Object Orientation with Design Patterns

Lecture 7:
FlyWeight, Proxy

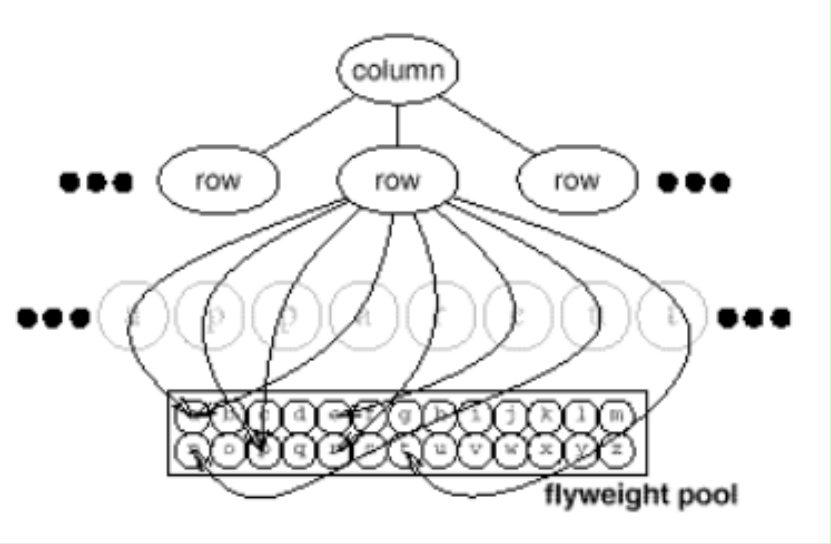
The Flyweight Pattern

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

Flyweight Motivation

- Flyweights model concepts or entities that are normally **too plentiful** to represent with objects.

- For example, a document editor can create a flyweight for each letter of the alphabet. Each flyweight stores a character code, but its coordinate position in the document and its typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears. The character code is **intrinsic** state, while the other information is **extrinsic**.



Extrinsic versus Intrinsic state

- A **flyweight** is a **shared object** that can be used in multiple contexts simultaneously.

- The flyweight acts as an **independent object** in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate.

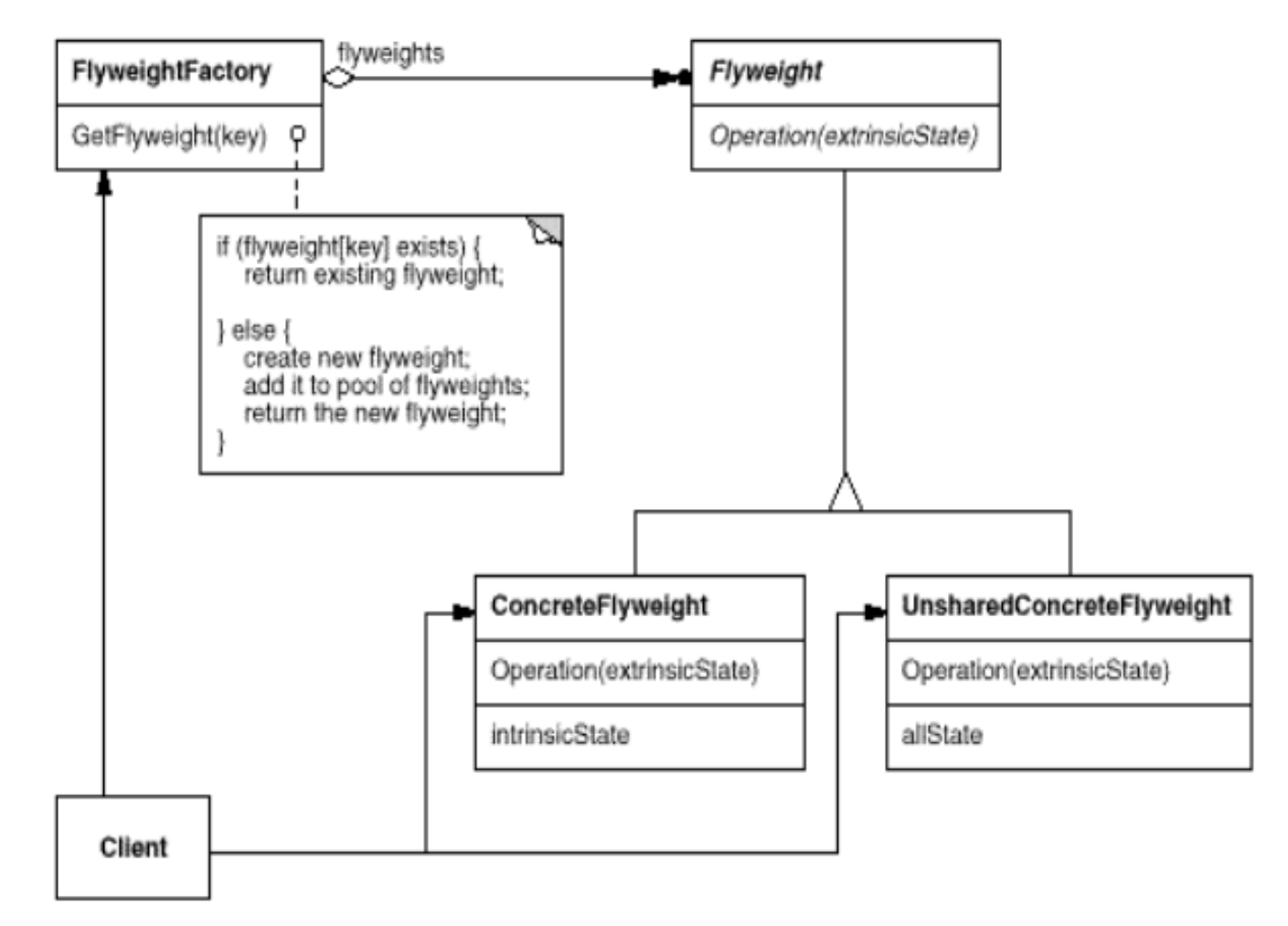
- The key concept here is the distinction between **intrinsic** and **extrinsic** state.

Extrinsic versus Intrinsic state

- **Intrinsic state** is stored in the flyweight; it consists of **information that's independent** of the flyweight's context making it **sharable.**

- **Extrinsic state** depends on and varies with the flyweight's context and therefore **can't be shared**. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

The Flyweight Structure



The Flyweight Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and act on **extrinsic** state.
- **ConcreteFlyweight (Folder)**
  - implements the Flyweight interface and adds storage for intrinsic state, if any. **A ConcreteFlyweight object must be sharable**. Any state it stores must be **intrinsic**; that is, it must be independent of the ConcreteFlyweight object's context.

- **UnsharedConcreteFlyweight**
  - not all Flyweight subclasses **need to be shared**. The Flyweight interface *enables* sharing; it doesn't enforce it.

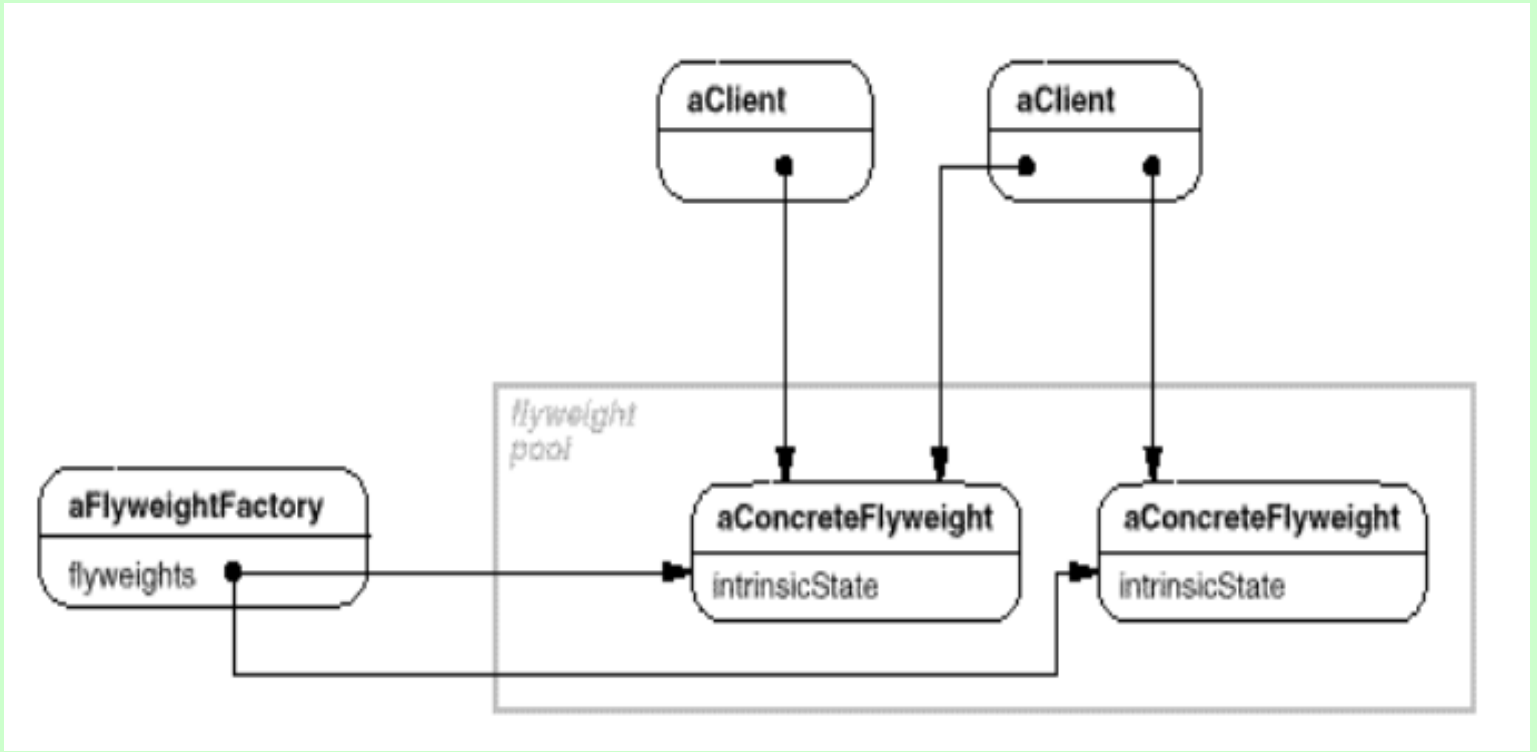
The Flyweight Participants

- **FlyweightFactory (FolderFactory)**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.

- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the **extrinsic state** of flyweight(s).

The Flyweight Structure

- The following object diagram shows how flyweights are shared:



The Flyweight Pattern Example

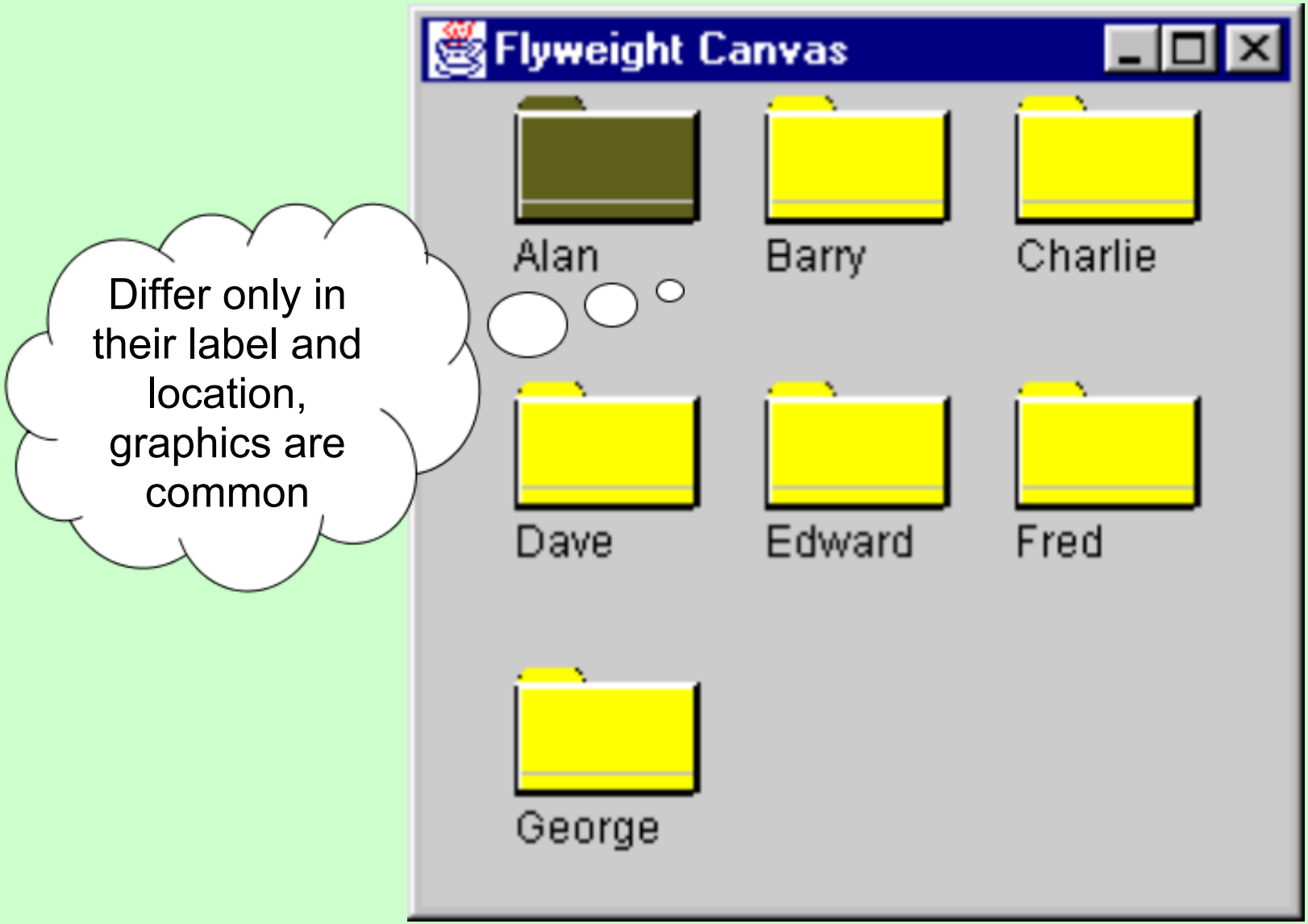
- Sometimes it is necessary to create a **very large number of small class instances to represent data.** You can greatly **reduce** the number of different classes that you need to instantiate if you can **determine that the instances are fundamentally the same**, except for a few parameters.

- The Flyweight pattern provides an approach for handling such classes. It refers to the instance's ***intrinsic*** **data** that makes the **instance unique** and the ***extrinsic*** **data** that is **passed as arguments.**

- The Flyweight is appropriate for **small , finegrained classes** such as those used for individual characters or icons in the screen.

- For example you might be drawing a series of icons which represent folders as shown on the next slide.

The Flyweight Pattern



The Flyweight Pattern

- A Flyweight is a **sharable instance** of a class. At first glance, each class might appear to be a **Singleton**.
  In fact, a small number of instances might exist, such as one for every character or one for every icon type.

- The **number of instances** that are allocated must be **decided as the class instances are needed**; usually a **FlyWeightFactory** class does this.

- The **factory class** is usually **a singleton** since it needs to **keep track of whether a particular instance** has been **generated.** It then **returns** either **a new instance** or a **reference to one that already exists**.

The Flyweight Pattern

- To decide whether some part of a program is **a candidate** for using the **flyweight pattern** consider whether it is **possible to remove some data from the class** and make it *extrinsic* (pass it in as a parameter).

- If this is possible then it **will greatly reduce the number of class instances** your program needs to maintain.

- Suppose you want to draw a small folder icon with a name under it for each person in an organization. If the organization is large , there could be many such icons; however they are actually all the same graphically. Even if we have **two icons**, one for **'Selected'** and one for **'Not Selected'.**

- In the following example having a **separate object** (icon) for each person is a **waste of resources**.

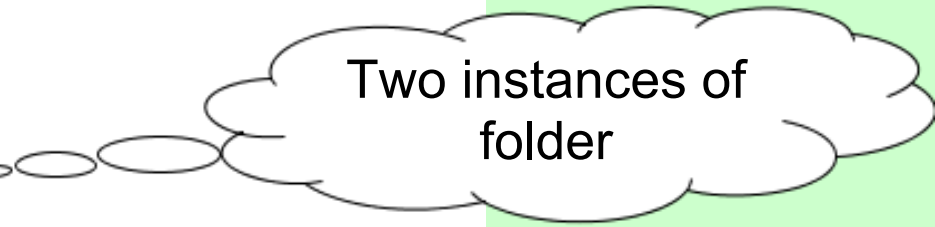
The Flyweight Pattern

- Instead, we will create a FolderFactory that **returns** either the **selected** or the **unselected folder** drawing class but does **not create additional instances** once one of each has been created. Since this is a **simple case** we can create the two instances at the outset and return one or the other.

```
import java.awt.*;

public class FolderFactory {
    Folder unSelected, Selected;
    public FolderFactory() {
        Color brown = new Color(0x5f5f1c);
        Selected =  new Folder(brown);
        unSelected = new Folder(Color.yellow);
    }

    public Folder getFolder(boolean isSelected) {
        if (isSelected)
            return Selected;
        else
            return unSelected;
    }
}
```



The Flyweight Pattern

- When **more instances** could exist, **the factory could keep a table** of the ones that it had **already created**.

- The unique thing about the flyweight pattern is that we **pass the coordinates** and the **name to be drawn** under the folder. These coordinates are the ***extrinsic data*** that allow us share the folder objects.

- The complete folder class shown on the next slide creates a **folder instance** with **one background color** or **the other** and has a public *draw* method that draws the folder at the point we specify.

```
public class Folder extends JPanel {
    private Color color;
    final int W = 50, H = 30;
    final int tableft = 0, tabheight=4, tabwidth=20, tabslant=3;
    public Folder(Color c) {
        color = c;
    }
    public void draw(Graphics g, int tx, int ty, String name) {
        g.setColor(Color.black);                    //outline
        g.drawRect(tx, ty, W, H);
        g.drawString(name, tx, ty + H+15);  //title
        g.setColor(Color.white);
        g.drawLine (tx, ty, tx+W, ty);
        Polygon poly = new Polygon();
        poly.addPoint (tx+tableft,ty);
        poly.addPoint (tx+tableft+tabslant, ty-tabheight);
        poly.addPoint (tx+tabwidth-tabslant, ty-tabheight);
        poly.addPoint (tx+tabwidth, ty);
        g.setColor(Color.black);
        g.drawPolygon (poly);
        g.setColor(color);                          //fill rectangle
        g.fillRect(tx+1, ty+1, W-1, H-1);
        g.fillPolygon (poly);
        g.setColor(Color.white);
        g.drawLine (tx, ty, tx+W, ty);
        g.setColor(Color.lightGray);                //bend line
        g.drawLine(tx+1, ty+H-5, tx+W-1, ty+H-5);
        g.setColor(Color.black);                    //shadow lines
        g.drawLine(tx, ty+H+1, tx+W-1, ty+H+1);
        g.drawLine(tx+W+1, ty, tx+W+1, ty+H);
        g.setColor(Color.white);                    //highlight lines
        g.drawLine(tx+1, ty+1, tx+W-1, ty+1);
        g.drawLine(tx+1, ty+1, tx+1, ty+H-1);
    }
}
```

The Flyweight Pattern

- To use the flyweight class like this our calling **program (FlyCanvas)** must **calculate the position** of each folder as part of its paint routine and then **pass the coordinates to the folder instance.**

- This is an advantage as we need the folders to have a different layout depending on the size of the window. Each folders position is computed dynamically during the paint routine.

- The following slide shows the paint method of the calling program.

The Flyweight Pattern

```
public void paint(Graphics g) {
    Folder f;
    String name;

    int j = 0;        //count number in row
    int row = Top;    //start in upper left
    int x = Left;

    //go through all the names and folders
    for (int i = 0; i< names.size(); i++) {
        name = (String)names.elementAt(i);
        if (name.equals(selectedName))
            f = fact.getFolder(true);
        else
            f = fact.getFolder(false);
        //have that folder draw itself at this spot
        f.draw(g, x, row, name);

        x = x + HSpace;     //change to next posn
        j++;
        if (j >= HCount) { //reset for next row
            j = 0;
            row += VSpace;
            x = Left;
        }
    }
}
```

Selecting a Folder

- Since we have **two folder instances**, **selected** and **unselected**, we would like to be able to select folders by moving the mouse over them. In the previous paint routine, we simply remember the name of the folder that was selected and ask the factory to return a “selected” folder for it.

- Because the folders are not **individual** instances **we cannot listen for mouse motion within each folder instance.**

- Instead we listen for **mouse events** at the **window level**. If the mouse is found to be within a Rectangle we make that **corresponding folder the selected one**.

Selecting a Folder

```
public void mouseMoved(MouseEvent e) {
     int j = 0;          //count number in row
     int row = Top;  //start in upper left
     int x = Left;

     //go through all the names and folders
     for (int i = 0; i< names.size(); i++) {
          //see if this folder contains the mouse
          Rectangle r = new Rectangle(x,row,W,H);
          if (r.contains(e.getX(), e.getY())) {
               selectedName=(String)names.elementAt(i);
               repaint();
          }
          x = x + HSpace;          //change to next posn
          j++;
          if (j >= HCount) {    //reset for next row
               j = 0;
               row += VSpace;
               x = Left;
          }
     }
}
```

Flyweights in Java

- Flyweights are **not often used at application level in Java**. They are more of a **system resource management technique** that is used at lower levels then Java.

- Some objects within the Java language could be implemented as flyweights. For example if two instances of the class **String** are created with the same literal values they could refer to the same storage location.

- To **prove the absence of flyweight classes** we could use the following code.

```
String fred1 = new String("Fred");
String fred2 = new String("Fred");

System.out.println(fred1 == fred2);
```

Flyweights in Java

- The **output** of such a test **will be false** because the two reference variables fred1 and fred2 are **referencing different objects** (different memory locations).

- Remember the == operator **compares actual object references** rather then the = which checks for equality of value.

- **Layout managers** in Java are flyweights since the only **difference** between one gridlayout, for example, and another is the **list of components** it contains and some attribute values. When the layout functionality is required the components and attributes are passed to the single shared instance (i.e. you feed specific context to the shared instance..the client is responsible for context specific information)

Flyweight Applicability

Apply the Flyweight pattern when *all* of the following are true:

- An application uses a **large number of objects** (identical or nearly identical)
- **Storage costs are high** because of the sheer quantity of objects.
- Most object state can be made **extrinsic** (non-identical parts)
- The application **doesn't depend on object identity**. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Flyweight Consequences

- Flyweights may introduce **run-time costs** associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state

- However, **such costs** are **offset by space savings**, which increases as more flyweights are shared.

- Storage savings are a function of several factors:

- the **reduction** in the **total number of instances** that comes from sharing
- the **amount of intrinsic state** per object
- whether **extrinsic state** is **computed or stored**.

Proxy Pattern

Proxy Pattern

- **Intent:**
  Provide a **surrogate** or placeholder for another object **to control access to it.**

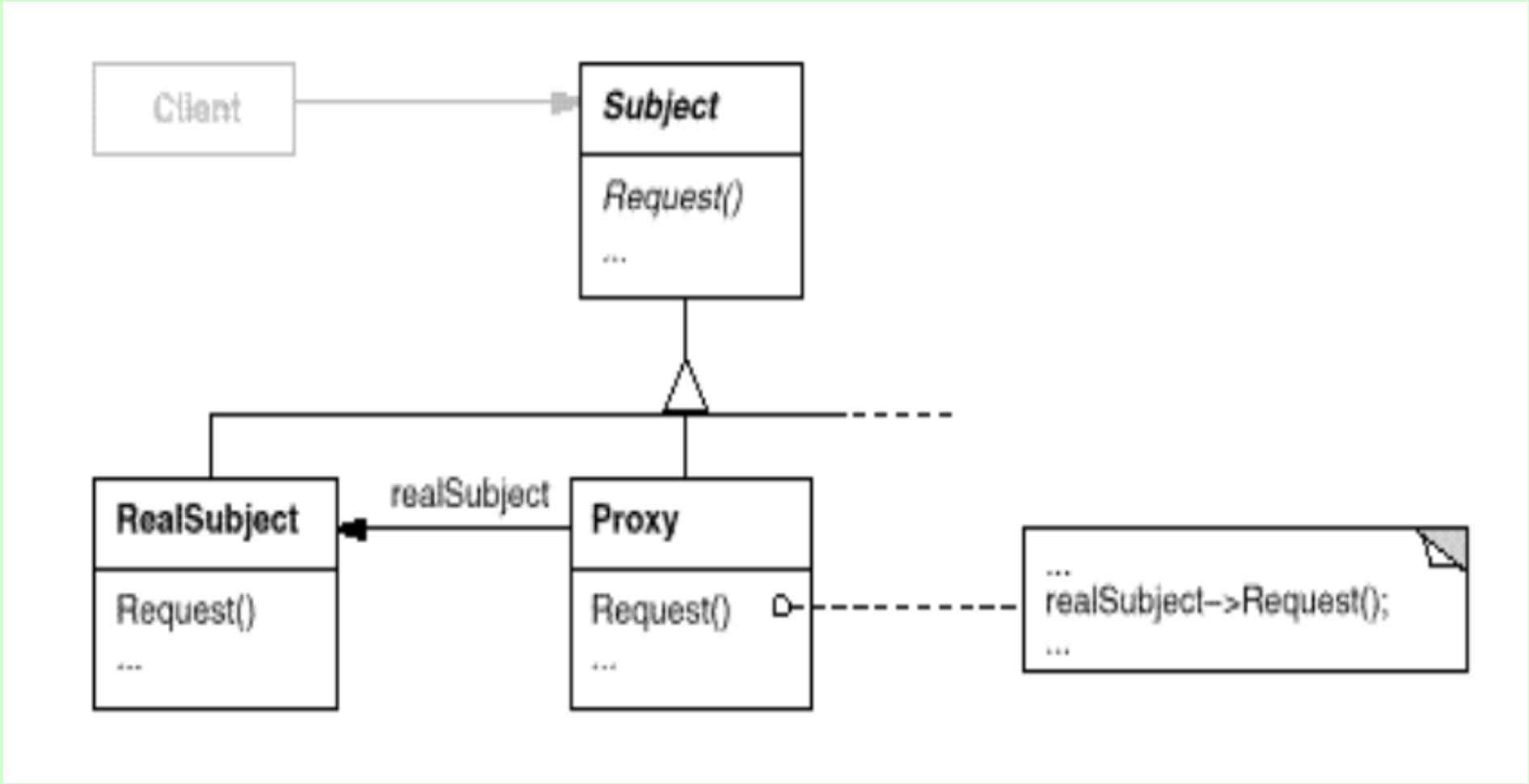
- **AKA:** Surrogate

- The **proxy pattern** is used to **represent** with **a simpler object** an object that is **complex or time-consuming to create**.
  If creating an object is expensive in time or computer-resources, a proxy **allows you to postpone** this creation until you actually **need the object**.

Proxy Pattern

- A **proxy** usually has **the same methods** as the **full object** that it represents. Once that full object is loaded, the Proxy passes on the method calls to the full object.
- There are several cases where a proxy can be useful
  - If an object such as a **large image** takes a long time to load. When the program starts, some indication that an image is to be displayed is needed so that the screen is set out correctly. However the actual image display can be postponed until the actual image is loaded.
  - If the object is on a **remote machine** and **loading it over a network might be slow,** especially during peak network load periods.
  - If the **object** has **limited access rights**. The proxy can then validate the access permissions for that user

Proxy Structure



Proxy Pattern Participants

- **Proxy**

- maintains a **reference** that lets the proxy access the real subject.

- provides **an interface** identical to Subject's so that a **proxy** can be **substituted** for the **real subject**.

- **controls access** to the **real subject** and may be responsible for **creating** and **deleting it.**

Proxy Pattern Participants

- **Subject**

- defines the **common interface for RealSubject** and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

- **RealSubject**

- **defines the real object that the proxy represents**.

Proxy Pattern Applicability

- Proxy is **applicable** whenever there is a **need** for a more versatile or **sophisticated reference** to an object than a **simple pointer**. Here are several common situations in which the Proxy pattern is applicable:

- A **remote proxy** provides a local representative for an object in a different address space.

- A **virtual proxy** creates expensive objects on demand.

- A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights.

Exercise – Lecture 7

Create an application that contains
the following buttons (do not use the button objects used in swing, draw your own graphic, a simple rectangle will do!):
Add, Update, Delete, Find.
When the mouse is over the button it
changes to red – when the mouse leaves
the button it goes back to green.
Use the flyweight pattern to implement the
above requirement. Draw the button graphic using the drawRect method