B16 Design Patterns

Lecture 3

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
We use a simplified version of C++ (no pointers 😊).

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

public interface Animal {
    void MakeSound();
}

class Cat : Animal {
    public string name;
    public void MakeSound();
    
    public class Animal {
        private string name;
        public abstract 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.
Behavioral Patterns: Strategy

• You want the client app to be able to select (dynamically) from a family of related algorithms/strategies.

• Example:
  – Sorting algorithms, line fitting approaches, search algorithm, encryption algorithms. Remember the examples from Lecture 1 – Creational Patterns.
  – Most problems have multiple algorithms ...
The **Strategy** abstract class has two implementations, **Strategy_A** and **Strategy_B**, and the client app can select either.
Behavioral Patterns: Strategy

```java
public interface Strategy {
    void Algorithm();
}

public class Strategy_A : Strategy {
    public void Algorithm() { /* strategy A algo */ }
}

public class Strategy_B : Strategy {
    public void Algorithm() { /* strategy B algo */ }
}

class Example {
    static void Main(string[] args) {
        Strategy a = new Strategy_A();
        Strategy b = new Strategy_B();

        a.Algorithm(); // runs Strategy A
        b.Algorithm(); // runs Strategy B
    }
}
```
Behavioral Patterns: State

• You need to switch between multiple algorithms/strategies, based on some (dynamic, internal) properties.

• Example:
  – Lots of state machine-type methods (hence the name): e.g. opening an automatic door or dealing with a request over a network.
Behavioral Patterns: State

• The states **State_A** and **State_B** implement the State interface and can both `DoThings()`.

• The **Context** class manages the live **State**, and is able to `goNext()` to the next state in the state machine, when certain internal conditions are met.
public class State_A : State {
    public void DoThing() { /* state A algo */ }
}

public class State_B : State {
    public void DoThing() { /* state B algo */ }
}

public class Context {
    public State liveState;

    public void GoNext() {
        if (liveState is State_A) liveState = new State_B();
        else liveState = new State_A();
    }

    public Context() {
        liveState = new State_A();
    }
}

class Example {
    static void Main(string[] args) {
        Context context = new Context();
        context.liveState.DoThing();
        context.GoNext();
    }
}
Behavioral Patterns: Template Method

You want the client app to be able to select (dynamically) from a family of related algorithms/strategies.

The various methods share well-defined parts of their operation.

Example:

- Frameworks for numerical optimization.
- Parts of the Windows Paint app, e.g. drawing brushes of different shapes.
Behavioral Patterns: Template Method

- The **EnergyFunctionMinimizer** implements, eg gradient descent or Newton’s method, and requires methods to `ComputeEnergyFunction()`, `ComputeGradient()` and `ComputeHessian()`.

- These are implemented specifically for each energy function in classes such as **Function_A** and **Function_B**.
Behavioral Patterns: Template Method

```csharp
public abstract class EnergyFunctionMinimizer {
    public abstract void ComputeEnergyFunction();
    public abstract void ComputeGradient();
    public abstract void ComputeHessian();

    public void Minimize() {
        ComputeEnergyFunction();
        ComputeGradient();
        ComputeHessian();

        // do something with the energy, gradient and hessian,
        // e.g new values for params = old values + alpha * gradient
    }
}

class NonLinearLSQ : EnergyFunctionMinimizer {
    public override void ComputeEnergyFunction() { /* ef compute for function A */ }
    public override void ComputeGradient() { /* gradient compute for function A */ }
    public override void ComputeHessian() { /* hessian compute for function A */ }
}

class Example {
    static void Main(string[] args) {
        EnergyFunctionMinimizer nonLinearLSQ = new NonLinearLSQ();
        nonLinearLSQ.Minimize();
    }
}
```
Behavioral Patterns: Strategy, State and Template Method

• Strategy and State are quite similar, except that:
  – Strategy does not assume any link between the various algorithms.
  – State changes the functionality (generally) based on a predefined structure.

• Template Method is similar to Strategy, except that some of the computation is shared between the various strategies, and contained within the base class.

• They can be combined, e.g. an non-linear optimization framework would likely combined Strategy (different techniques) with Template Method (different energy functions).
Behavioral Patterns: Visitor

• You want to add behavior to a class, but do not want to integrate the new code into that class.

• Examples:
  – Code that’s already written and cannot easily be modified too much, e.g. is already in production, used by other developers, extra code might not fit there, etc.
Behavioral Patterns: Visitor

- Each **Shape** does not implement the text export directly, but relegates the functionality to the specific **Visitors**.
- Each **Shape** links to the specific visitor through the `accept()` method, and the **TXTExportVisitor** implements the functionality for each specific type of **Shape**, via the `visit*()` methods.
```java
public interface Shape {
    void draw();
    void accept(Visitor v);
}

public class Circle : Shape {
    public void draw() { /* draw circle */}
    public void accept(Visitor v) { v.VisitCircle(this); }
}

public class Rectangle : Shape {
    public void draw() { /* draw circle */}
    public void accept(Visitor v) { v.VisitRectangle(this); }
}

public interface Visitor {
    void VisitCircle(Circle c);
    void VisitRectangle(Rectangle r);
}

public class TXTExportVisitor : Visitor {
    public void VisitCircle(Circle c) {
        /* export the circle params to txt */
    }
    public void VisitRectangle(Rectangle r) {
        /* export the rectangle params to txt */
    }
}

class Example {
    static void Main(string[] args) {
        Circle c = new Circle();
        TXTExportVisitor txtExport = new TXTExportVisitor();
        c.accept(txtExport);
    }
}
```
Behavioral Patterns: Memento

• You want to save, and later restore, the internal state of an object.

• Example:
  – Computer games saves.
  – Neural network parameters during training.
  – Undo / Redo.
  – …
Behavioral Patterns: Memento

- **Originator**: The object that has the state to be saved / loaded.
- **Caretaker**: The object that knows when and why the Originator needs to save and restores itself.
- **Memento**: The object that represents the Originator’s saved state.
public class Memento {
    public string textToSave;
}

public class Originator {
    public string textToSave;
    public Originator() {
        textToSave = "save me";
    }
    public void SetMemento(Memento memento) {
        this.textToSave = memento.textToSave;
    }
    public Memento CreateMemento() {
        Memento m = new Memento();
        m.textToSave = textToSave;
        return m;
    }
}

class Caretaker {
    static void Main(string[] args) {
        Originator originator = new Originator();
        Memento m = originator.CreateMemento();
        originator.SetMemento(m);
    }
}
Behavioral Patterns: Iterator

• You want to access the elements of a collection sequentially, but do not want to fix (or assume to know) the way the collection is structured.

• Example:
  – Most list/collection accesses should be done through iterators, unless performance is absolutely critical. This is especially important on structures that can change shape.
Each collection we want to be able to traverse should inherit an **IterableCollection** interface.

Each **IterableCollection** implementation should be able to create an Iterator for the specific collection, e.g. **ListIterator** and **VectorIterator**.

Each **Iterator** should be able to, e.g. move to **first()** element in the collection, the **next()** element and check if the enumeration **isDone()**.
```java
public interface Iterator {
    void first();
    void next();
    boolean isDone();
}

public interface IteratableCollection { Iterator CreateIterator(); }

public class VectorCollection : IteratableCollection {
    private int[] values;
    private int capacity;

    public class VectorIterator : Iterator {
        private VectorCollection collection;
        private int currentIndex;

        public VectorIterator(VectorCollection collection) {
            this.collection = collection;
        }

        public void first() { currentIndex = 0; }
        public void next() { currentIndex++; }
        public boolean isDone() { return !(currentIndex < collection.capacity - 1); }

        public int currentValue() { return collection.values[currentIndex]; }
        public void setCurrentValue(int value) { collection.values[currentIndex] = value; }
    }

    public Iterator CreateIterator() {
        return new VectorIterator(this);
    }

    public VectorCollection(int capacity) {
        this.capacity = capacity;
        values = new int[capacity];
    }
}

class Example {
    static void Main(string[] args) {
        VectorCollection vectorCollection = new VectorCollection(100);
        VectorCollection.VectorIterator iterator = (VectorCollection.VectorIterator)vectorCollection.CreateIterator();
        while (!iterator.isDone()) {
            iterator.setCurrentValue(10);
            Console.Out.WriteLine(iterator.currentValue());
            iterator.next();
        }
    }
}
```
Behavioral Patterns: Mediator

• You want two (or more) objects to communicate, without knowing each other’s identities so, to allow, the communication to change independently of the objects.

• Examples:
  – Applications for computer remote control (eg TeamViewer), since we cannot assume there’s exists a direct (i.e. non-firewalled) path from one computer to the other.
  – UI design, where each UI element (e.g. Button) would find it easier to communicate with a single Mediator then with multiple other UI elements.
Behavioral Patterns: Mediator

- Each **PingClient** has a reference to the **Mediator** object, and can call `Notify()`.
- The **Mediator** has references to all the **PingClients**, and can relay messages.
Behavioral Patterns: Mediator

```java
public interface Client {
    void SendPing();
    void ReceivePing();
}

public interface Mediator {
    void Notify();
}

public class PingClient : Client {
    private Mediator mediator;
    /* could have explicit address here */
    public void SendPing() { mediator.Notify(); }
    public void ReceivePing() { Console.Out.WriteLine("ping"); }
    public PingClient(Mediator mediator) {
        this.mediator = mediator;
    }
}

public class PingMediator : Mediator {
    public List<Client> clients;
    public void Notify() {
        /* could do something intelligent here, or */
        foreach (Client client in clients)
            client.ReceivePing();
    }
    public PingMediator() {
        clients = new List<Client>();
    }
}

class Example {
    static void Main(string[] args) {
        PingMediator m = new PingMediator();
        Client c1 = new PingClient(m);
        Client c2 = new PingClient(m);
        m.clients.Add(c1);
        m.clients.Add(c2);
        /* could have explicit address here */
        c1.SendPing();
    }
}
```

The code above demonstrates a simple implementation of the Mediator pattern in C#. The `Client` interface defines two methods, `SendPing` and `ReceivePing`, which a client can use to communicate. The `Mediator` interface has a single method, `Notify`, which is implemented in the `PingMediator` class. The `PingClient` class implements the `Client` interface and uses the mediator to notify clients when a ping should be sent. The `Main` method creates two `PingClient` objects and adds them to the mediator. When `c1.SendPing()` is called, the mediator notifies both clients to process the ping.
Behavioral Patterns: Observer

• You want objects to “observe” other objects, ie be notified of changes.

• Example:
  – Message boards.
  – Twitter
Behavioral Patterns: Observer

- **Subject** is a class whose state can change arbitrarily.
- **Observes** want to be notified of changes in state, so they can:
  - Subscribe (i.e. *Attach()* to notifications.
  - Unsubscribe (i.e. *Detach()* from notifications.
- **Notify()** will report the change of state to the various subscribed **Observers**.
Behavioral Patterns: Observer

```csharp
public interface Observer { void Update(); }

public class SpecificObserver : Observer {
    public void Update() { Console.Out.WriteLine("I am updated!"); }
}

public class Subject {
    private List<Observer> observers;

    public Subject() { observers = new List<Observer>(); }

    public void Attach(Observer o) { observers.Add(o); }
    public void Detach(Observer o) { observers.Remove(o); }
    public void Notify() {
        foreach (Observer o in observers)
            o.Update();
    }
}

class Example {
    static void Main(string[] args) {
        Subject s = new Subject();
        Observer o1 = new SpecificObserver();
        Observer o2 = new SpecificObserver();
        s.Attach(o1); s.Attach(o2);
        s.Notify();
    }
}
```

```
```
Observer and Mediator are very similar patterns, but:

• Mediator is centralized – all communication goes through the mediator classes.

• Observer distributes information: each object could subscribe to notification posted by all other objects.
Behavioral Patterns: Chain of Responsibility

• You want to build an object oriented version of the \texttt{if} ... \texttt{else if} ... \texttt{else if} ......... \texttt{else} ... \texttt{endif} idiom.

• Example:
  – Decryption algorithms when the encryption method is not known.
  – Input initialization (e.g. cameras, control devices, ...), when the input type is dynamic.

• Can be seen as a way of aggregating Decorator style objects.
Behavioral Patterns: Chain of Responsibility

https://refactoring.guru/design-patterns/chain-of-responsibility
Behavioral Patterns: Chain of Responsibility

• The **Handler** abstract class defines the prototype of each object in the chain, and may contain:
  – a reference to the `nextHandler` in the chain.
  – a reference to the handler above in the chain ie `super`.

• Each chain object, eg **Handler_A** or **Handler_C** executes a different `handle()` function.
public abstract class Decryptor {
    public Decryptor nextDecryptor;
    public abstract void Decrypt();
}

public class DecryptorAES : Decryptor {
    public override void Decrypt() {
        if (this is AES encryption) {
            /* AES decryption */
            return;
        }
        nextDecryptor.Decrypt();
    }
}

public class DecryptorRSA : Decryptor {
    public override void Decrypt() {
        if (this is RSA encryption) {
            /* RSA decryption */
            return;
        }
        nextDecryptor.Decrypt();
    }
}

class Example {
    static void Main(string[] args) {
        Decryptor aes = new DecryptorAES();
        Decryptor rsa = new DecryptorRSA();
        aes.nextDecryptor = rsa;
        aes.Decrypt();
    }
}
Behavioral Patterns: Command

• You need to handle object requests without knowing anything about the operation being requested or the receiver of the request.

• Example:
  – The callback mechanism e.g. dealing with UI calls (e.g. what happens when a button is clicked).
  – Copy and Paste functionality.
Behavioral Patterns: Command

https://refactoring.guru/design-patterns/command
Behavioral Patterns: Command

https://refactoring.guru/design-patterns/command
Behavioral Patterns: Command

https://refactoring.guru/design-patterns/command
Behavioral Patterns: Command

The **OnClickCommand** links the `Click()` action from the **Button** with the `ProcessClick()` method in the **OnClickReceiver**.

Related commands should be linked through inheritance, e.g. here **OnClickCommand** and **OnDoubleClickCommand** inherit **ClickCommand**.
Behavioral Patterns: Command

```java
public class OnClickReceiver {
    public void ProcessClick()
    { /* click is processed here */ }
}

public interface ClickCommand { void Execute(); }

public class OnClickCommand : ClickCommand {
    public OnClickReceiver clickReceiver;
    public void Execute() { clickReceiver.ProcessClick(); }
    public OnClickCommand(OnClickReceiver clickReceiver) {
        this.clickReceiver = clickReceiver;
    }
}

public class Button {
    public ClickCommand onClickCommand;
    public void Click() { onClickCommand.Execute(); }
}

class Example {
    static void Main(string[] args) {
        OnClickReceiver receiver = new OnClickReceiver();
        OnClickCommand onClickCommand = new OnClickCommand(receiver);

        Button button = new Button();
        button.onClickCommand = onClickCommand;

        button.Click(); // will be processed in the receiver
    }
}
```
Behavioral Patterns – Summary (1)

• **Strategy**: allows one of a family of algorithms to be selected live.
• **State**: allows an object to alter its behaviour when its internal state changes.
• **Template Method**: defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behaviour.
• **Visitor**: separates an algorithm from an object structure by moving the hierarchy of methods into one object.
• **Memento**: provides the ability to restore an object to its previous state (undo).

Behavioral Patterns – Summary (2)

- **Iterator**: accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator**: allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Observer**: is a publish/subscribe pattern which allows a number of observer objects to see an event.
- **Chain of responsibility**: delegates commands to a chain of processing objects.
- **Command**: creates objects which encapsulate actions and parameters, used to separate action initiator from action processor.