Proxy [GoF]

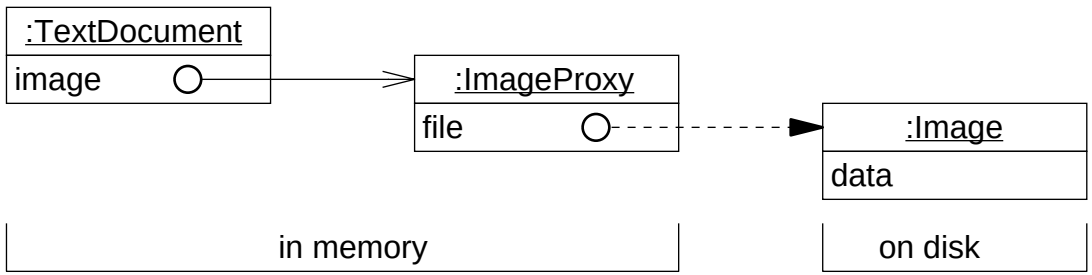
Intent

Provide a surrogate or placeholder for another object to control access to.

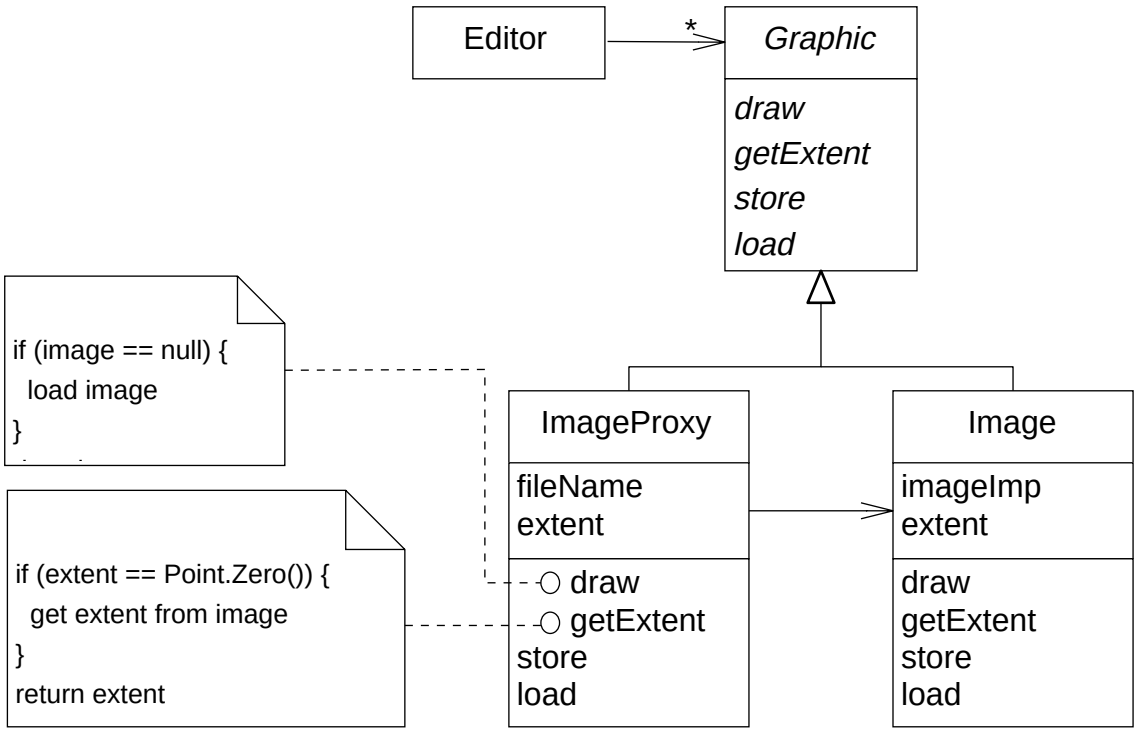
Motivation

Consider a document editor that can embed graphical objects in a document. Some graphical objects, like raster images, can be expensive to create. But opening a document should be fast. Thus, it should be avoided to create all “expensive” objects at once when the document is opened.

One solution for the above situation is to use another object, an image proxy, that acts as a stand-in for the real image object:



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. Assume that images are stored in separate files. Then the file names can be used as reference to the real object.

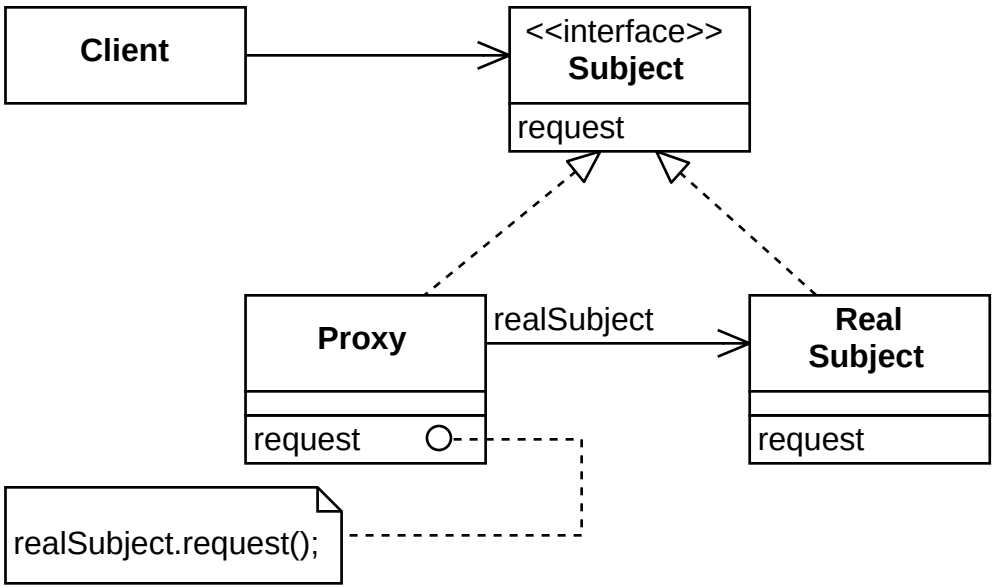


Applicability

A `Proxy` can be used as follows:

- 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.
- A *smart reference* is a replacement for a bare pointer that performs additional actions when an object is accessed.

Structure


Client
<<interface>>
Subject
request
Proxy
request
realSubject
Real
Subject
request
realSubject.request();

Participants

- **`Proxy`** (**`ImageProxy`**):
  - maintains a reference that lets the proxy access the real subject
  - provides an interface identical to `Subject`'s interface so that a proxy can be substituted for the real subject
  - provides an interface identical to Subject's interface so that a proxy can be substituted for the real subject
  - controls access to the real subject
  - other responsibilities depend on the kind of proxy such as encoding requests and its arguments, information caching, and access protection
- **`Subject`** (**`Graphic`**)
  - defines the common interface for `RealSubject` and `Proxy` so that `Proxy` can be used anywhere a `RealSubject` is expected
- **`RealSubject`** (**`Image`**)
  - defines the real object the proxy represents

Collaborations

- `Proxy` forwards the request to the RealSubject when appropriate, depending on the kind of proxy.

Consequences

- A proxy introduces a level of indirection.
- This indirection can be used to delay concrete actions on the `RealSubject`. For example, a *copy-on-demand* proxy can delay the creation of a large object until a client is actually going to modify it.

Implementation

- `Proxy` does not alway need to know the type of the `RealSubject`, since, in some situations, the `Proxy` can deal with its `Subject` solely through an abstract interface.
- `Proxy` needs to refer to the `RealSubject`, be it in memory or not. Thus, the Proyx needs a kind of key to get access to the right `RealSubject` object during the boot-strap procedure.

Sample Code

In this example, a virtual proxy is given. The abstract class `Graphic` defines the interface for graphical objects:

```
public abstract class Graphic {
   public abstract void draw(Point at);
   public abstract void handleMouse(Event event);
   public abstract Point getExtent ();
   public abstract void load(InputStream from);
   public abstract void save(OutputStream to);
}
```

The `Image` class implements the `Graphic` interface to display image files. `Image` overrides `handleMouse` to let users resize the image interactively.

```
public class Image extends Graphic {
   public Image(String fileName) {...} // loads image from file
   public void draw(Point at) {...}
   public void handleMouse(Event event) {...}
   public Point getExtent() {...}
   public void load(ObjectInputStream from) {...}
   public void save(ObjectOutputStream to) {...}
   // private stuff, omitted.
}
```

Class `Point` is a helper containing the coordinates of a point.

`ImageProxy` has the same interface as `Image`:

```
// Class ImageProxy:
public ImageProxy(String fileName)
{  this.fileName = fileName;
   extent = Point.zero();    // don't know extent yet
   image = null;
}
```

The protected method `getImage` instantiates an `Image` object, if not yet done, and returns the corresponding object reference:

```
// Class ImageProxy:
protected Image getImage() {
   if (image == null) {
      image = new Image(fileName);
   }
   return image;
}
```

The implementation of `getExtent` returns the cached extent if possible; otherwise the image is loaded from the file. `draw` loads the image, and `handleMouse` forwards the event to the real image:

```
// Class ImageProxy:
public Point getExtent() {
   if (extent.equals(Point.zero())) {
      extent = getImage().getExtent();
   }
   return extent;
}
public draw(Point at) {
   getImage().draw(at);
}
public handleMouse(Event event) {
    getImage().handleMouse(event);
}
```

The `save` method saves the cached image extent and the image file name to a stream. `load` retrieves this information and initializes the corresponding members.

```
// Class ImageProxy:
public void save(ObjectOutputStream to) {
   to.writeObject(extent);
   to.writeObject(fileName);
}
public void load(ObjectInputStream from) {
   extent = (Point) from.readObject();
   fileName = (String) from.readObject();
}
```

Finally, suppose there is a class `TextDocument` that can contain Graphic objects:

```
public class TextDocument {
   public TextDocument() {...}
   void insert(Graphic g) {...}
   // ...
}
```

An `ImageProxy` object can be inserted into a text document like this:

```
// A client:
TextDocument text = new TextDocument();
// ...
text.insert(new ImageProxy(“anImageFileName”));
```

Related Patterns

- Adapter: provides a different interface to the object it adapts. In contrast, proxy provides the same interface as its subject.
- Decorator: Can have similar implementation as the proxy, but has a different purpose. For example, a decorator adds responsibilities to the decorated object.