Design Patterns II

CS193D, 3/8/06
The Proxy Pattern

Proxies are stand-ins for other objects. You might use a proxy to perform access control, serve as a middleman, or mask issues with the underlying object.

*Virtual Proxy* – Allows you to create a low-overhead object until the full underlying object is needed.

*Remote Proxy* – Allows you to access an object on a network as though it were local.

*Access Proxy* – Allows you to check permissions before using the protected functionality.

*Smart Proxy* – Allows you to provide additional functionality.
Example: Apache Axis (Remote Proxy)

A SOAP library that can generate client-side proxy objects (stubs):

```cpp
// on the client
int main() {
    Calculator c;
    int intOut;
    c.add(20, 40, intOut);
    cout << "result is = " << intOut << endl;
}

// Calculator.cpp (pseudocode)
void Calculator::add(int x, int y, int& result) {
    SOAPEnvelope env("http://foo.com/action/add");
    env.addArgument("x", x);
    env.addArgument("y", y);
    SOAPResult response = env.sendRequest();
    result = response.getIntValue();
}
```
Example: Persisted Object Proxy (a Smart Proxy)

class User : public DatabaseObject {
    public:
        string getName();
        void setName(string inName);
};

class UserProxy {
    public:
        void setName(string inName) {
            if (!isNameValid(inName)) throw "bad name";
            mUser.setName(inName);
        }
        string getName() {
            if (mUser.getName() == "#NULL#") return "";
            return mUser.getName();
        }
    private:
        User mUser;
};
The Adapter (Wrapper) Pattern

An adapter or wrapper gives a new interface to an existing class. Usually, it's for compatibility reasons, but it could also be for ease of use.

Example: wrap a new version of a library with the old version's API:

class LibTranslateWrapper : public LibTranslate {
    public:
        string translate(string inText, string inLang) {
            // the new API uses language codes
            int langCode = lookupLangCode(inLang);
            return translate(inText, langCode);
        }
};
The Decorator Pattern

A decorator is a temporary embellishment to an object that adds new functionality or allows access through a new lens.

Example: Reading a stream that contains a particular type of data:

```
int imageFormat; string name;
inputFile >> imageFormat;
ImageReader ireader(inputFile);
Image* theImage = ireader.readImage();
inputFile >> name;
```
Example: Web Page Styles

HTML uses an XML-like syntax for styles:

```html
<B>For those about to rock, we salute you.</B>
<I><B>For those about to rock, we salute you.</B></I>
```

```cpp
class Paragraph {
public:
    Paragraph(const string& inText) : mText(inText) {}
    virtual string getHTML() const { return mText; }
protected:
    string mText;
};
```
class BoldParagraph : public Paragraph {
    public:
        BoldParagraph(const Paragraph& inPara) :
            Paragraph(""), mDecorated(inParagraph) {}

        virtual string getHTML() const {
            return "<B>" + mDecorated.getHTML() + "</B>";
        }
    protected:
        const Paragraph& mDecorated;
};

class ItalicParagraph : public Paragraph {
    public:
        ItalicParagraph(const Paragraph& inPara) :
            Paragraph(""), mDecorated(inParagraph) {}

        virtual string getHTML() const {
            return "<I>" + mDecorated.getHTML() + "</I>";
        }
    protected:
        const Paragraph& mDecorated;
};
Using the HTML Decorators

The decorators haven't changed the interface, just the functionality:

```cpp
Paragraph p("For those about to rock...");

// Bold
cout << BoldParagraph(p).getHTML() << endl;

// Bold Italic
cout << ItalicParagraph(BoldParagraph(p)).getHTML() << endl;

<B>For those about to rock...</B>
<I><B>For those about to rock...</B></I>
```
Decorating versus Subclassing

Styled paragraphs as *types* of paragraphs:

Decorators allow you to apply behavior without introducing new types.
The Chain of Responsibility Pattern

This is a behavioral pattern that gives you a way to give multiple objects a crack at handling a particular task.
void Rectangle::handleMessage(int msg) {
    switch (msg) {
    case kDraw:
        drawSelf();
        break;
    default:
        // Rectangle doesn't handle this message
        Shape::handleMessage(msg);
    }
}

void Shape::handleMessage(int msg) {
    switch (msg) {
    case kClick:
        gCurrentSelection = this;
        break;
    default:
        // Shape doesn't handle this message
        mCanvas.handleMessage(msg);
    }
}
Notes about Chain of Responsibility

- The chain often mirrors an object hierarchy, but doesn't have to (e.g. Canvas is not the superclass of Shape)

- It can be brittle in some implementations. One class could break the chain or cause an infinite loop.

- It's a reasonable way to model layering, where some objects should have the first shot at a command.

- It's a common way of handling menu selections in a graphical application.
The Observer Pattern

The Observer Pattern, like the Chain of Responsibility, provides a way for objects to react to some sort of event.

An Observer registers a priori for the events it is interested in, and is notified when the event is broadcast. This is often called publish-subscribe.

Example: Alert tool that listens for certain system events

Example: Graphical application where multiple objects might want to do something when a button is pressed.
The Abstract Factory Pattern

A factory is an object whose job is to create other objects:

class EchoTaskFactory
{
    public:
        Task* make() {
            return new EchoTask();
        }
};
Factories versus Constructors

The advantages of Factories over constructors are:

1) Centrality. By creating a separate object to create instances, you can easily create an object pool.

2) Polymorphic Creation. You can structure factories into a hierarchy and create objects where the type is determined at runtime.

The abstract factory is just a formalization of one approach to HW3.
Abstract Factory Hierarchies

class TaskFactory {
    public:
        [etc]
        virtual Task* make() = 0;
    }
};

class EchoTaskFactory : class EchoTask :
    public TaskFactory {
        public Task {
            public:
                [etc]
                virtual Task* make() {
                    return new EchoTask();
                }
            }
        }
    };

class CompileTaskFactory : class CompileTask :
    public TaskFactory {
        public Task {
            public:
                [etc]
                virtual Task* make() {
                    return new CompileTask();
                }
            }
        }
    };

The New Parallel Hierarchies

TaskFactory

EchoTask Factory  CompileTask Factory

Task

EchoTask  CompileTask
So?

By building polymorphic objects that create subclasses of Tasks, we can determine the type of task to create at runtime.

```c++
TaskFactory currentFactory = getCurrentTaskFactory();
Task* theTask = currentFactory.make();

// What is theTask? Could be an EchoTask... could be a CompileTask. Could be peaches, could be lunchmeat.
```

Next step: Determining what factory to use.
Storing Factories in a Table

The method for dealing with which factory to use varies from application to application.

- Factories based on resource availability (TCP versus AppleTalk)
- Factories based on user preferences (Valentines Card versus BDay)
- Factories stored for later lookup (XML-driven)

```cpp
void TaskManager::init()
{
    sTaskMap["compile"] = new CompileTaskFactory();
    sTaskMap["echo"] = new EchoTaskFactory();
}
```
Factories for Modal Behavior and Data
Factories are often used to group together a collection of data or functionality that varies based on some mode, like language.

```cpp
void handleError(int errNum) {
    if (errNum == kFileError) {
        cerr << gLanguageFactory.getFileErrorString();
    } else if (errNum == kLoadError) {
        cerr << gLanguageFactory.getLoadErrorString();
    }
}

class FrenchLanguageFactory : public LanguageFactory {
    public:
        string getFileErrorString() const {
            return "Je suis un essuie-glace";
        }
    [etc]
};
```