Ingineria programării

Curs 7 – 3 April

Din Cursurile Trecute

▸ Design Patterns: Definitions, Elements

▸ GOF: Creational Patterns, Structural Patterns, Behavioral Patterns

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton
- Lazy initialization
- Object pool
- Multiton
- Resource acquisition

Creational Patterns – Introduction

- Abstract the instantiation process
- Help **make a system independent of how its objects are created, composed, and represented**
- There are two recurring themes in these patterns:
  - First, they all encapsulate knowledge about which concrete classes the system uses
  - Second, they hide how instances of these classes are created and put together
- The creational patterns give a lot of flexibility in *what* gets created, *who creates it, how it gets created, and when*

Creational Patterns – Abstract Factory

- **Intent** – Provide an interface for creating families of related or dependent objects without specifying their concrete classes

- **Also Known As** – Kit

- **Motivation**
  - Consider a user interface toolkit that supports multiple look-and-feel standards, defines different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons
  - To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel

Abstract Factory 3

▸ **Applicability** – Use this pattern when:
  ◦ a system should be independent of how its products are created, composed, and represented

  ◦ a system should be configured with one of multiple families of products

  ◦ a family of related product objects is designed to be used together, and you need to enforce this constraint

  ◦ you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

CP – Abstract Factory

▶ **Abstract Factory** – maze, computer components

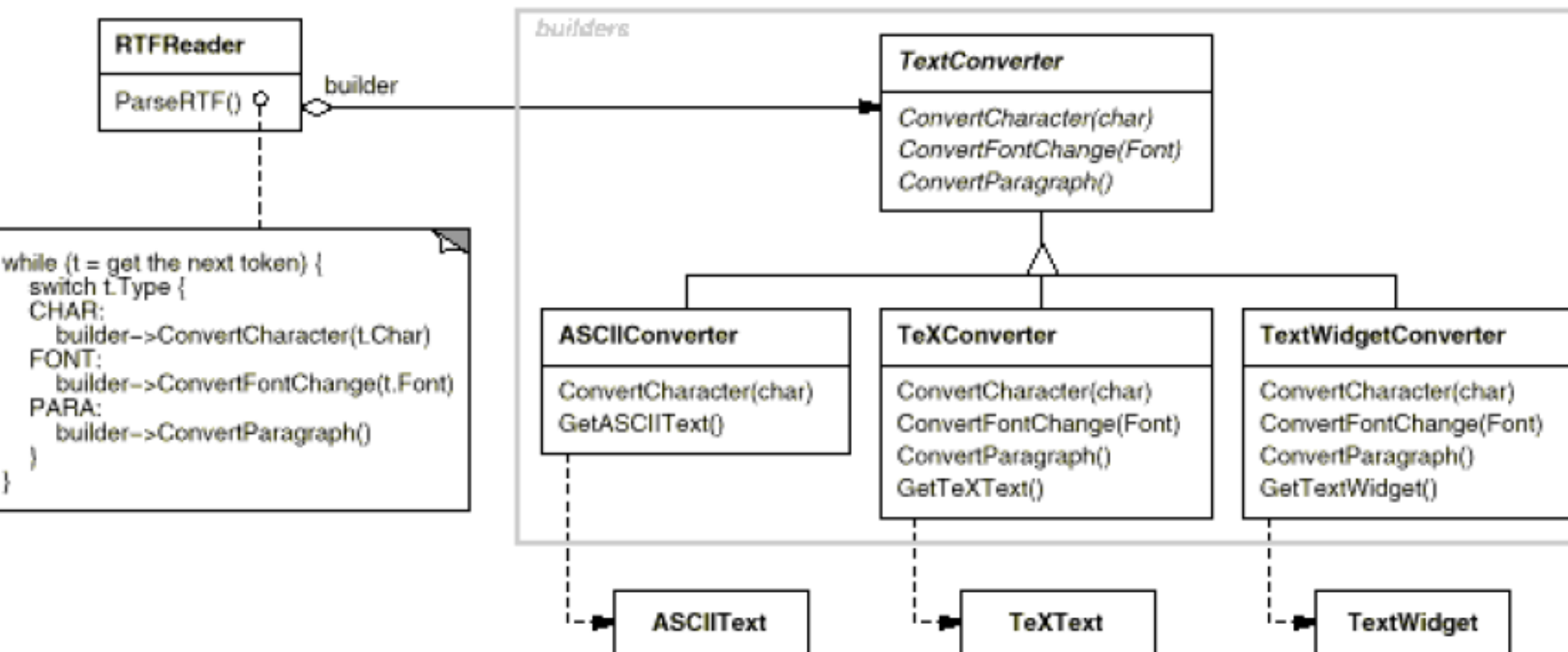
cd: Abstract Factory Implementation - UML Class Diagram
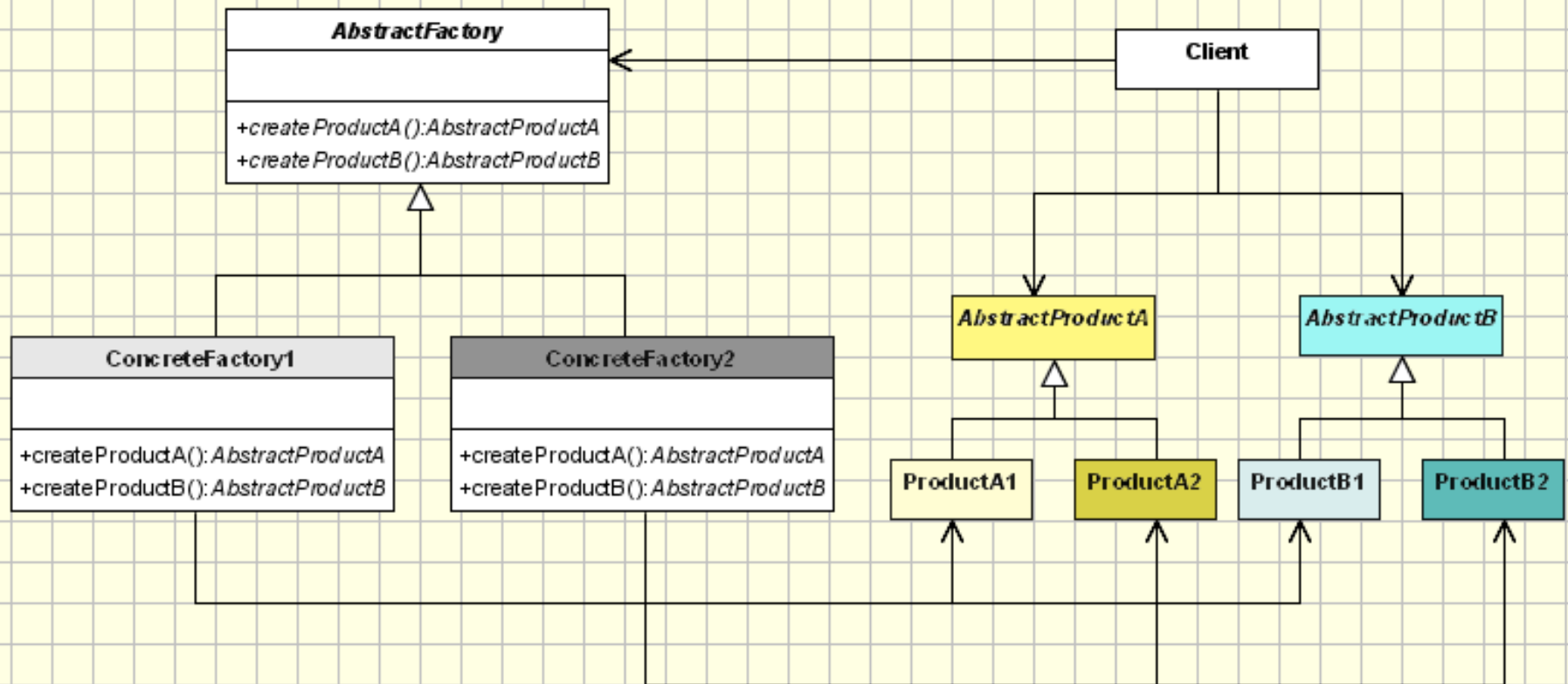
Creational Patterns – Builder

▶ **Intent - Separate the construction of a complex object from its representation** so that the same construction process can create different representations

▶ **Motivation**

◦ A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats.

◦ The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.
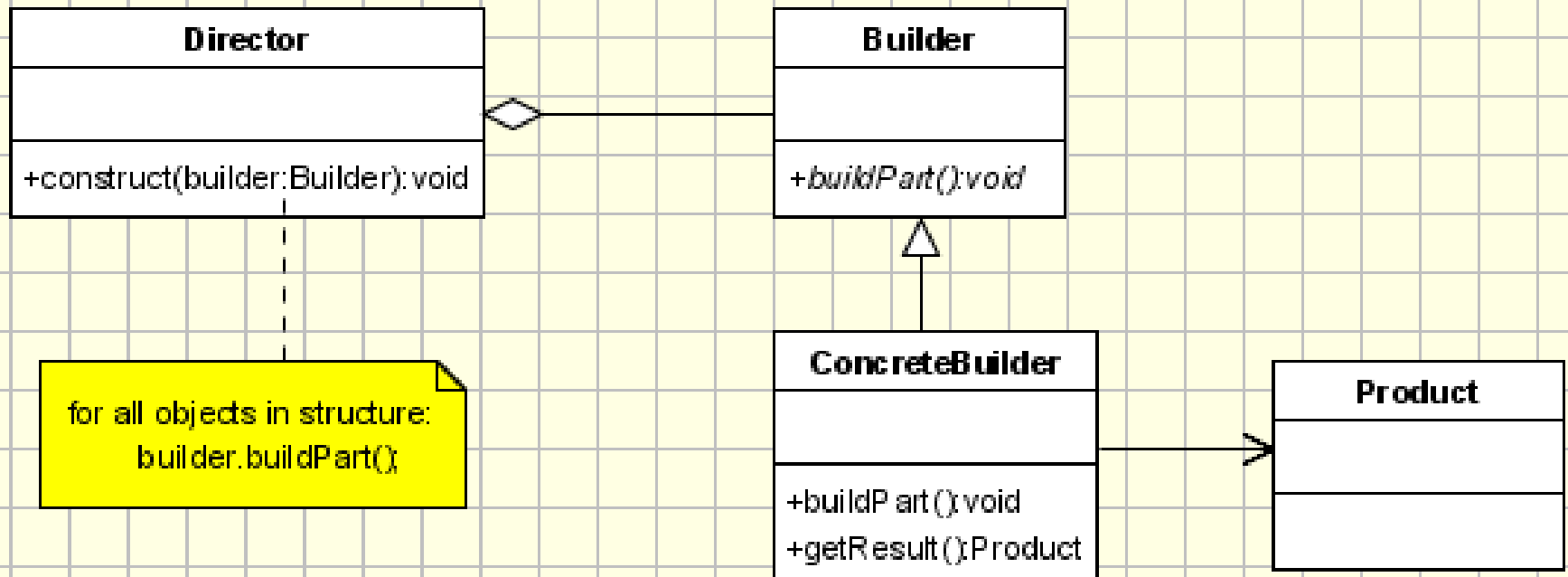
Builder 2

- Each kind of converter class takes the mechanism for creating and assembling
- The converter is separate from the reader

CP – Builder

- **Builder** – RTF Reader, Happy Meal

cd: Builder Implementation - UML Class Diagram

Director

+construct(builder:Builder):void

for all objects in structure:
builder.buildPart();

Builder

+buildPart():void

ConcreteBuilder
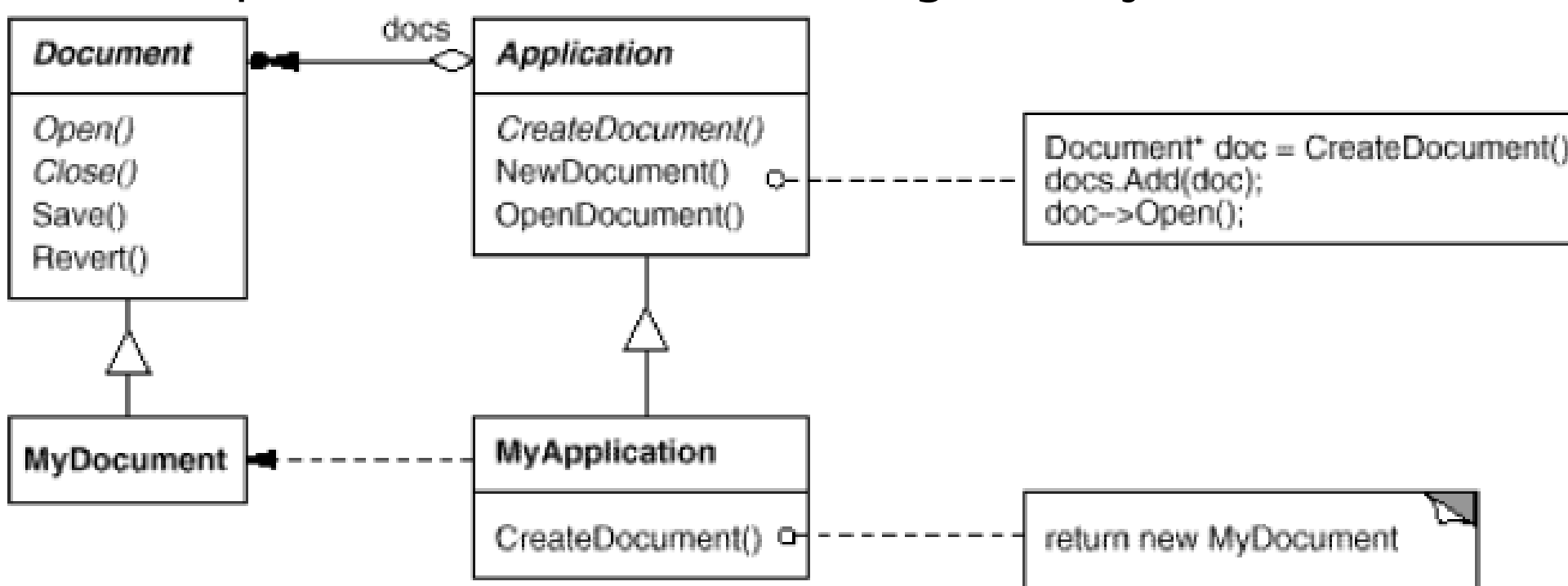
+buildPart():void
+getResult():Product

Product

Creational Patterns – Factory Method

- **Intent** – Define an interface for creating an object, but let subclasses decide which class to instantiate
- **Also Known As** – Virtual Constructor
- **Motivation** – To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required (at Open or New from a menu, for example)

Factory Method 2

- Subclasses redefine an abstract CreateDocument
- It can then instantiate application-specific Documents without knowing their class
- CreateDocument is a **factory method because it's** responsible for "manufacturing" an object

```
Document                    docs        Application
Open()                                  CreateDocument()        Document* doc = CreateDocument();
Close()                                 NewDocument()           docs.Add(doc);
Save()                                  OpenDocument()          doc->Open();
Revert()

MyDocument                              MyApplication

                                        CreateDocument()        return new MyDocument
```

Factory Method 3

▶ **Applicability** – Use this pattern when

◦ a class can't anticipate the class of objects it must create

◦ a class wants its subclasses to specify the objects it creates

◦ classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate
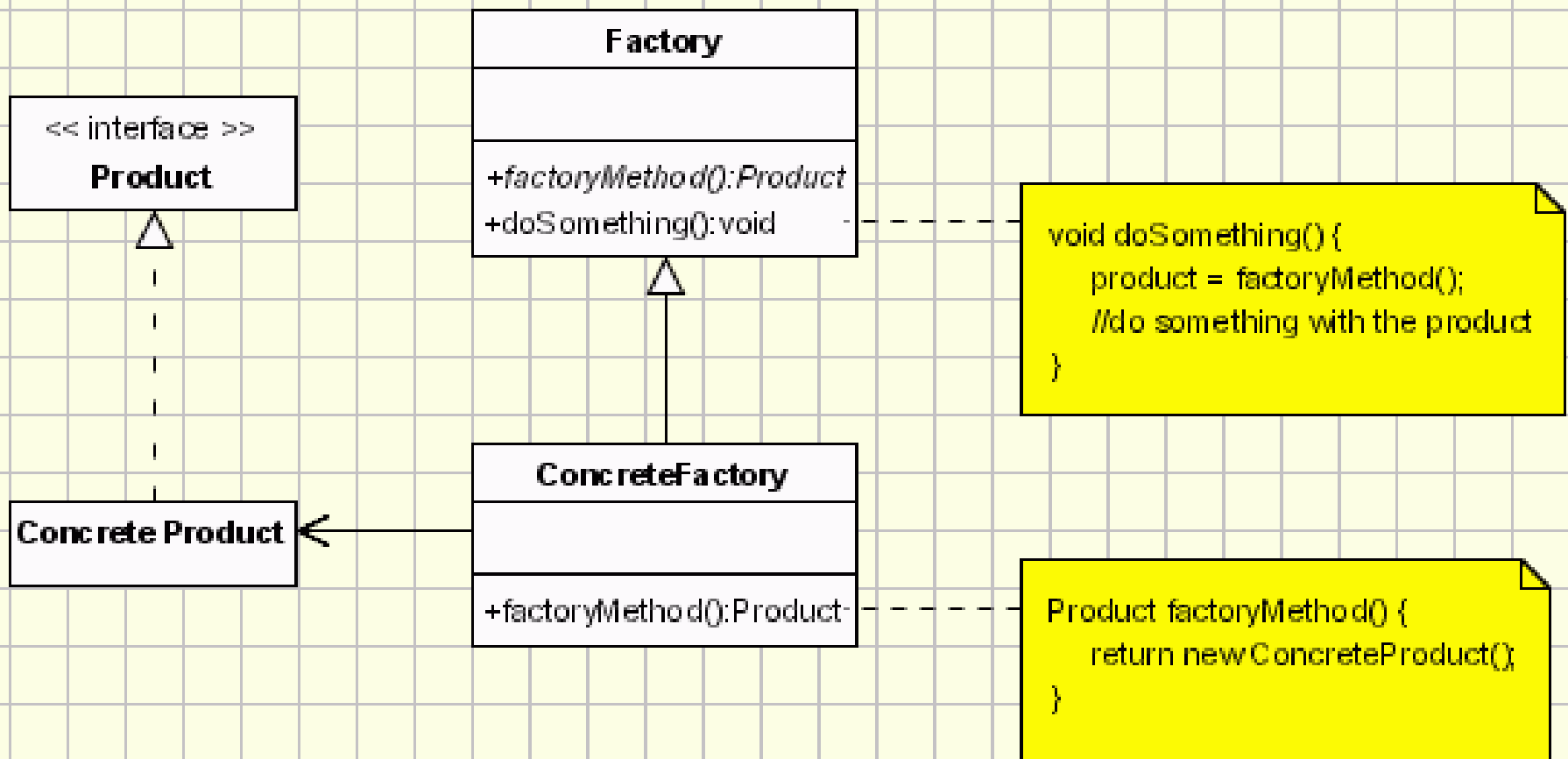
Factory Method 5

▶ **Consequences**

1. *Provides interconnections for subclasses*

2. *Connects parallel class hierarchies*

CP – Factory Method

▸ **Factory Method** – Open/New Project, Hello Mr/Ms

Creational Patterns – Prototype

- **Intent** – Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- **Motivation** – You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves
- The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects

Prototype 2

- We can use the Prototype pattern to reduce the number of classes
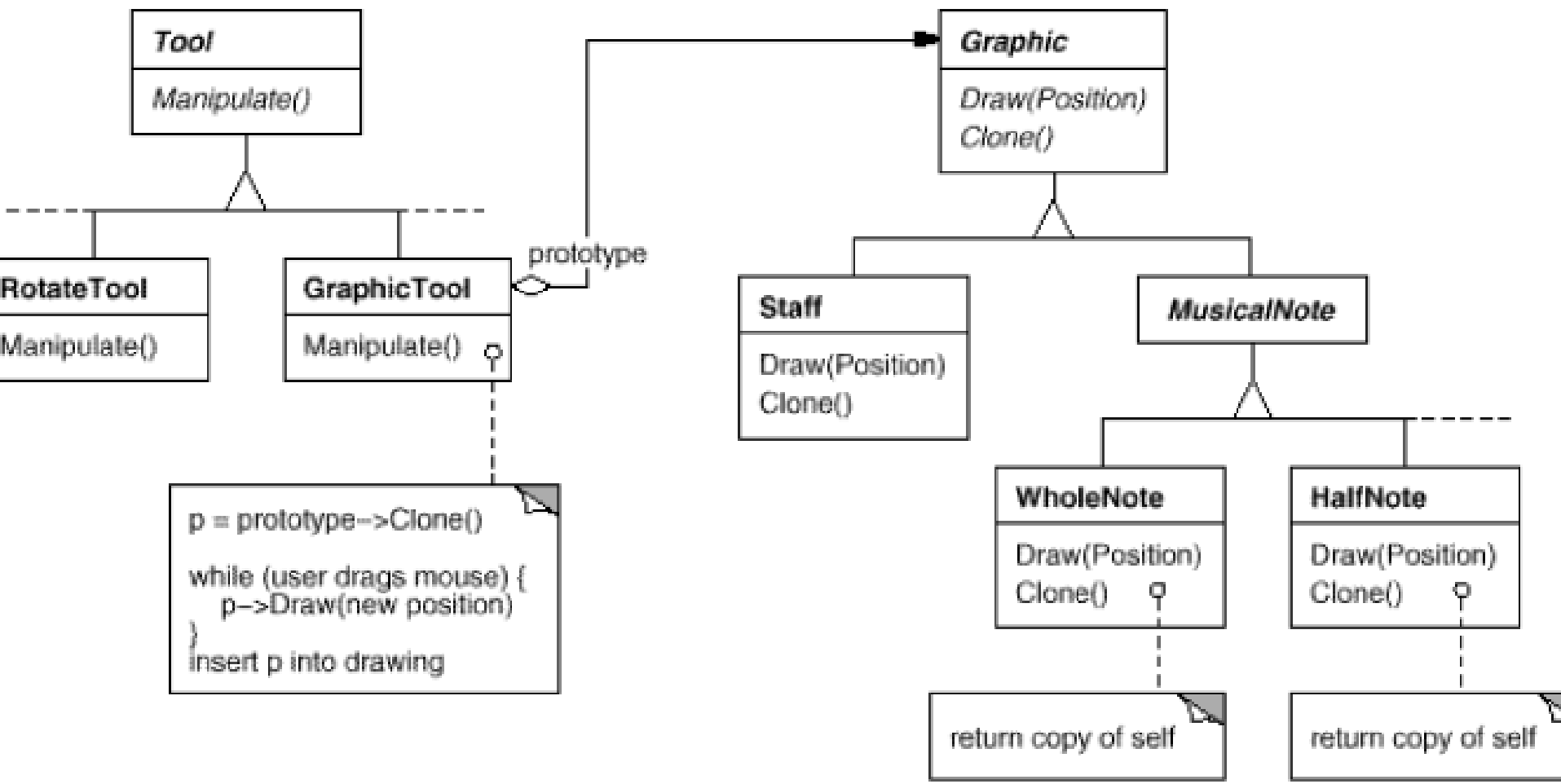
Prototype 3

▸ **Applicability** – Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; *and*

◦ when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*

◦ to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*

◦ when instances of a class can have one of only a few different combinations of state

# CP – Prototype

- **Prototype** – Music editor, Clonable
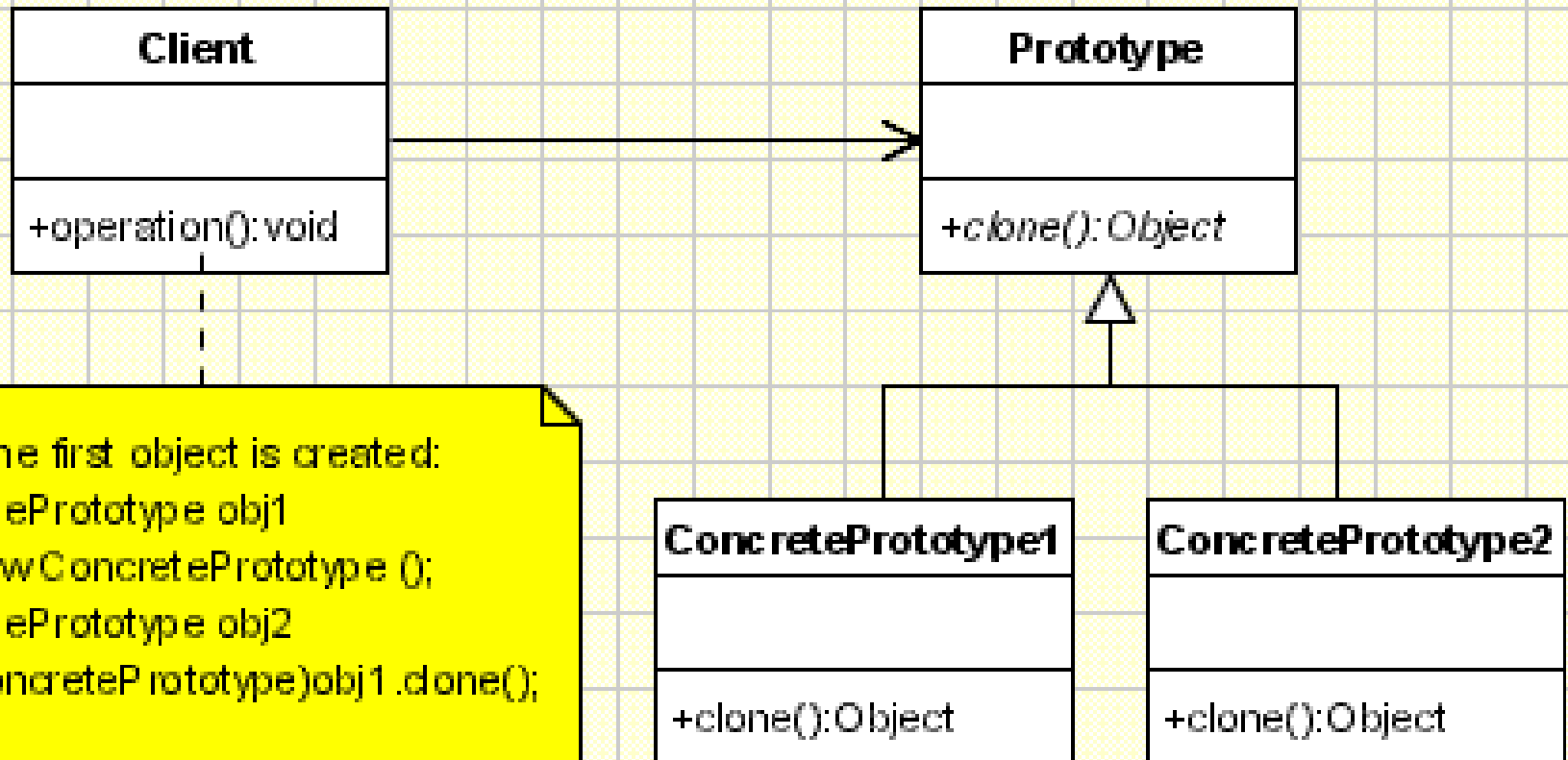
Creational Patterns – Singleton

- **Intent** – Ensure a class only has one instance, and provide a global point of access to it

- **Motivation** – It's important for some classes to have exactly one instance. There should be only one file system and one window manager. An accounting system will be dedicated to serving one company.

- How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

CP – Singleton

▸ **Singleton** – Logger

cd: Singleton Implementation- UML Class diagram

Singleton
-instance:Singleton
-Singleton():   +getInstance():Singleton
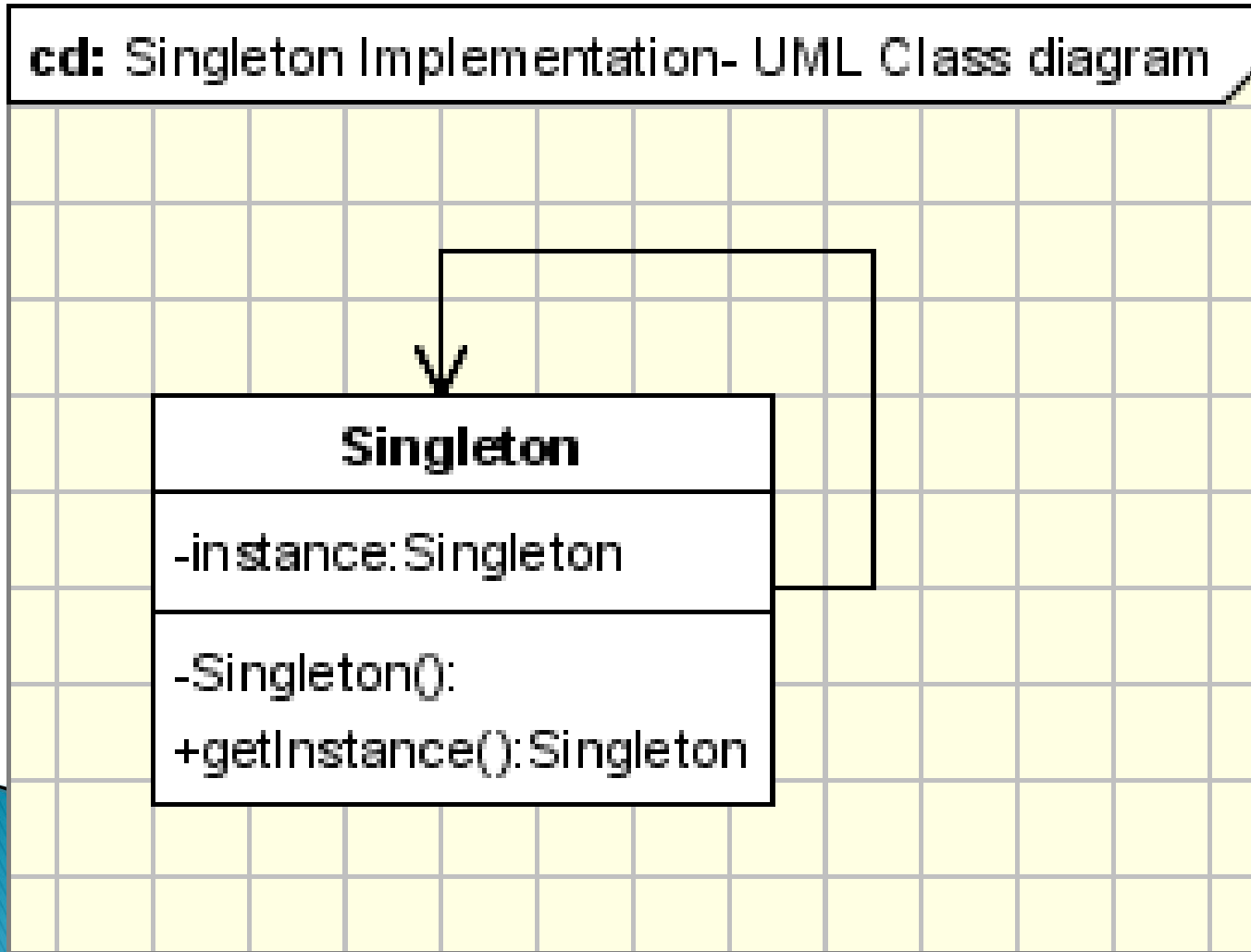
CP – Lazy initialization

- **Lazy initialization** is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed

- How? With a **flag**
- Each time when the object is called, the **flag** is tested:
  - If it is **ready**, it is **returned**
  - If **not**, it is **initialized**

Lazy initialization – Java 1

```
public class Fruit {
  private static final Map<String,Fruit> types = new
  HashMap<String, Fruit>();
  private final String type;

  //using a private constructor to force use of the factory method
  private Fruit(String type) { this.type = type; }

  public static synchronized Fruit getFruit(String type) {
    if(!types.containsKey(type)) {
      types.put(type, new Fruit(type)); // Lazy initialization
    }
    return types.get(type);
  }
}
```
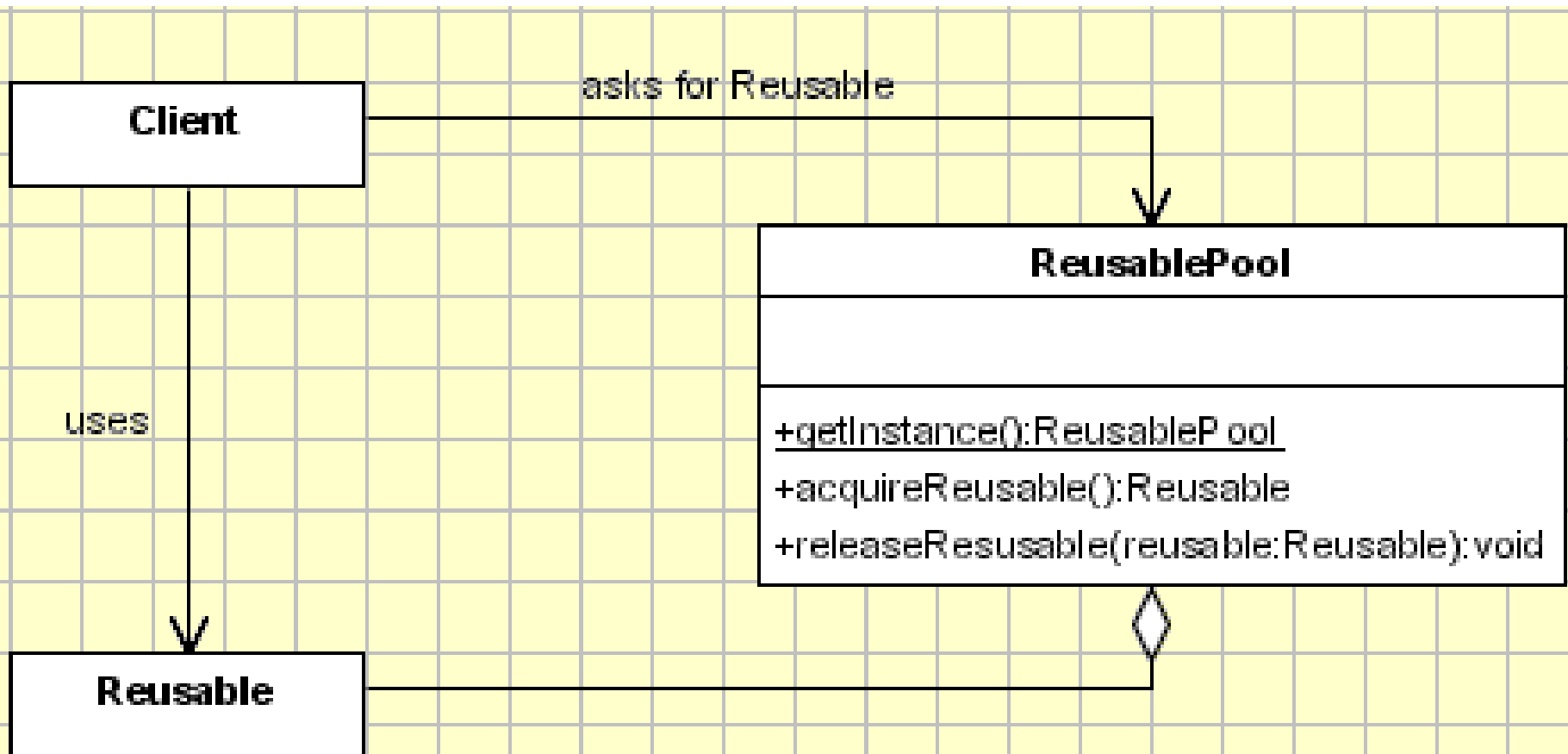
CP – Object Pool

- **Intent**: reuse and share objects that are expensive to create.

- **Motivation**: Performance can be sometimes the key issue during the software development and **the object creation (class instantiation) is a costly step**. The Object Pool pattern offer a mechanism to reuse objects that are expensive to create

- **Why use it?** Basically, we'll use an object pool whenever there are several clients who needs the same stateless resource which is expensive to create

Object Pool – Implementation (1)

▶ When a client asks for a Reusable object, the pool performs the following actions:
  ◦ **Search** for an available Reusable object and **if it was found** it will be **returned** to the client.
  ◦ **If no Reusable object was found** then it **tries to create a new one.** If this actions **succeeds** the new Reusable object will be **returned** to the client.
  ◦ If the pool **was unable to create a new Reusable**, the pool will **wait** until a reusable object will be released.

Object Pool – Implementation (2)

Client —— asks for Reusable —→ ReusablePool

ReusablePool

+getInstance():ReusablePool
+acquireReusable():Reusable
+releaseResusable(reusable:Reusable):void

Client —— uses —→ Reusable

Object Pool - Applicability

▸ Lets' take the example of the **database connections**. It's obviously that opening too many connections might affect the performance:
  ◦ **Creating a connection is an expensive** operation
  ◦ When there are **too many connections opened** it **takes longer to create a new one** and the database server will become overloaded

▸ Here the **object pool comes to manage the connections** and provide a way to reuse and share them. It can also limit the maximum number of objects that can be created

Structural Patterns 1

▸ **Structural patterns are concerned with how classes and objects are composed to form larger structures**

▸ **Example**: consider how **multiple inheritance** mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

Structural Patterns 2

- Rather than composing interfaces or implementations, structural *object patterns* describe ways to compose objects **to realize new functionality**
- **The added flexibility of object composition** comes from the ability to change the composition at run-time

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Structural Patterns – Adapter

- **Intent**: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Also Known As**: Wrapper
- **Motivation**: Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.

Adapter 1

▸ A TextShape subclass that can display and edit text is considerably more difficult to implement

▸ We suppose the existence of a TextView class for displaying and editing text. So we can consider TextShape derived from these classes
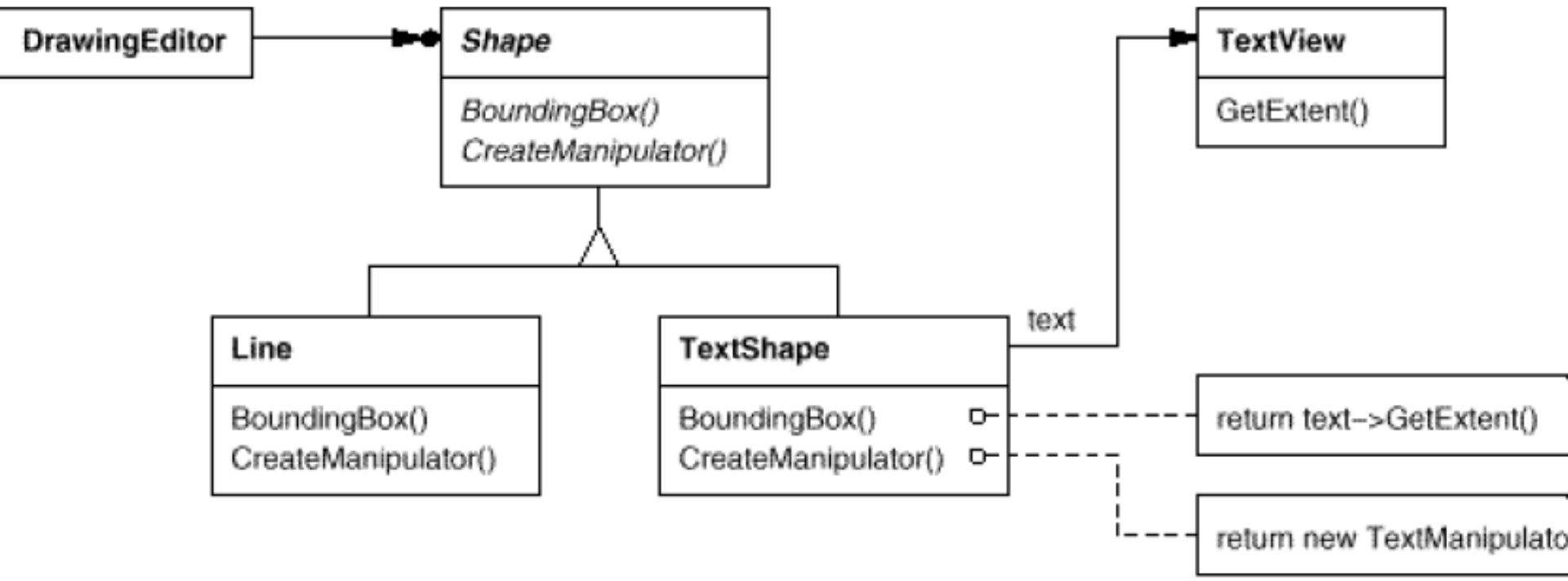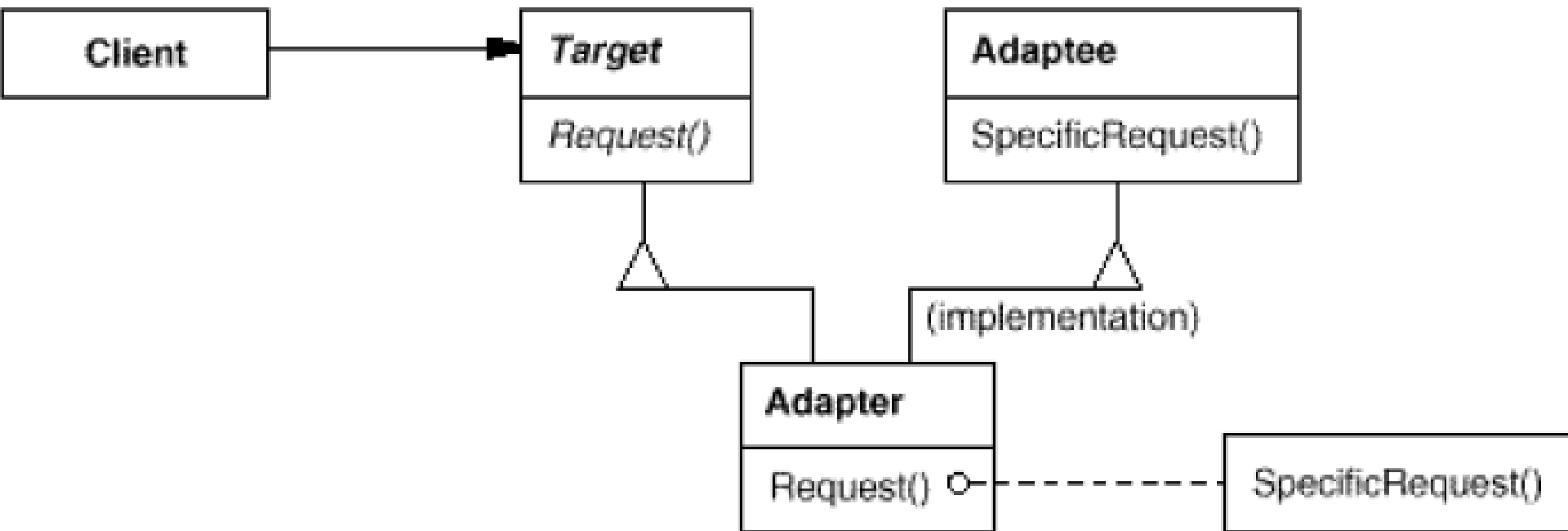
Adapter 2

▶ **Applicability** – Use the Adapter pattern when
  ◦ you want to use an existing class, and its interface does not match the one you need

  ◦ you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces

  ◦ *(object adapter only) you need to use several existing subclasses, but it's* impractical to adapt their interface by sub classing every one.

Adapter 3

▸ Structure - The Adapter pattern is used so that two unrelated interfaces can work together. This is something like we convert interface of one class into interface expected by the client.

Adapter – Example

- We all have electric sockets in our houses of different sizes and shapes. I will take an example of a socket of 15 Ampere. This is a bigger socket and the other one which is smaller is of 5 Ampere. A 15 Amp plug cannot fit into a 5 Amp socket => we will use an Adapter

- We have a 5 Amp plug and want a 5 Amp socket so that it can work. We DO NOT have a 5 Amp socket, what we have is a 15 Amp socket in which the 5 Amp plug cannot fit. The problem is how to cater to the client without changing the plug or socket.

Adapter – Java 1

```java
/** The socket class has a specs for 15 AMP.*/
public interface Socket {
	public String getOutput();
}


public class Plug {
  private String specification = "5 AMP";
  public String getInput() {
      return specification;
  }
}
```

Adapter – Java 2

```java
public class ConnectorAdapter implements Socket
{
    public String getOutput() {
        Plug plug = new Plug();
        String output = plug.getInput();
        return output;
    }
}
```

Structural Patterns – Bridge

- **Intent** – Decouple an abstraction from its implementation so that the two can vary independently

- **Also Known As** – Handle/Body

- **Motivation** – Consider the abstraction of shapes, each with its own properties. One thing all shapes can do is draw themselves. Drawing graphics to a screen can be dependent on different graphics implementations or operating systems. Shapes have to be able to be drawn on many types of operating systems

Bridge 1

▸ The bridge helps by allowing the creation of new implementation classes that provide the drawing implementation. Shape class provides methods for getting the size or properties of a shape. Drawing class provides an interface for drawing graphics
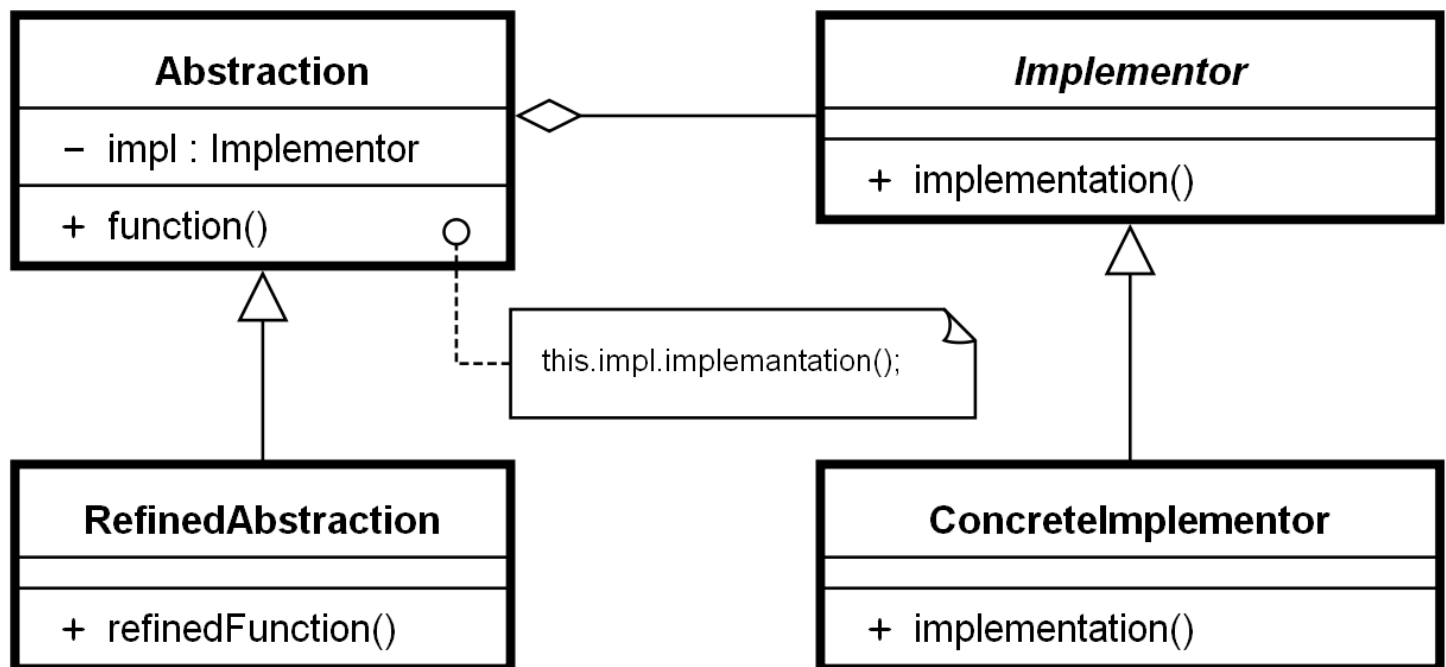
Bridge – Java 1

```java
interface DrawingAPI {/** "Implementor" */
    public void drawCircle(double x, double y, double
    radius);
}
/** "ConcreteImplementor" 1,2 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double
    radius) { System.out.printf("API1" + x + y + radius);}
}
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double
    radius) { System.out.printf("API2" + x + y + radius);}
}
```

Bridge – Java 2

```java
interface Shape {/** "Abstraction" */
   public void draw();
   public void resizeByPercentage(double pct); }

/** "Refined Abstraction" */
class CircleShape implements Shape {
   private double x, y, radius;
   private DrawingAPI drawingAPI;
   public CircleShape(double x, double y, double radius,
   DrawingAPI drawingAPI) {
        this.x = x; this.y = y; this.radius = radius;
        this.drawingAPI = drawingAPI;
   }
   public void draw() {drawingAPI.drawCircle(x, y, radius); }
   public void resizeByPercentage(double pct) { radius *= pct; }
}
```

# Bridge – Java 3

```
/** "Client" */
class BridgePattern {
  public static void main(String[] args) {
        Shape[] shapes = new Shape[2];
        shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
        shapes[1] = new CircleShape(5, 7, 8, new DrawingAPI2());
        for (Shape shape : shapes) {
                shape.resizeByPercentage(2.5);
                shape.draw();
        }
  }
}
```

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
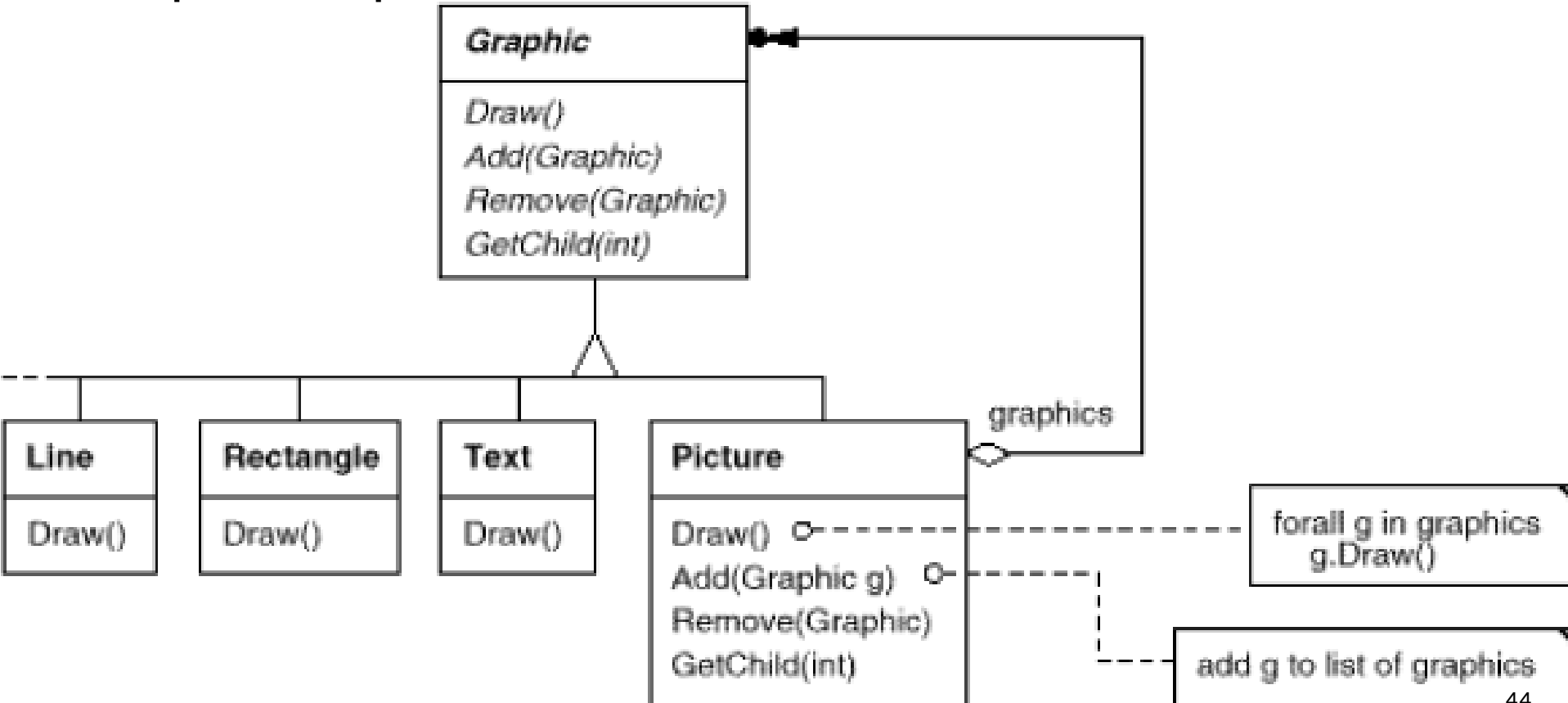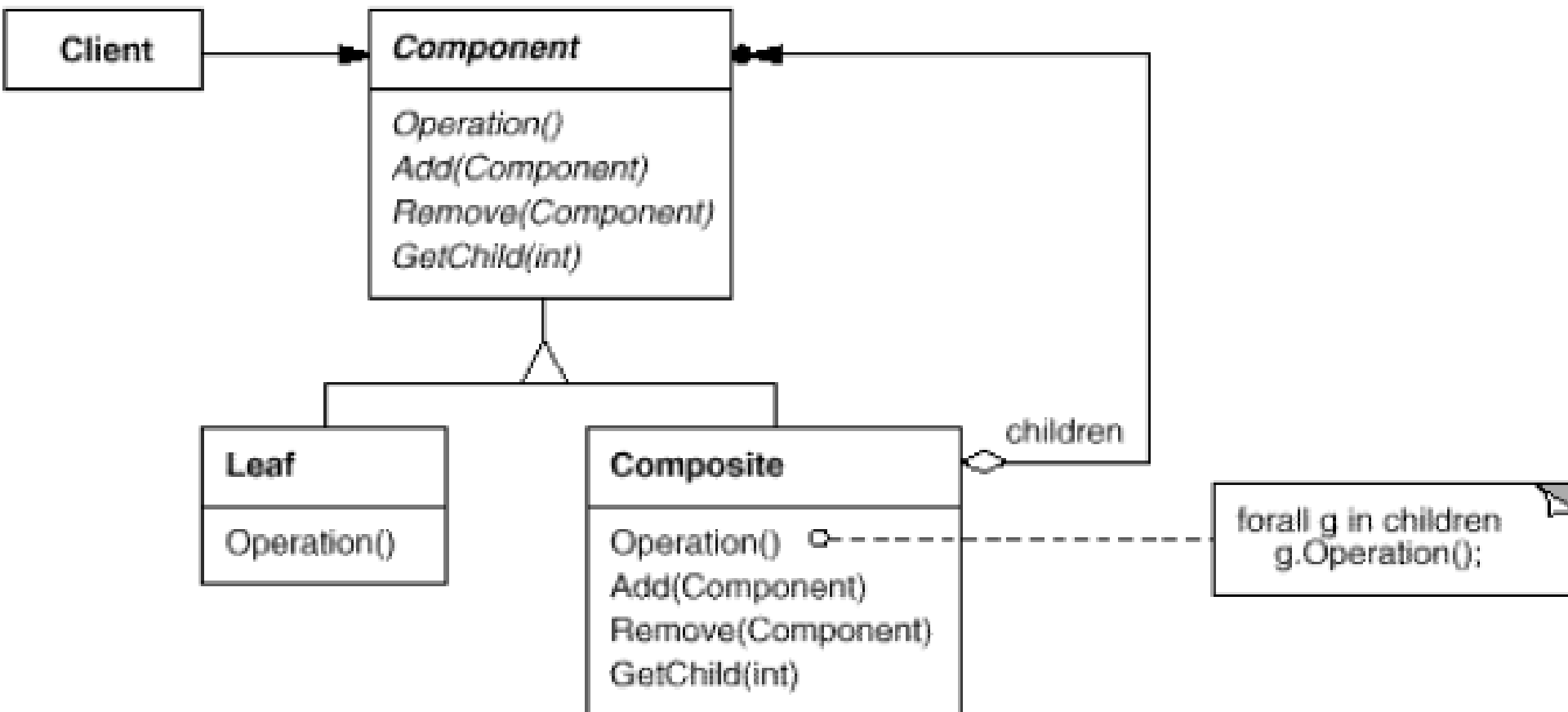
Composite 2

▸ **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

```java
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());
```

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
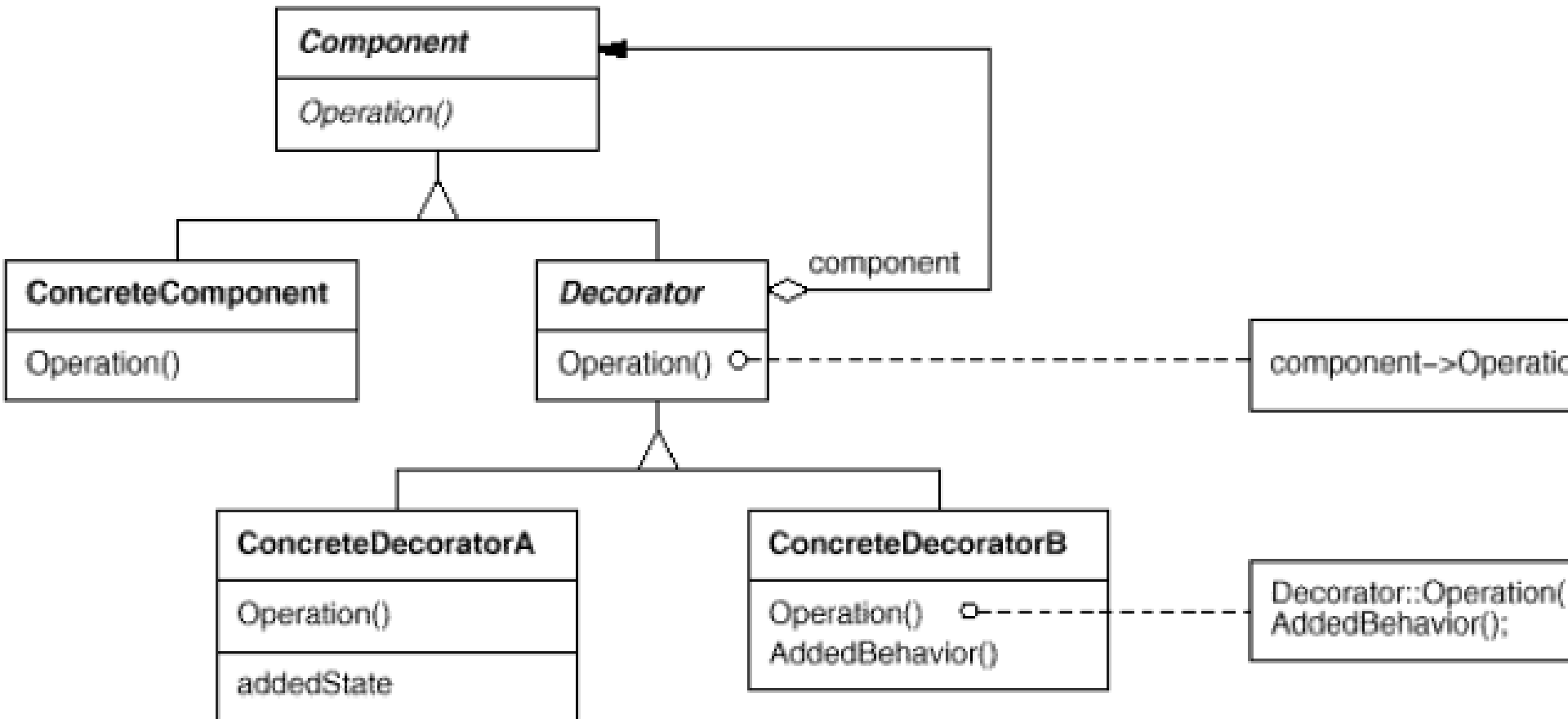
Decorator 1

▸ A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**
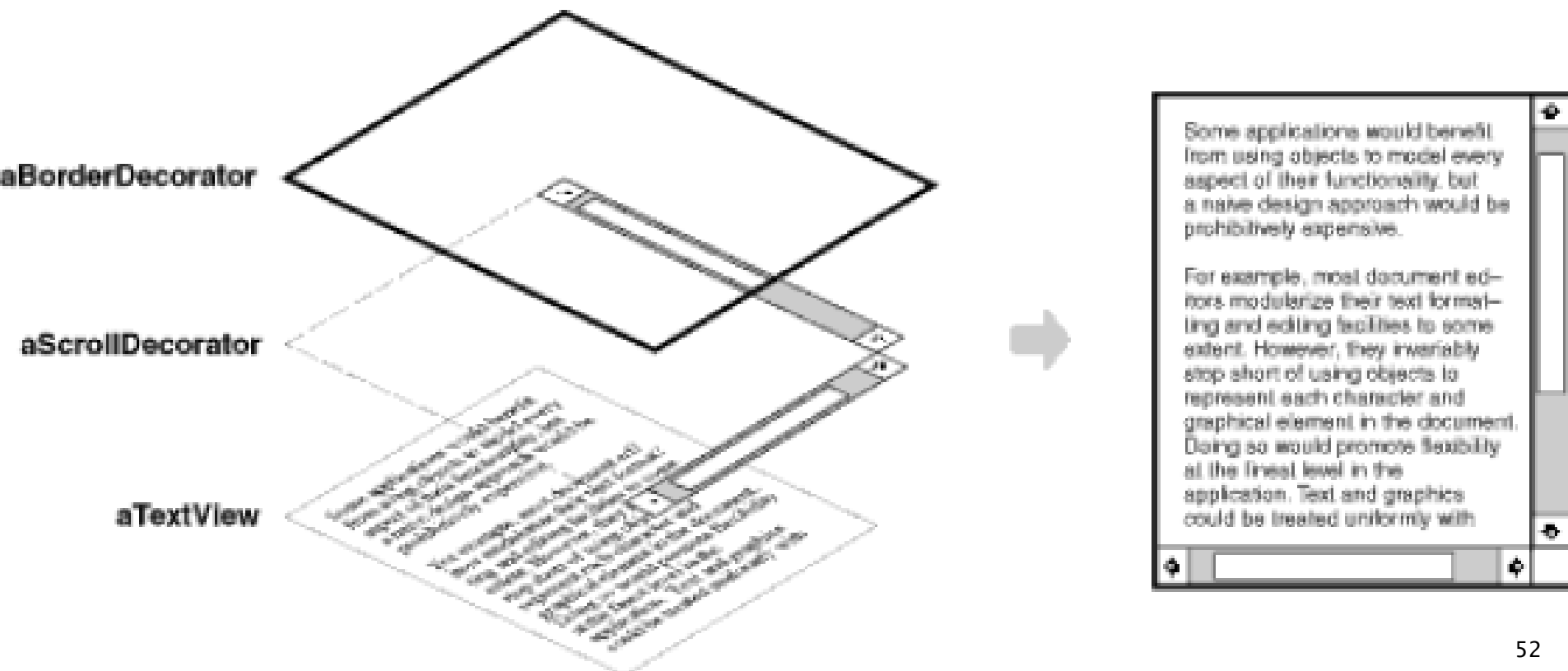
Decorator 2

▸ 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

```java
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

```java
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.

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
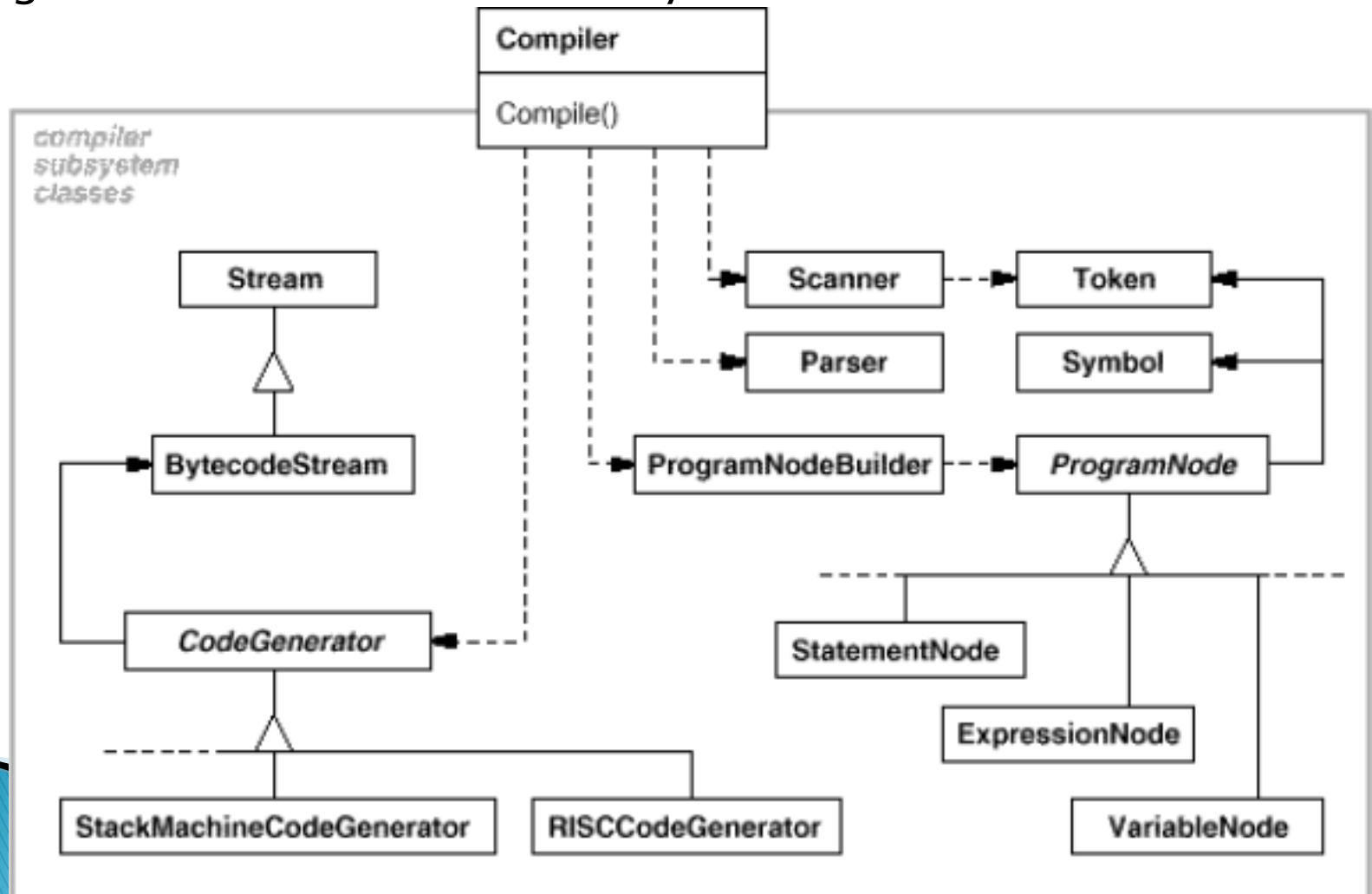
Façade 1

- A common design goal **is to minimize the communication and dependencies between** subsystems
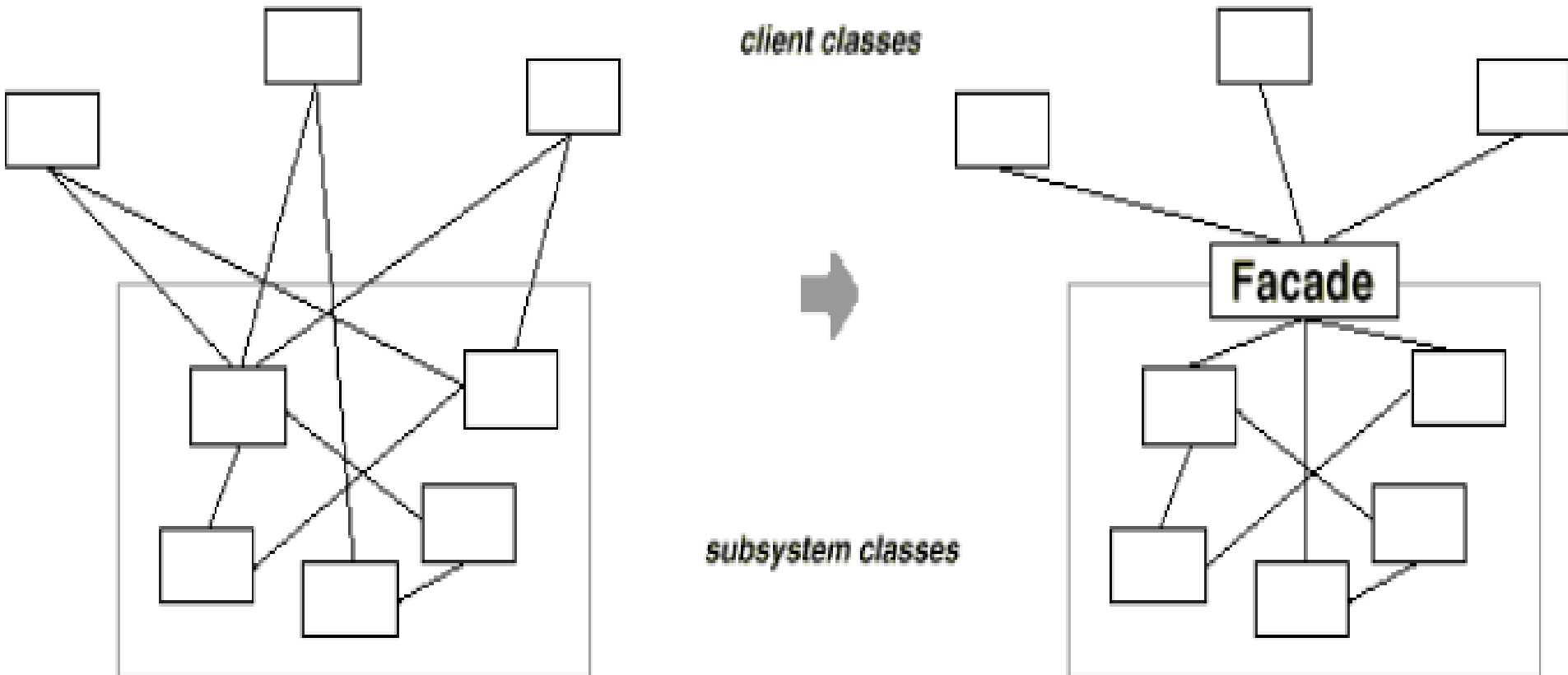
Façade 2

▸ 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

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


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

Façade – Java 2

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

Façade – Java 3

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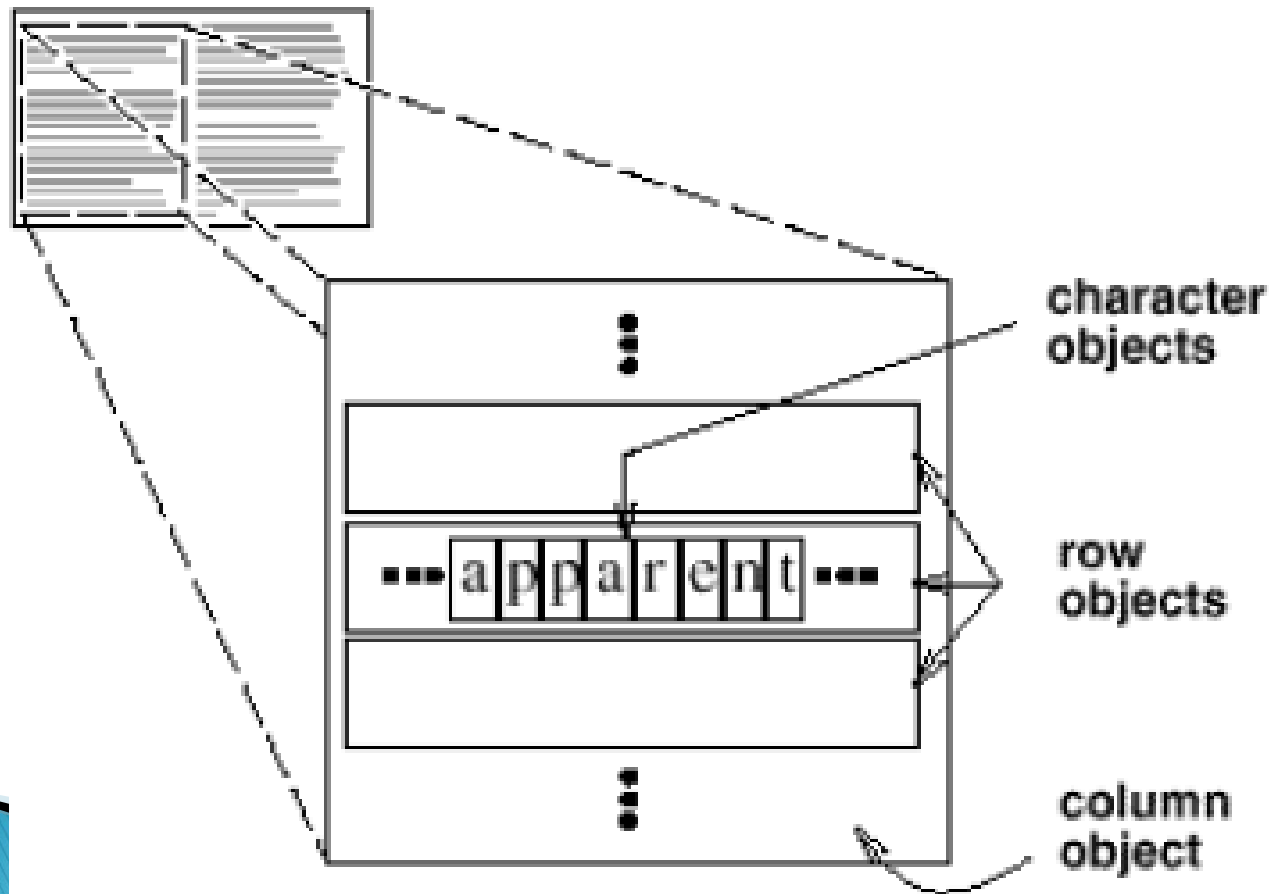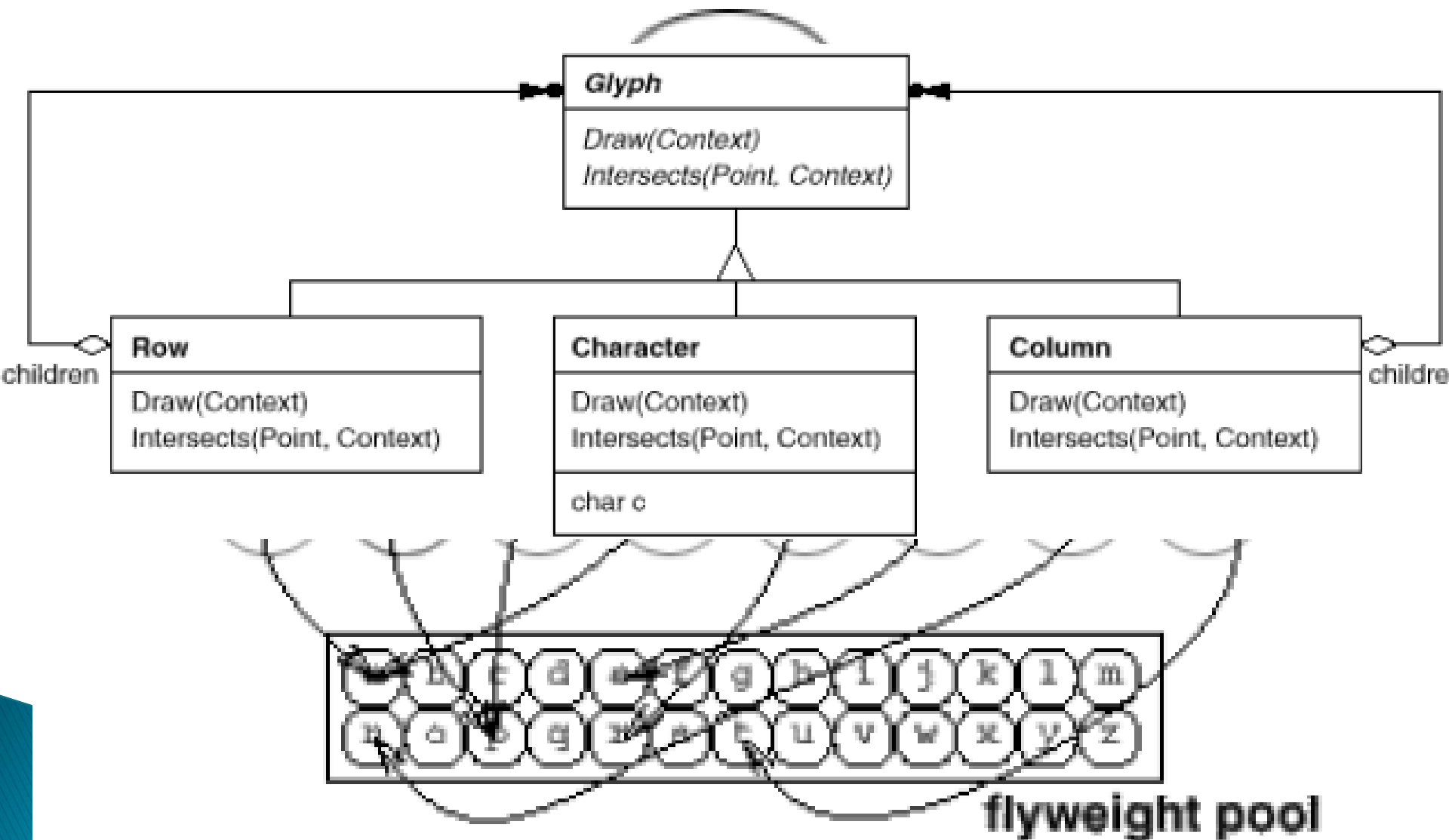
Structural Patterns - Flyweight

- 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

character objects

row objects

column object

Flyweight 2

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

```java
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

```java
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();
}
```

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
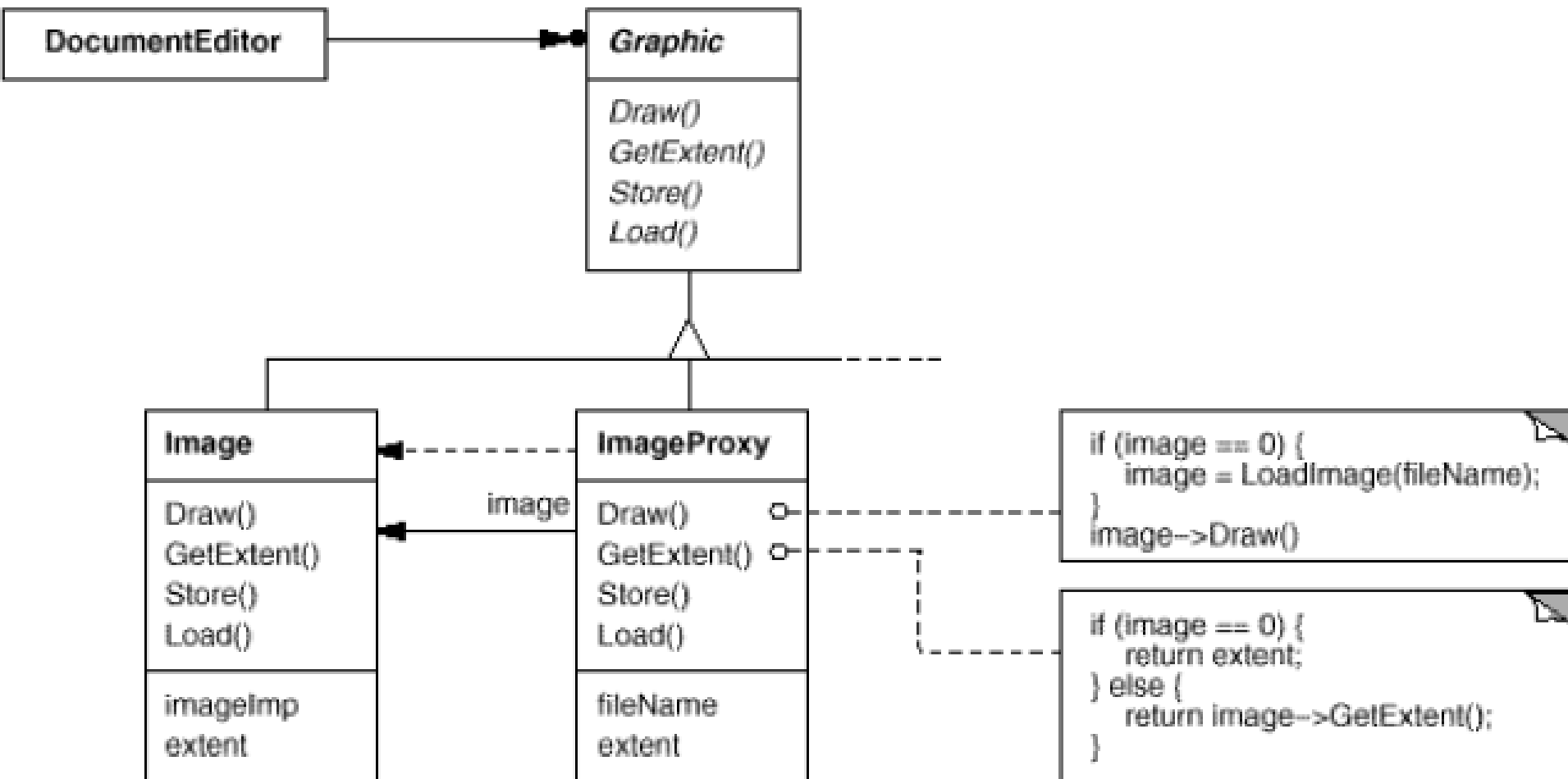
Proxy 1

▸ 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.

aTextDocument — image ● ⟶ anImageProxy — fileName ● ⤏ anImage — data

in memory ——— on disk
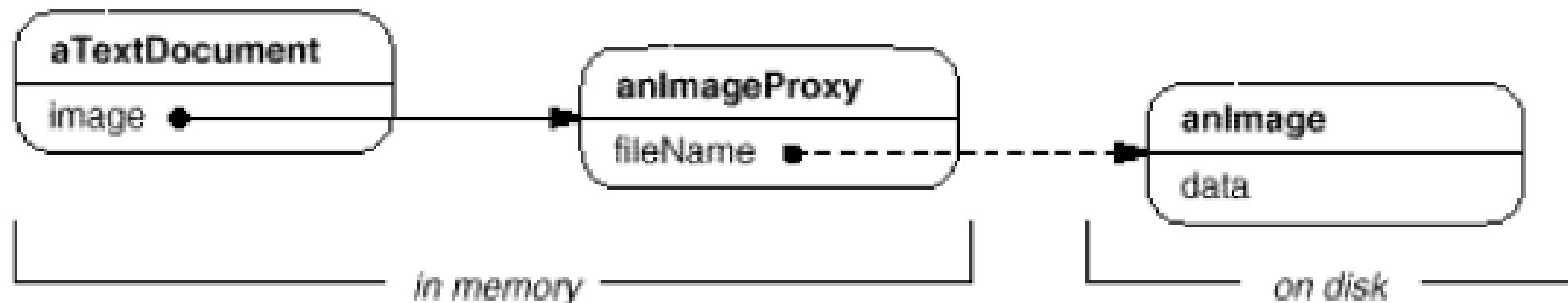
Proxy 2

▸ The following class diagram illustrates this example in more detail

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

```java
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

```java
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;
  }
}
```

Bibliografie

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

Links

- Design Patterns: http://www.oodesign.com/
- 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
- Lazy initialization: http://en.wikipedia.org/wiki/Lazy_initialization
- Use Lazy Initialization to Conserve Resources: http://www.devx.com/tips/Tip/18007