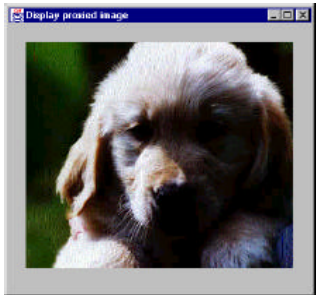
CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Proxy Pattern

Apr. 10, 2007

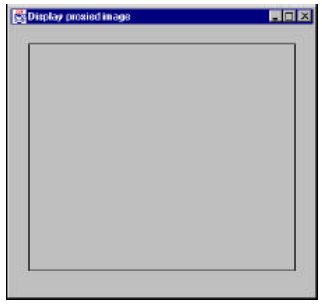
Example

- A program needs to load and display a large image (perhaps over a network).
  - Very slow process
- When the program starts, there must be some indication that an image is to be displayed so that the screen lays out correctly, but the actual image display can be postponed until the image is completely loaded.
- This is particularly important in programs such as word processors and web browsers that lay out text around the images even before the images are available.



A Proxy

- An image ***proxy*** can begin loading the image in the background, while drawing a simple rectangle or other symbol to represent the image's extent on the screen before it appears.
- The proxy can even delay loading the image at all until it receives a paint request, and only then begin the process.



A Proxy Pattern

- The Proxy pattern is used when you need to represent a complex object by a simpler one.
- If creating an object is expensive in time or computer resources, Proxy allows you to postpone this creation until you need the actual object.
- A Proxy usually has the same methods as the object it represents, and once the object is loaded, it passes on the method calls from the Proxy to the actual 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.
  - If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
  - If the object has limited access rights, the proxy can validate the access permissions for that user.
- Proxies can also be used to distinguish between requesting an instance of an object and the actual need to access it. For example, program initialization may set up a number of objects which may not all be used right away. In that case, the proxy can load the real object only when it is needed.

Strategy

- We create a simple program to display an image on a JPanel when it is loaded.
- Rather than loading the image directly, we use a class we call ImageProxy to defer loading and draw a rectangle around the image area until loading is completed.
- We create the instance of the ImageProxy just as we would have for an Image, and that we add it to the enclosing JPanel as we would an actual image.

Lets See Some Code!

```
public class ProxyDisplay extends JxFrame
{
  public ProxyDisplay()
  {
  super("Display proxied image");
  JPanel p = new JPanel();
  getContentPane().add(p);
  p.setLayout(new BorderLayout());
  ImageProxy image = new ImageProxy(this, "elliott.jpg",
                                                  321,271);
  p.add("Center", image);
  setSize(400,400);
  setVisible(true);
  }
```


Designing the ImageProxy Class

- The ImageProxy class sets up the image loading and creates a MediaTracker object to follow the loading process within the constructor.
- The MediaTracker class is a utility class to track the status of a number of media objects.
  - Media objects could include audio clips as well as images, though currently only images are supported.
  - To use a media tracker, create an instance of MediaTracker and call its addImage method for each image to be tracked.
  - In addition, each image can be assigned a unique identifier. This identifier controls the priority order in which the images are fetched. It can also be used to identify unique subsets of the images that can be waited on independently. Images with a lower ID are loaded in preference to those with a higher ID number.

The Code

```
public ImageProxy(JFrame f, String filename,
                  int w, int h)
{
height = h;
width = w;
frame = f;

tracker = new MediaTracker(f);
img = Toolkit.getDefaultToolkit().getImage(filename);
tracker.addImage(img, 0);     //watch for image loading

imageCheck = new Thread(this);
imageCheck.start();           //start 2nd thread monitor

//this begins actual image loading
try{
    tracker.waitForID(0,1);
 }
catch(InterruptedException e){}
}
```

More Details!

- The *waitForID* method of the MediaTracker actually initiates loading.
- In this case, we put in a minimum wait time of 1 msec so that we can minimize apparent program delays.
- The constructor also creates a separate thread *imageCheck* that checks the loading status every few milliseconds, and starts that thread running.

run()

```
public void run()
   {
   //this thread monitors image loading
   //and repaints when the image is done
   try{
   Thread.sleep(1000);
   while(! tracker.checkID(0))
      Thread.sleep(1000);
   }
   catch(Exception e){}
   repaint();
   }
```

- We slowed the polling time down to 1 second to see the program draw the rectangle and then refresh the final image.

The Proxy Needs a paint() Method

- The Proxy is derived from the JPanel component, and therefore, naturally has a *paint* method.
- In this method, we draw a rectangle if the image is not loaded. If the image has been loaded, we erase the rectangle and draw the image instead.

paint()

```
public void paint(Graphics g)
   {
    if (tracker.checkID(0))
      {
      height = img.getHeight(frame);   //get height
      width = img.getWidth(frame);     //and width
      g.setColor(Color.lightGray);     //erase box
      g.fillRect(0,0, width, height);
      g.drawImage(img, 0, 0, frame);   //draw image
      }
   else
      {
      //draw box outlining image if not loaded yet
      g.drawRect(0, 0, width-1, height-1);
      }
   }
```

Copy-on-write

- You can also use proxies to keep copies of large objects that may or may not change.
- If you create a second instance of an expensive object, a Proxy can decide there is no reason to make a copy yet.
  - It simply uses the original object.
- Then, if the program makes a change in the new copy, the Proxy can copy the original object and make the change in the new instance.
- This can be a great time and space saver when objects do not always change after they are instantiated.

Summary of Structural Patterns

- The **Adapter** pattern, used to change the interface of one class to that of another one.
- The **Bridge** pattern, intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. You can then change the interface and the underlying class separately.
- The **Composite** pattern, a collection of objects, any one of which may be either itself a Composite, or just a primitive object.
- The **Decorator** pattern, a class that surrounds a given class, adds new capabilities to it, and passes all the unchanged methods to the underlying class.
- The **Façade** pattern, which groups a complex object hierarchy and provides a new, simpler interface to access those data.
- The **Flyweight** pattern, which provides a way to limit the proliferation of small, similar class instances by moving some of the class data outside the class and passing it in during various execution methods.
- The **Proxy** pattern, which provides a simple place-holder class for a more complex class which is expensive to instantiate.