Flyweight [GoF]

Intent

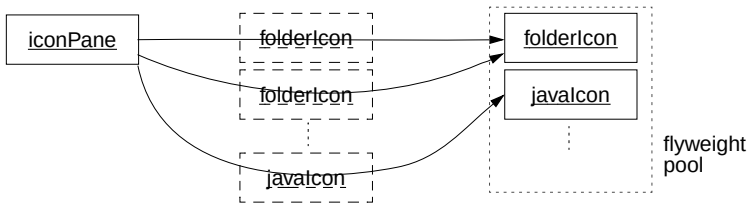
Object sharing to support large set of objects.

Motivation

Suppose you'll write an application that displays a large number of icons:

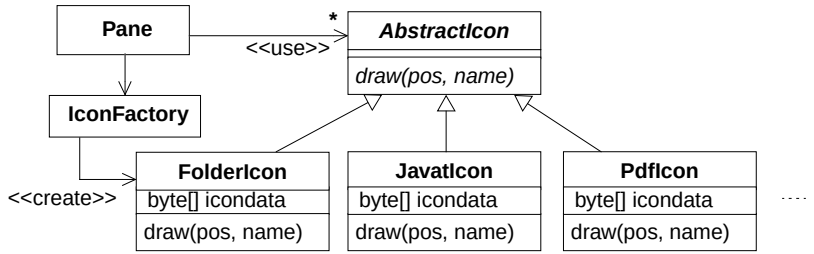
To manipulation the icons in the displayed pane, you want them to be ordinary objects. A naive implementation, however, would be prohibitively expensive.

A **flyweight** is a *shared* object that can be used in multiple contexts simultaneously. In the example application given above, the icon pane (the view) conceptually contains many folder, Java source file, and other icon *objects*. However, there is only *one* real object instance for each icon type:



The key concept is the distinction between *intrinsic* and *extrinsic* state:

- **Intrinsic state:** Stored in the flyweight, independent of the flyweight's context.
- **Extrinsic state:** State that depends on an varies with the flyweight's context.

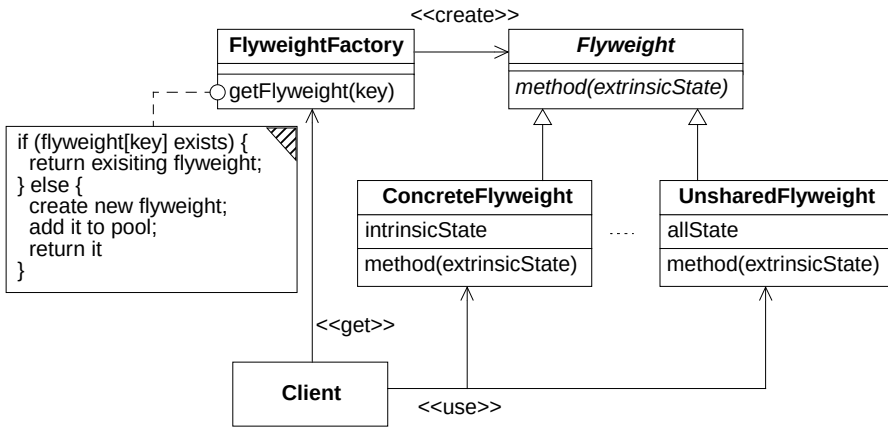


Applicability

Use the Flyweight pattern when all of the following are true:

- An application uses a large number of object.
- Storage costs are high.
- Application-aware object state can be made extrinsic.
- Many groups of objects can be replaced by a few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity.

Structure



Participants

- **`Flyweight`** (`AbstractIcon`)
  - declares an interface through which clients act on any kind of flyweights
  - protects against access by objects other than the Originator
  - has effectively two interfaces: a narrow interface for Caretaker, and a wide interface for Originator
- **`ConcreteFlyweight`** (`FolderIcon`)
  - implements the Flyweight interface, adds intrinsic state (if any)
  - must be sharable, i.e., its intrinsic state must be independent of the flyweight's context
- **`UnsharedFlyweight`** (`...`)
  - not all Flyweight subclasses must be shared. The Flyweight interface enables sharing; it doesn't enforce it
- **`FlyweightFactory`** (`IconFactory`)
  - creates and pools flyweight objects
  - ensures that flyweights are shared properly
- **`Client`** (`Pane`)
  - maintains a reference to flyweight(s)
  - computes or stores the extrinsic state of flyweight(s)

Implementation

Consider the following issues:

- *Managing share objects.* Because objects are share, clients should not instantiate them directly. Use a factory instead.
- *Factory method.* There are two approaches to consider when implementing the factory:
  - Use a separate factory method for each kind of Flyweight object you create:
    ```
    // FlyweightFactory:
    public Flyweight createConcreteFlyweight1() {...}
    public Flyweight createConcreteFlyweight2() {...}
    ...
    ```
  - Use a generic factory method, parameterized with a key to distinguish the kind of Flyweight to create and return:
    ```
    // Alternative FlyweightFactory:
    public Flyweight createFlyweight(String key) {...}
    ```


Collaborations

- Flyweight state is separated into intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; extrinsic state is stored or computed in the client objects. Clients pass the extrinsic state to the flyweight when they invoke its methods.
- Clients must not instantiate Flyweights directly. Clients must obtain Flyweights exclusively from `FlyweightFactory` objects to ensure that they are shared properly.

Consequences

Flyweights may introduce run-time costs because of the finding, computing, and transferring of extrinsic state of Flyweight objects. These costs are offset by space savings which is a function of several factors:

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

Sample Code

Let's consider a sketch of the code of the example in the Motivation section.

A base class for icons of the application might look like:

```
public abstract class AbstractIcon {
   public abstract void draw(Graphics g,
                             int tx, int ty,
                             String name, boolean selected);
}
```

Notice that we could have used a Java interface here. Notice also the variables denoting the extrinsic state.

A concrete icon subclass stores its intrinsic state, and implements the `draw` method. In the example given here, there is also a distinction whether the icon is selected or not:

```
public class FolderIcon extends AbstractIcon {
    private final int H = 48;
    private ImageIcon iconSel, iconUnsel;
   ...
}
```

The constructor's access should be made restricted. We make it package-visible here preventing clients not being in this package to directly create instances:

```
// in class FolderIcon;
   FolderIcon() {
      URL iconURL =
         ClassLoader.getSystemResource("images/folder_o.png");
      if (iconURL != null) {
         iconSel = new ImageIcon(iconURL);
      } else ... // some kind or error handling
         iconURL =
            ClassLoader.getSystemResource("images/folder.png");
      if (iconURL != null) {
         iconUnsel = new ImageIcon(iconURL);
      } else ... // some kind or error handling
   }
```

Next, we have to implement all the methods the flyweights support, given the extrinsic state. In our example, the only method is method `draw`:


```
// in class FolderIcon;
   public void draw(Graphics g, int tx, int ty,
               String name, boolean sel)
   {
      // Clear the space occupied by either of the icons
      ...
      if (sel) {
         iconSel.paintIcon(null, g, tx, ty);
      } else {
         iconUnsel.paintIcon(null, g, tx, ty);
      }
      g.drawString(name, tx, ty + H + 15);  //title
   }
```

A factory class for the creation of the concrete flyweight icon objects might look like:

```
public class IconFactory {
   private Map iconmap = new HashMap();
   // Typically a Singleton, Singleton code not shown:
   ...
}
```


The factory class in this example has one method to create any one of the icon objects it supports:

```
// in class IconFactory:
   public AbstractIcon createIcon(String key) {
      AbstractIcon icon = (AbstractIcon) iconmap.get(key);
      if (icon == null) {
         icon = makeIcon(key); iconmap.put(key, icon);
      }
      return icon;
   }
```

The private helper method `makeIcon` actually create the application's icons:

```
// in class IconFactory:
   private static AbstractIcon makeIcon(String key) {
      AbstractIcon icon = null;
      if (key.equals("dir")) {
         icon = new FolderIcon();
      } else if (key.equals("java")) { ... }
      else icon = new UnknownIcon();
      return icon;
   }
```


A client uses the flyweights via the factory class. For example:

```
// Client code, excerpt:
   // helper class:
   private class Item {
      String type;
      String name;
      AbstractIcon icon;
      Item(String type, String name, AbstractIcon icon) {
         this.type = type;
         this.name = name;
         this.icon = icon;
      }
   }
```

In a client, application icons can then be created:

```
   // icon creation:
   IconFacory factory = IconFactory.getInstance();
   Item item1 = new Item("dir","Folder1",
                         factory.createIcon("dir"));
   ...
```

In the client, the icons can then be displayed:

```
// draw the icon:
Graphics g = ...;
// for each icon to draw:
   Item item = ...;
   boolean selected = ...;
   xpos = ...; // calculate the X position
   ypos = ...; // calculate the Y position
   item.icon.draw(g, xpos, ypos, item.name, selected);
```

Related Patterns

- Flyweights are often combined with the Composite pattern.
- State and Strategy patterns can sometimes be implemented as flyweights.