B16 Design Patterns

Lecture 2

Victor Adrian Prisacariu
http://www.robots.ox.ac.uk/~victor
Course Content

I. Code Design Patterns
   1. Motivation, Classification, UML
   2. Creational Patterns
   3. Structural Patterns
   4. Behavioral Patterns

II. Algorithm Design Patterns

Slides on Weblearn
UML Reminder

Diagram showing class relationships:
- **Human**
  - +name: string
  - +Talk()

- **Animal**
  - +MakeSound()
  - \(0..1\) plays with \(0..*\)

- **Food**

- **Toy**

- **Tail**

- **Cat**
  - +name: string
  - +MakeSound()

- **Dog**
  - +name: string
  - +MakeSound()

- **Kitten**
  - +name: string
  - +MakeSound()
  - +GrowUp()
We use a simplified version of C++ (no pointers 😊).

```java
public class Animal {
    public string name;
    public void MakeSound() {}
}
```

```java
public class Animal {
    private string name;
    public abstract void MakeSound();
}
```

```java
public interface Animal {
    void MakeSound();
}
```

```java
public class Animal {
    private string name;
    public void MakeSound();
}
```

```java
public class Cat : Animal {
    public string name;
    public void MakeSound();
}
```
Code Design Pattern Categories

Three fundamental groups:

• **Creational**
  – They abstract the instantiation process.
  – Make systems independent on how objects are compared, created and represented.

• **Structural**
  – Focus on how classes and objects are composed to form (relatively) large structures.
  – Generally use inheritance.

• **Behavioral**
  – Describe how different objects work together.
  – Focus on
    • The algorithms and assignment of responsibilities among objects.
    • The communication and interconnection between objects.
Creational Patterns – Summary

- **Singleton**: limits object creation to only one instance.
- **Prototype**: creates objects by cloning existing objects.
- **Factory method**: method that creates related objects.
- **Builder**: separates construction and representation.
- **Abstract factory**: groups object factories that have a common theme.

Code Design Pattern Categories

Three fundamental groups:

• **Creational**
  - They abstract the instantiation process.
  - Make systems independent on how objects are compared, created and represented.

• **Structural**
  - Focus on how classes and objects are composed to form (relatively) large structures.
  - Generally use inheritance.

• **Behavioral**
  - Describe how different objects work together.
  - Focus on
    • The algorithms and assignment of responsibilities among objects.
    • The communication and interconnection between objects.
Structural Patterns: Decorator

• You want to add functionality to a pre-existing object, and:
  – The new functionalities are independent of each other.
  – The object is unaware of the new functionalities.

• Example:
  – You have a **ByteWriter** class, that writes an array of bytes to a file on disk, and inherits on a generic Writer interface (i.e. fully abstract class).
  – You want to add a writer for text files and a writer that encrypts the array of bytes.
**Structural Patterns: Decorator**

**ByteWriter** is the original class, implementing the **Writer** interface.

**Decorator_TextWriter** is an example decorator that:
- Contains one instance of **ByteWriter**.
- Adds methods specific to text writing.

**Decorator_EncryptedWriter** is an example decorator that:
- Contains one instance of **ByteWriter**.
- Adds methods specific to encryption.

The various decorators can be chained.
public interface Writer { void Write(); }

public class ByteWriter : Writer {
    public void Write() { /* byte writer */ }
}

public class Decorator_TextWriter : Writer {
    public Writer localWriter;
    public void Write() {
        /* do something for text */
        localWriter.Write();
    }
}

public class Decorator_EncryptedWriter : Writer {
    public Writer localWriter;
    public void Write() {
        /* do something to encrypt */
        localWriter.Write();
    }
}

class Example {
    static void Main(string[] args) {
        char bytes = new char[199];
        ByteWriter bw = new ByteWriter();
        bw.Write(bytes); // writes the bytes
        Decorator_EncryptedWriter ew = new Decorator_EncryptedWriter();
        ew.Write(bytes); // encrypts and writes the bytes
    }
}
Structural Patterns: Proxy

• You want to mediate access to an object.
• We might also want to provide additional functionality when the object is accessed.

• Examples:
  – Access control, e.g. on iOS when you want to access the camera, the software checks if the app has permissions.
  – Accessing remote files, e.g. the files would need to be downloaded locally before they could be opened, as in Google Drive.
Structural Patterns: Proxy

- **FileReader** is a class responsible for reading files, that can be accessed from the client app.
- **LocalReader** is our target object.
- **Proxy_RemoteReader** is our proxy class, that mediates access to the local reader.
public interface FileReader { void Read(); }

class LocalReader : FileReader {
    public void Read() { /* byte reader */ }
}

class Proxy_RemoteReader : FileReader {
    public LocalReader localReader;
    public void Read() {
        /* do something for connection, download the data */
        localReader->Read();
    }
}

class Example {
    static void Main(string[] args) {
        Proxy_RemoteReader* rr = new Proxy_RemoteReader();
        rr->Read(); // reads from proxy
    }
}
public interface FileReader { void Read(); }

public class LocalReader : FileReader {
    public void Read() { /* byte reader */ }
}

public class Proxy_RemoteReader : FileReader {
    private LocalReader localReader;
    public void Read() {
        /* do something for connection, download the data */
        localReader->Read();
    }
}

class Example {
    static void Main(string[] args) {
        Proxy_RemoteReader* rr = new Proxy_RemoteReader();
        rr->Read(); // reads from proxy
    }
}
Structural Patterns: Bridge

• You might want to decouple an abstraction from an implementation, to make them independent of each other.

• Example:
  – You want to support multiple versions of your code.
  – You want to support multiple implementation of the same code (e.g. across multiple devices).
• **BitcoinMiner** is the main access point.
• It can use either **BitcoinMinerImplementation**, ie either **BitcoinMiner_Software** or **BitcoinMiner_Hardware**.
public interface BitcoinMinerImplementation { void MineBitcoin(); }

public class BitcoinMiner_Software : BitcoinMinerImplementation {
    public void MineBitcoin() { /* software miner */ }
}

public class BitcoinMiner_Hardware : BitcoinMinerImplementation {
    public void MineBitcoin() { /* hardware miner */ }
}

public class BitcoinMiner {
    private BitcoinMinerImplementation implementation;
    public void MineBitcoin() {
        implementation->MineBitcoin();
    }
    public BitcoinMiner() {
        /* some magical way of deciding between implementations */
    }
}

class Example {
    static void Main(string[] args) {
        BitcoinMiner* mn = new BitcoinMiner();
        mn->MineBitcoin();
    }
}
Structural Patterns: Decorator, Proxy and Bridge

```java
public interface File Reader { void Read(); }

public class LocalReader : FileReader {
    public void Read() { /* byte reader */ }
}

public class Proxy_RemoteReader : FileReader {
    private LocalReader localReader;
    public void Read() {
        /* do something for connection, download the data */
        localReader->Read();
    }
}

class Example {
    static void Main(string[] args) {
        Proxy_RemoteReader* rr = new Proxy_RemoteReader();
        rr->Read(); // reads from proxy
    }
}
```

```java
public interface BitcoinMinerImplementation { void MineBitcoin(); }

public class BitcoinMiner_Software : BitcoinMinerImplementation {
    public void MineBitcoin() { /* software miner */ }
}

public class BitcoinMiner {
    private BitcoinMinerImplementation implementation;
    public void MineBitcoin() {
        implementation->MineBitcoin();
    }
}

class Example {
    static void Main(string[] args) {
        BitcoinMiner* mn = new BitcoinMiner();
        mn->MineBitcoin();
    }
}
```

Proxy  Bridge
Structural Patterns: Decorator, Proxy and Bridge

```java
public interface Writer { void Write(); }

public class ByteWriter : Writer {
    public void Write() { /* byte writer */ }
}

public class Decorator_EncryptedWriter : Writer {
    public Writer localWriter;
    public void Write() {
        /* do something to encrypt */
        localWriter->Write();
    }
}

class Example {
    static void Main(string[] args) {
        char bytes = new char[199];
        Decorator_EncryptedWriter ew = new Decorator_EncryptedWriter();
        ew->Write(bytes); // encrypts and writes the bytes
    }
}

public interface FileReader { void Read(); }

public class LocalReader : FileReader {
    public void Read() { /* byte reader */ }
}

public class Proxy_RemoteReader : FileReader {
    private LocalReader localReader;
    public void Read() {
        /* do something for connection, download the data */
        localReader->Read();
    }
}

class Example {
    static void Main(string[] args) {
        Proxy_RemoteReader* rr = new Proxy_RemoteReader();
        rr->Read(); // reads from proxy
    }
}
```

Decorator

Proxy
Structural Patterns: Decorator, Proxy and Bridge

• Decorator, Proxy and Bridge are quite similar, both from pov of code and UML.

• The purpose is different:
  – Decorator uses interface to add extra functionality.
  – Proxy uses interface to mediate functionality.
  – Bridge uses the interface to switch functionality.
Structural Patterns: Adapter

• You have an old piece of code, and want to use it in your current system.

• Example:
  – Math libraries, where there are existing old implementations in e.g Fortran.
  – Game console emulators or Apple’s move from PowerPC to Intel.
• The `AppThatNeedsOldCode` calls the `ComputeSVD()` method in from the generic `MathLibrary` interface.

• The `Adapter` class translates these calls into calls to `ComputeSVDFortran()`. 
**Structural Patterns: Adapter**

```java
public interface MathLibrary { void ComputeSVD(); }

public class OldMathLibrary {
    public void ComputeSVDFortran() { /* magically runs the old code */ }
}

public class Adapter : MathLibrary {
    OldMathLibrary oldMathLibrary;

    public void ComputeSVD() { oldMathLibrary.ComputeSVDFortran(); }
}

class Example {
    static void Main(string[] args) {
        MathLibrary ml = new Adapter();
        ml.ComputeSVD(); // runs OldMathLibrary code
    }
}
```
Structural Patterns: Adapter vs Proxy

UMLs are potentially similar, but:
• Adapter modifies the interface to the target object, proxy maintains it.
• Adapter is specifically designed to change the interface, proxy is specifically designed to maintain it.
Structural Patterns: Façade

• You have a complicated framework with lots of subsystems and want to provide a simplified higher-level view.

• Example:
  – Most source code / SDK releases.
Structural Patterns: Façade

- You might have a large collection of code (or library) with various subsystems.
- You would create a class or collection of classes, eg Façade, which can access the subsystems, and expose functionality via DoSomething().
Fun Example: Neural Nets for Live Segmentation
Structural Patterns: Composite

• You want to manipulate a hierarchical collection of objects.
• You’d want to treat both complex (i.e. composite tree components) and primitive (i.e. tree leaves) objects the same.

• Example:
  – Managing music in a playlist.
  – Displaying user interface elements.
The interface (here **Music**) specifies the behavior for both leaf and composite nodes, e.g. **Play()**.

**Song** is the leaf node, implementing specific behavior.

**Playlist** is the composite node:
- Supports the addition (and removal, etc) of nodes.
- Can iterate and call interface-specific methods for the nodes.
public interface Music { void Play(); }

class Song : Music {
    public void Play() { /* plays song */ }
}

class Playlist : Music {
    List<Music> songs;
    public void Play() {
        foreach (Music music in musics)
            music.Play();
    }
    public void AddSong(Music music) {
        songs.Add(music);
    }
}

class Example {
    static void Main(string[] args) {
        Music playlist = new Playlist();
        Music song = new Song();
        ((Playlist)playlist).AddSong(song);
        ((Playlist)playlist).AddSong(song);
        playlist.Play(); // iterates through songs and plays them
    }
}
Structural Patterns: Flyweight

• You need to support a large number of objects, but want to avoid large memory / processor / bandwidth requirements.

• You can separate each object into an “intrinsic” (state-dependent) and an “extrinsic” (state-independent) component.

• The intrinsic state uses fewer resources than the extrinsic one, and, potentially, it also has fewer variations than the extrinsic.

• Example:
  – Rendering scenes in computer games.
  – Drawing photo albums.
Structural Patterns: Flyweight

- We removed the intrinsic, repeatable component of a class `Tree` into a class `TreeType`.
- `Tree` now is much smaller.
- `TreeType` is still big, but we don’t need as many.
- The types are maintained inside a `TreeFactory`.

https://refactoring.guru/design-patterns/flyweight
public class TreeType {
    public string name;
    public string color;
    public byte[] texture;

    public void Draw(int2 location)
    { /* draws a tree of a this type at a location */ } }

public class Tree {
    int2 location;
    TreeType treeType;

    public void Draw() { treeType.Draw(location); }
    public Tree(TreeType tt) { this.treeType = tt; }
}

public class TreeFactory {
    private TreeType[] treeTypes;
    public TreeType GetTreeType(int id) { return treeTypes[id]; }
    public TreeFactory() { /*somehow populate treeTypes */ }
}

class Example {
    static void Main(string[] args)
    {
        TreeFactory tf;

        // can have lots of tree and they won't take
        // much memory
        Tree tree = new Tree(tf.GetTreeType(0));

        // now we can draw the tree
        tree.Draw();
    }
}
Structural Patterns: Composite and Flyweight

- Both patterns deal structure multiple objects.
- Composite deals with accessing data in an uniform way.
- Flyweight can save space.
Structural Patterns – Summary

- **Decorator**: dynamically adds/overrides behaviour in an existing object.
- **Proxy**: provides a placeholder for another object to control access, reduce cost, and reduce complexity.
- **Bridge**: decouples abstraction from implementation.
- **Adapter**: allows classes with incompatible interfaces to work together.
- **Façade**: provides a simplified interface to a large body of code.
- **Composite**: composes similar objects so they can be manipulated as one.
- **Flyweight**: reduces the cost of creating and manipulating a large number of similar objects.