# Ingineria programării

Curs 9 – 19-20 Aprilie

# Recapitulare

- GOF: Creational Patterns, Structural Patterns, Behavioral Patterns
- Creational Patterns
- Structural Patterns

# Recapitulare – CP

- **Abstract Factory** – computer components
- **Builder** – children meal
- **Factory Method** – Hello <Mr/Ms>
- **Prototype** – Cell division
- **Singleton** – server log files

# Recapitulare – SP

- **Adapter** – socket-plug
- **Bridge** – drawing API

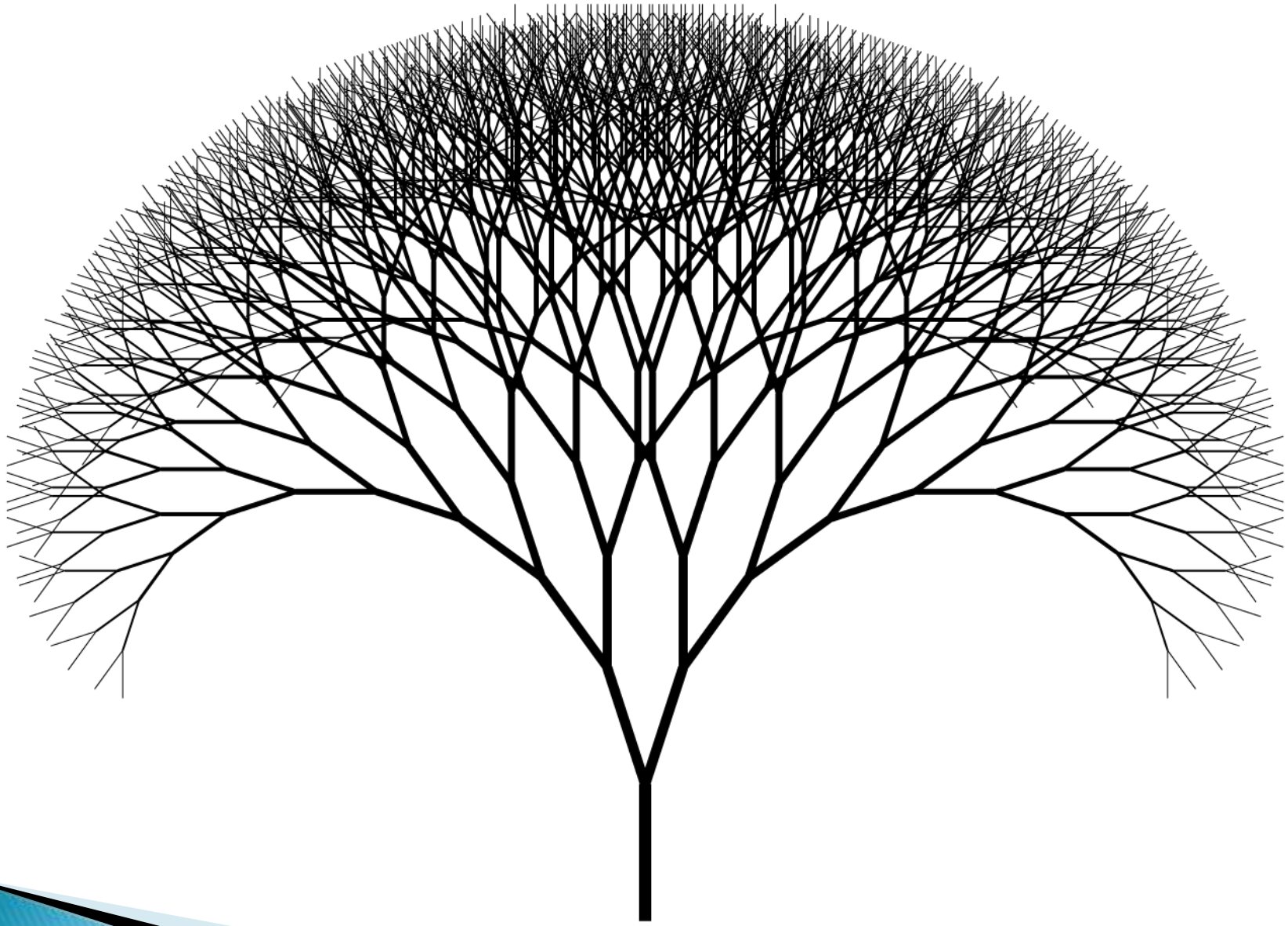
# Structural Patterns

- **Composite** – employee hierarchy
- **Decorator** – Christmas tree
- **Façade** – store keeper
- **Flyweight** – FontData
- **Proxy** – ATM access

# Structural Patterns – Composite

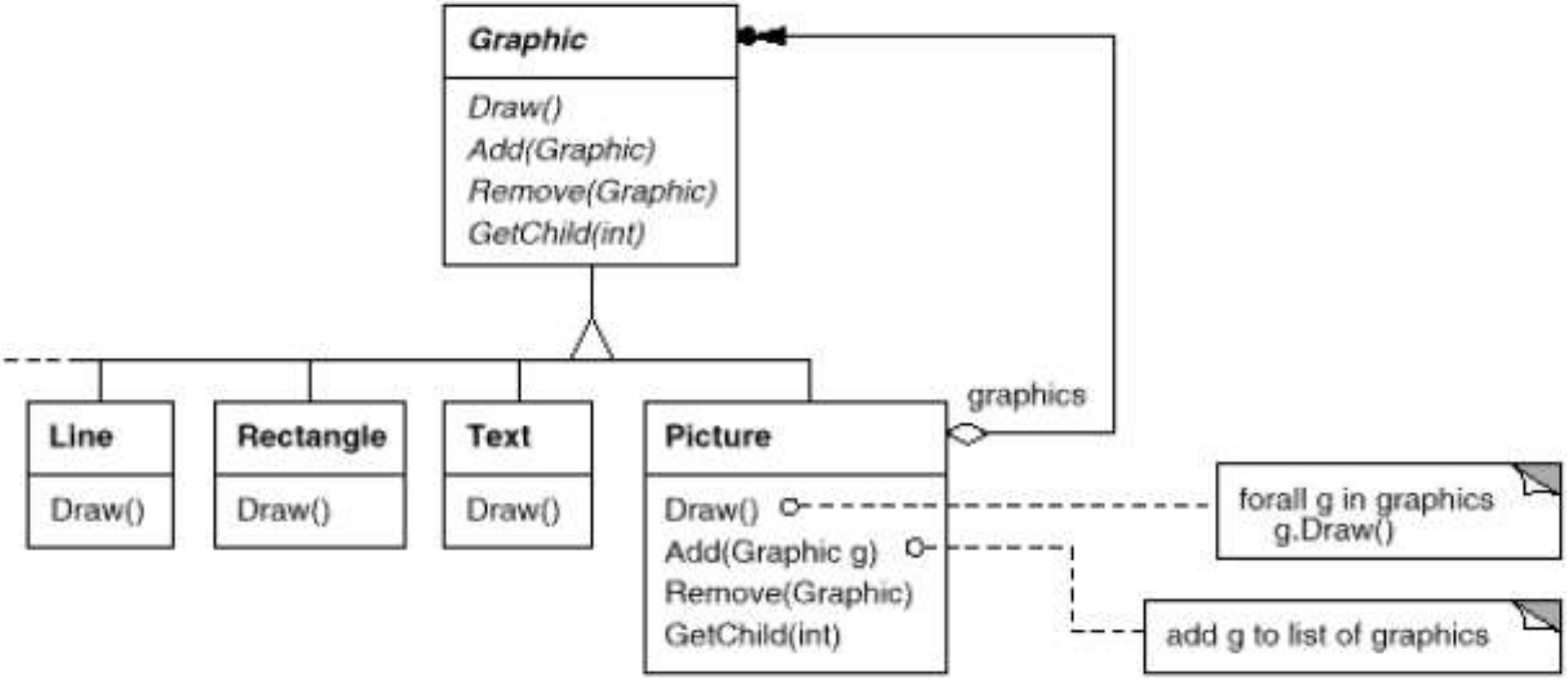
- **Intent** – Compose objects into tree structures to represent part-whole hierarchies. **Composite lets clients treat individual objects and compositions of objects uniformly**

- **Motivation** – Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically.



# Composite 1

- Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components

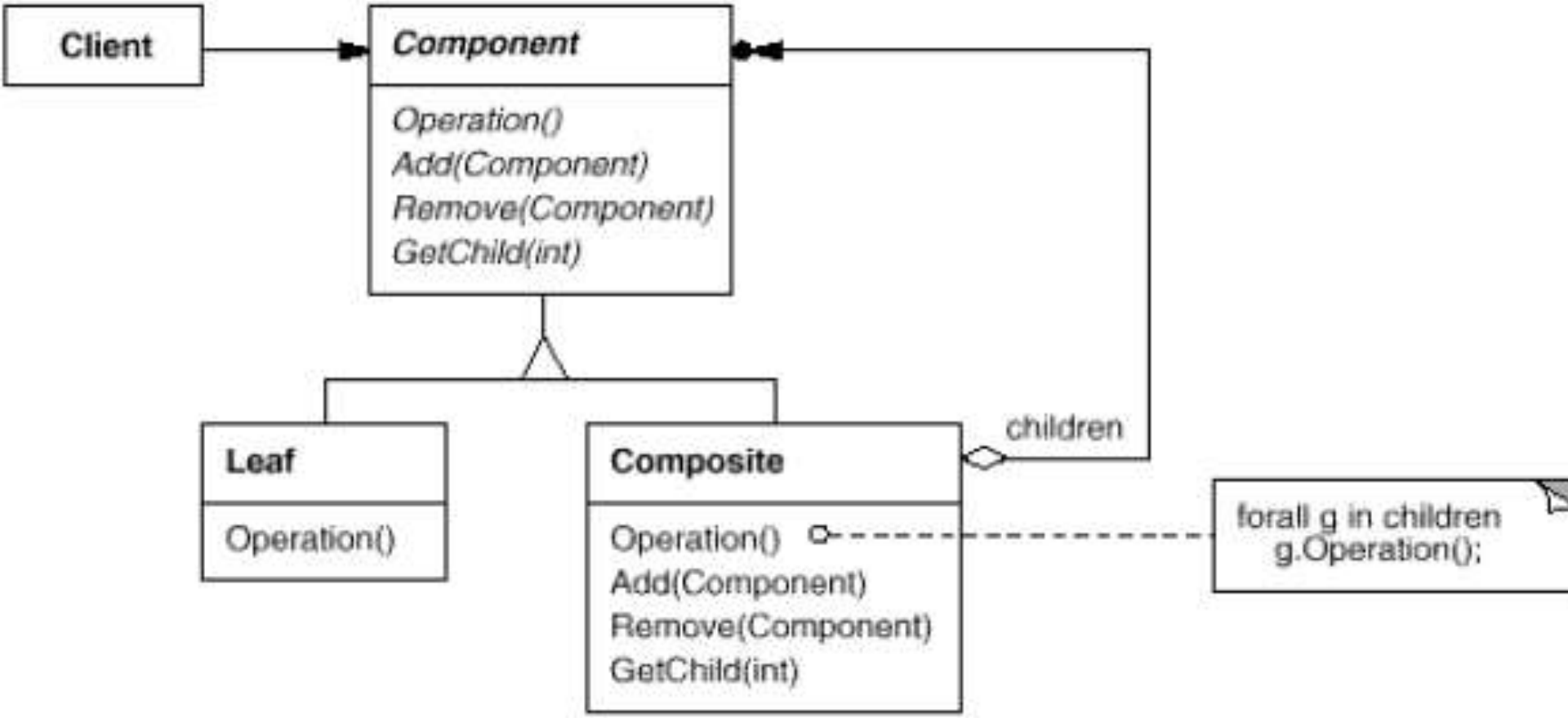


# Composite 2

- **Applicability** – Use this pattern when
  - you want to represent part-whole hierarchies of objects
  - you want clients to be able to ignore the difference between compositions of objects and individual objects
  - Clients will treat all objects in the composite structure uniformly

# Composite 3

- Structure



# Composite – Example

- The most common example in this pattern is of a company's employee hierarchy

- The employees of a company are at various positions. Now, say in a hierarchy, the manager has subordinates; also the Project Leader has subordinates, i.e. employees reporting to him/her. The developer has no subordinates

# Composite – Java 1

```
public class Employee {
        private String name; private double salary;
        private Vector subordinates;

        public Vector getSubordinates() {return subordinates;}
        public void setSubordinates(Vector subordinates) {
                this.subordinates = subordinates;}

        public Employee(String name, double sal) {
                setName(name);setSalary(sal);
                subordinates = new Vector();
        }
        public void add(Employee e) {subordinates.addElement(e);}
        public void remove(Employee e) {subordinates.remove(e);}
}
```

# Composite – Java 2

```
private void addEmployeesToTree() {
   Employee CFO = new Employee("CFO", 3000);
   Employee headFinance1 = new Employee("HF. North", 2000);
   Employee headFinance2 = new Employee("HF. West", 2200);
   Employee accountant1 = new Employee("Accountant1", 1000);
   Employee accountant2 = new Employee("Accountant2", 900);
   Employee accountant3 = new Employee("Accountant3", 1100);
   Employee accountant4 = new Employee("Accountant4", 1200);

   CFO.add(headFinance1); CFO.add(headFinance2);
   headFinance1.add(accountant1);headFinance1.add(accountant4);
   headFinance2.add(accountant2);headFinance2.add(accountant3);
}
```

*CFO = chief financial officer*

# Composite – Java 3

- Once we have filled the tree up, now we can get the tree for any employee and find out whether that employee has subordinates with the following condition.

```
Vector subOrdinates = emp.getSubordinates();
if (subOrdinates.size() != 0)
    getTree(subOrdinates);
else
    System.out.println("No Subordinates for the
Employee: "+emp.getName());
```

# Composite – The Good, The Bad …

- Preserves O (from SOLID), because new classes can easily be added to the hierarchy
- Using polymorphism and inheritance works well for complex tree structures
- Prone to overgeneralization
- Difficult to provide a common interface

# Structural Patterns – Decorator

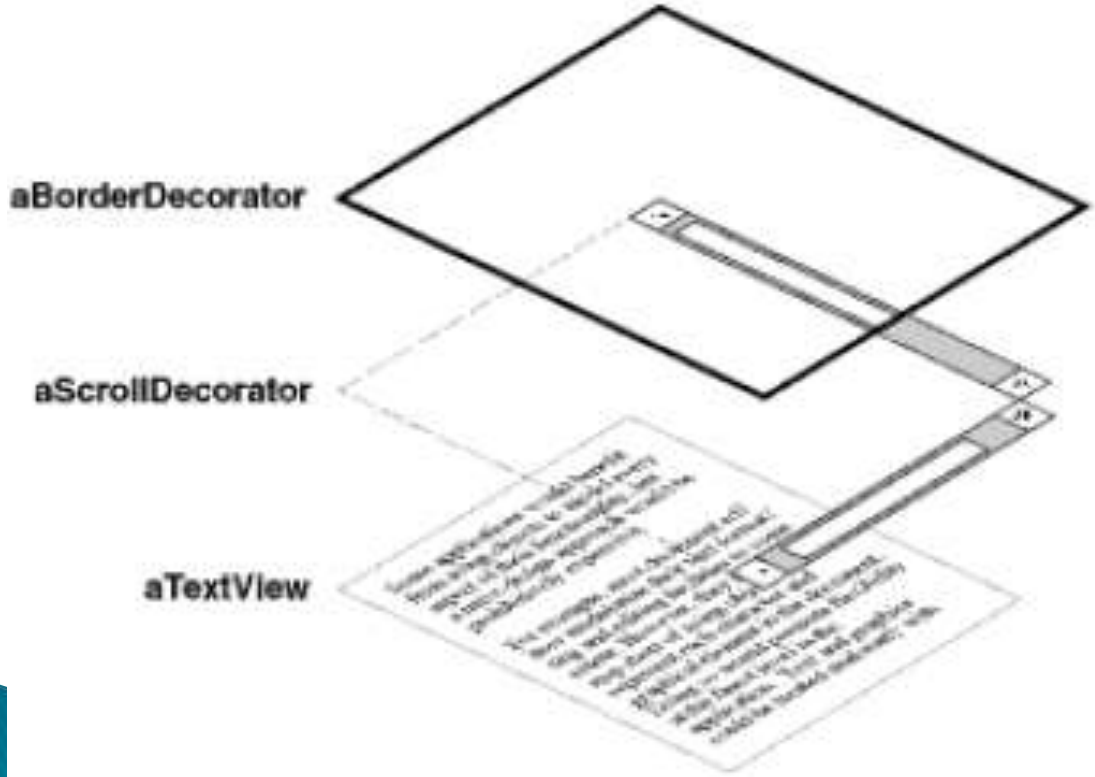
- **Intent – Attach additional responsibilities to an object dynamically**. Decorators provide a flexible alternative to subclassing for extending functionality
- **Also Known As** – Wrapper (*similar Adapter*)
- **Motivation** – Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component

Component

# Decorators

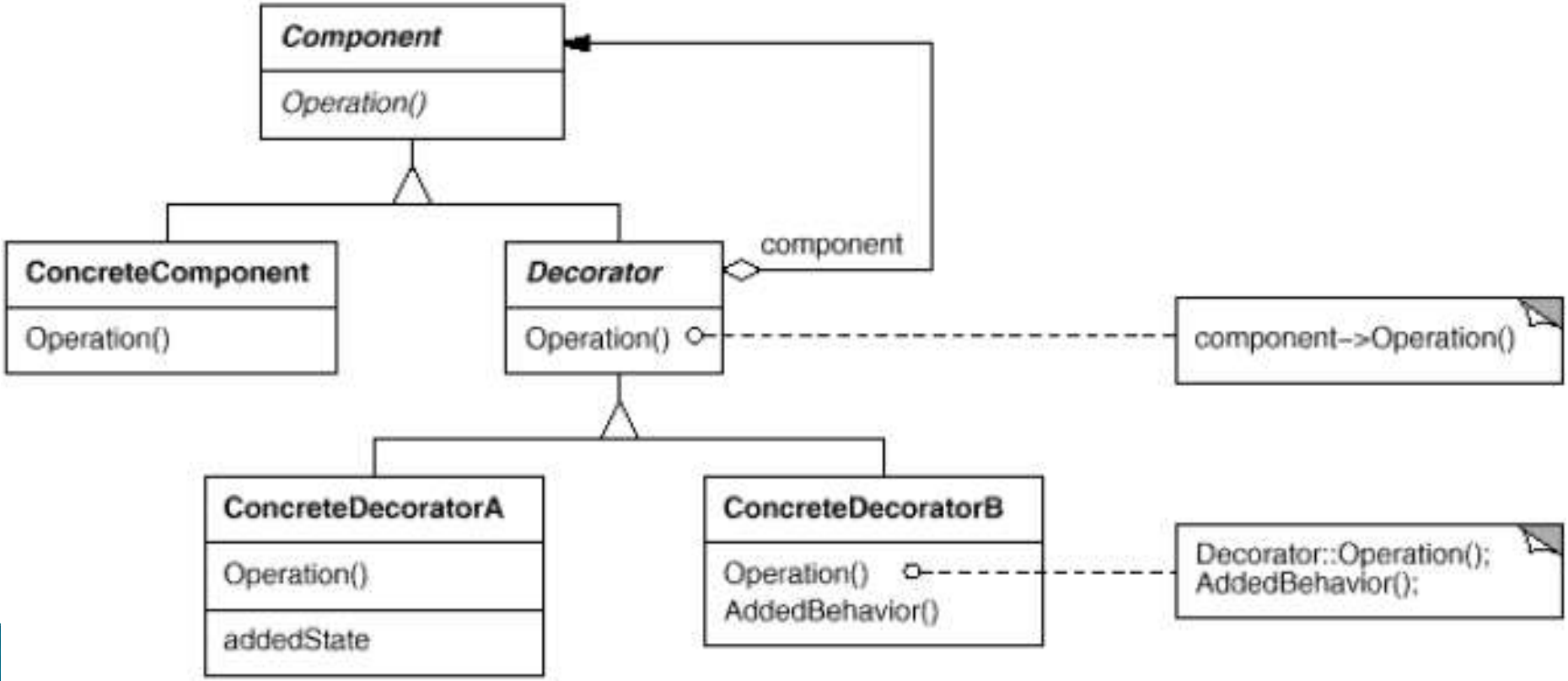
# Decorator 1

- A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**




# Decorator 2

- Structure



# Decorator 3

- **Applicability** – Use Decorator
  - to add responsibilities to individual objects dynamically and transparently
  - for responsibilities that can be withdrawn
  - when extension by subclassing is impractical

# Decorator – Example

- Suppose we have some 6 objects and 2 of them need a special behavior, we can do this with the help of a decorator

- Let's take an example of a **Christmas tree**. There is a need to decorate a Christmas tree. Now we have many branches which need to be decorated in different ways

# Decorator – Java 1

```
public abstract class Decorator {
   /** The method places each decorative item on
   the tree. */
   public abstract void place(Branch branch);
}
public class ChristmasTree {
   private Branch branch;
   public Branch getBranch() {
        return branch;
   }
}
```

# Decorator – Java 2

```
public class BallDecorator extends Decorator {
  public BallDecorator(ChristmasTree tree) {
      Branch branch = tree.getBranch();
      place(branch);
  }

  public void place(Branch branch) {
      branch.put("ball");
  }
}
```

# Decorator – Java 3

- Similarly, we can make StarDecorator and RufflesDecorator

```
StarDecorator decorator = new StarDecorator(new
  ChristmasTree());
```

- This way the decorator will be instantiated and a branch of the Christmas tree will be decorated.

# Decorator – The Good, The Bad …

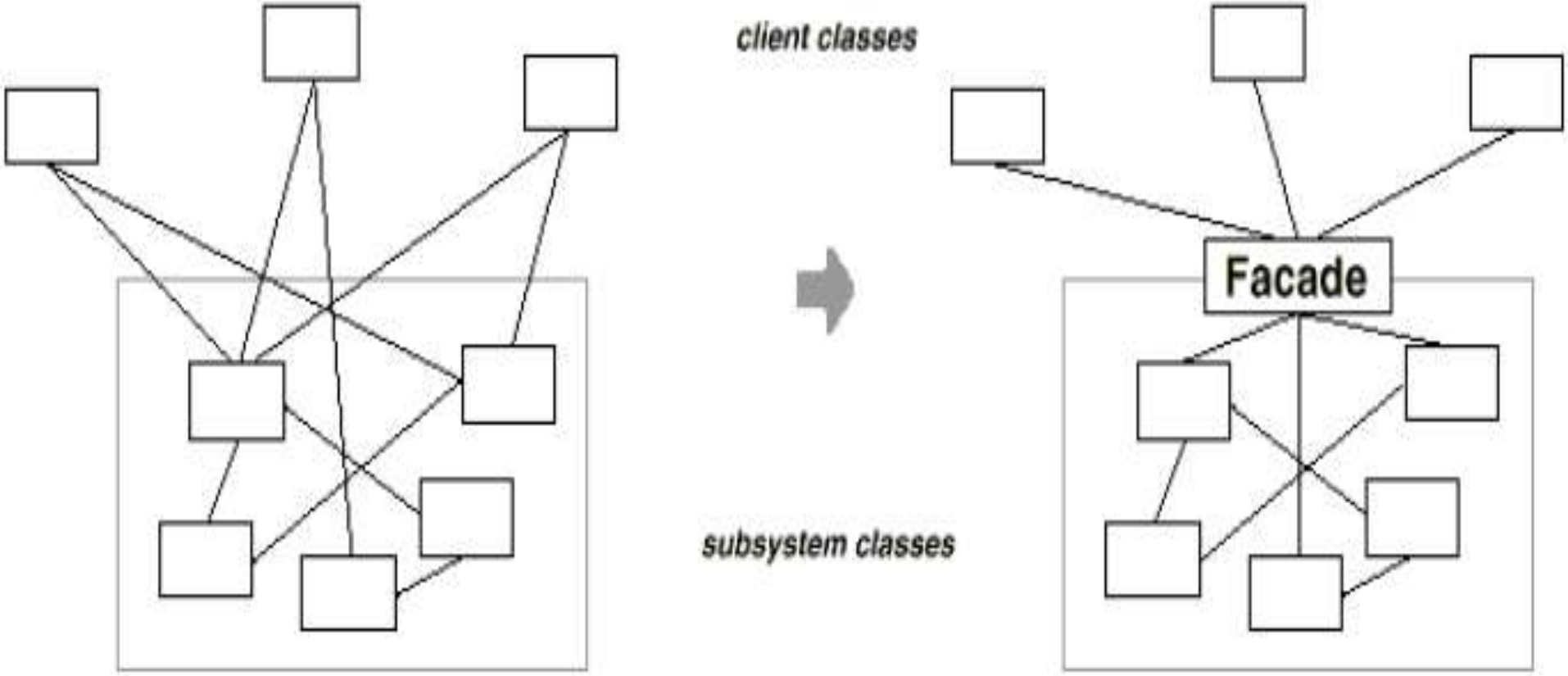
- Change object behavior without inheritance
- Preserves S (from SOLID) – a large class can be split in smaller ones
- Can add multiple behaviors by adding multiple decorations
- Change an object's responsibilities at runtime
- The sequence of applying decorators matters
- Difficult to remove a decorator from the middle of the pile
- The code is hard to manage

# Structural Patterns – Façade

- **Intent – Provide a unified interface to a set of interfaces in a subsystem**
- **Motivation** – Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as *Scanner, Parser, ProgramNode, BytecodeStream,* and *ProgramNodeBuilder* that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler want to compile some code

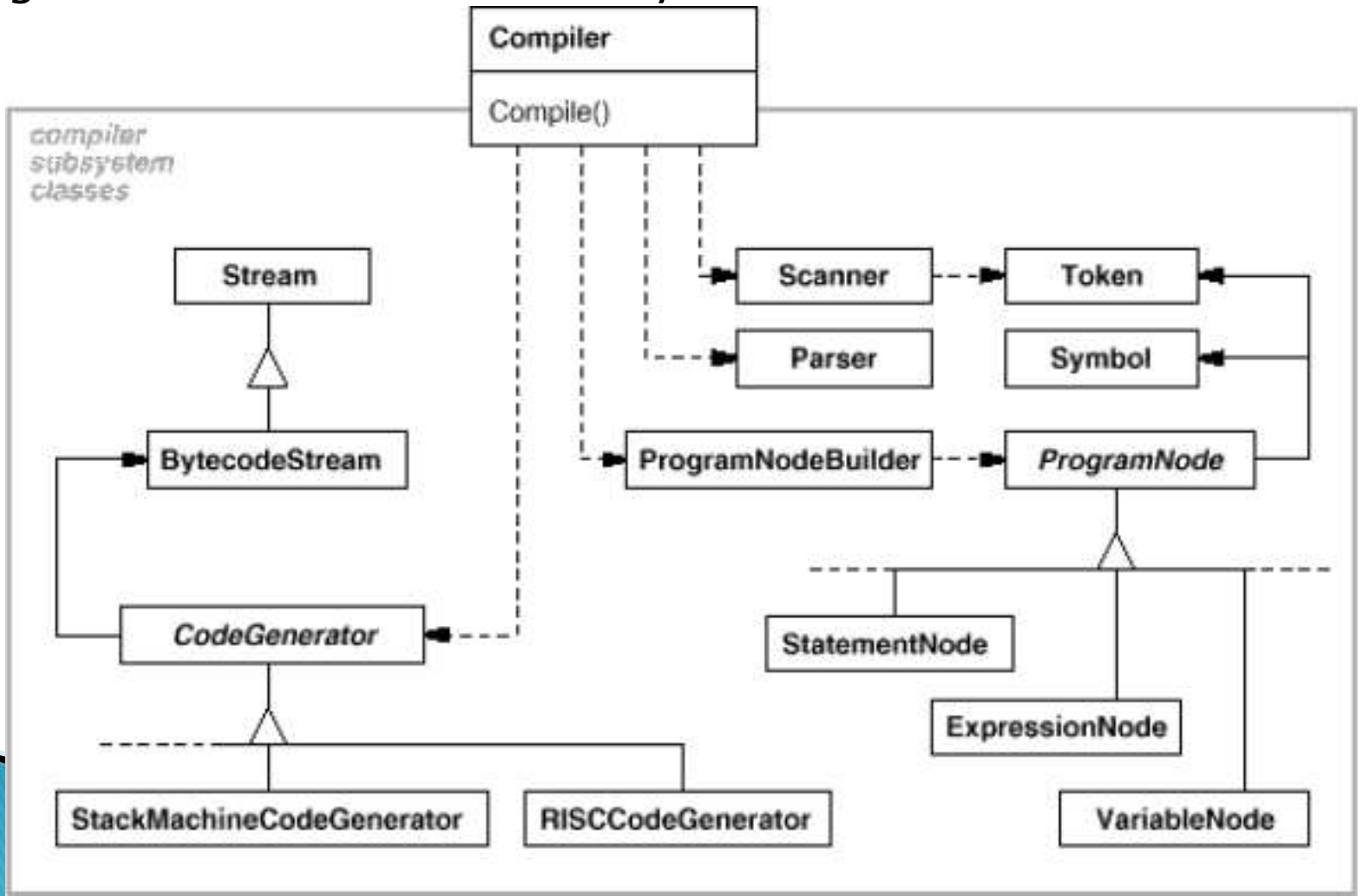
# Façade 1

- A common design goal **is to minimize the communication and dependencies between** subsystems



# Façade 2

- The compiler facade makes life easier for most programmers without hiding the lower-level functionality from the few that need it



# Façade 3

- **Applicability** – Use the Facade pattern when
  - you want to provide a simple interface to a complex subsystem
  - there are many dependencies between clients and the implementation classes of an abstraction
  - you want to layer your subsystems

# Façade – Example 1

- Facade as the name suggests means the face of the building. The people walking past the road can only see this glass face of the building. The face hides all the complexities of the building and displays a friendly face.
- Facade hides the complexities of the system and provides an interface to the client from where the client can access the system. In Java, the interface JDBC can be called a façade
- Other examples?

# Façade – Example 2

- Let's consider a **store**. This store has a store keeper. In the storage, there are a lot of things stored e.g. **packing material, raw material and finished goods**.
- You, as client want access to different goods. You do not know where the different materials are stored. You just have access to store keeper who knows his store well. Here, the store keeper acts as the facade, as he hides the complexities of the system Store.

# Façade – Java 1

```
public interface Store {
  public Goods getGoods();
}

public class FinishedGoodsStore implements Store
{
  public Goods getGoods() {
      FinishedGoods finishedGoods = new FinishedGoods();
      return finishedGoods;
  }
}
```

# Façade – Java 2

```
public class StoreKeeper {
  public RawMaterialGoods getRawMaterialGoods() {
      RawMaterialStore store = new RawMaterialStore();
      RawMaterialGoods rawMaterialGoods =
  (RawMaterialGoods)store.getGoods();
      return rawMaterialGoods;
  }
  …
}
```

# Façade – Java 3

```
public class Client {
  public static void main(String[] args) {
      StoreKeeper keeper = new StoreKeeper();
      RawMaterialGoods rawMaterialGoods =
      keeper.getRawMaterialGoods();
  }
}
```

# Façade – The Good, The Bad …

- Isolates and mask system complexity from the user
- The façade class runs the risk of being coupled to everything

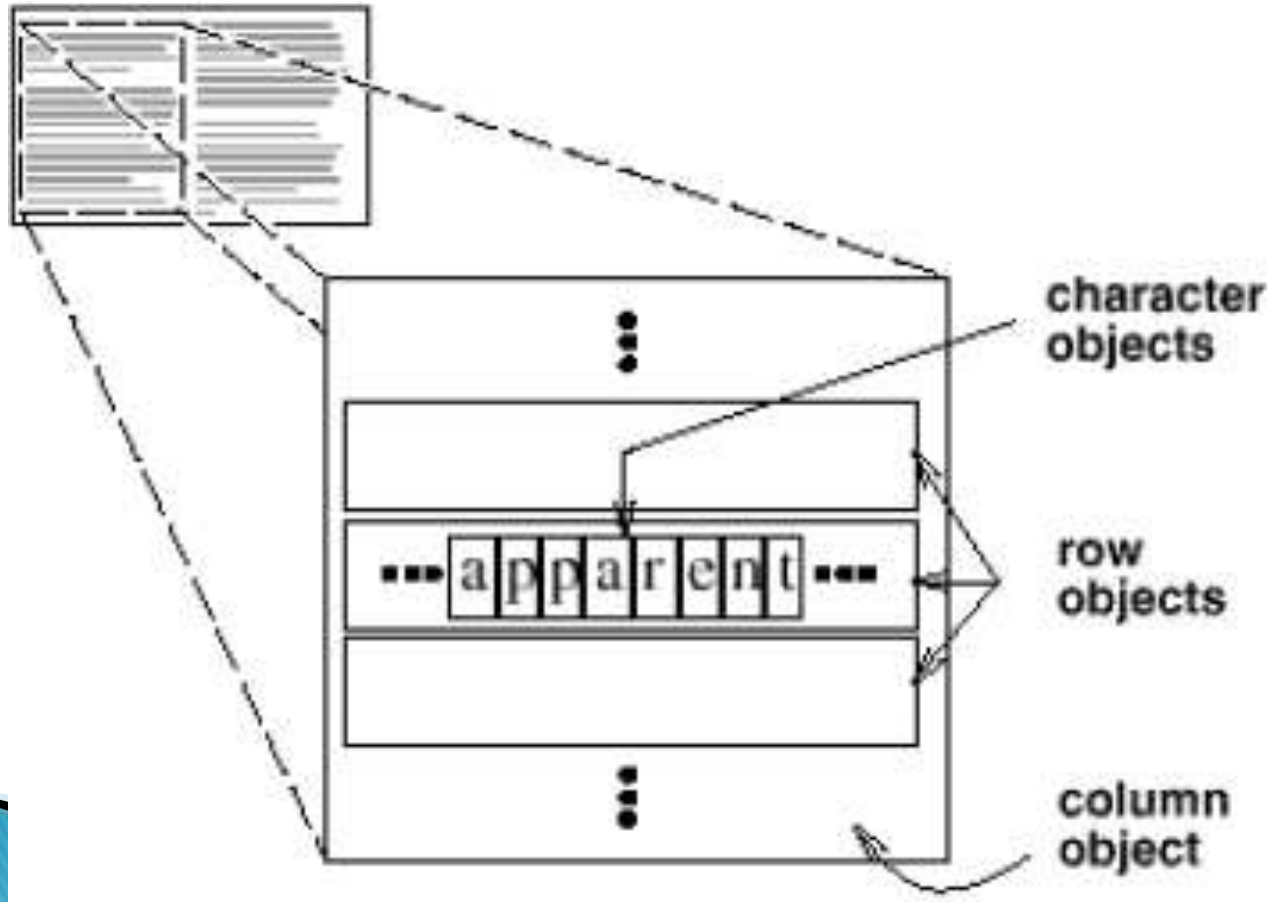
# Structural Patterns – Flyweight

- **Intent – Use sharing to support large numbers of fine-grained objects efficiently**

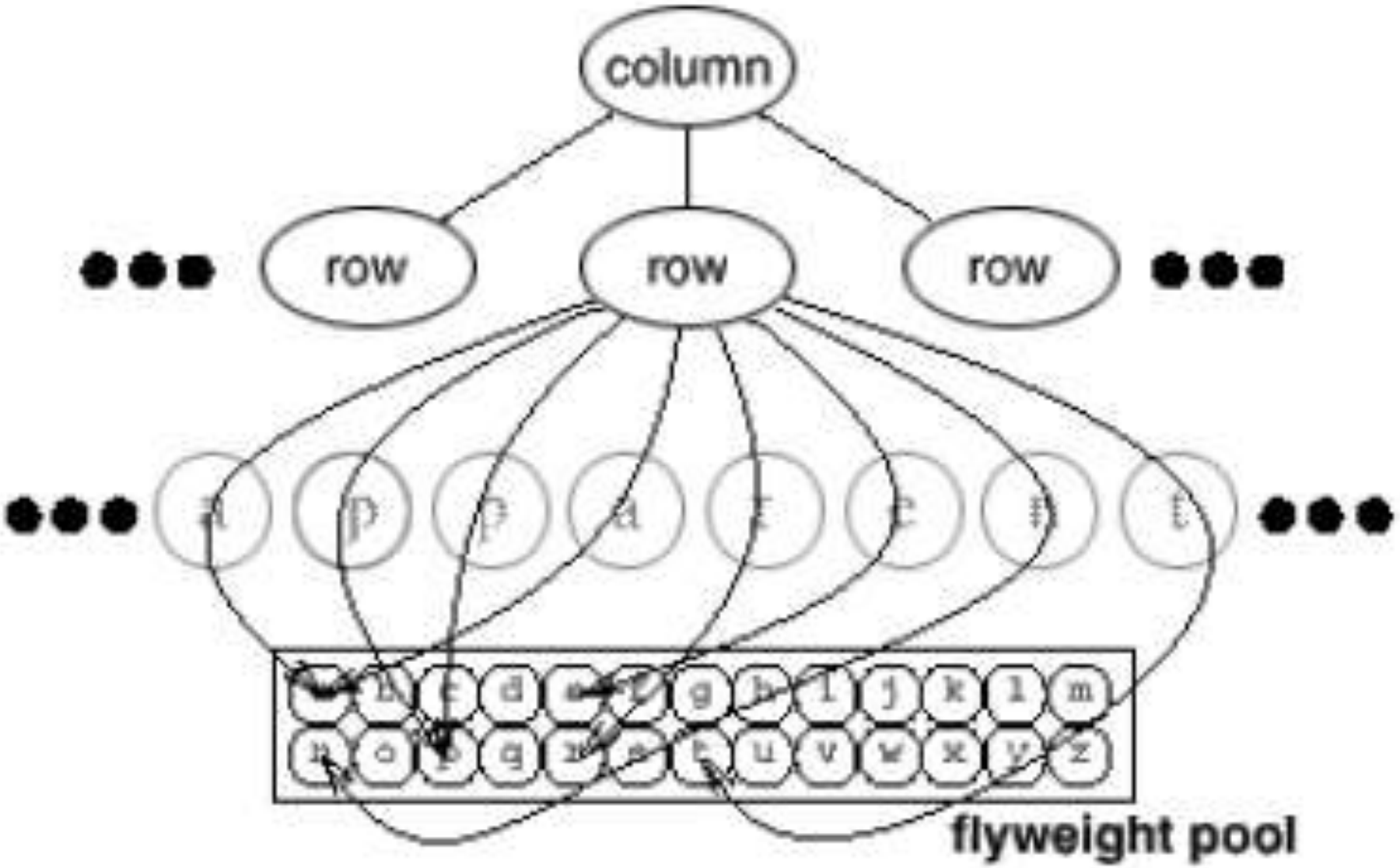
- **Motivation –** Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.
- For example, most document editor implementations have text formatting and editing facilities that are modularized to some extent.

# Flyweight 1

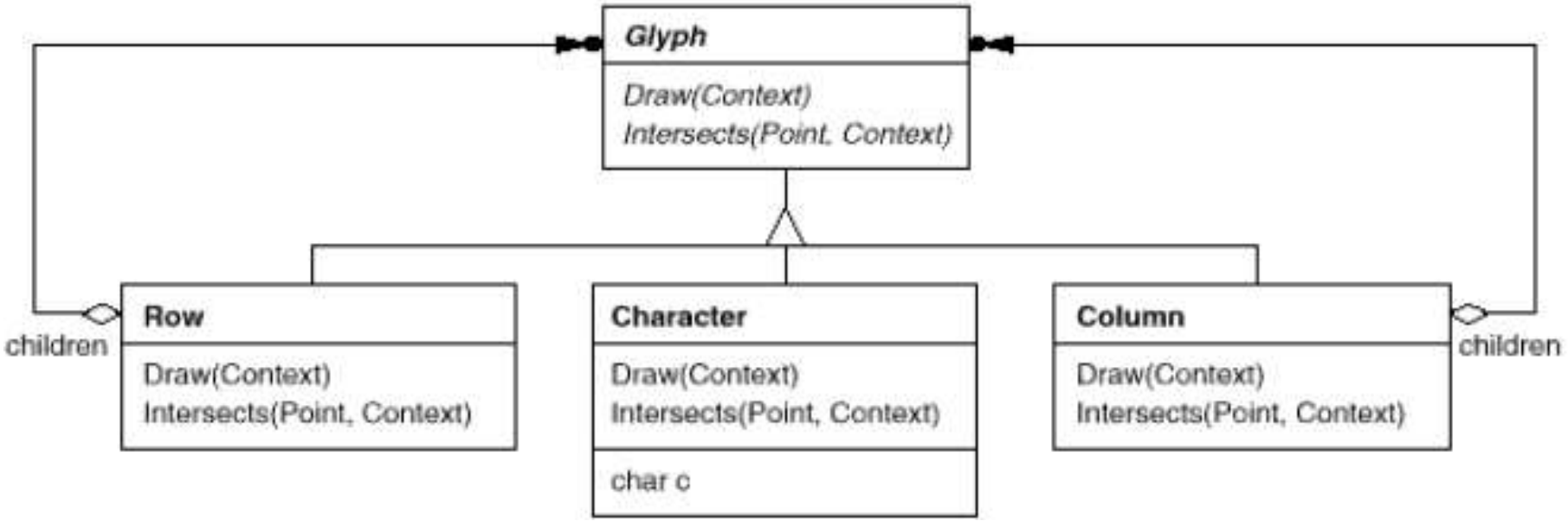
- The following diagram shows how a document editor can use objects to represent characters



# Flyweight 2



# Flyweight 3


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
children

# Flyweight 4

- **Applicability** – Use the Flyweight pattern when
  - Supporting a large number of objects that:
    - Are similar
    - Share at least some attributes
    - Are too numerous to easily store whole in memory

# Flyweight – Example

- **A Flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects**

- A classic example usage of the flyweight pattern are the data structures for graphical representation of characters in a word processor. It would be nice to have, for each character in a document, a glyph object containing its *font outline, font metrics*, and other formatting data, but it would amount to hundreds or thousands of bytes for each character. Instead, are used the flyweights called **FontData**

# Flyweight – Java 1

```
public enum FontEffect {
   BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}
public final class FontData {
   private static final WeakHashMap<FontData,
   WeakReference<FontData>> FLY_WEIGHT_DATA = new
   WeakHashMap<FontData, WeakReference<FontData>>();
   private final int pointSize;
   private final String fontFace;
   private final Color color;
   private final Set<FontEffect> effects;

   private FontData(int pointSize, String fontFace, Color color,
   EnumSet<FontEffect> effects) {
         this.pointSize = pointSize;
         this.fontFace = fontFace;
         this.color = color;
         this.effects = Collections.unmodifiableSet(effects);
   }
```

# Flyweight – Java 2

```
public static FontData create(int pointSize, String
  fontFace, Color color, FontEffect... effects) {
  EnumSet<FontEffect> effectsSet =
  EnumSet.noneOf(FontEffect.class);
  for (FontEffect fontEffect : effects) {
  effectsSet.add(fontEffect); }
  FontData data = new FontData(pointSize, fontFace,
  color, effectsSet);
  if (!FLY_WEIGHT_DATA.containsKey(data)) {
       FLY_WEIGHT_DATA.put(data, new
       WeakReference<FontData> (data));
  }
  return FLY_WEIGHT_DATA.get(data).get();
}
```

# Flyweight – The Good, The Bad …

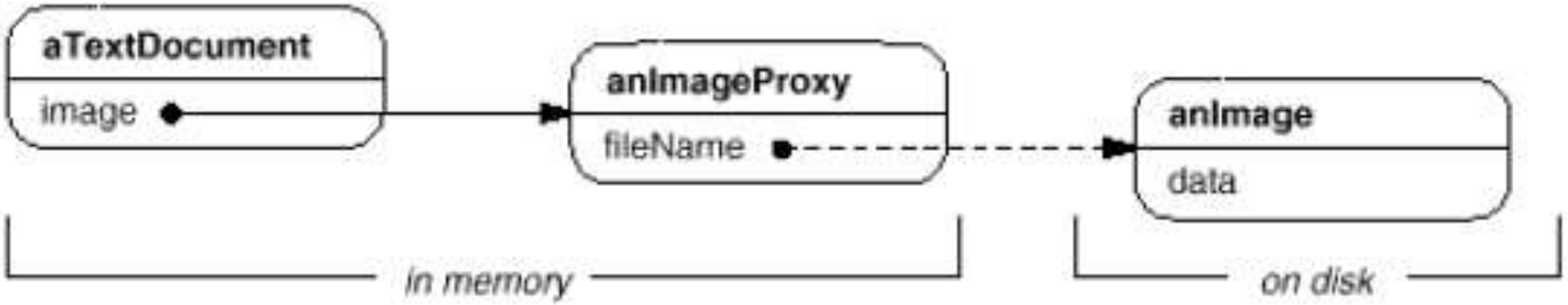
- Saves on memory in the case of large numbers of objects
- Becomes costly in processing time
- The code is complicated and not intuitive

# Structural Patterns – Proxy

- **Intent – Provide a surrogate or placeholder for another object to control access to it.**
- **Also Known As** – Surrogate
- **Motivation** – Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time

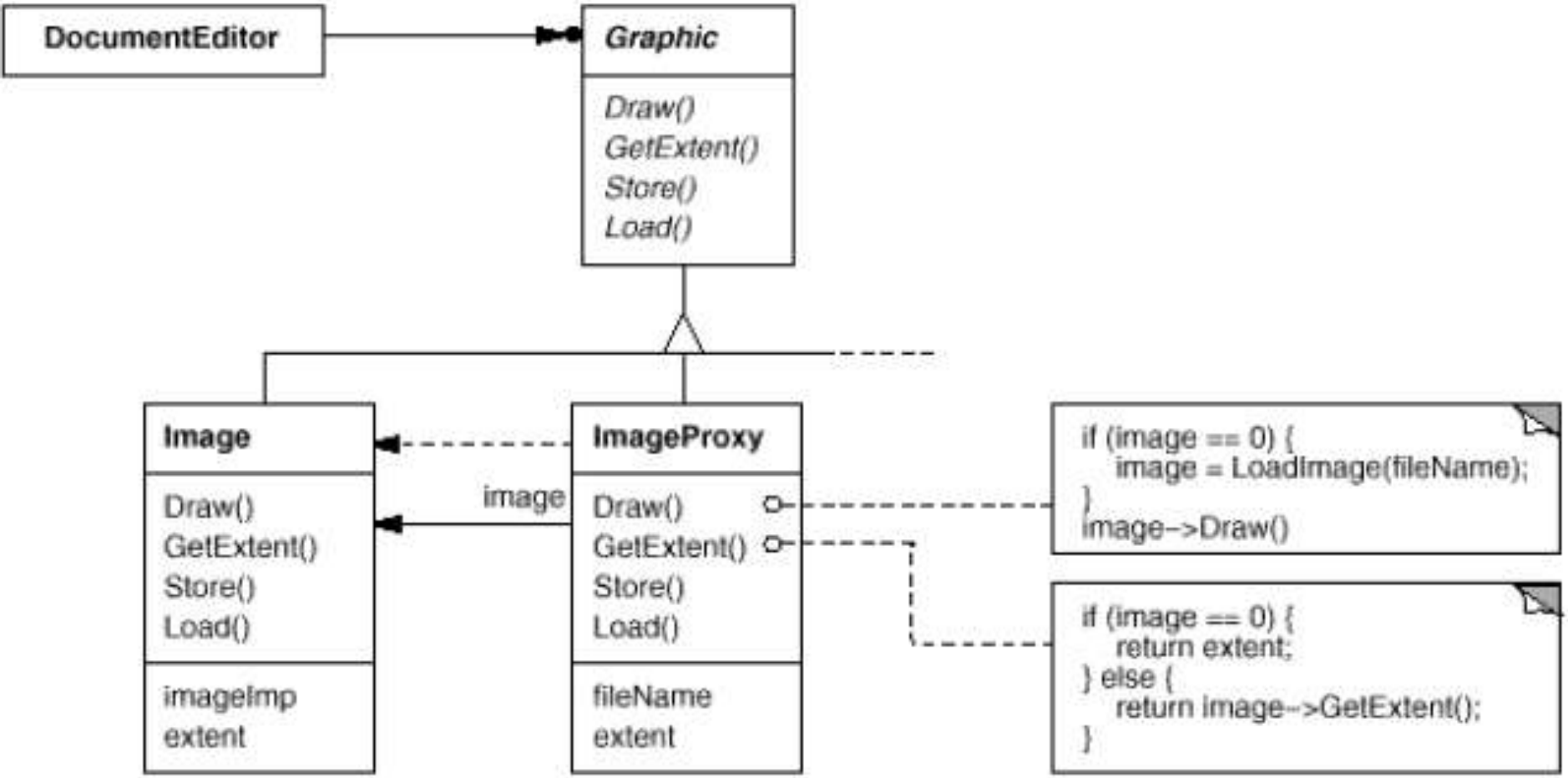
# Proxy 1

- The solution is to use another object, an image **proxy, that acts as a stand-in** for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



# Proxy 2

- The following class diagram illustrates this example in more detail



# Proxy 3

- **Applicability** – Use the Proxy pattern when
  - You need to provide some interposed service between the application logic and the client
  - Provide some lightweight version of a service or resource
  - Screen or restrict user access to a resource or service

# Proxy – Example

- Let’ say we need to withdraw money to make some purchase. The way we will do it is, go to an ATM and get the money, or purchase straight with a cheque.
- In old days when ATMs and cheques were not available, what used to be the way??? Well, get your passbook, go to bank, get withdrawal form there, stand in a queue and withdraw money. Then go to the shop where you want to make the purchase.
- In this way, we can say that ATM or cheque in modern times act as proxies to the Bank.

# Proxy – Java 1

```
public class Bank {
   private int numberInQueue;

   public double getMoneyForPurchase(double amountNeeded) {
         You you = new You("Prashant");
         Account account = new Account();
         String accountNumber = you.getAccountNumber();
         boolean gotPassbook = you.getPassbook();
         int number = getNumberInQueue();

         while (number != 0) {number--; }

         boolean isBalanceSufficient =
   account.checkBalance(accountNumber, amountNeeded);
         if(isBalanceSufficient)
                  return amountNeeded;
         else
                  return 0;
   }

   private int getNumberInQueue() {
         return numberInQueue;}
}
```

# Proxy – Java 2

```
public class ATMProxy {
  public double getMoneyForPurchase(double amountNeeded){
    You you = new You("Prashant");
    Account account = new Account();
    boolean isBalanceAvailable = false;
    if(you.getCard()) {
        isBalanceAvailable =
        account.checkBalance(you.getAccountNumber(),
        amountNeeded);
   }

   if(isBalanceAvailable)
        return amountNeeded;
   else
        return 0;
   }
}
```

# Proxy – The Good, The Bad …

- The provided service can be changed without affecting the client
- The proxy is available even if the base service or resource may be unavailable
- Preserves O (from SOLID) – you can add new proxies without changing the service or client
- It usually delays the response to the client
- The code is complicated because of increased number of classes

# Behavioral Patterns 1

- Behavioral patterns are concerned with **algorithms** and the **assignment of responsibilities between objects**
- These patterns **characterize complex control flow** that's difficult to follow at run-time
- They shift your focus away from flow of control to let you **concentrate just on the way objects are interconnected**

# Behavioral Patterns 2

- **Encapsulating variation** is a theme of many behavioral patterns
- When an **aspect of a program changes frequently**, these patterns define an object that encapsulates that aspect
- Then other **parts of the program can collaborate with the object** whenever they depend on that aspect

# Behavioral Patterns 3

- These patterns describe **aspects of a program that are likely to change**
- Most patterns have two kinds of objects:
  - the **new object**(s) that encapsulate the aspect,
  - and the **existing object**(s) that use the new ones
- Usually the **functionality of new objects would be an integral part of the existing objects** were it not for the pattern

# Patterns

- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

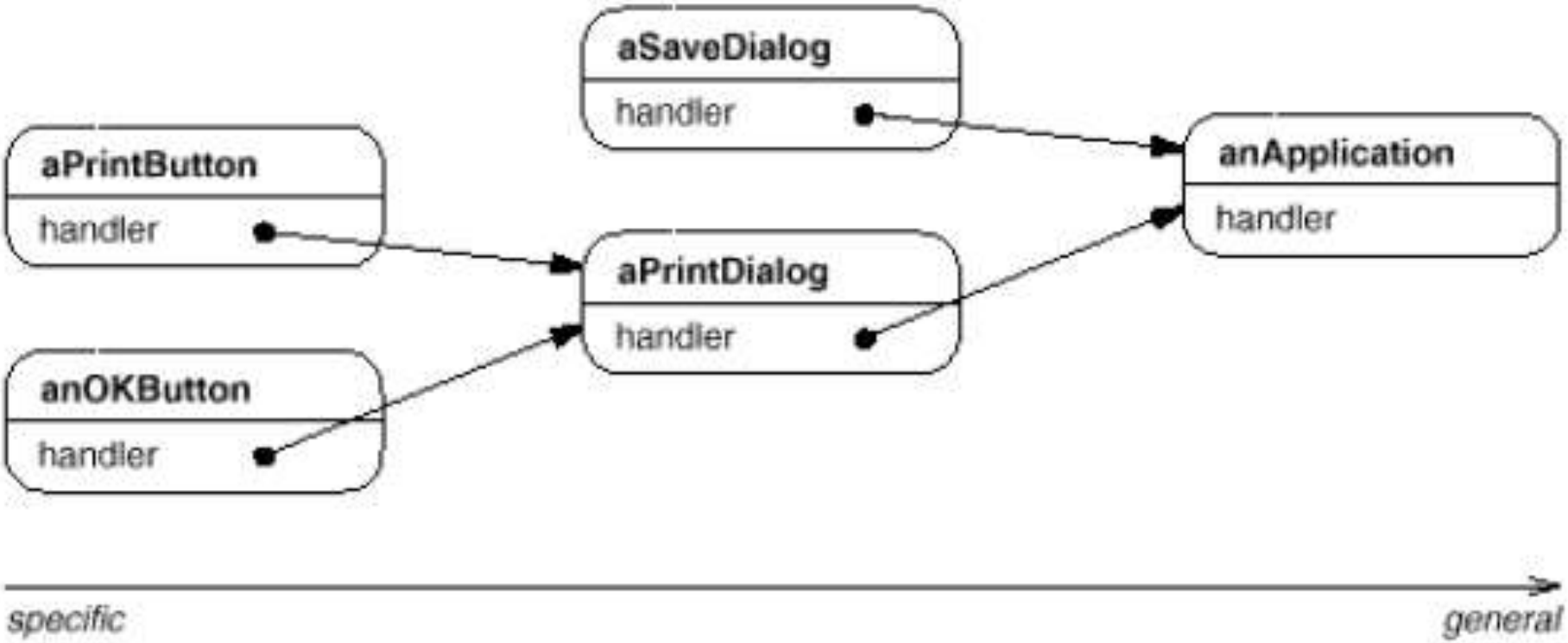
# Chain of Responsibility

- **Intent** – Chain the receiving objects and **pass the request along the chain until an object handles it**

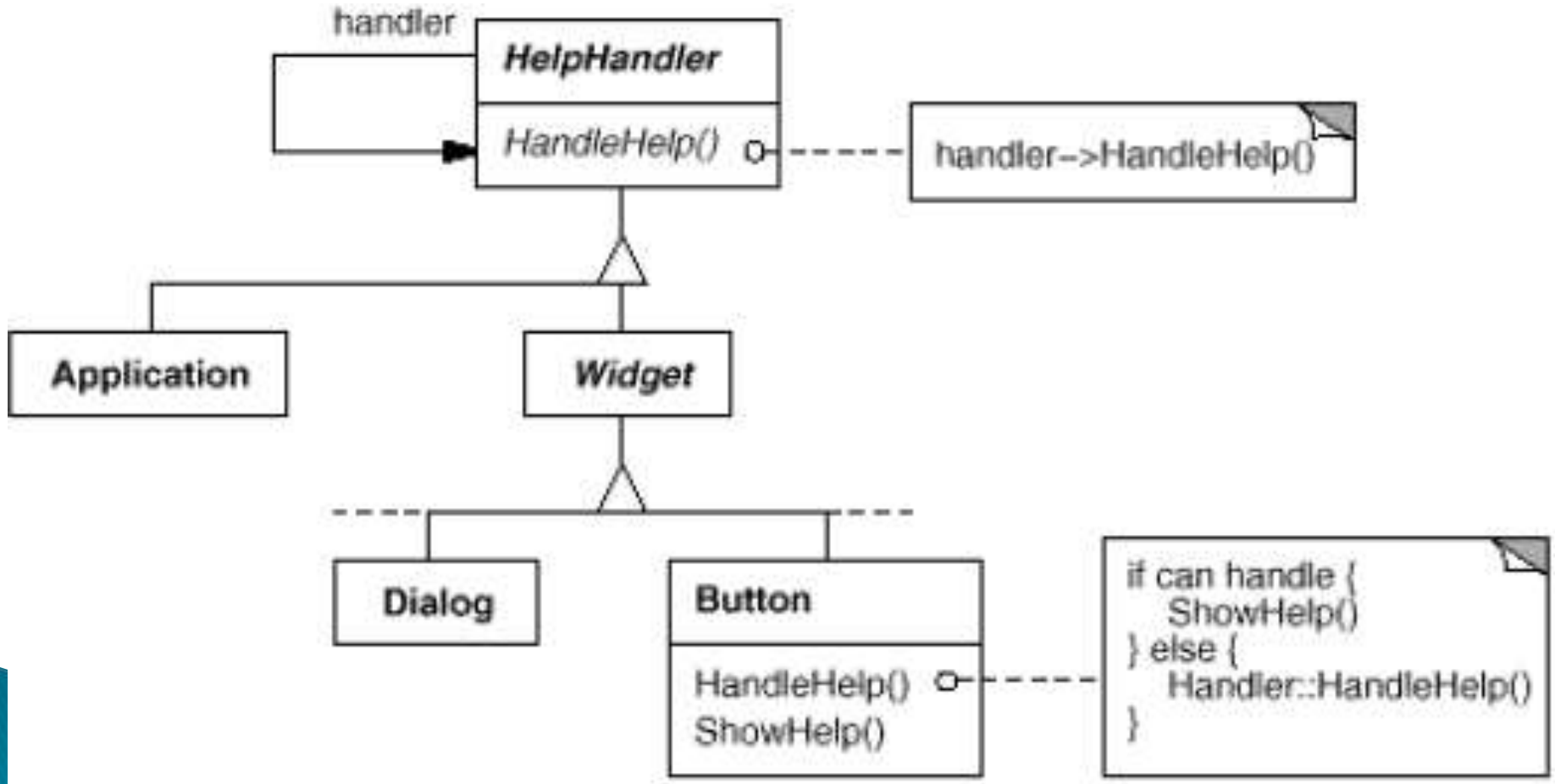
- **Motivation** – Consider a context-sensitive help facility for a graphical user interface. The help that's provided depends on the part of the interface that's selected and its context. If no specific help information exists for that part of the interface, then the help system should display a more general help message about the immediate context

# Chain of Responsibility – Idea

- It's natural to organize help information **from the most specific to the most general**



# Chain of Responsibility – Structure



# Chain of Responsibility

- **Applicability** – Use this pattern when
  - more than one object may handle a request, and the handler isn't known *a priori*
  - you want to issue a request to one of several objects without specifying the receiver explicitly
  - the set of objects that can handle a request should be specified dynamically

# Chain of Responsibility – Example

- Suppose, we have a multi level filter and gravel of different sizes and shapes. We need to filter this gravel of different sizes to approx size categories
- We will put the gravel on the multi-level filtration unit, with the filter of maximum size at the top and then the sizes descending. The gravel with the maximum sizes will stay on the first one and rest will pass, again this cycle will repeat until, the finest of the gravel is filtered and is collected in the sill below the filters
- Each of the filters will have the sizes of gravel which cannot pass through it. And hence, we will have approx similar sizes of gravels grouped

# Chain of Responsibility – Java 1

```
public class Matter {
   private int size;
   private int quantity;

   public int getSize() {return size;}

   public void setSize(int size) {this.size = size;}

   public int getQuantity() {return quantity;}

   public void setQuantity(int quantity) {
        this.quantity = quantity;
   }
}
```

# Chain of Responsibility – Java 2

```
public class Sill {
   public void collect(Matter gravel) {}
}

public class Filter1 extends Sill {
   private int size;

   public Filter1(int size) {this.size = size;}

   public void collect(Matter gravel) {
   for(int i = 0; i < gravel.getQuantity(); i++) {
         if(gravel.getSize() < size) {
                  super.collect(gravel);}
         else {
   //collect here. that means, only matter with less size will
   pass
         }
   }}}
```

# Chain of Responsability– The Good, The Bad …

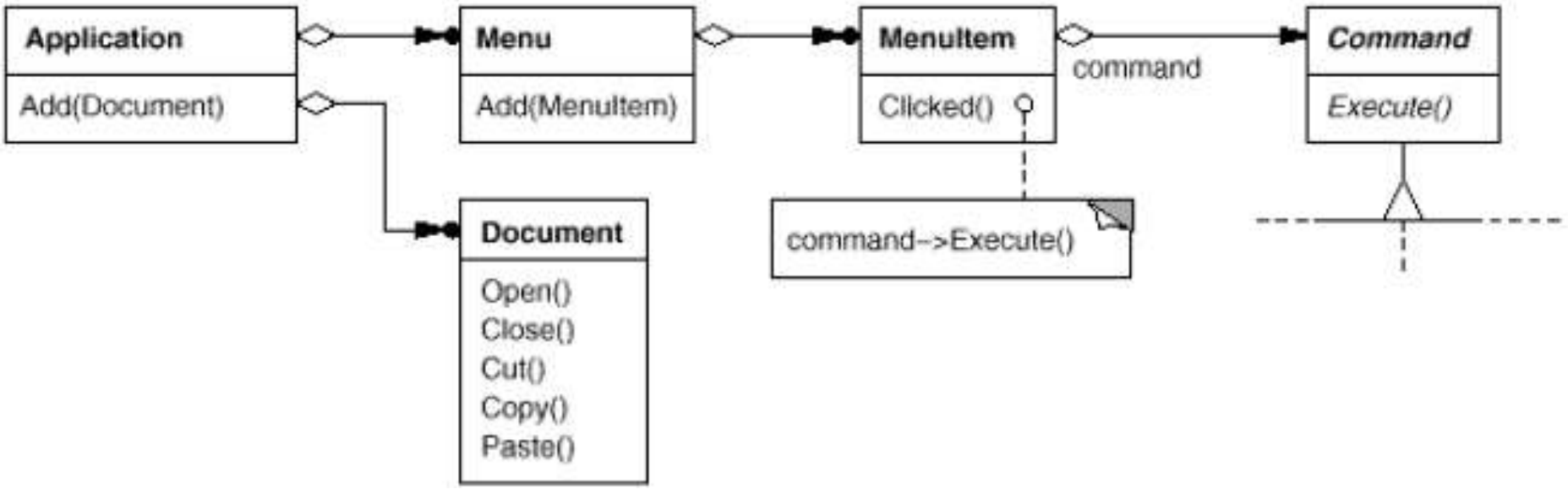
- Control the sequence of handler calls
- Preserves S and O (from SOLID)
- Some requests may not be handled by any class

# Command

- **Intent – Encapsulate a request as an object**, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- **Also Known As** – Action, Transaction
- **Motivation** – Sometimes **it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request**
- For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input. **But the toolkit can't implement the request explicitly in the button or menu**, because only applications that use the toolkit know what should be done on which object

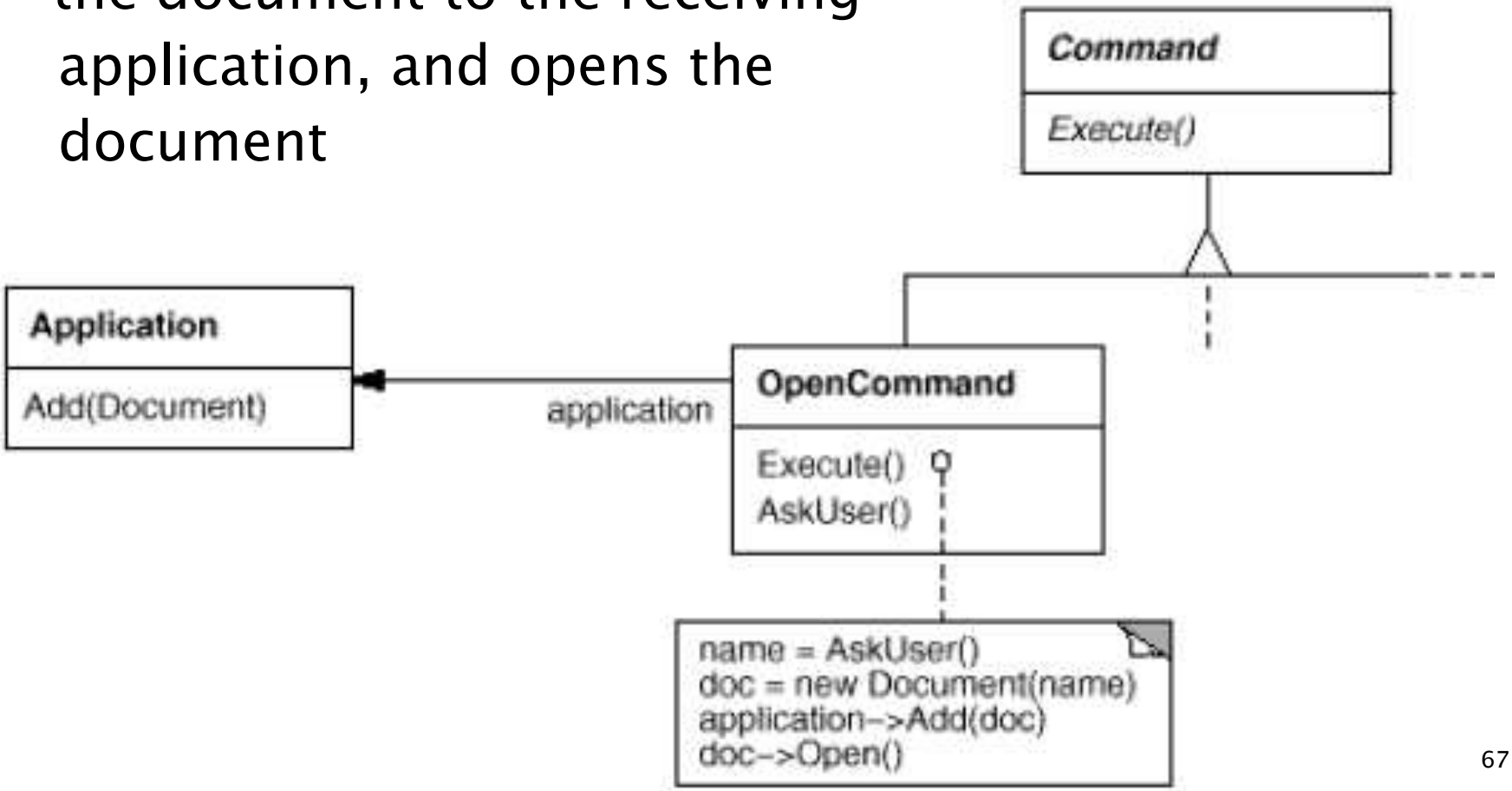
# Command 2

- The key to this pattern is an abstract Command class, which declares an interface for executing operations



# Command – Structure

- OpenCommand prompts the user for a document name, creates a corresponding Document object, adds the document to the receiving application, and opens the document



# Command – Example

- A classic example of this pattern is a restaurant:
  - A **customer** goes to restaurant and orders the food according to his/her choice
  - The **waiter/ waitress** takes the order (command, in this case) and hands it to the *cook* in the kitchen
  - The **cook** can make several types of food and so, he/she prepares the ordered item and hands it over to the *waiter/waitress* who in turn serves to the *customer*

# Command – Java 1

```
public class Order {
  private String command;
  public Order(String command) {
      this.command = command;
  } }
public class Waiter {
  public Food takeOrder(Customer cust, Order
  order) {
      Cook cook = new Cook();
      Food food = cook.prepareOrder(order, this);
      return food;
  } }
```

# Command – Java 2

```
public class Cook {
  public Food prepareOrder(Order order, Waiter
  waiter) {
      Food food = getCookedFood(order);
      return food;
  }
  public Food getCookedFood(Order order) {
      Food food = new Food(order);
      return food;
  }
}
```

# Command– The Good, The Bad …

- Supports undo/redo types of operations
- Preserves S and O (from SOLID)
- Combine simple commands into a single complex one
- Allows delaying execution
- Code becomes complicated because of an extra layer of code between caller and service

# Bibliography


- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* (GangOfFour)

# Links

- Structural Patterns: http://www.oodesign.com/structural-patterns/
- Gang-Of-Four: http://c2.com/cgi/wiki?GangOfFour, http://www.uml.org.cn/c%2B%2B/pdf/DesignPatterns.pdf
- Design Patterns Book: http://c2.com/cgi/wiki?DesignPatternsBook
- About Design Patterns: http://www.javacamp.org/designPattern/
- Design Patterns – Java companion: http://www.patterndepot.com/put/8/JavaPatterns.htm
- Java Design patterns: http://www.allapplabs.com/java_design_patterns/java_design_patterns.htm
- Overview of Design Patterns: http://www.mindspring.com/~mgrand/pattern_synopses.htm https://refactoring.guru/design-patterns