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 NOT a finished algorithm that can be directly translated into program code.
- IT IS a template for how to solve a problem that has been used in many different situations.
- Object-Oriented design patterns show interactions between classes and objects without the specific the program code that implements the pattern.

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 C++ and SmallTalk 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 within an interface 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:
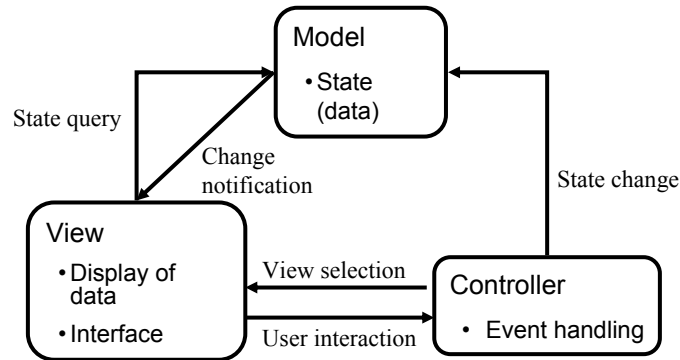
I.   Sun Microsystems: http://java.sun.com/blueprints/patterns/MVC-detailed.html

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

The Model-View-Controller Pattern[1]

• With this pattern the logic required to maintain the data is separated (database, text file) from how the data is viewed (graph, numerical) vs. how the data can be interacted with (GUI, command line).

```
                          ┌──────────────┐
                          │ Model        │
                          │  • State     │◄──────────┐
                          │    (data)    │           │
                          └──────────────┘           │
         State query    ▲        ╲                   │
                        │         ╲ Change            │ State change
                        │          ╲ notification     │
                        │           ▼                 │
   ┌──────────────────┐              ┌────────────────────┐
   │ View             │  View selection │ Controller       │
   │  • Display of    │◄──────────────  │                  │
   │    data          │                 │  • Event handling│
   │  • Interface     │  User interaction──►                │
   └──────────────────┘              └────────────────────┘
```

James Tam

Model-View Controller Pattern (2)

• With many client applications the view and the controller may be viewed as one entity.

• With web-based applications the view and the controller may be very well defined:
   – View: client browser program
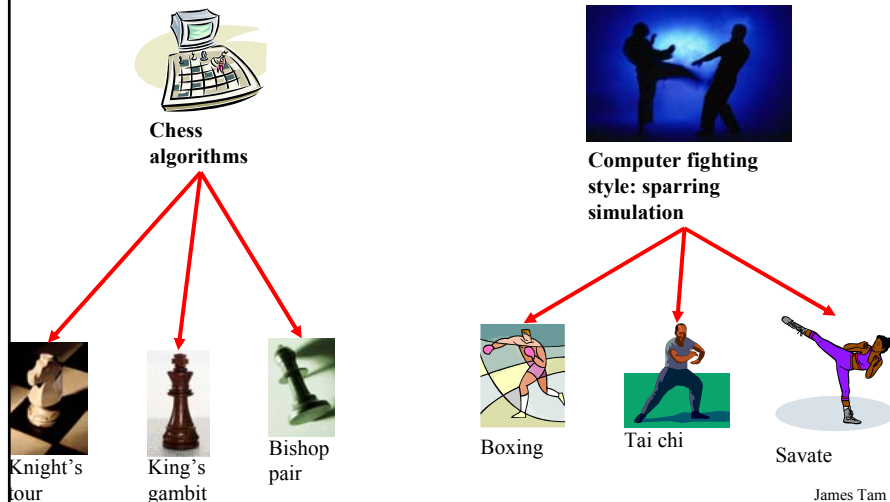   – Controller: the server side applications that handle the web requests

James Tam

Model-View-Controller Pattern (3)

- 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

Computer fighting style: sparring simulation

Knight's tour

King's gambit

Bishop pair

Boxing

Tai chi

Savate

The Strategy Pattern (2)

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

The Strategy Algorithm: Example

The complete online example can be found in UNIX under:
/home/233/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)

```java
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)

```java
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.
  - For the algorithm, the algorithm may be used in a number of different contexts/containers (e.g., sorting algorithms).

You Should Now Know

- What is a design pattern
- How two example design patterns work