5. Structural Patterns - 2

*SW Design Patterns*,
by Boyan Bontchev,
FMI - Sofia University
2006/2011
Annotation

- Structural patterns
- Definitions
- Properties
- Intent, motivation, structure, participants, collaborations, consequences, implementation issues about:
  - Façade
  - Flyweight
  - Proxy
- Examples in Java
References

- Gamma, Helm, Johnson, Vlissides ("Gang of Four" - GoF) *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995
- *Design Patterns Explained*, by Allan Shalloway and James Trott, Prentice Hall, 2001
- THE DESIGN PATTERNS JAVA COMPANION, by JAMES W. COOPER, Adiscon-Wesley, October 2, 1998
## Design pattern catalog - GoF

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Factory Method</td>
<td>Adapter</td>
<td>Interperter</td>
</tr>
<tr>
<td></td>
<td>Abstract Factory, Builder, Prototype, Singleton</td>
<td>Bridge, Composite, Decorator, Facade, Flyweight, Proxy</td>
<td>Chain of Responsibility, Command, Iterator, Mediator, Template Method, Memento, Observer, State, Strategy, Visitor</td>
</tr>
<tr>
<td>Scope</td>
<td>Object</td>
<td>Object</td>
<td>Object</td>
</tr>
<tr>
<td>Scope</td>
<td>Class</td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>Scope</td>
<td>Object</td>
<td>Object</td>
<td>Object</td>
</tr>
</tbody>
</table>
Structural Design Pattern

Structural patterns describe how classes and objects can be combined to form larger structures.

- Class patterns describe how inheritance can be used to provide more useful program interfaces.

- Object patterns describe how objects can be composed into larger structures using object composition or the inclusion of objects within other objects.
Let follow a paper…

- **Non-Software Examples of Software Design Patterns**, by Michael Duell, in AG Communication Systems e-zine:

  [http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm](http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm)
The 7 Structural Design Patterns

1/2

- Adapter (GoF 139) - can be used to make one class interface match another to make programming easier.
- Bridge (GoF 151) - separates an object's abstraction from its implementation so that you can vary them independently.
- Composite (GoF 163) - a structural object pattern; describes how to build a class hierarchy made up of objects each of which may be either simple or itself a composite object. The composite objects let you compose primitive and other composite objects into arbitrarily complex structures.
- Decorator (GoF 175) - describes how to add responsibilities to objects dynamically; composes objects recursively to allow an open-ended number of additional responsibilities. For example, a Decorator object containing a user interface component can add a decoration like a border or shadow to the component, or it can add functionality like scrolling and zooming.
The 7 Structural Design Pattern

2/2

- Façade (GoF 185) - shows how to make a single object represent an entire subsystem. A façade is a representative for a set of objects. The façade carries out its responsibilities by forwarding messages to the objects it represents.

- Flyweight (GoF 195) - defines a structure for sharing objects. Objects are shared for at least two reasons: efficiency and consistency. Flyweight focuses on sharing for space efficiency. Applications that use lots of objects must pay careful attention to the cost of each object. Substantial savings can be had by sharing objects instead of replicating them.

- Proxy (GoF 207) - acts as a placeholder for another object, as a local representative for an object in a remote address space. It can represent a large object that should be loaded on demand. It might protect access to a sensitive object; it can restrict, enhance, or alter object properties.
The Façade (Facade) Pattern [1]

**Intent** - provides a unified interface to a set of interfaces in a subsystem - defines a higher level interface that makes the subsystem easier to use.

**Motivation** – structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.
Motivation – Compiler Interface

- Consider applications access to a 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 generally don't care about details like parsing and code generation; they merely want to compile some code but not low-level interfaces.

- To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class defining a unified interface to the compiler's functionality and thus, acts as a facade: it offers clients a single, simple interface to the compiler subsystem. It glues together the classes that implement compiler functionality without hiding them completely.
The Solution
Applicability

Use the Façade pattern when

- you need a simple interface to a complex subsystem. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but harder to use for clients that don't need to customize it. A façade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

- there are many dependencies between clients and the implementation classes of an abstraction. A façade decouples the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.

- subsystems layering. A facade defines an entry point to each subsystem level. If subsystems are dependent - make them communicating with each other solely through their facades.
Participants and Structure

- **Façade (Compiler)**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.

- **Subsystem classes (Scanner, Parser, ProgramNode, etc.)**
  - implement subsystem functionality.
  - handle work assigned by the Façade object.
  - have no knowledge of the façade; that is, they keep no references to it.
Collaborations

- Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the façade may have to do work of its own to translate its interface to subsystem interfaces.

- Clients that use the façade don't have to access its subsystem objects directly.
Consequences and Benefits

- Façade shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use. Examples in real life...

- It promotes *weak coupling* between the subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients. Façade layer eliminates complex or circular dependencies.
  - Reducing compilation dependencies with façades can limit the re-compilation needed for a small change in an important subsystem.
  - A façade can also simplify porting systems to other platforms, because it's less likely that building one subsystem requires building all others.

- It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.
Implementation Issues

1. Reducing client-subsystem coupling.
   - The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.
   - An alternative to subclassing is to configure a Facade object with different subsystem objects (delegation). To customize the facade, simply replace one or more of its subsystem objects.

2. Public versus private subsystem classes. A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulates classes. Thus, we can think of the public and private interface of a subsystem.
   - The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders.
   - The Facade class is part of the public interface, of course, but it’s not the only part. Other subsystem classes are usually public as well. For example, the classes Parser and Scanner in the compiler subsystem are part of the public interface.
Java Façade for JDBC Classes [3]

• java.sql package provide an example of a set of quite low level classes that interact in a convoluted manner
• to connect to a database, you use an instance of the Connection class. Then, to find out the names of the database tables and fields, you need to get an instance of the DatabaseMetadata class from the Connection.
• to issue a query, you compose the SQL query string and use the Connection to create a Statement class. By executing the statement, you obtain a ResultSet class, and to find out the names of the column rows in that ResultSet, you need to obtain an instance of the ResultSetMetadata class.
Solution

By designing a Façade consisting of a **Database** class and a **resultSet** class (note the lowercase “r”), we can build a much more usable system.
Instead of...

```java
try{
    Class.forName(driver);
} //load the Bridge driver
catch (Exception e) {
    System.out.println(e.getMessage());
}
try {
    con = DriverManager.getConnection(url);
    DatabaseMetaData dma = con.getMetaData(); //get the meta data
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
Vector tname = new Vector();
try {
    ResultSet results = dma.getTables( .... ));
} catch (Exception e) {System.out.println(e);} while (results.hasMoreElements()) //gets the 
    tname.addElement( results.getString ("TABLE_NAME") );
//next – get columns for each table…
...Two simple enclosing classes which contain all of the significant methods of the Connection, ResultSet, Statement and Metadata classes:

Class **Database** {
    public Database(String driver()) //constructor
    public void Open(String url, String cat);
    public String[] getTableNames();
    public String[] getColumnNames(String table);
    public String getColumnValue(String table, String columnName);
    public String getNextValue(String columnName);
    public resultSet Execute(String sql);
}
Cont.

class resultSet {
    public resultSet(ResultSet rset) //constructor
    public String[] getMetaData();
    public boolean hasMoreElements();
    public String[] nextElement();
    public String getColumnValue(String columnName);
    public String getColumnValue(int i);
}

The Flyweight Pattern [1]

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

- **Motivation** – There are cases in programming where it seems that you need to generate a very large number of small class instances to represent data.
  - Sometimes you can greatly reduce the number of different classes that you need to instantiate if you can recognize that the instances are fundamentally the same except for a few parameters.
  - If you can move those variables outside the class instance and pass them in as part of a method call, the number of separate instances can be greatly reduced.
Motivating Example

- Object-oriented document editors typically use objects to represent embedded elements like tables and figures.
- However, they usually stop short of using an object for each character in the document, even though doing so would promote flexibility at the finest levels in the application.
- Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted.
The Solution

- *Everything is an object* - the drawback of such a design is its **cost**. Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of **memory** and may incur unacceptable **runtime overhead**.

- The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.

- **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.
Intrinsic and Extrinsic state

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

- Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable.
- Extrinsic state depends on and varies with the fly-weight's context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

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

- Logically there is an object for every occurrence of a given character in the document.

- Physically, however, there is one shared flyweight object per character, and it appears in different contexts in the document structure. Each occurrence of a particular character object refers to the same instance in the shared pool of flyweight objects.
The Effect

- Glyph is the abstract class for graphical objects, *some of which may be flyweights*. *Operations that may depend on extrinsic state have it passed to them as a parameter*. For example, Draw and Intersects must know which context the glyph is in before they can do their job. Clients supply the context-dependent information that the flyweight needs to draw itself.

- Because the number of different character objects is far less than the number of characters in the document, the total number of objects is substantially less than what a naive implementation would use.
Applicability

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application does not depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.
Structure

```
FlyweightFactory
  GetFlyweight(key) {
    if (flyweight[key] exists) {
      return existing flyweight;
    } else {
      create new flyweight;
      add it to pool of flyweights;
      return the new flyweight;
    }
  }

Flyweight
  Operation(extrinsicState)

ConcreteFlyweight
  Operation(extrinsicState)
  intrinsicState

UnsharedConcreteFlyweight
  Operation(extrinsicState)
  allState
```
Sharing
Participants

- **Flyweight (Glyph)**
  - declares an interface through which flyweights can receive and act on extrinsic state.

- **ConcreteFlyweight (Character)**
  - 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 (Row, Column)**
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing but it *does not enforce* it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a fly-weight, 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). 
Collaborations

- The *state* that a flyweight needs to function must be characterized as either *intrinsic* or *extrinsic*:
  - intrinsic state is stored in the ConcreteFlyweight object;
  - extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations.

- Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.
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 increase 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.

- The more flyweights are shared, the greater the storage savings. The savings increase with the amount of shared state.
Implementation Issues 1/2

1. Removing extrinsic state from shared objects. The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state.

2. Managing shared objects. Because objects are shared, clients shouldn't instantiate them directly. FlyweightFactory lets clients locate a particular flyweight. FlyweightFactory objects often use an associative store to let clients look up flyweights of interest. For example, the flyweight factory in the document editor example can keep a table of flyweights indexed by character codes. The manager returns the proper flyweight given its code, creating the flyweight if it does not already exist.
Implementation Issues 2/2

3. Sharability also implies some form of reference counting or garbage collection to reclaim a flyweight's storage when it's no longer needed. However, neither is necessary if the number of flyweights is fixed and small (e.g., flyweights for the ASCII character set). In that case, the flyweights are worth keeping around permanently.

4. The Flyweight pattern is often combined with the Composite (GoF163) pattern to represent a hierarchical structure as a graph with shared leaf nodes. A consequence of sharing is that flyweight leaf nodes cannot store a pointer to their parent. Rather, the parent pointer is passed to the flyweight as part of its extrinsic state.
Java Example [3]

• Need to draw a small folder icon with a name under it for each person in an organization. If this is a large organization, there could be a large number of such icons, but they are actually all the same graphical image. Even if we have two icons, one for “is Selected” and one for “not Selected” the number of different icons is small.

• In such a system, having an icon object for each person, with its own coordinates, name and selected state is a waste of resources.

• Instead, we’ll 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 such a simple case, we just create them both at the outset and then return one or the other:
The FolderFactory Class

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;
    }
}
Sharing the Folder class

- For cases where more instances could exist, the factory could keep a table of the ones it had already created and only create new ones if they weren’t already in the table.

- The unique thing about using Flyweights, however, is that we pass the coordinates and the name to be drawn into the folder when we draw it. These coordinates are the extrinsic data that allow us to share the folder objects, and in this case create only two instances. The complete Folder class simply creates a folder instance with one background color or the other and has a public Draw method that draws the folder at the point you specify.
class Folder extends JPanel {
    private Color color;
    final int W = 50, H = 30;
    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); //fill rectangle
        g.fillRect(tx+1, ty+1, W-1, H-1);
        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);
    }
}
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;
        }
    }
}

We can have generated an array or Vector of folders at the outset and simply scan through the array to draw each folder. Such an array is not as wasteful as a series of different instances because it is actually an array of references to one of only two folder instances.
The Proxy Pattern [1]

- **Intent** - provides a *surrogate* or *placeholder* for another object to control access to it.
- **Also Known As** - Surrogate
- **Motivation** - one reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. 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.
Suggestion

These constraints would suggest creating each expensive object *on demand*, which in this case occurs when an image becomes visible.

- But what do we put in the document in place of the image?

- And how can we hide the fact that the image is created on demand so that we don't complicate the editor's implementation? This optimization shouldn't impact the rendering and formatting code, for example.
Solution

- 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.

- The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation. The proxy forwards subsequent requests directly to the image. It must therefore keep a reference to the image after creating it.

- If the images are stored in separate files, we can use the file name as the reference to the real object. The proxy also stores its **extent**, that is, its width and height. The extent lets the proxy respond to requests for its size from the formatter without actually instantiating the image.

![Diagram showing relationships between aTextDocument, anImageProxy, and anImage]
The document editor accesses embedded images through the interface defined by the abstract **Graphic** class.

- **ImageProxy** is a class for images that are created on demand. **ImageProxy** maintains the file name as a reference to the image on disk - passed as an argument to the **ImageProxy** constructor. **ImageProxy** also stores the bounding box of the image and a reference to the real Image instance. This reference won't be valid until the proxy instantiates the real image. The Draw operation makes sure the image is instantiated before forwarding it the request.

- **GetExtent** forwards the request to the image only if it's instantiated; otherwise **ImageProxy** returns the extent (the screen slice) it stores.
Structure

Object diagram of a proxy structure at run-time ↓:

```
... realSubject->Request();
...
```
Participants

- **Proxy** (ImageProxy) - maintains a reference that lets the proxy access the real subject.
  - may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - 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.

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

- **RealSubject** (Image) - defines the real object that the proxy represents.
Proxy’s responsibilities

Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy. Proxy’s responsibilities depend on the kind of proxy:

- **remote proxies** are responsible for encoding a request and its arguments and for *sending the encoded request to the real subject in a different address space.*

- **virtual proxies** may *cache additional information about the real subject* so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.

- **protection proxies** check that the *caller has the access permissions required to perform a request.*
Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer:

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

2. **A virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.

3. **A protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. E.g., for protected access to operating system objects.

4. **A smart reference** is a replacement for a simple pointer that performs additional actions when an object is accessed. Typical uses include:
   - counting the number of references to the real object so that it can be freed automatically – **garbage collection**
   - loading a persistent object into memory when it's first referenced.
   - checking that the real object is locked before it's accessed to ensure that no other object can change it – **critical sections**
Consequences

A) The Proxy pattern introduces a level of indirection when accessing an object:

1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

B) There's another optimization that the Proxy pattern can hide from the client. It's called **copy-on-write**, and it's related to creation on demand. Copying a large and complicated object can be an expensive operation. If the copy is never modified, then there's no need to incur this cost. By using a proxy to postpone the copying process, we ensure that we pay the price of copying the object only if it's modified.

To make copy-on-write work, the subject must be **reference counted**. Copying the proxy will do nothing more than increment this reference count. Only when the client requests an operation that modifies the subject does the proxy actually copy it. In that case the proxy must also decrement the subject's reference count. When the reference count goes to zero, the subject gets deleted.
Overloading the member access operator in C++. C++ supports overloading operator->, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced. The proxy behaves just like a pointer.

Using doesNotUnderstand in Smalltalk. Smalltalk provides a hook that you can use to support automatic forwarding of requests. Smalltalk calls doesNotUnderstand: aMessage when a client sends a message to a receiver that has no corresponding method. The Proxy class can redefine doesNotUnderstand so that the message is forwarded to its subject.

Proxy doesn't always have to know the type of real subject. If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly. But if Proxies are going to instantiate RealSubjects (such as in a virtual proxy), then they have to know the concrete class.

How to refer to the subject before it's instantiated - some proxies have to refer to their subject whether it's on disk or in memory. That means they must use some form of address space-independent object identifiers. We used a file name for this purpose in the Motivation.
Java Example [3]

In this example program, 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.

```java
public class ProxyDisplay extends JFrame {
    public ProxyDisplay() {
        super("Display proxied image");
        JPanel p = new JPanel();
        getContentPane().add(p);
        p.setLayout(new BorderLayout());
        ImageProxy image = new ImageProxy(this, "elliott.jpg", 2321, 2271);
        p.add("Center", image);
        setSize(2400,2400);
        setVisible(true);
    }
}
```
The `ImageProxy` class sets up the image loading and creates a `MediaTracker` object to follow the loading process within the constructor:

```java
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); // watch for image loading
    imageCheck.start(); // start 2nd thread monitor
    // this begins actual image loading
    try {
        tracker.waitForID(0, 1);
    } catch(InterruptedException e) { ... }
}
```

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.
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();
}

For the purposes of this illustration program, we slowed the
polling time down to 1 second so you can see the program
draw the rectangle and then refresh the final image.
Finally, 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.

```java
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);
    }
}
```