Introduction To Design Patterns

You will learn about design techniques that have been successfully applied to different scenarios.

What Is A Design Pattern?

- A general and reusable solution to a commonly occurring problem in the design of software.
- IT IS a template for how to solve a problem that has been used in many different situations.
- IT IS NOT a finished algorithm that can be directly translated into program code.
- The various Object-Oriented design patterns show interactions between classes and objects without being tied to the specific the program code that implements the pattern (language independent)
  - e.g., Information hiding, inheritance etc.

Origin Of Design Patterns

- The foundation for design patterns come from the original patterns specified in the book "*Design Patterns: Elements of Reusable Object-Oriented Software*"
- Authors: "The gang of four" (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides).
- Although examples of the patterns were provided in the C++ and SmallTalk programming languages the patterns can be applied to any Object-Oriented language.

The Model-View-Controller Pattern[1]

- Sometimes the same data may have to be accessed under different contexts e.g., powerful desktop, web, mobile device.
- Each context may require a different interface (e.g., web page on a mobile device, software on a computer).
- Even the context of a single program running on a single device there may be a desire to see different views of the data e.g., financial analysts may want to see details (spreadsheet and/or financial statement) whereas the shareholders or management may focus on overview views (graphs)

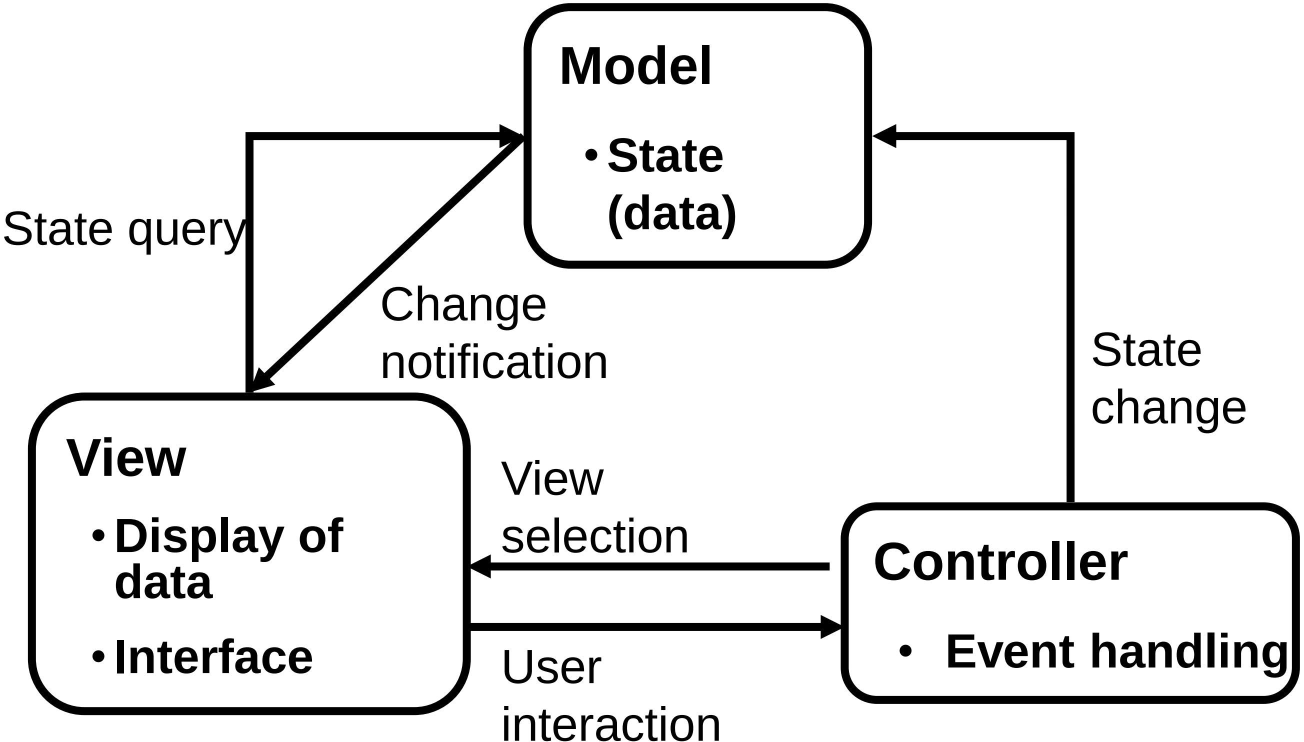
[1] Some additional sources that describe the model-view controller pattern:

I. Sun Microsystems: http://java.sun.com/blueprints/patterns/MVC-detailed.html

II. Microsoft: http://msdn.microsoft.com/en-us/library/ms978748.aspx

The Model-View-Controller Pattern[1]

- With this pattern, different parts are separate and independent:
  - Model: The data (database, text file):
  - View: How the data appears or the perspective under which it is viewed (graph, numerical)
  - Controller: How the data can be interacted with (GUI, command line).

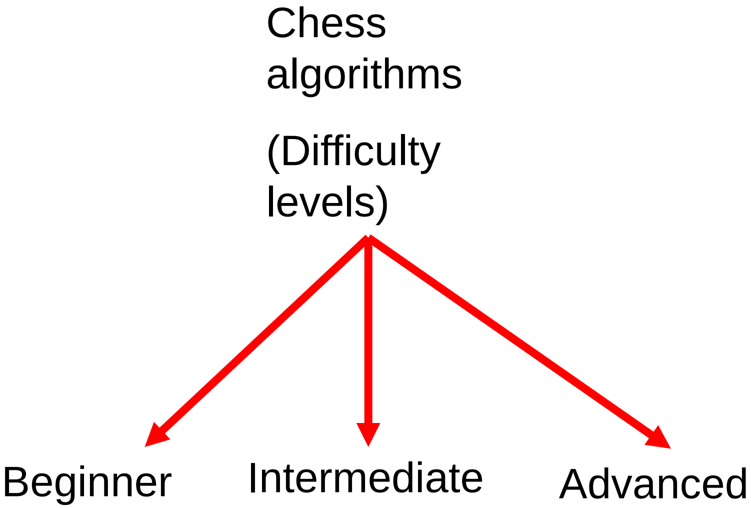


Model-View-Controller Pattern (2)

- Implementing different parts that are decoupled (minimized dependencies) provides many benefits:
  - One part may be changed independent of the other parts e.g., updates to the interface can have minimal impact on the data.
  - It's seldom that one person will have a deep understanding of all parts (e.g., knowledge of Accounting to create the financial statements vs. knowledge of web design to create the web interface). Different people with different areas of expertise can work on the different parts.
  - One version of the data can be created and maintained and as needed different ways of interacting and viewing data can be developed.

The Strategy Pattern

- The algorithm is determined at run time.



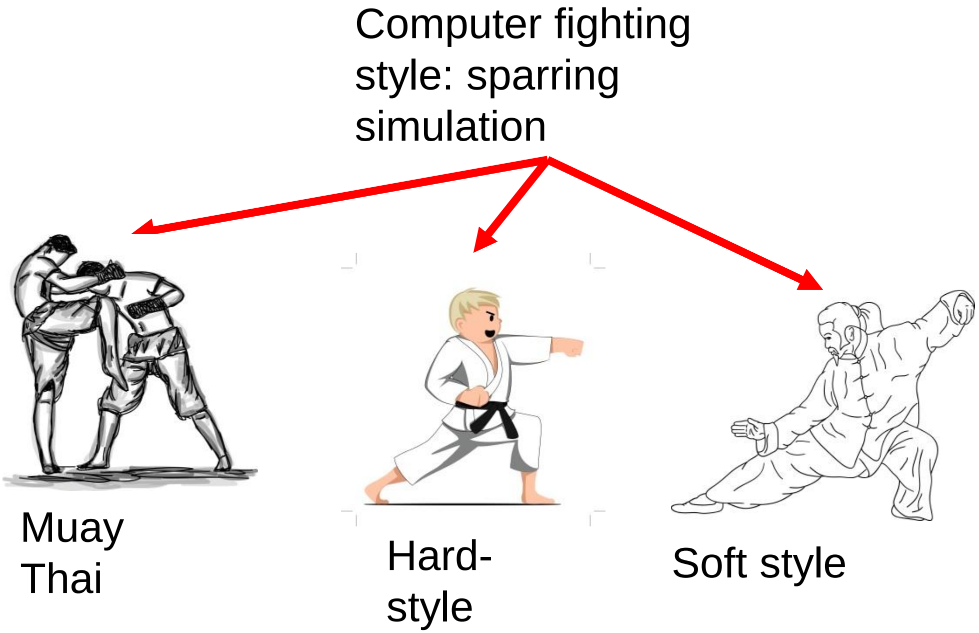
Chess algorithms

(Difficulty levels)

Beginner

Intermediate

Advanced



Computer fighting style: sparring simulation

Muay Thai

Hard-style

Soft style

Images: colourbox.com

The Strategy Pattern (2)

- One object contains a reference to another object.
- The second object determines the algorithm to execute.

The Strategy Algorithm: Example

Location of the example:

`/home/219/examples/designPatterns/strategy`

```
public class Driver {
   public static void main (String [] args) {
       MyContainer aContainer = null;

       // First algorithm
       aContainer = new MyContainer (new AddAlgorithm());
       System.out.println(aContainer.executeAlgorithm(2,5));

       // Second algorithm
       aContainer = new MyContainer (new MultiplyAlgorithm());
       System.out.println(aContainer.executeAlgorithm(2,5));
   }
}
```

The Strategy Algorithm: An Example (2)

```
public class MyContainer
{
   private Algorithm anAlgorithm;
   public MyContainer (Algorithm anAlgorithm)
   {
      this.anAlgorithm = anAlgorithm;
   }
   public int executeAlgorithm (int x, int y)
   {
      return(anAlgorithm.execute(x,y));
   }
}
```

The Strategy Algorithm: An Example (3)

```
public interface Algorithm {
   public int execute (int x, int y);
}

public class AddAlgorithm implements Algorithm {
   public int execute (int x, int y) {
      return (x+y);
   }
}

public class MultiplyAlgorithm implements Algorithm {
   public int execute (int x, int y) {
      return (x*y);
   }
}
```

Advantages Of The Strategy Pattern

- It decouples the context/container from the algorithm used by the context/container.
  - For the container it may allow the context/container to easily substitute additional algorithms.
    - ‘Expansion packs’
  - For the algorithm, the algorithm may be used in a number of different contexts/containers (e.g., sorting algorithms).

Side Note: Static Attributes

- Static attributes of a class are initialized when the Java virtual machine ("java") loads a class into memory.
- This must be done before any of the methods of the class can be called (even the constructor).
- Location of an illustrative example:

/home/219/examples/designPatterns/static

Static Attributes: Driver Class

```
public class Driver
{
    public static void main (String [] args)
    {
        Foo aFoo = new Foo();
    }
}
```

Static Attributes: Class Foo & Bar

```
public class Foo
{
    private static Bar aBar =
        new Bar();
    public Foo()
    {
        System.out.println(">>>
          Trace only: constructor
          Foo() <<<");
    }
}
```

```
public class Bar
{
    public Bar()
    {
        System.out.println(">>>
          Trace only: constructor
          Bar() <<<");
    }
}
```

The Singleton Pattern

- Singleton class: there is only one instance of the class (one object).
- That object provides a common set of operations for the rest of the program and globally accessible (variable) data.
- It is not the same as a purely static class.
  - Static methods but no variable attributes.
- The Singleton pattern is enforced by making the constructor private.
- Example singleton class: Random number generator.
  - For testing/debugging it is desirable to generate the same sequence of random numbers.

Singleton Example

- Location of the example:

```
/home/219/examples/designPatterns/singleton
```

Singleton: Driver

```
public class DriverSingleRandom
{
    public static void main(String [] args)
    {
        SingleRandom aSingleRandom = SingleRandom.getInstance();
        aSingleRandom.setSeed(1);
        for (int i = 0; i < 10; i++)
            System.out.println(i + ": " +
              aSingleRandom.nextInt());
        System.out.println();
    }
}
```

Class SingleRandom

```
public class SingleRandom
{
    private Random generator;
    private static SingleRandom instance = new SingleRandom();

    private SingleRandom()
    {
        System.out.println(">>> Trace only: this.SingleRandom()
          <<<");
        generator = new Random();
    }

    public static SingleRandom getInstance()
    {
        System.out.println(">>> Trace only:
                SingleRandom.getInstance() <<<");
        return(instance);
    }
```

1st

2nd

3rd

Class SingleRandom (2)

```
    public void setSeed(int seed)
    {
        System.out.println(">>> Trace only: ref.setSeed() <<<");
        generator.setSeed(seed);
    }


    public int nextInt()
    {
        System.out.println(">>> Trace only: ref.nextInt() <<<");
        return (generator.nextInt());
    }

}
```

Discussions/Resources: Singleton Pattern

- http://msdn.microsoft.com/en-us/library/ee817670.aspx

You Should Now Know

- What is a design pattern
- How the three example design patterns work