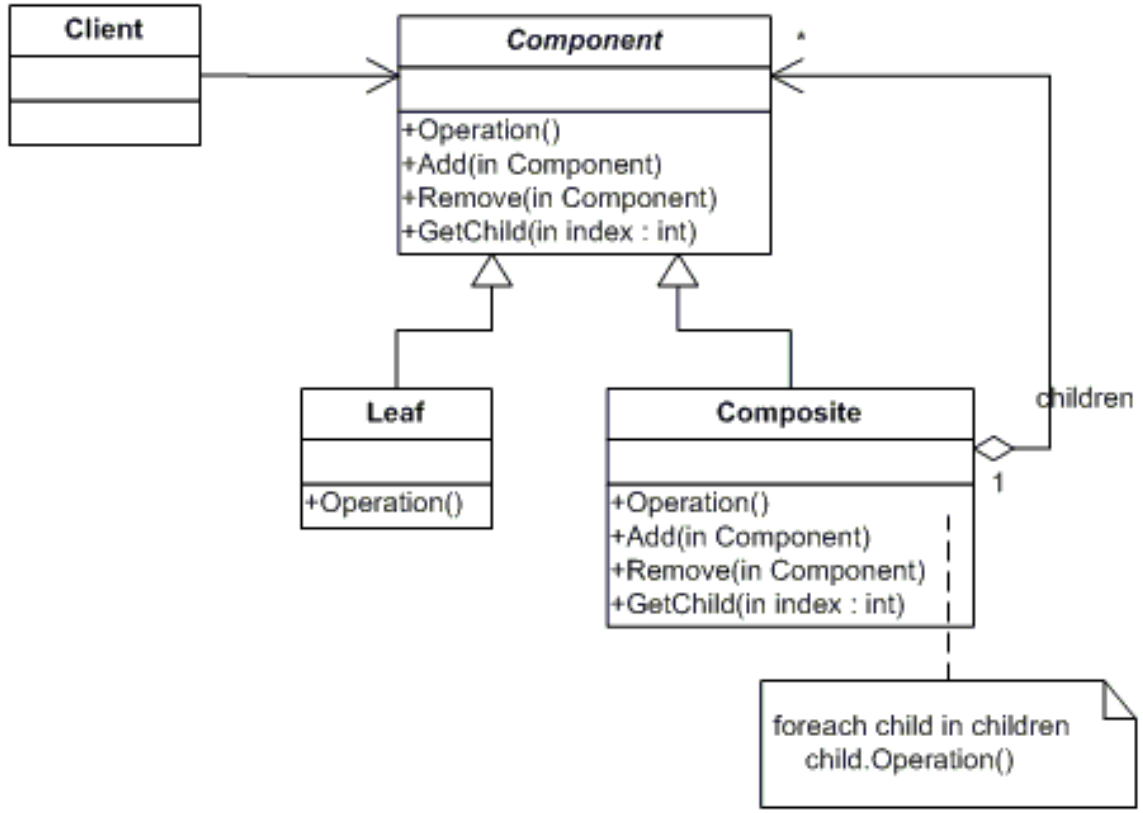
The Bridge Pattern - consequences

- We can **switch** abstractions and implementations in runtime!
- We are encouraged to use **layers** of code
  - A higher layer only knows the abstraction and the implementer
- We can **extend** the abstractions independently of implementations
- We can **extend** the implementations independently of abstractions
- Implementation is **hidden** from the client

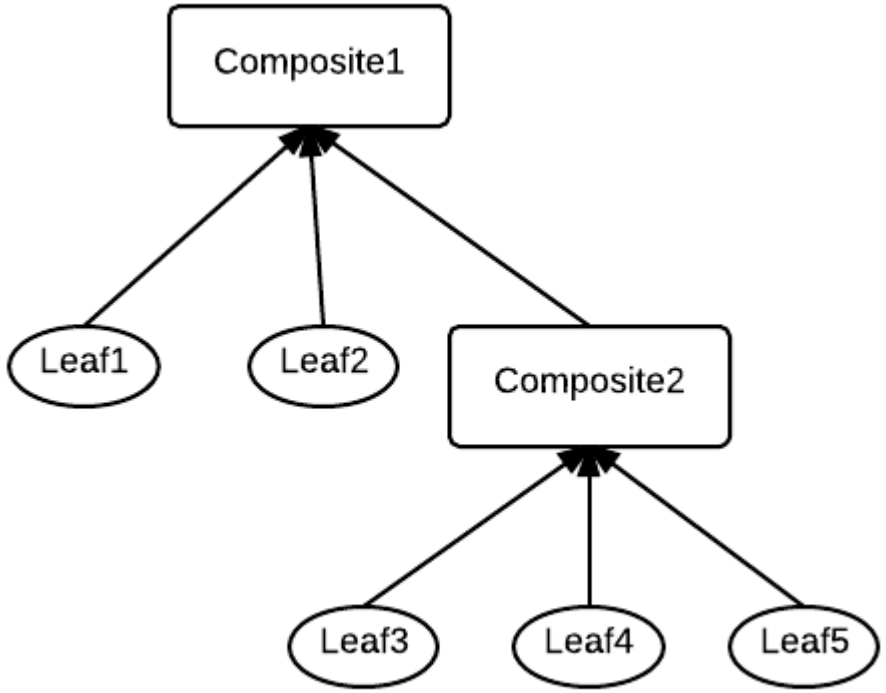
Composite Design Pattern

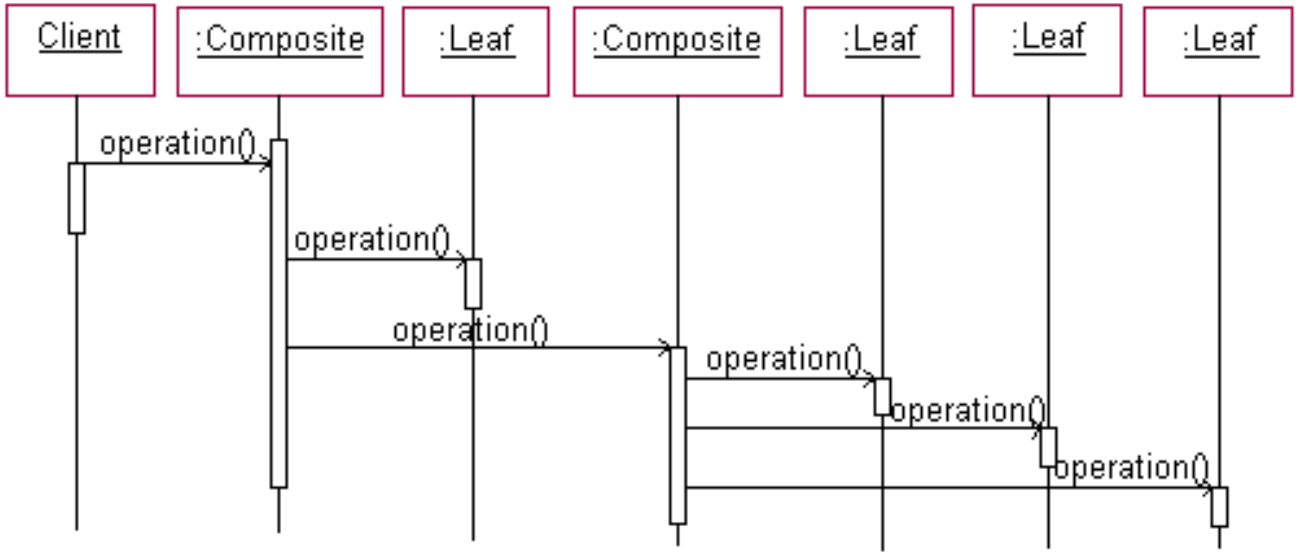
The Composite Design Pattern

- Creates a "Tree"
- Simple to create a recursive hierarchy
- Composite and Leaf are
  - Interchangeable to the client
  - Easy to be added

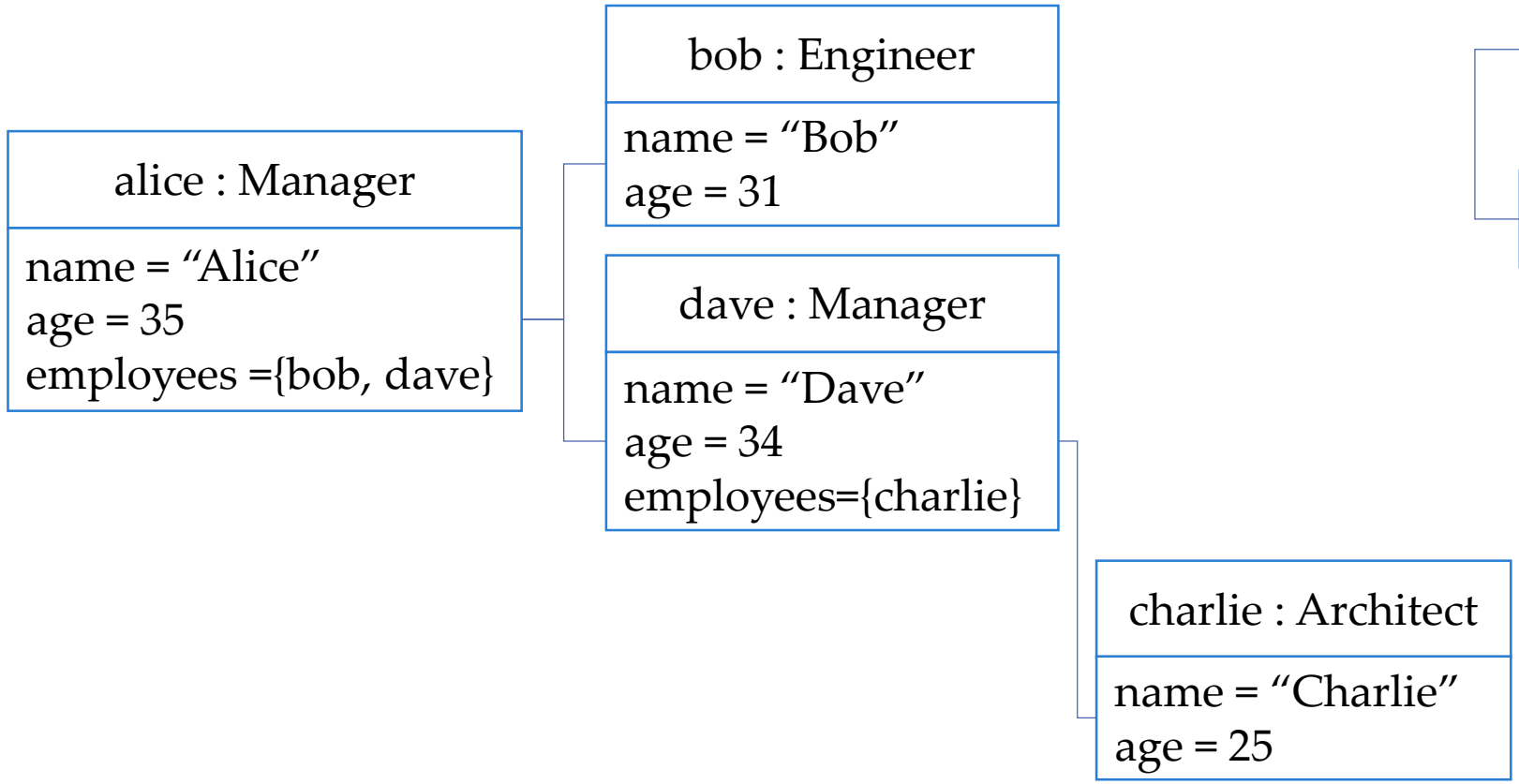


The Composite Design Pattern

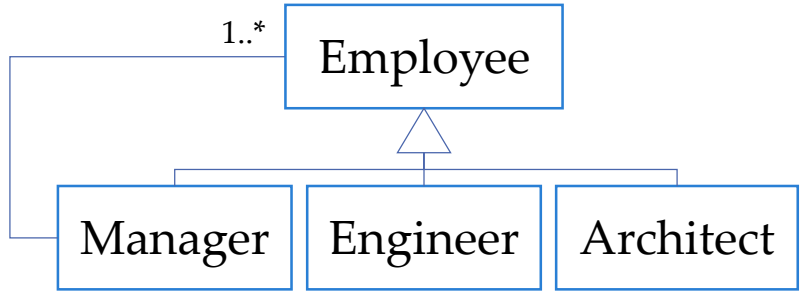




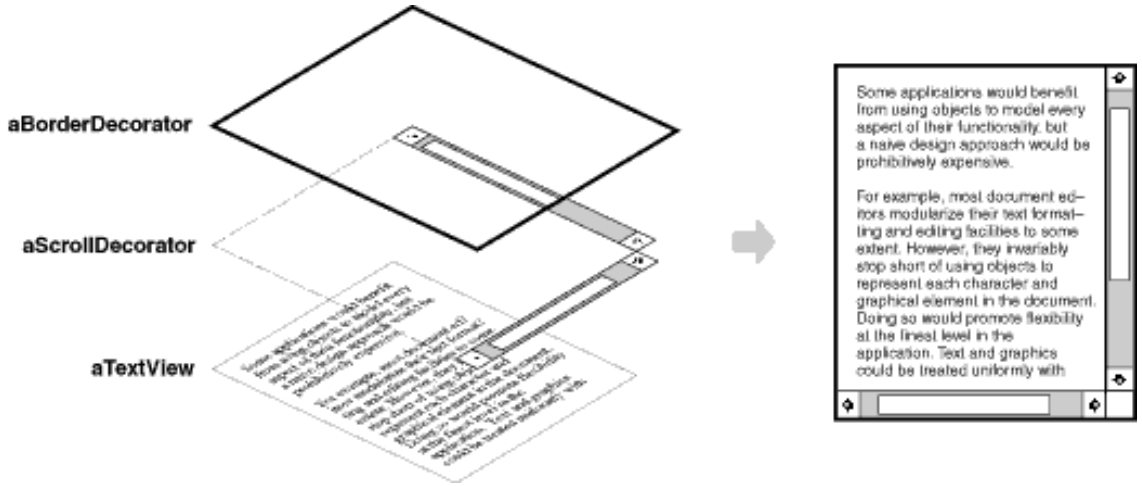
Example

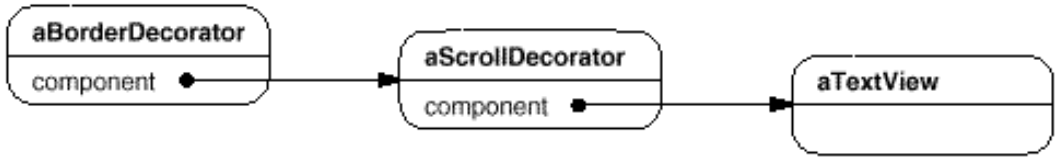


Class diagram:

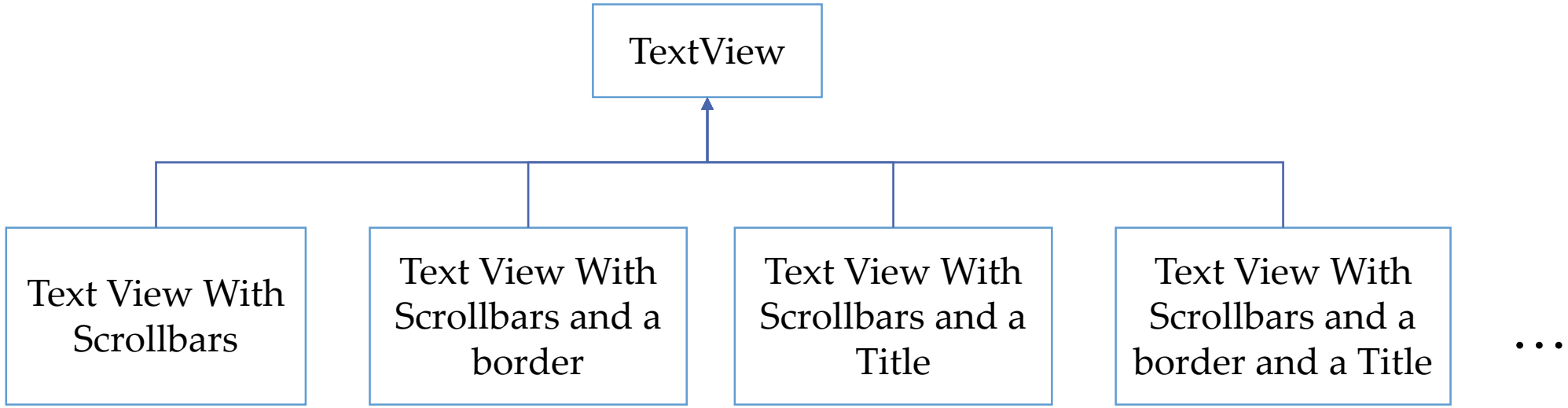


Decorator Design Pattern



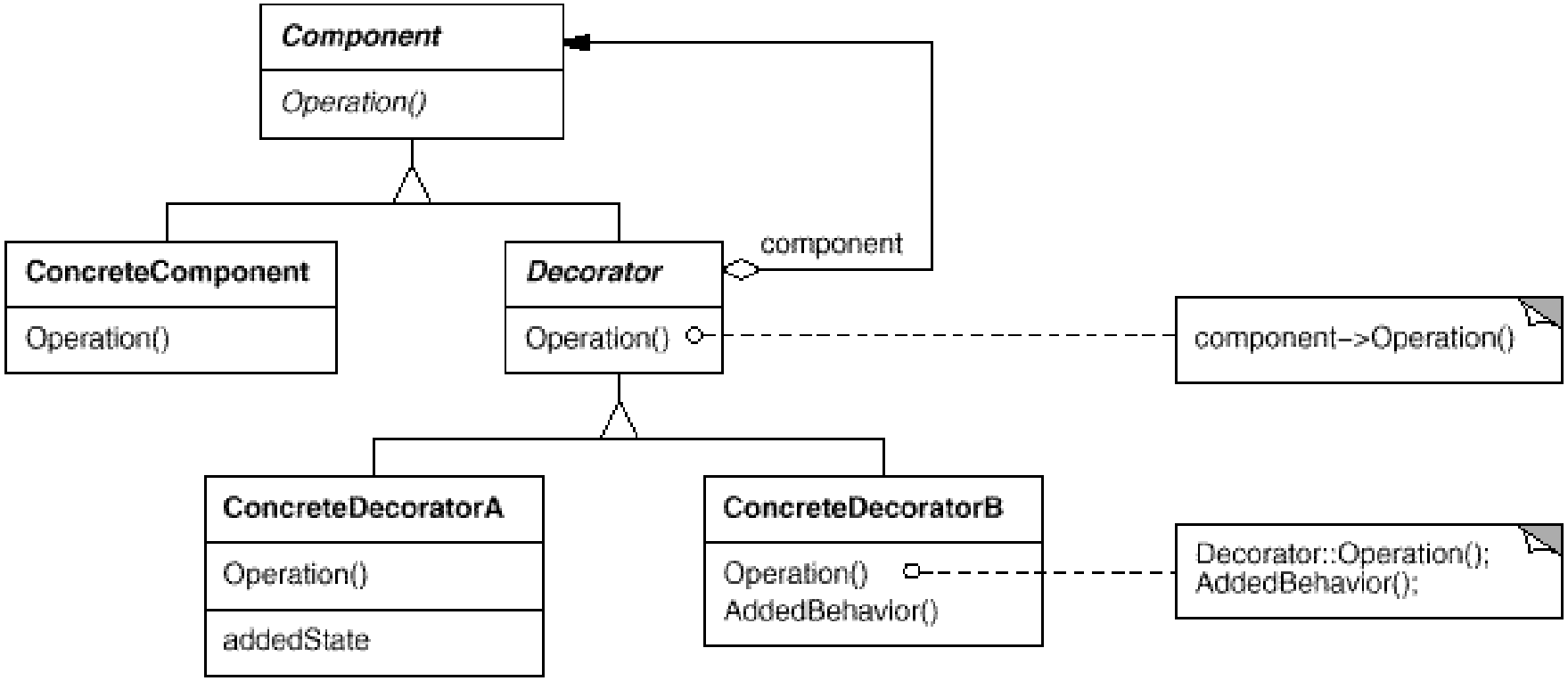


Problem:

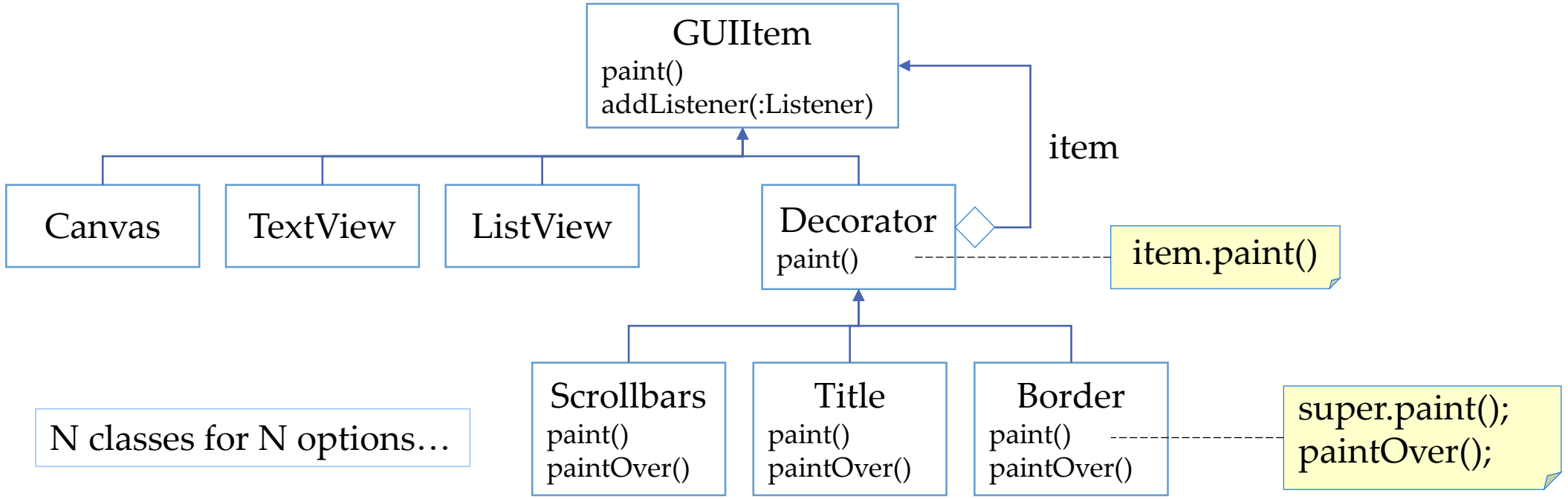


2^N classes for N options…

The Decorator Design Pattern



The Decorator Design Pattern



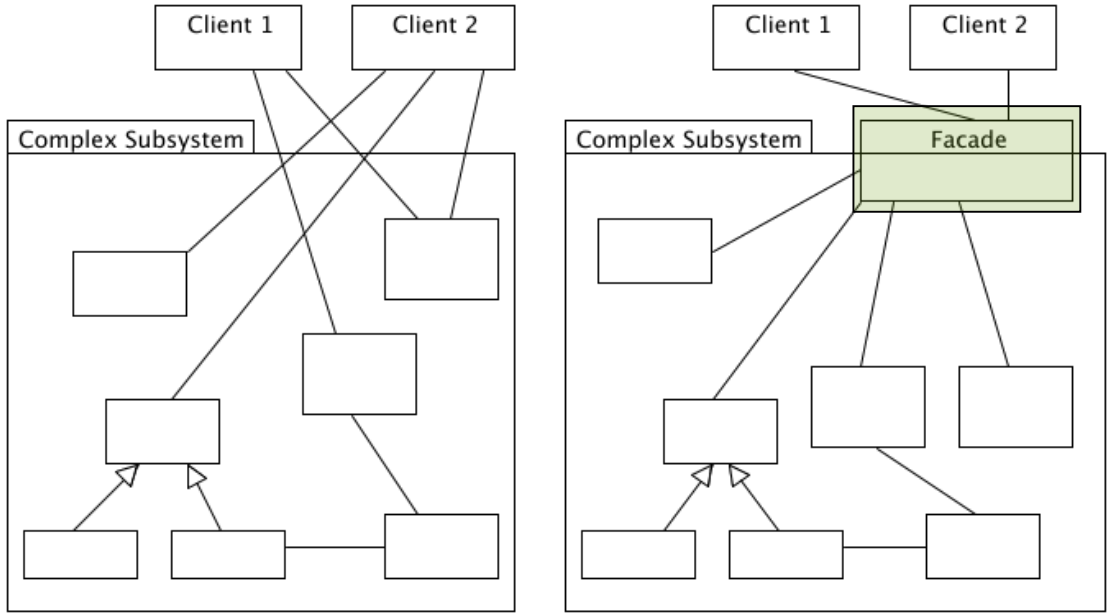

```
GUIItem g1=new Scrollbars(new Border(new TextView()));
GUIItem g2=new Scrollbars(new Border(new Title(new Canvas())));
g1.paint();
g2.paint();
```

The Decorator Design Pattern

- **N classes** instead of 2^N classes
- We can add and **mix** decorations, even add the same decorations twice
- With inheritance you create all the options in advance,
- With Decorators you create **just what you need** (in runtime)

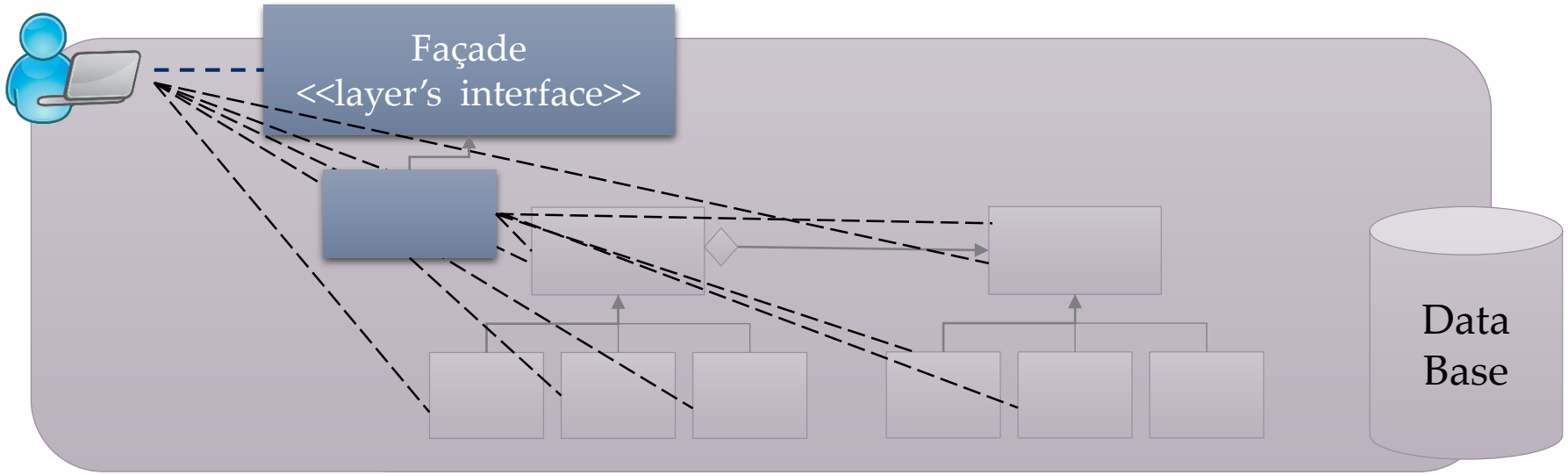
- A lot of small classes
- Easy to maintain to those who understand it, not to others…

Façade


Client 1
Client 2
Complex Subsystem
Client 1
Client 2
Complex Subsystem
Facade

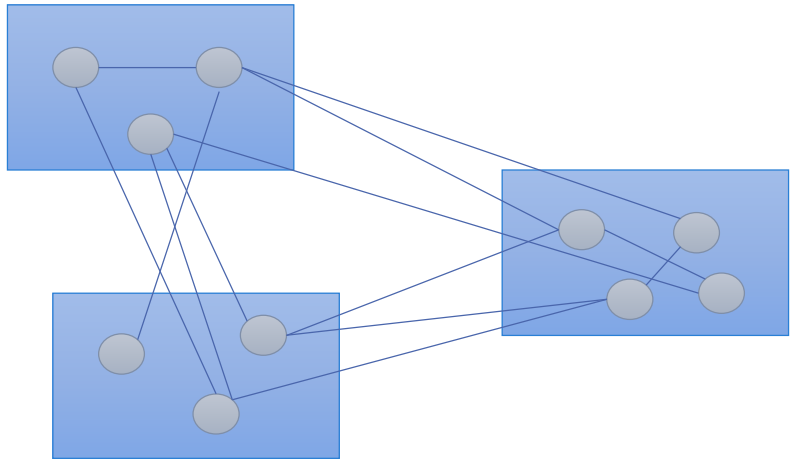
The Façade Design Pattern

- A client can be separated from a layer of code with the use of a façade
- A client can command a layer of code to do something
- The client does not have to know about particulars in the layer

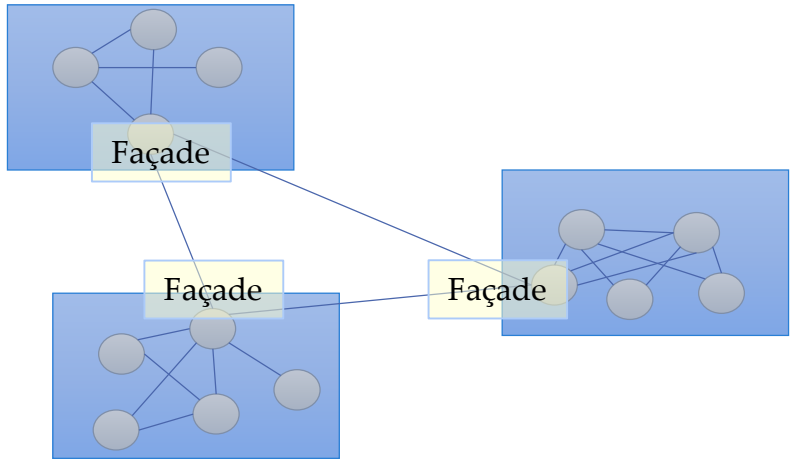


Low Coupling / High Cohesion

- **Coupling:** the level of dependencies between objects
- **Decoupling:** the process of reducing these dependencies
- **Cohesion:** how focused is a class around a single responsibility
- We want high cohesion & low coupling



Low Cohesion, High Coupling



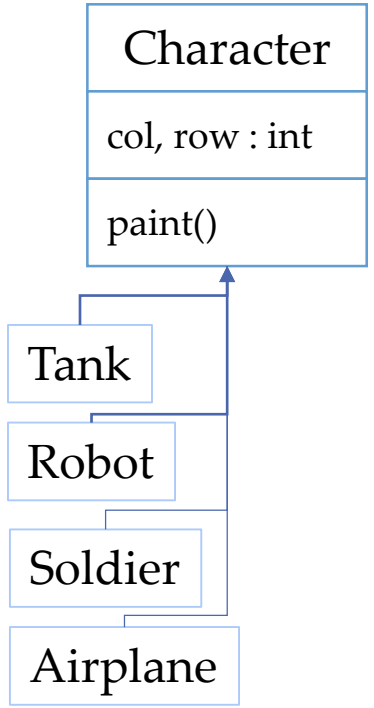
High Cohesion, Low Coupling

Flyweight Design Pattern

Problem:

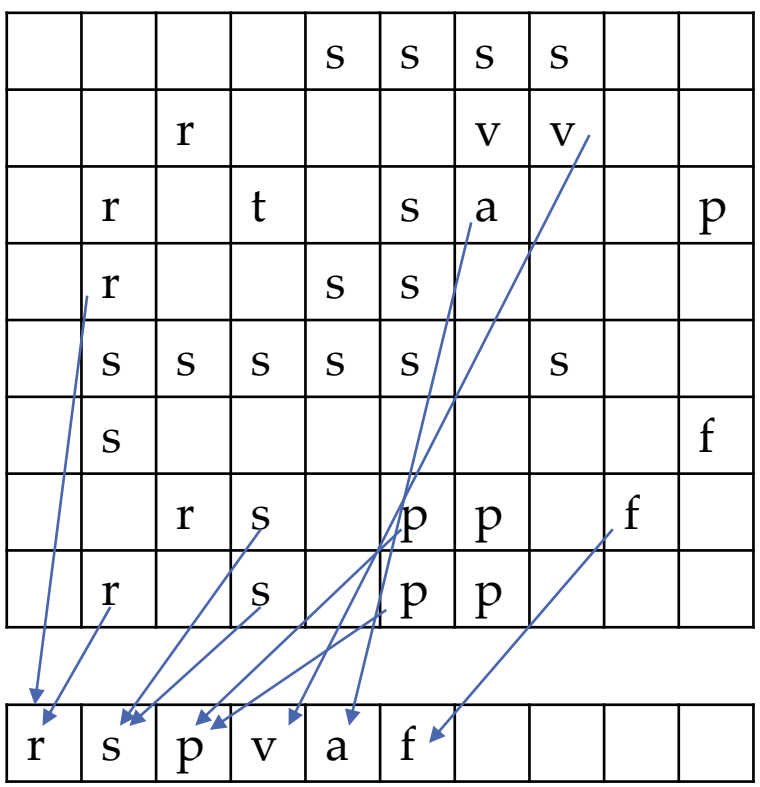
- How can we minimize the number of objects?

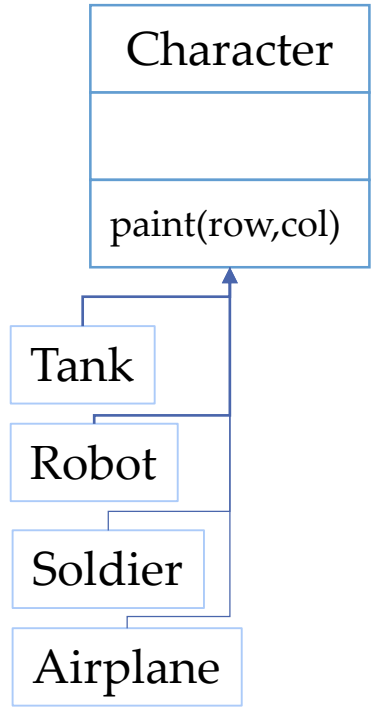
				s	s	s	s		
		r				v	v		
	r		t		s	a			p
	r			s	s				
	s	s	s	s	s		s		
	s								f
		r	s		p	p		f	
	r		s		p	p			



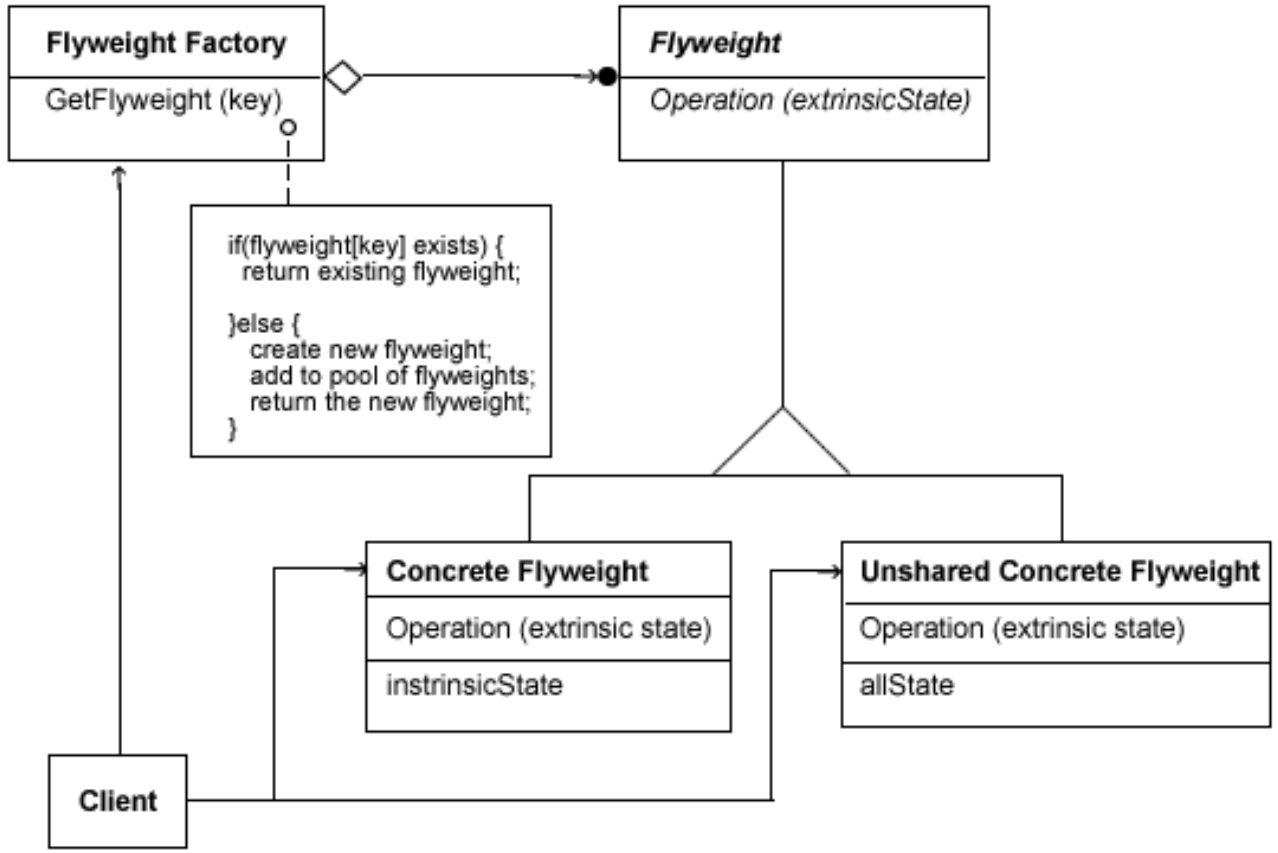
A solution

- A state becomes a parameter
- row, col → parameters of paint()
- Each type appears as one object
- A lot of references
  - instead of a lot of objects
- Time over Space…





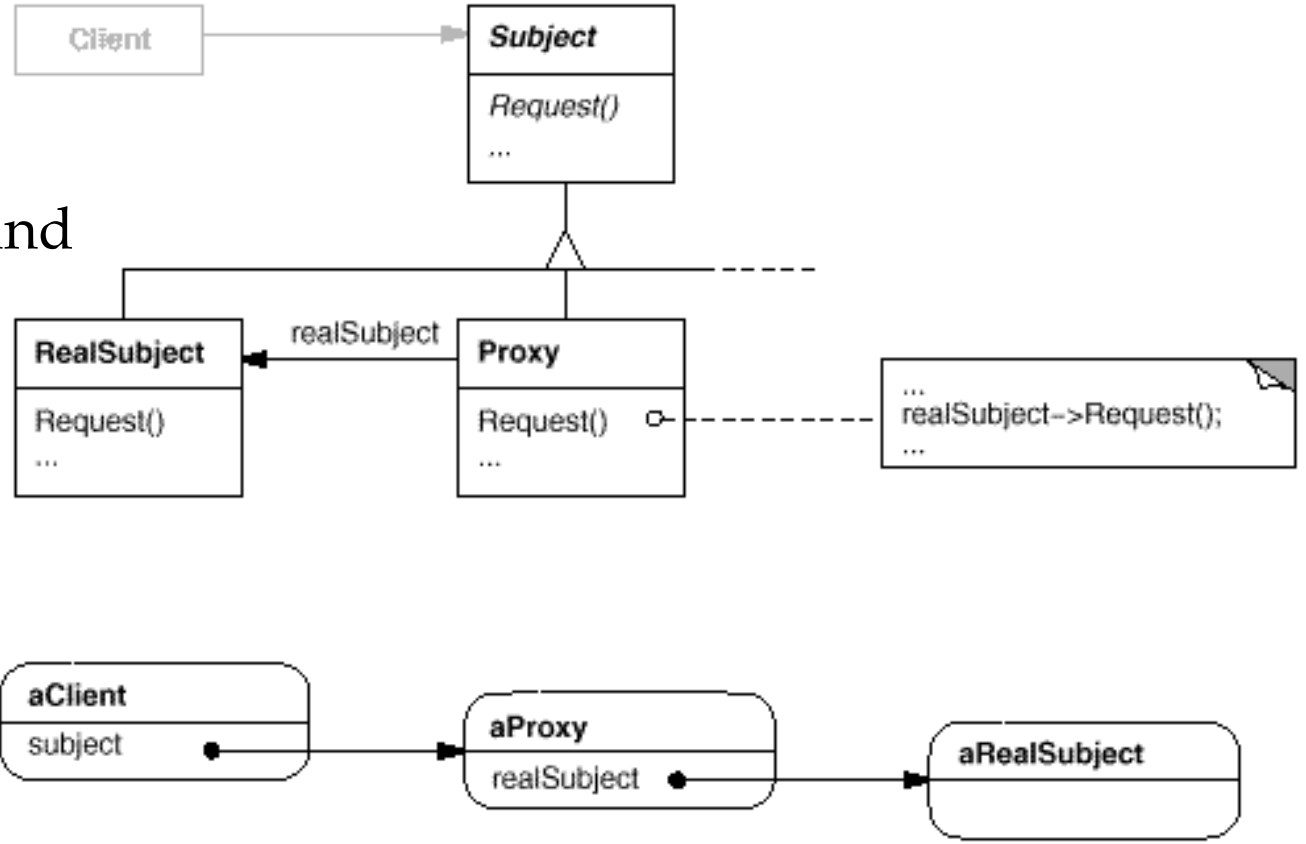
The Flyweight Pattern



Proxy Design Pattern

Proxy Design Pattern

- Governs the access for the real subject
- Remote proxy
  - Applies caching
- Virtual proxy
  - Create “expensive” objects by demand
- Protection proxy
  - Manages access
- Smart reference
  - Count references
  - Manage memory
  - Etc.

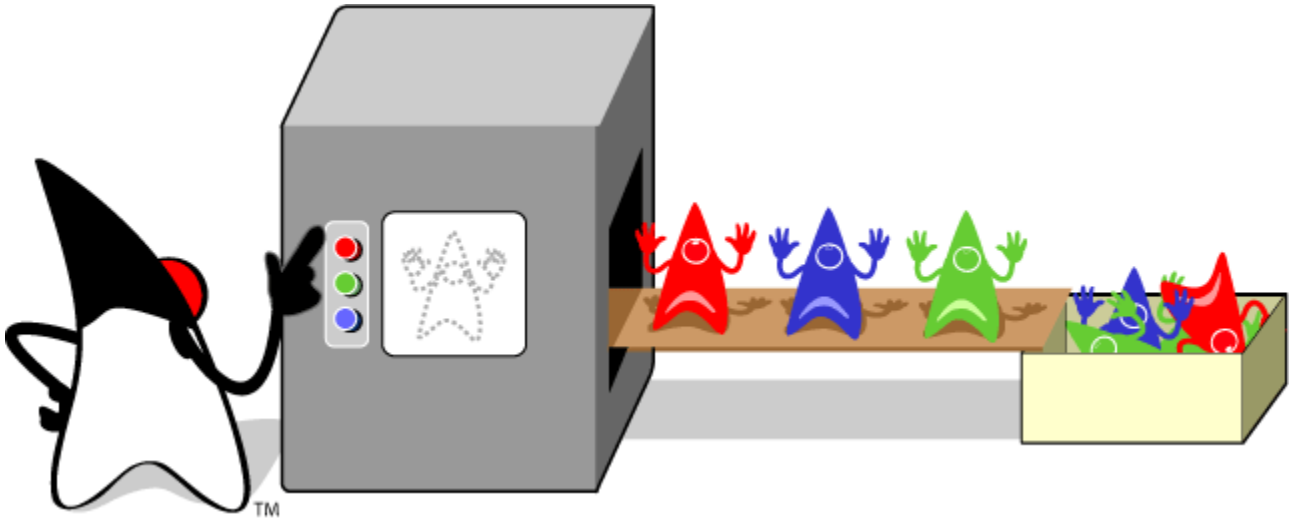


Exercise

1. Why use a façade?
2. Which design pattern separates abstractions from implementations?
3. What are the advantages and drawbacks of Decorator?
4. Does Composite patterns allows the creation of a graph?
5. Which design pattern allows us to manage cache?

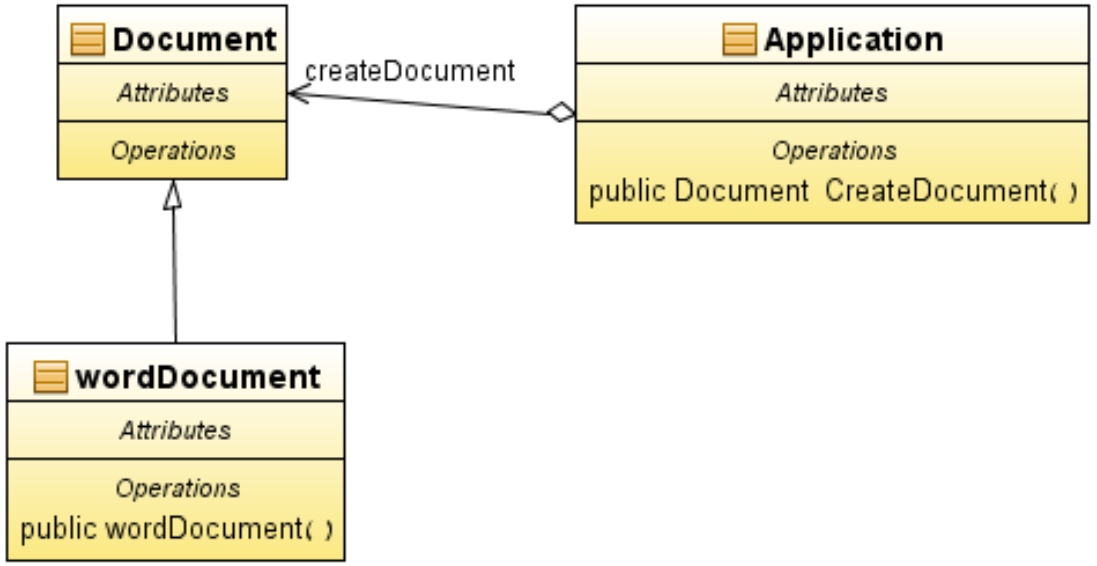
Creational Patterns

Factory Design Pattern



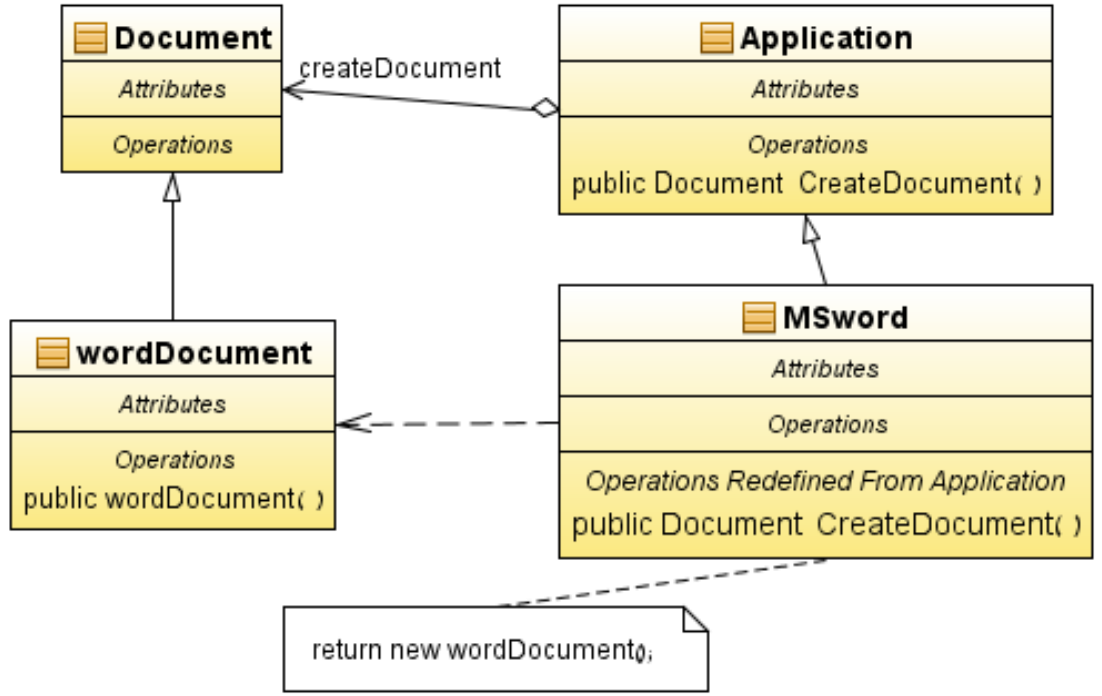
Factory Pattern – the problem

- The Application class knows ***when*** it is required to create a Document
- It does not know ***which type*** of Document should be created



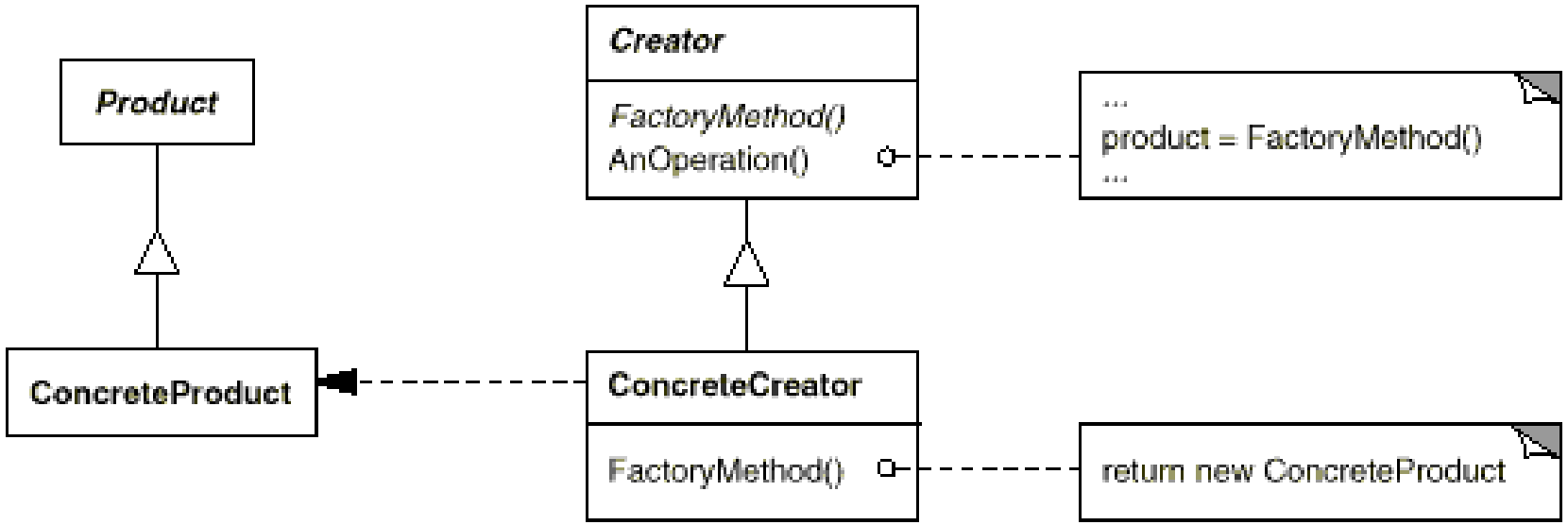
Factory Pattern – the solution

- A concrete Application class creates a concrete Document
  - MSword creates a wordDocument
  - MSexcel creates an excelDocument
  - …

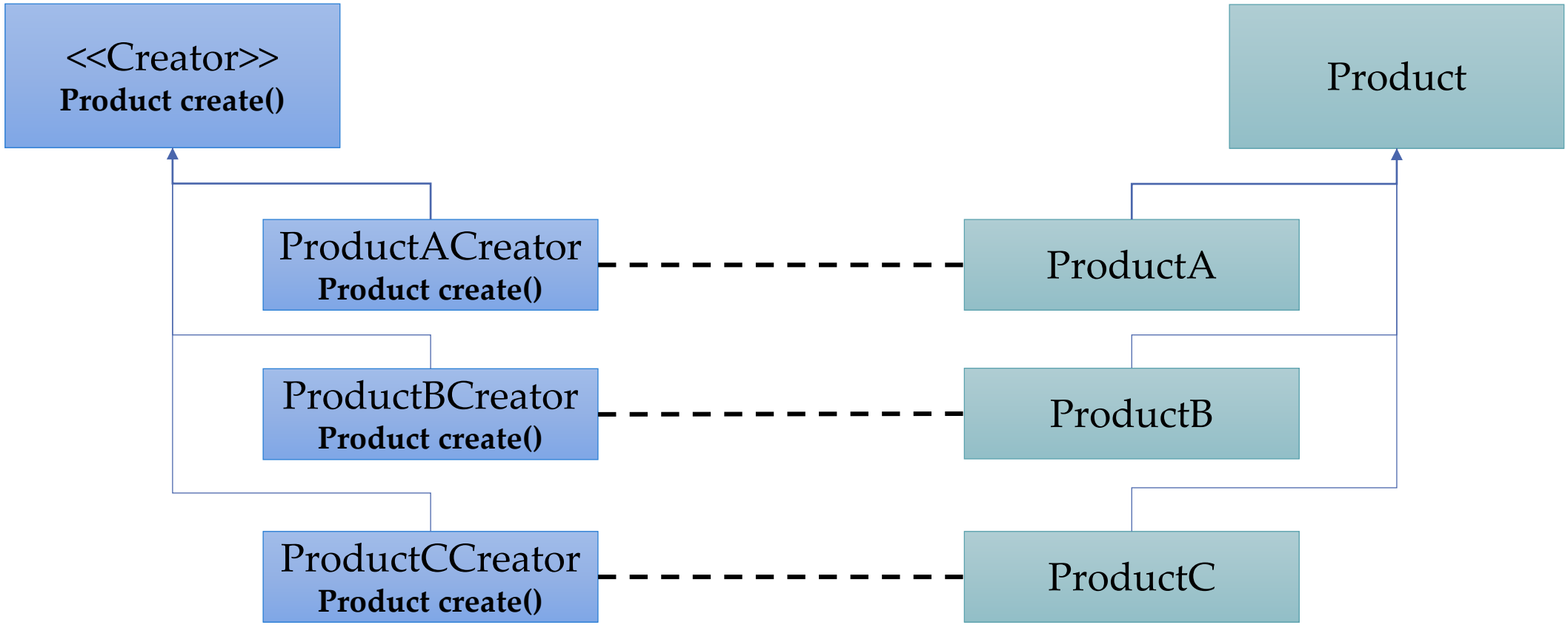


Factory Pattern – the solution

- Generally:

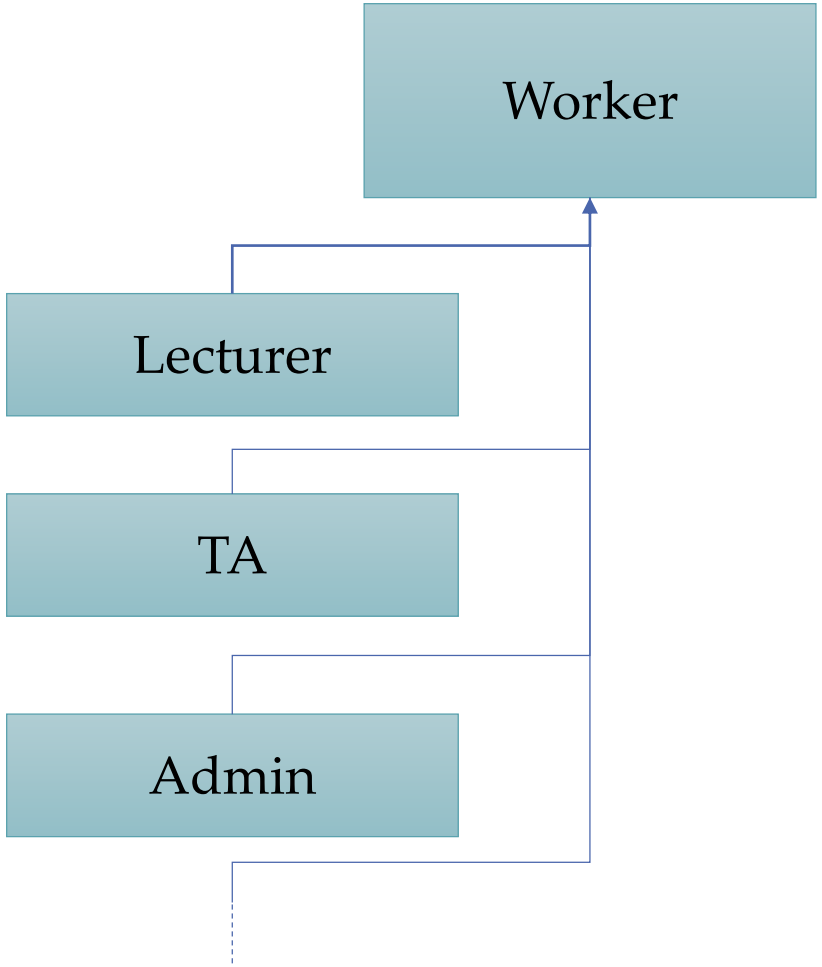


Factory Design Pattern



Quiz…

- Let's say we have *n* types of workers
- And when the user inputs the type, the right object needs to be instanced
- Creating *n* "if" statements takes O(n) time
- It is also not very object oriented…
- Utilize the **factory pattern** and a **container** to return the new worker in O(1)



Factory Pattern

- First, we implement the interface and the classes inside the factory:

- For each type of *Worker*, we create a *Creator* class

```
public class WorkerFactory {
 private interface Creator{
  public Worker create();
 }
 private class AdminCreator implements Creator{
  public Worker create() {
   return new Admin();
  }
 }
 private class TACreator implements Creator{
  public Worker create() {
   return new TA();
  }
 }
 private class LecturerCreator implements Creator{
  public Worker create() {
   return new Lecturer();
  }
 }
...
```

this is a functional interface…

Factory Pattern

- Next, we create a HashMap!
- String→Creator
- The **key** is exactly the user's parameter
- The **value** is a creator
- We instantiate each class once, O(n) mem'

- Notice how *createWorker* takes O(1) to return a new instance of Worker of the given type!

```
HashMap<String,Creator> workersCreators;

public WorkerFactory() {
 workersCreators=new HashMap<String, Creator>();
 workersCreators.put("admin", new AdminCreator());
 workersCreators.put("ta", new TACreator());
 workersCreators.put("lecturer",new LecturerCreator());
 // notice, takes O(n) memory
}

}
```

Factory Pattern

- Next, we create a HashMap!
- String→Creator
- The **key** is exactly the user's parameter
- The **value** is a creator
- We instantiate each class once, O(n) mem'

- Notice how *createWorker* takes O(1) to return a new instance of Worker of the given type!

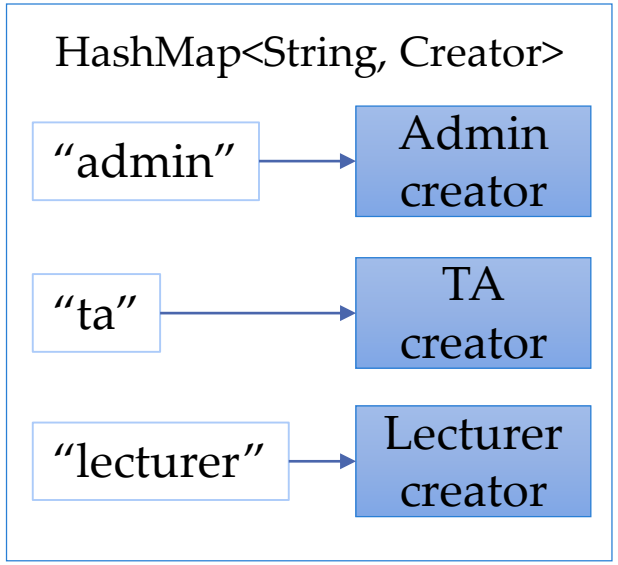
```
HashMap<String,Creator> workersCreators;

public WorkerFactory() {
 workersCreators=new HashMap<String, Creator>();
 workersCreators.put("admin", ()->new Admin());
 workersCreators.put("ta", ()->new TA());
 workersCreators.put("lecturer",()->new Lecturer());
 // notice, takes O(n) memory
}

public Worker createWorker(String type){
 Creator c=workersCreators.get(type);
 // takes O(1) time!
 if(c!=null) return c.create();
 return null;
}
}
```

Factory Pattern

- Usage example:

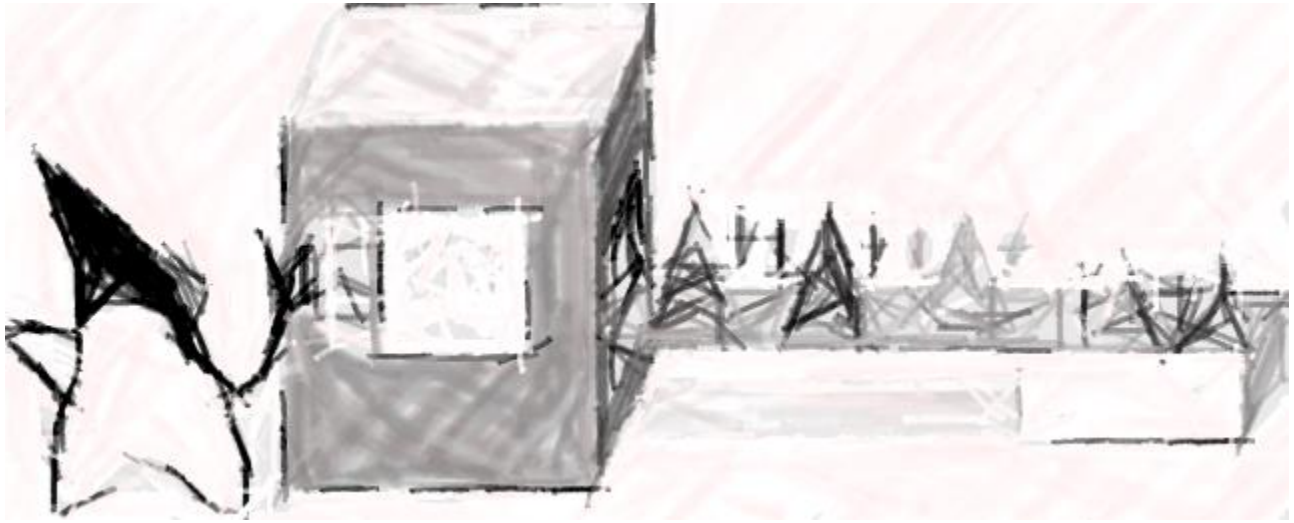
```
WorkerFactory fac=new WorkerFactory();
String userInput;
//...
Worker w=fac.createWorker(userInput);
 if(w!=null)
   System.out.println(w.getClass()+" was created!");
 else
   System.out.println("wrong type of worker!");
}
```

enter types of workers:
admin
class Admin was created!
ta
class TA was created!
ceo
wrong type of worker!
exit



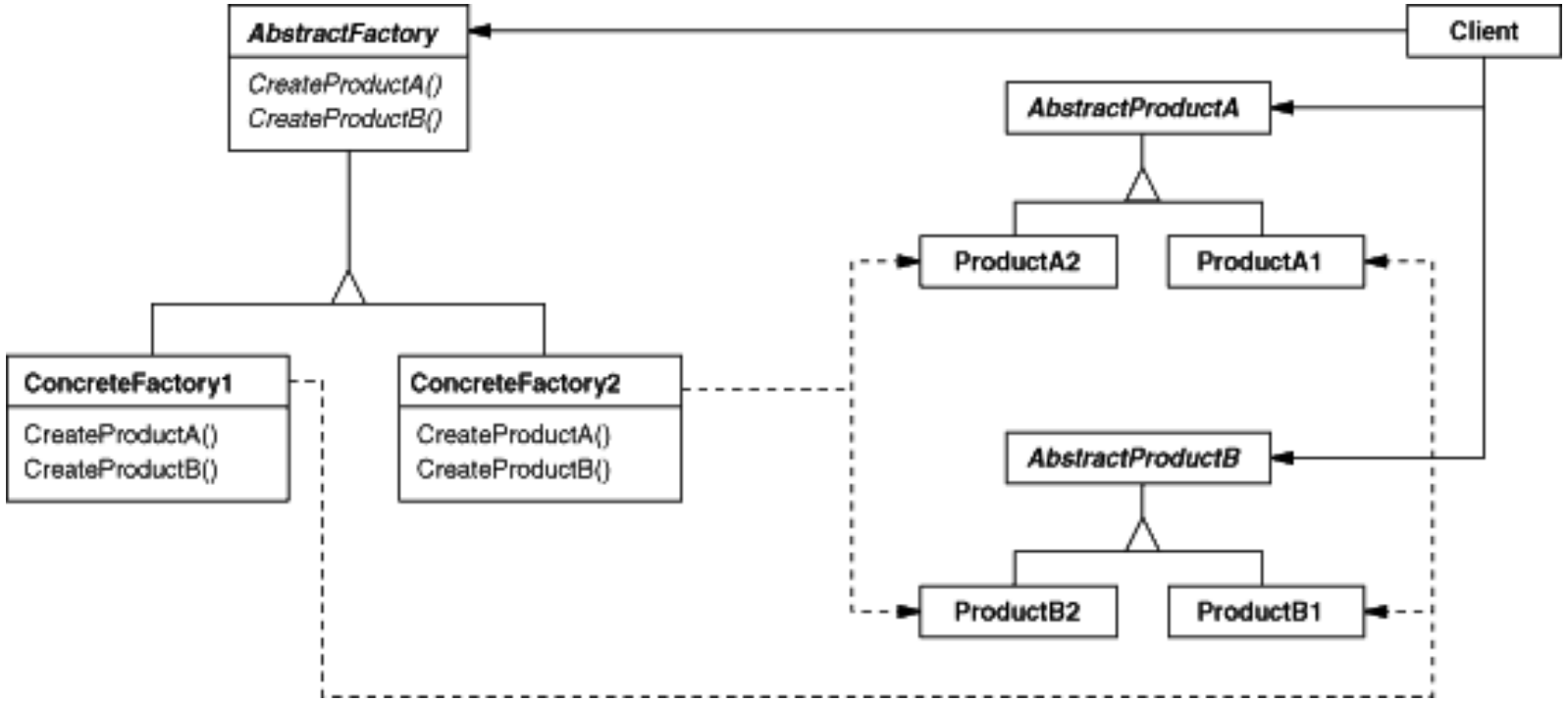

```
public Worker createWorker(String type){
 Creator c=workersCreators.get(type);
 // takes O(1) time!
 if(c!=null) return c.create();
 return null;
}
```

Abstract Factory Design Pattern



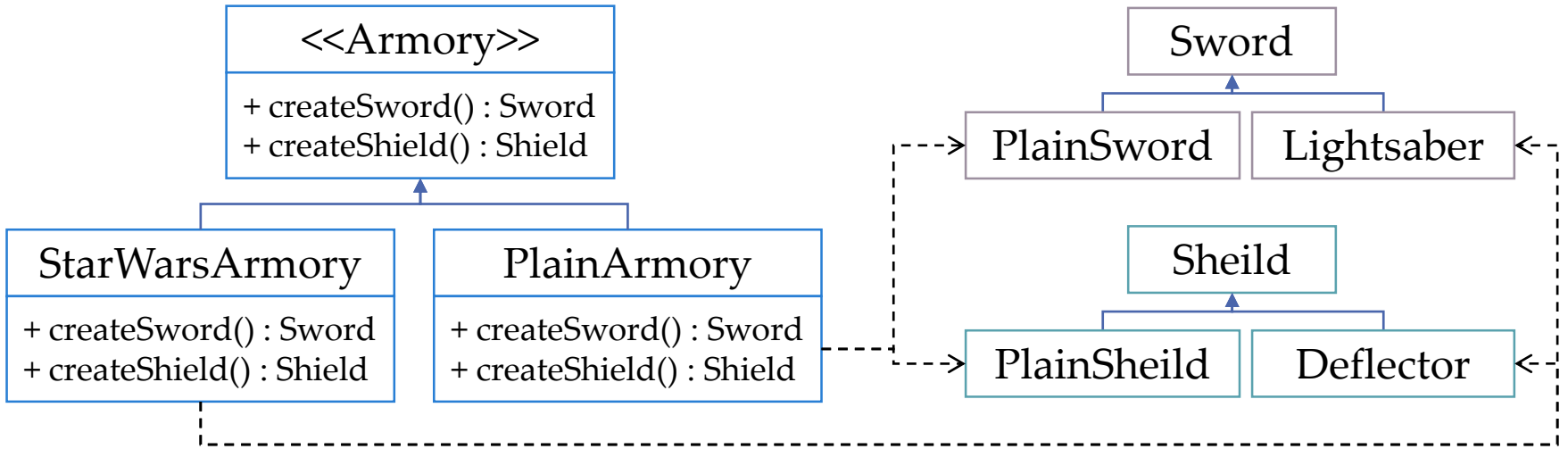
Abstract Factory Pattern

- Provides a way to encapsulate a group of individual factories
- that have a common theme
- without specifying their concrete classes



Abstract Factory Pattern - Example

- The ***star wars*** armory and ***plain*** armory are interchangeable
- Each can return a sword and a shield
  - The star wars armory returns a light saber and a deflector shield
  - The plain armory returns a plain sword and a plain shield



Abstract Factory Pattern – consequences

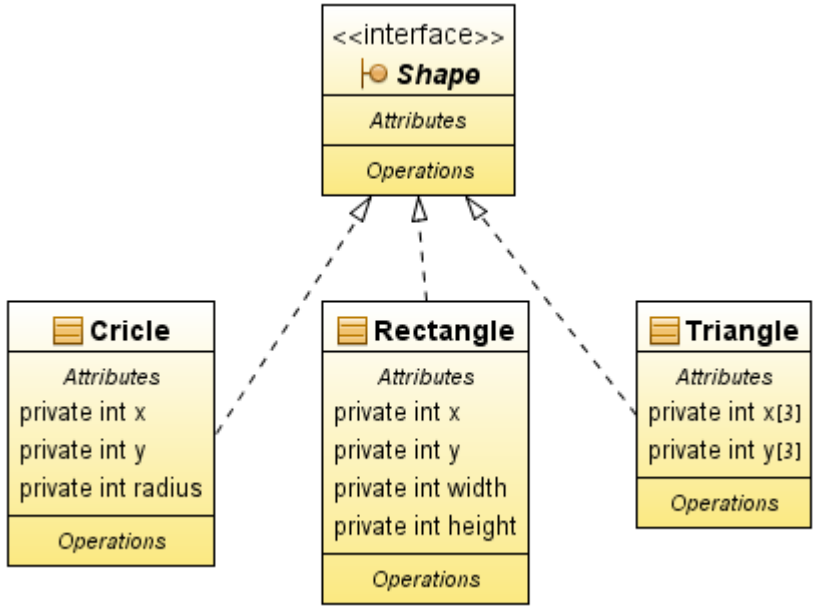
- The client does not have to know bout concrete classes
  - The clients knows an abstract armory, a sword, and a shield
- The abstract factory allows to define “families” of related objects
- These families are interchangeable, yet unmixed

```
Armory a = new StarWarsArmory(); // or new PlainArmory();
Sword sword = a.createSword();
Shield shield = a.createShield();
```

Prototype Design Pattern

Prototype Pattern – the problem

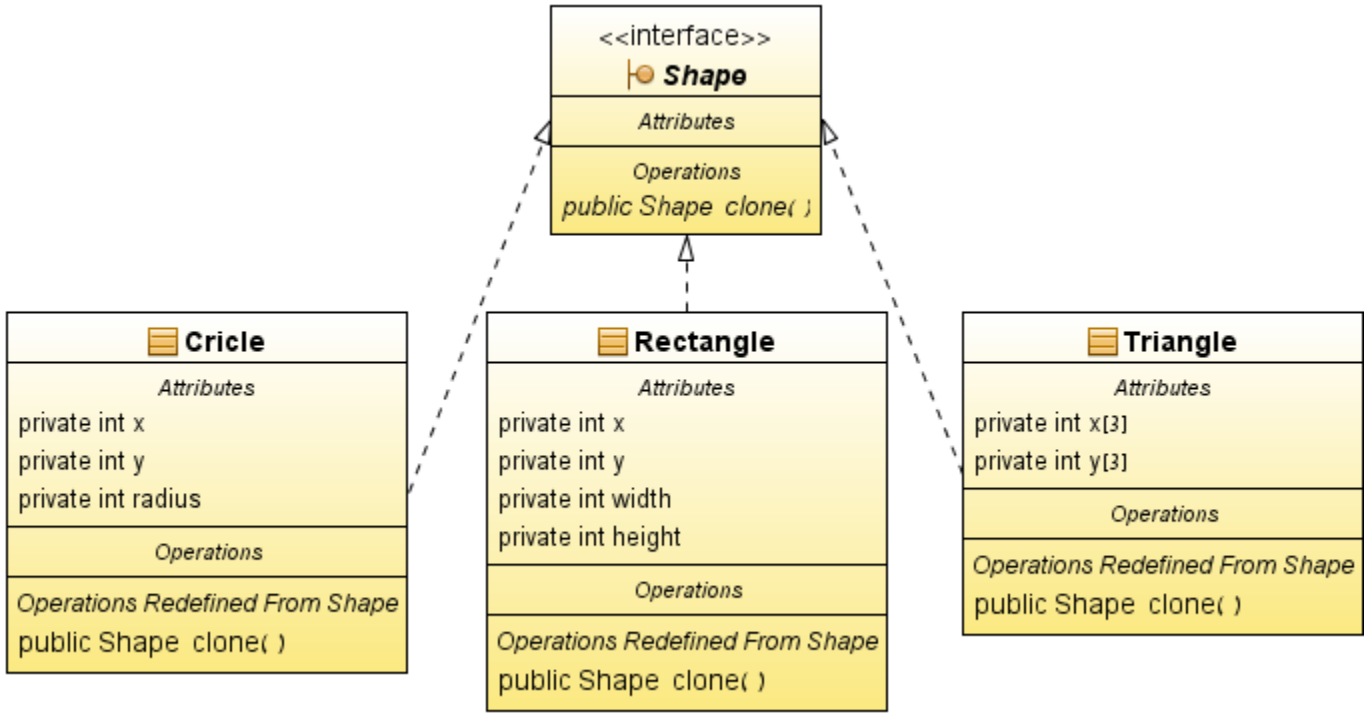
- What do you add?




```
class ShapesHolder{
  ArrayList<Shape> list;
  public void addAcopy(Shape s){
     list.add(new ???);
  }
}
```

Prototype Pattern

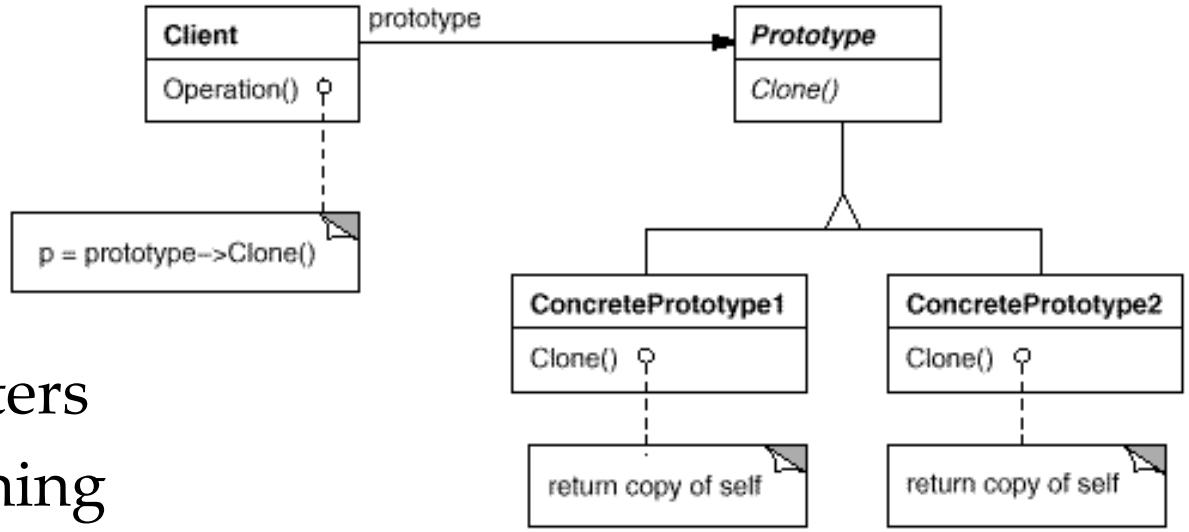
- Each concrete class implements its own close() method




```
class ShapesHolder{
  ArrayList<Shape> list;
  public void addAcopy(Shape s){
     list.add(s.clone());
  }
}
```

Prototype Pattern – consequences

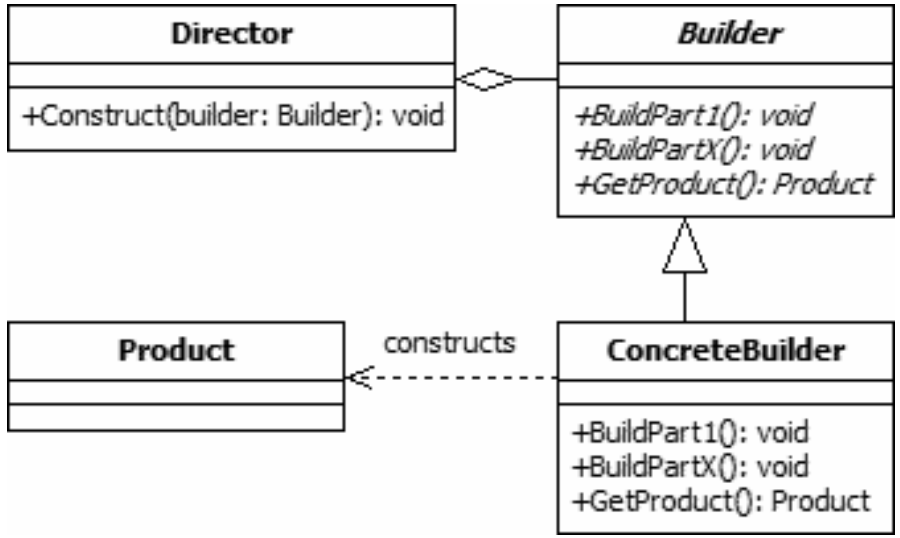
- Advantages
  - Independence of concrete types
  - Quick – we do not have to query the type
- Disadvantages
  - clone() cannot have (many) parameters
  - Setters have to be used after the cloning
  - Shallow vs. deep copy



Builder Design Pattern

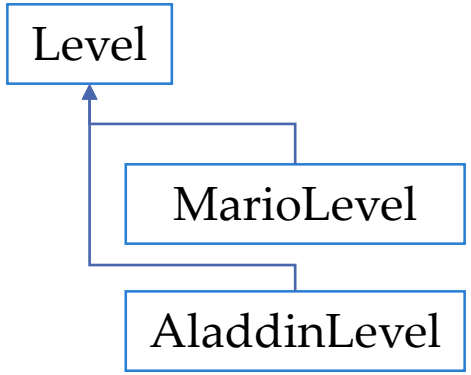
Builder Design Pattern

- The intent is to separate the **construction**
- of a **complex** object
- from its **representation**
- The same construction process can create different representations

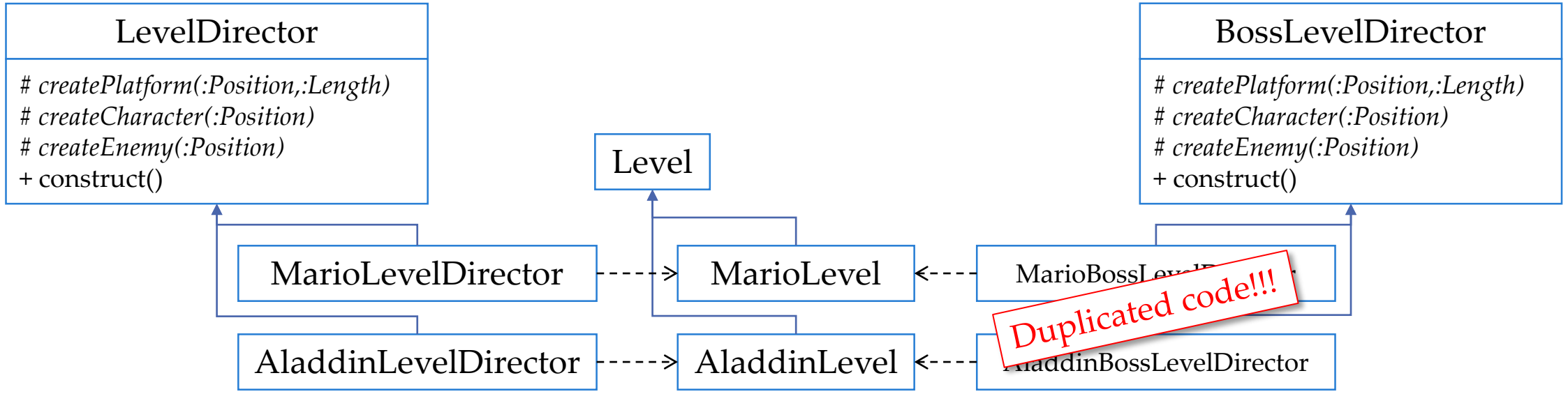


Builder Design Pattern - problem


```
for(int i=0;i<10;i++)
    createPlatform(new Position(d[i].x,d[i].y),d[i].l);

createCharacter(new Position(d[0].x,d[0].y));

Random r=new Random();
for(int i=0;i<100;i++)
   createEnemy(new Position(r.nextInt(100), r.nextInt(100)));
```



Bad solution…



Bad solution…

LevelDirector

+ *createPlatform(:Position,:Length)*
+ *createCharacter(:Position)*
+ *createEnemy(:Position)*
+ *getLevel() : Level*
+ construct()

```
for(int i=0;i<10;i++)
   createPlatform(new Position(d[i].x,d[i].y),d[i].l);

createCharacter(new Position(d[0].x,d[0].y));

Random r=new Random();
for(int i=0;i<100;i++)
  createEnemy(new Position(r.nextInt(100), r.nextInt(100)));
```

NoviceLevelDirector

ExpertLevelDirector

MasterLevelDirector

BossLevelDirector

Level

MarioNoviceLevel

AladdinNoviceLevel

MarioBossLevel

AladdinBossLevel

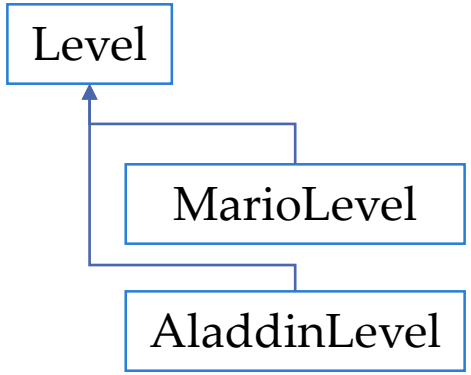
MarioLevel

AladdinLevel

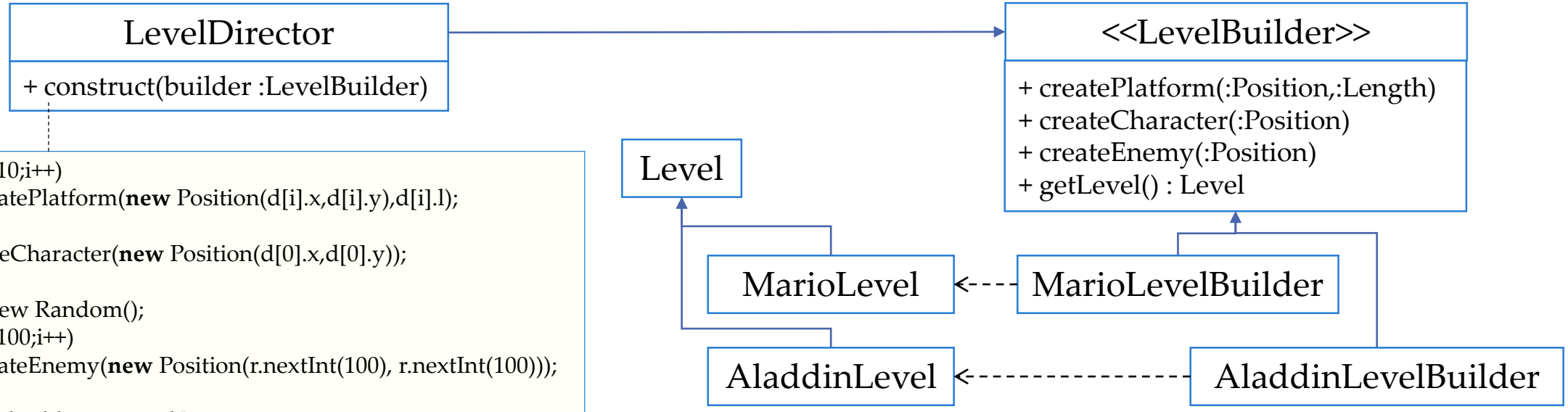
Duplicated code!!!

Builder Design Pattern - problem


```
for(int i=0;i<10;i++)
    createPlatform(new Position(d[i].x,d[i].y),d[i].l);

createCharacter(new Position(d[0].x,d[0].y));

Random r=new Random();
for(int i=0;i<100;i++)
   createEnemy(new Position(r.nextInt(100), r.nextInt(100)));
```

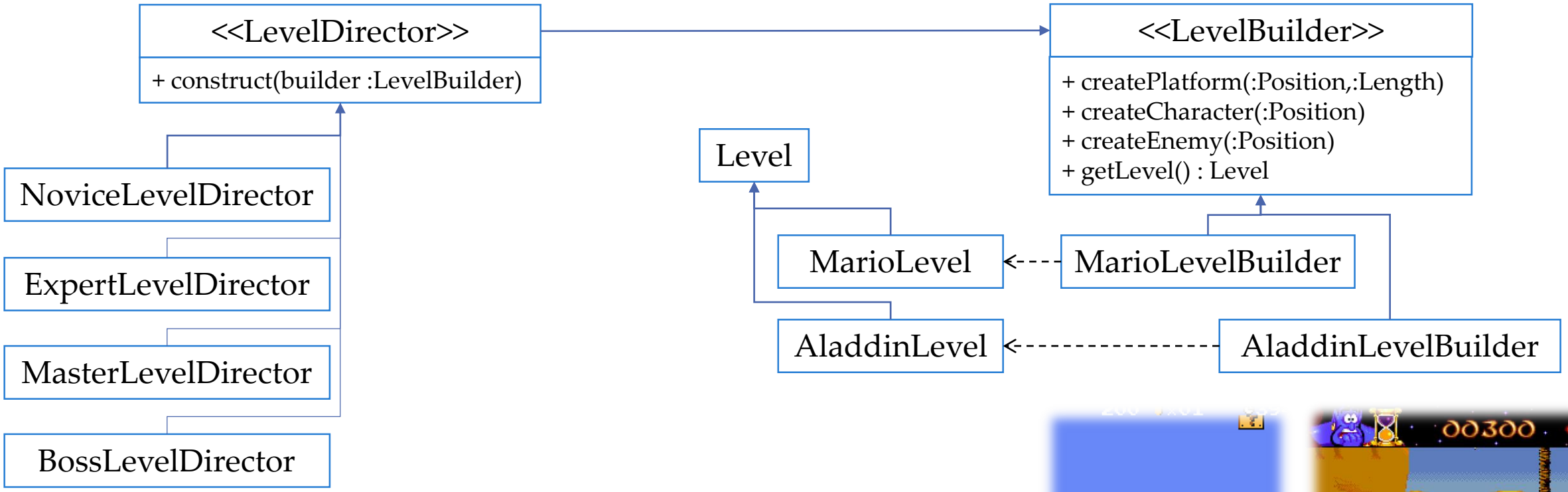


Builder Design Pattern - Solution

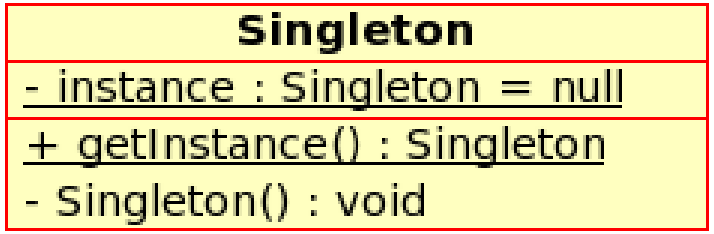



```
for(int i=0;i<10;i++)
  builder.createPlatform(new Position(d[i].x,d[i].y),d[i].l);

builder.createCharacter(new Position(d[0].x,d[0].y));

Random r=new Random();
for(int i=0;i<100;i++)
  builder.createEnemy(new Position(r.nextInt(100), r.nextInt(100)));

Level level = builder.getLevel();
```

Builder Design Pattern - Extended



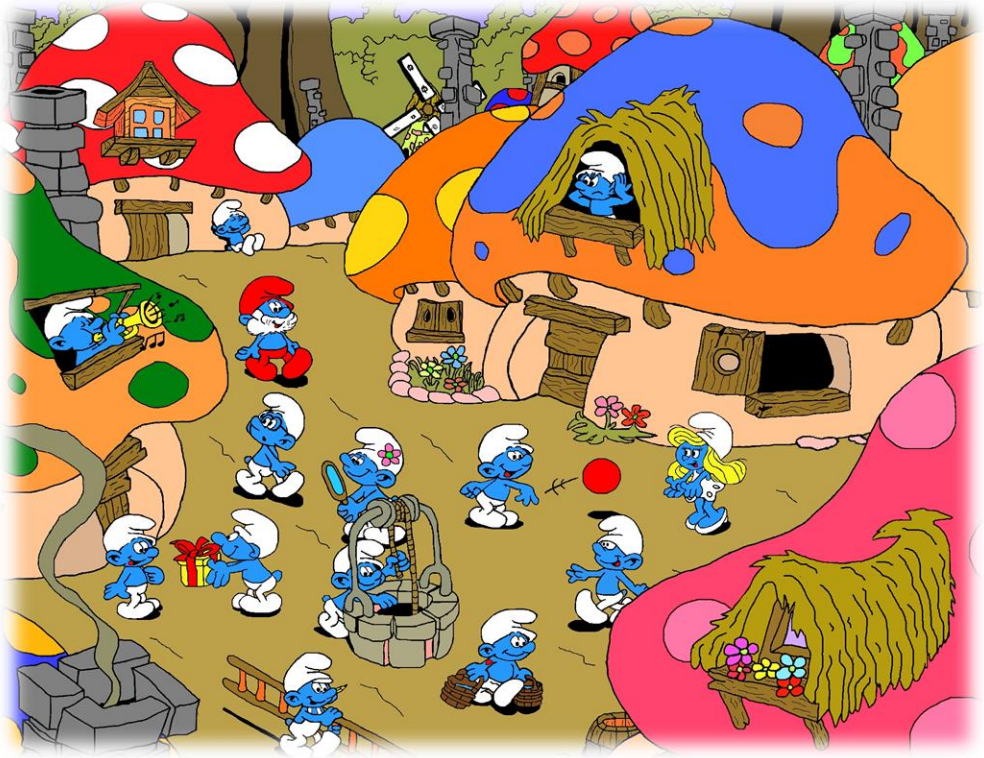
Singleton Design Pattern ☺



Exercise…

1. What is the difference between **Builder**, **Factory**, and **Abstract Factory**?
2. Think of 2 ways to avoid creating N classes in **Factory**.
3. Create an activity diagram for the **Singleton** Object.
4. How **Builder** is connected to **immutable** classes? Fluent programming?
    1. Code example

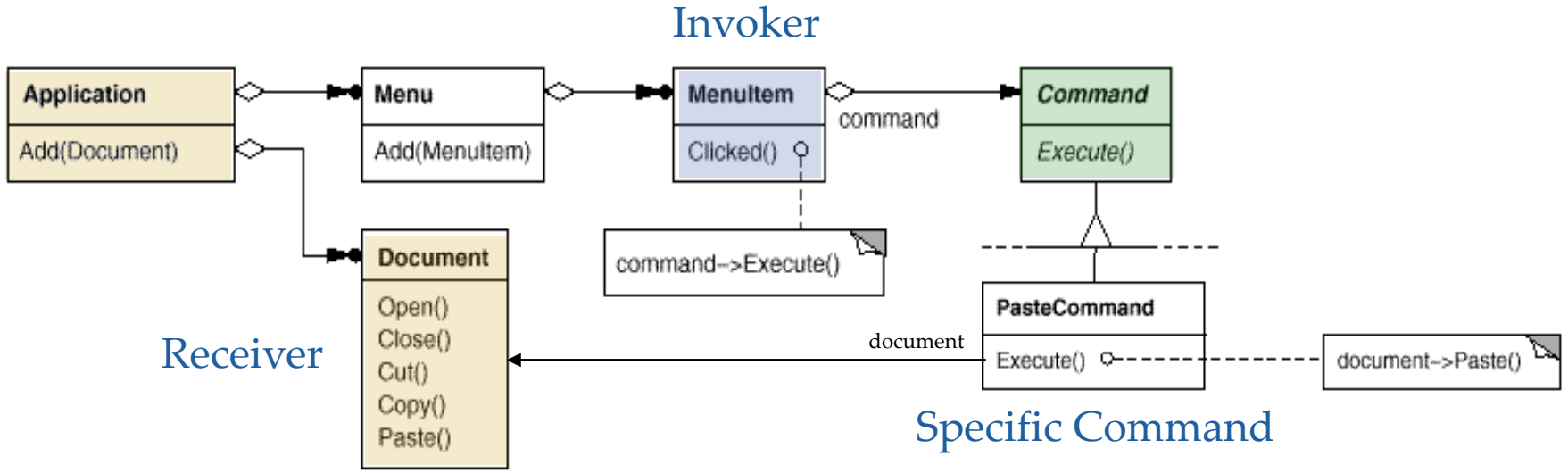
Behavioral Patterns



Command Pattern

Command Pattern

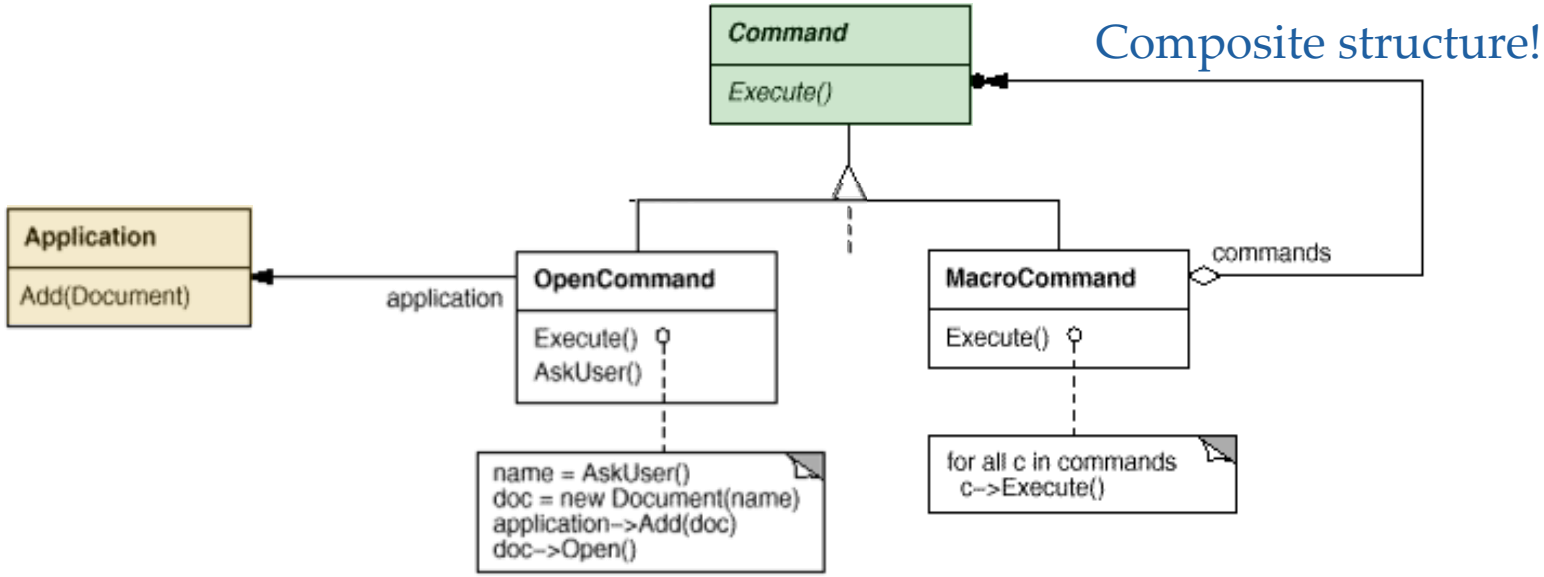
- Every request is served in a command object
- Commands can inherit other commands
- Command objects can be put into a queue, manage a log, support “undo” etc.
- Example:



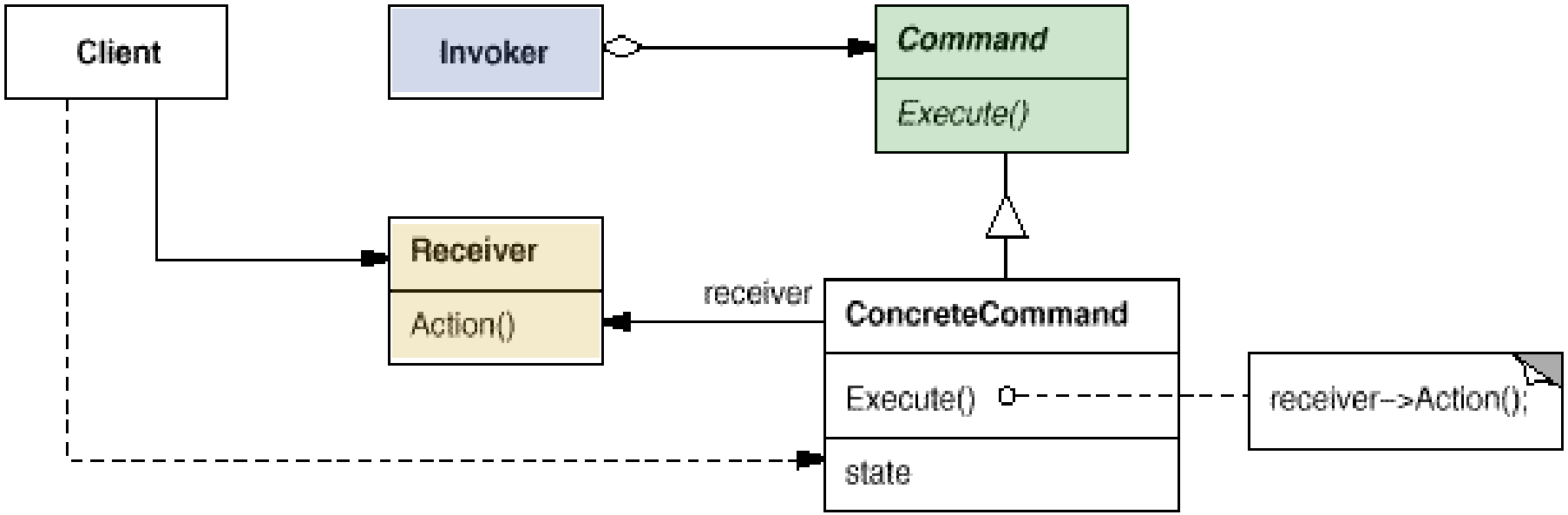
Other invokers can invoke the same specific command; the command code appears only once

Command Pattern

- Every request is served in a command object
- Commands can inherit other commands
- Command objects can be put into a queue, manage a log, support “undo” etc.
- Example:



Command Pattern



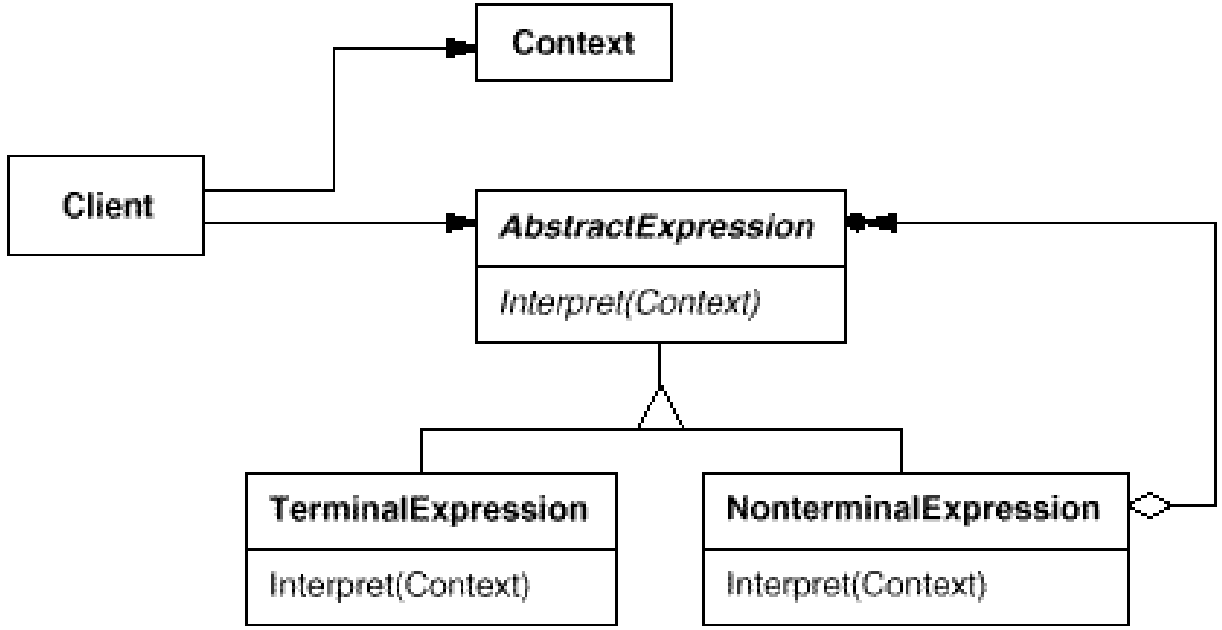
Example of a general Controller mechanism

- The Controller is a singleton and an observer
- Upon an update, a key is used to retrieve or create the Command object
- Existing Command objects are stored in a hash table (flyweight pattern)
- New Command objects are created by a Command-Pool (builder pattern)
- The Command object is inserted to a priority queue
- A different thread polls commands from the queue, if it is not empty
  - And execute each Command in a different thread via a Thread Pool

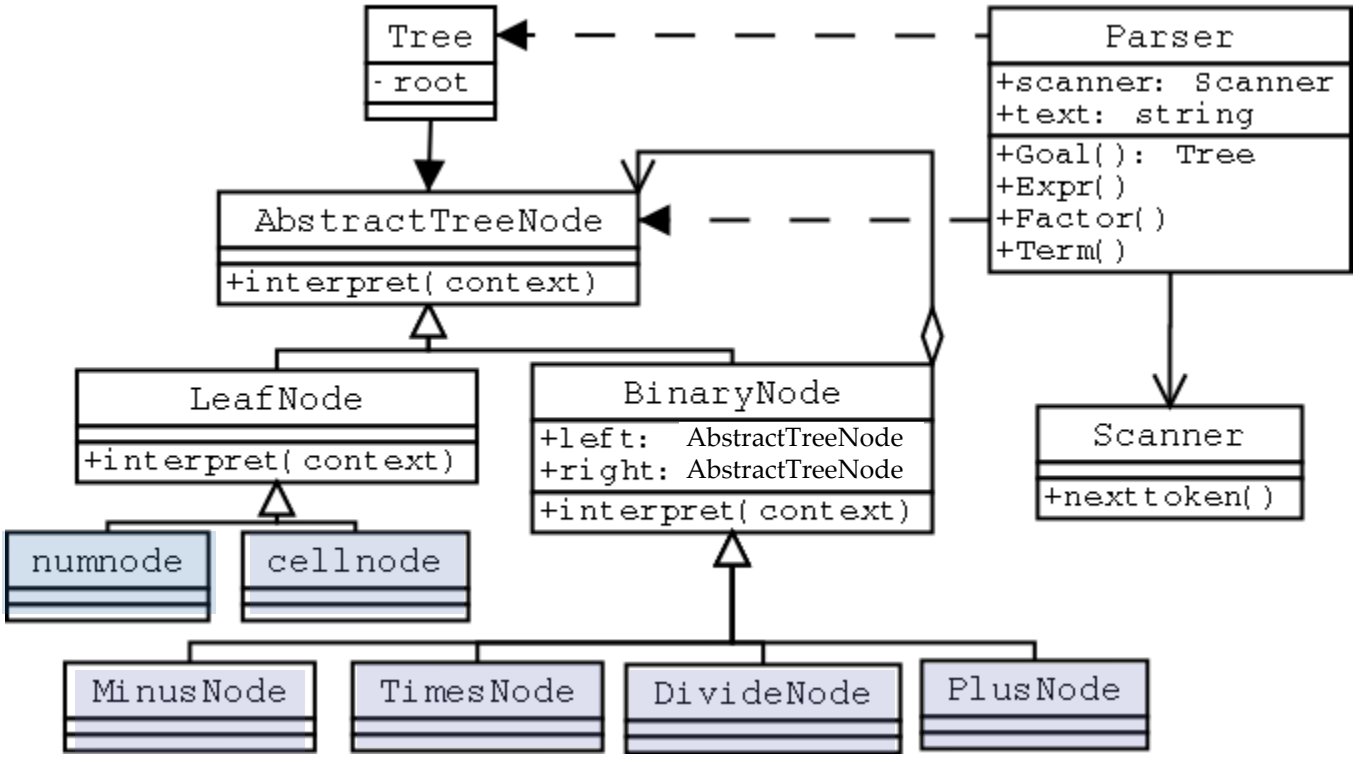
Interpreter Pattern

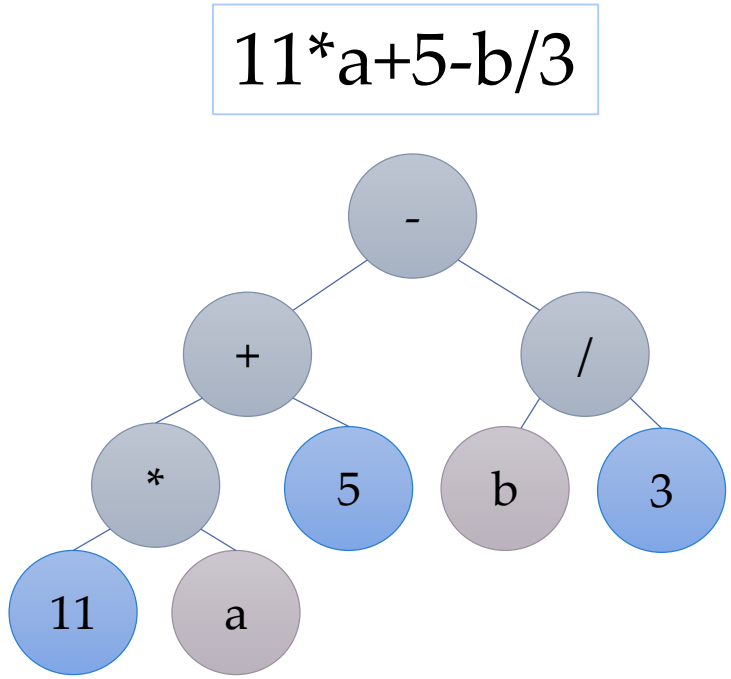
Interpreter Pattern

- Composite structure
- The behavior of *interpret()* is changed
- Can define a grammar



Interpreter Pattern Example



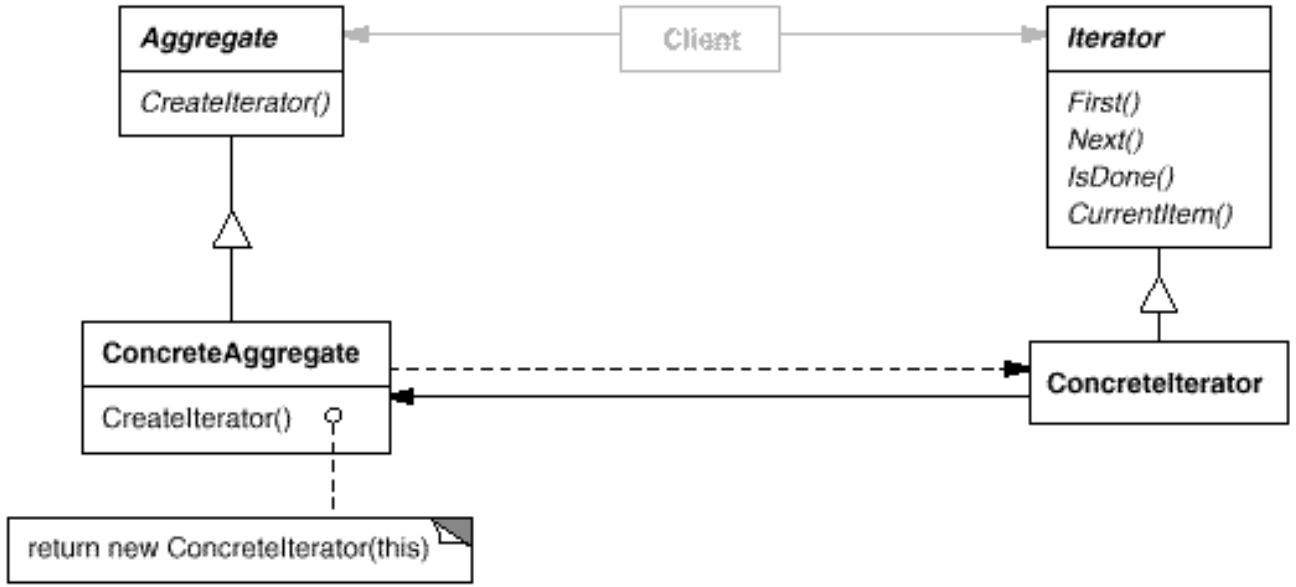


Iterator Pattern

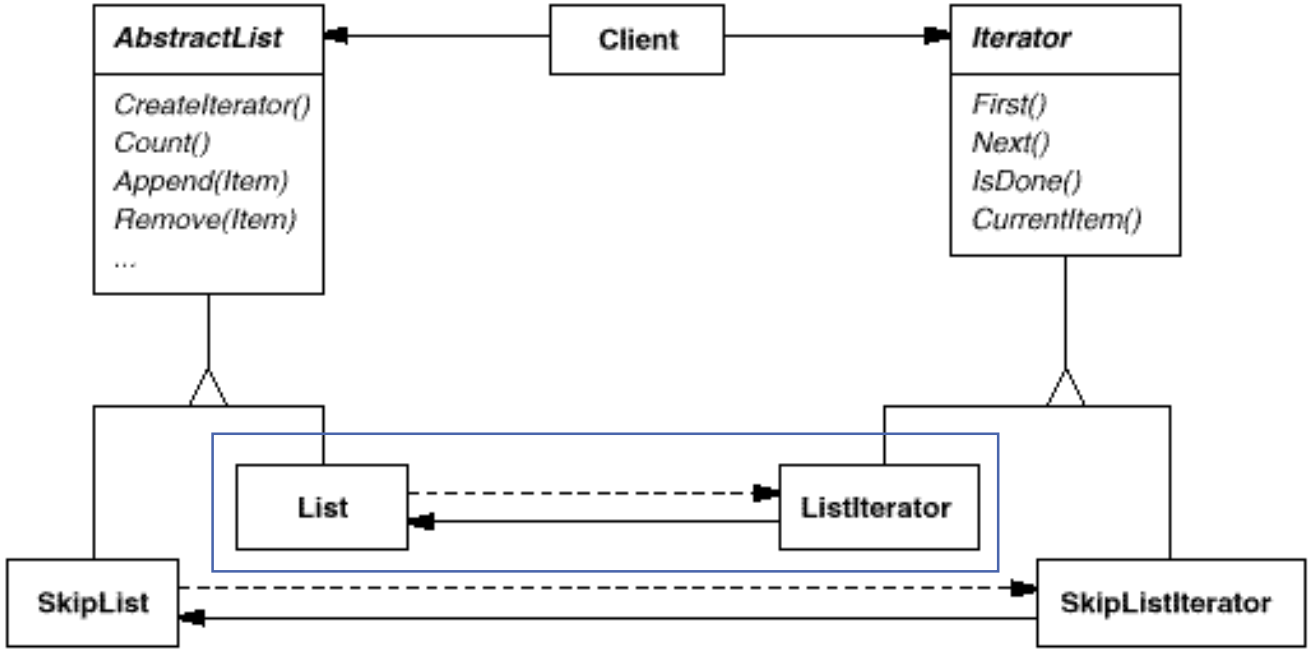
Iterator Pattern

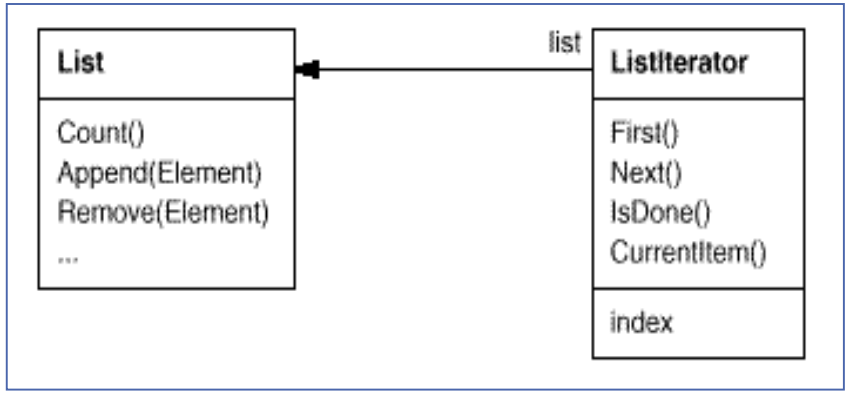
- Enables access to items of a complex object
- without exposing its inner structure
- Each container should return its own type of iterator

- Should consider:
  - Who defines the iteration alg'?
  - How robust is the iterator?
    - E.g., when deleting items

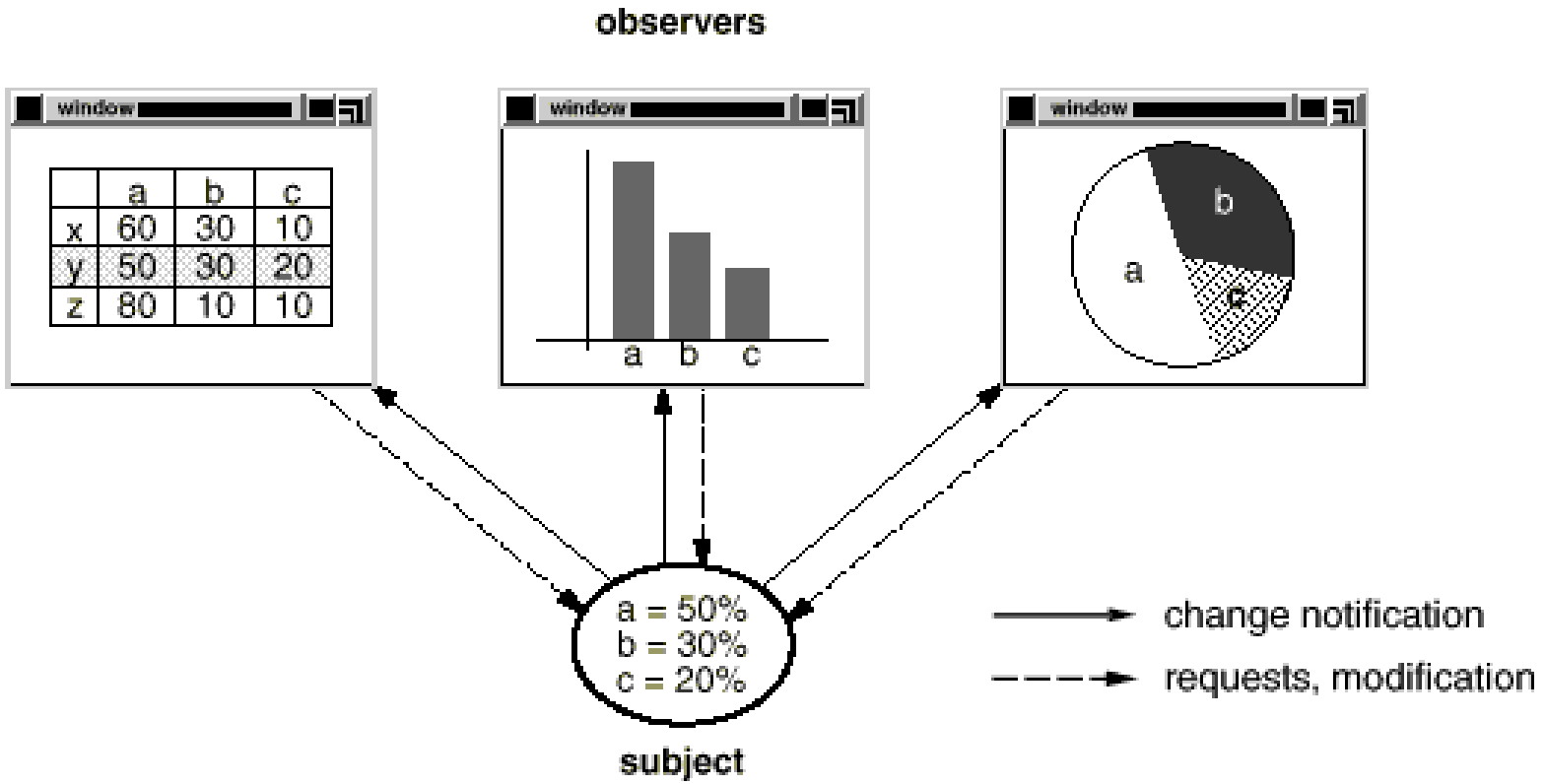


Iterator Pattern Example

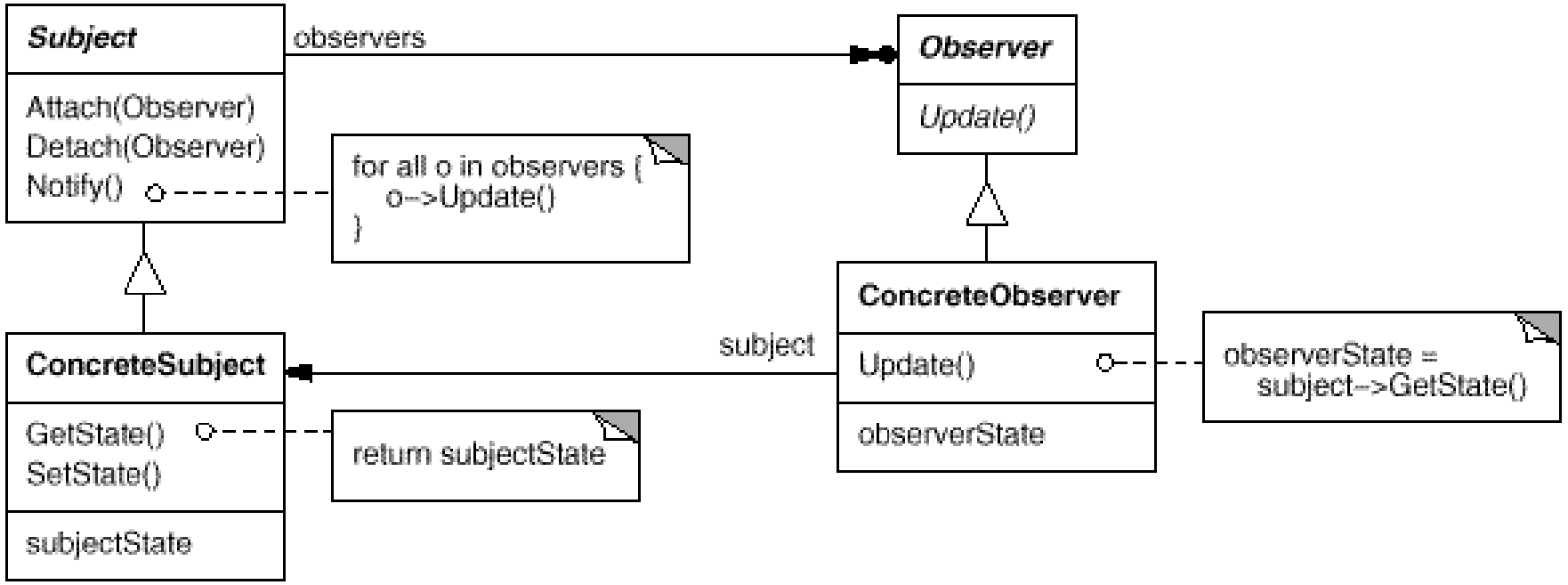




Observer Pattern

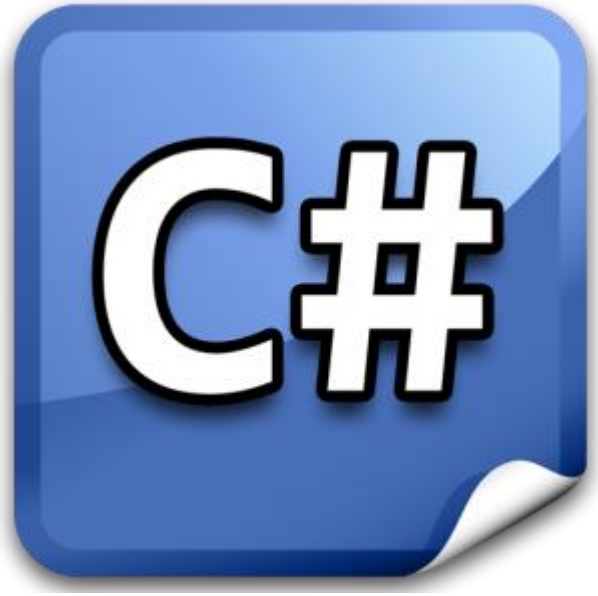


Observer Pattern



C# Example

Observer Pattern



Delegate variable vs. Event variable

- public void f(){…};
- public void g(){…};
- public **delegate** void myFunc();

```
myFunc x;
x=f;
x(); // activate f()
x=g;
x(); // activate g()
```

```
event myFunc x;
x+=f;
x+=g;
x(); // activate f() and g()
x-=f;
x(); // activate only g()
```

Observer Pattern - delegates & events in C#

- The ***Observable*** defines an event variable of some known delegate type
- The ***Observer*** registers its own delegates to the observable
- The observable activates all the registered delegates whenever it is needed


```
// when it is needed to notify all observers
notify(this, theEventArgs);
```

```
theObservable.notify +=
        delegate(Object sender, EventArgs e){
                // do something about the notification
        }
```

An Alarm Clock Example

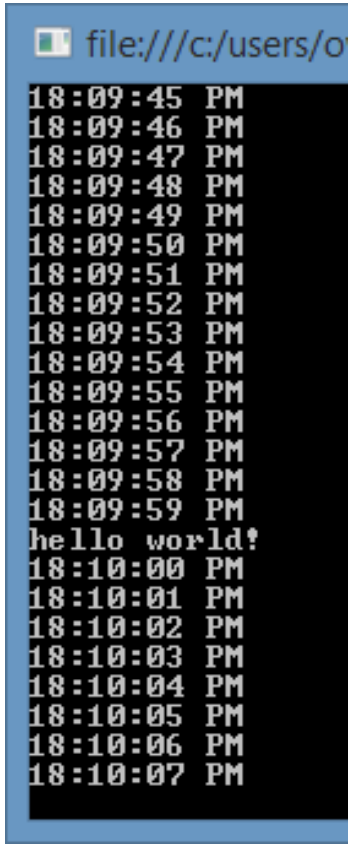
```
class AlarmClock {
    public Boolean stop;

    public delegate void whatToDo(String time);
    public event whatToDo customEvent;

    public void start()  {
        new Thread(
         delegate() {
           while (!stop) {
                String time = DateTime.Now.ToString("HH:mm:ss tt");

                customEvent(time);

                Thread.Sleep(1000);
           }
         }
        ).Start();
    }
}
```

An Alarm Clock Example

- Now we can use the event's **+=** operator to assign as many delegates as we wish
- The **-=** operator removes delegates from the event

```
static void Main(string[] args) {
    AlarmClock ac = new AlarmClock();

    ac.customEvent += delegate(String time) {
        if (time.Equals("18:10:00 PM")) {
            Console.WriteLine("hello world!");
        }
    };
    ac.customEvent += delegate(String time) {
        Console.WriteLine(time);
    };

    ac.start();
    Thread.Sleep(3*60*1000);
    ac.stop = true;
    Console.ReadKey();
}
```

We have added 2 event handlers

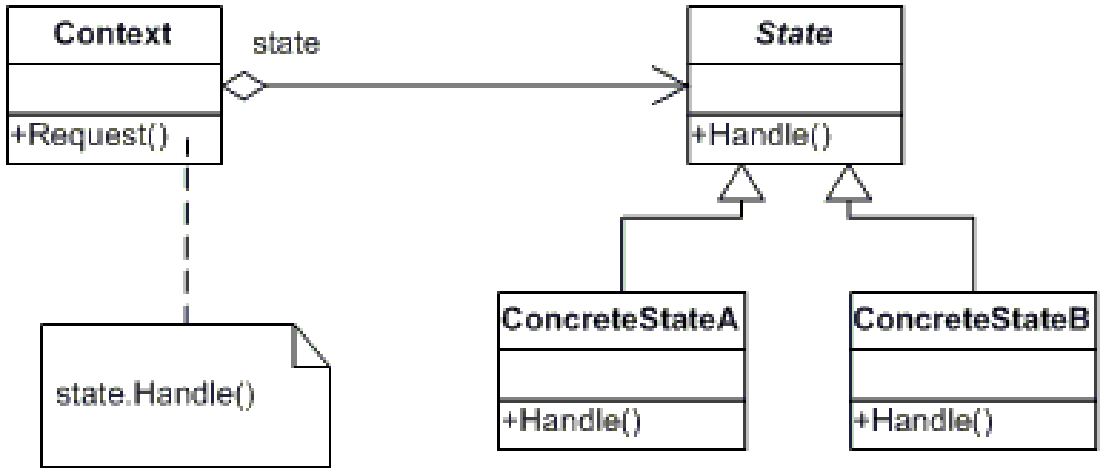


State Pattern

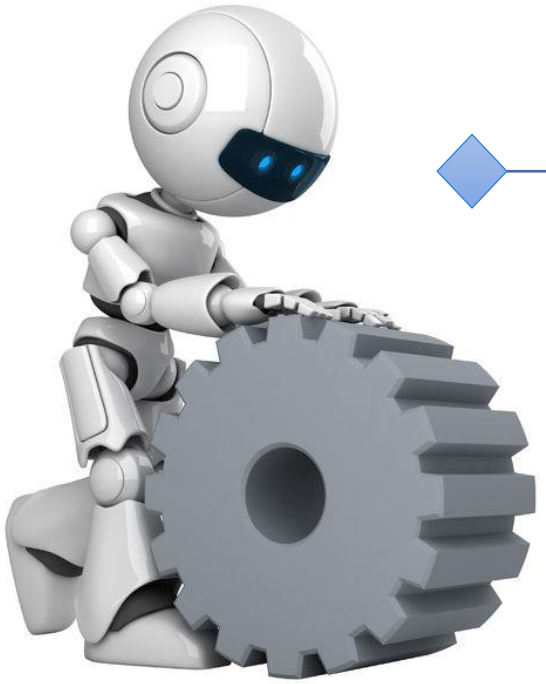
State Pattern

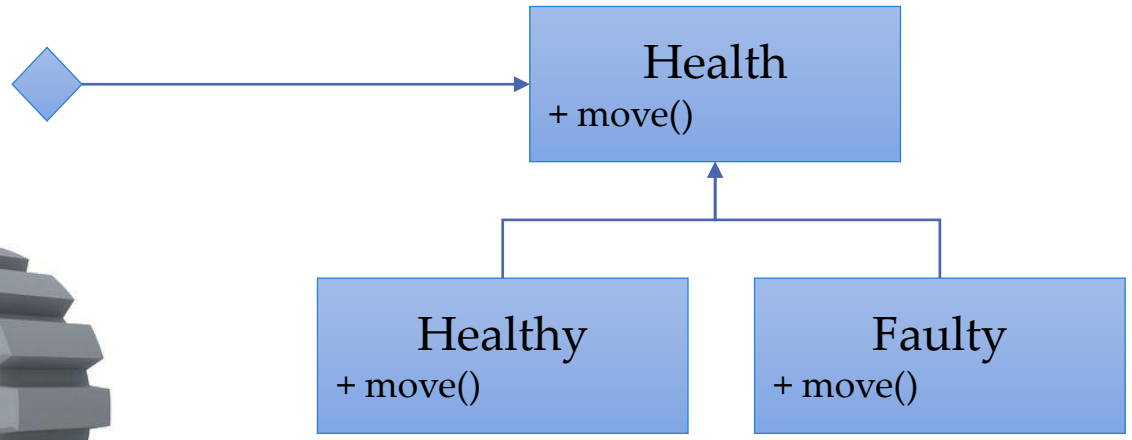
- The object changes its behavior when its state is changed
- All of the sate related behaviors are bound to the same class
- Easy to add new behaviors and maintain existing ones

- States can become flyweights
  - Sharing actions instead of data



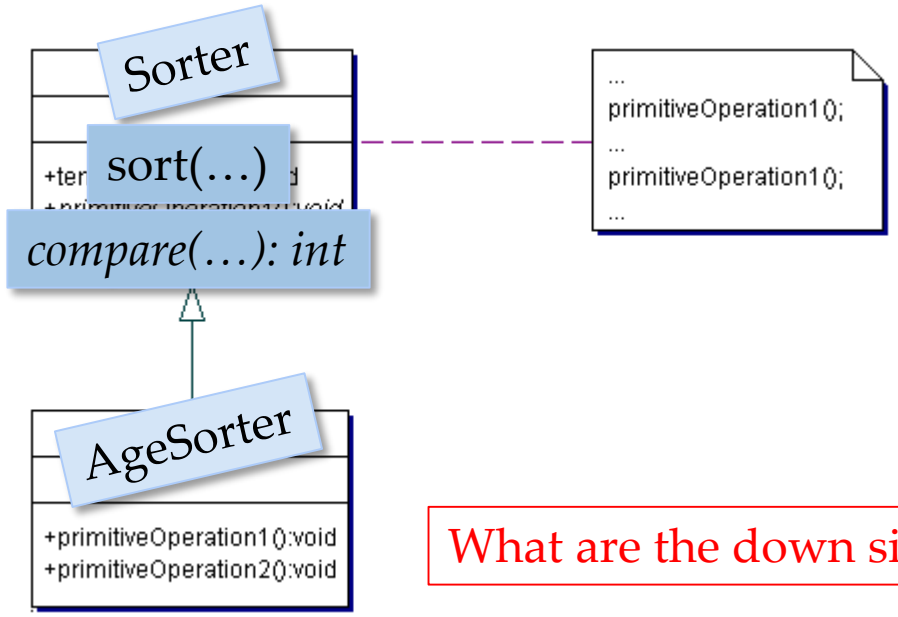
State Example





Template Method

Change the behavior of a (template) method by calling abstract methods



What are the down sides of this design pattern?

Strategy Pattern