Design Patterns

Designing object-oriented software is hard, and designing reusable object oriented software is even harder. You must

- find pertinent objects
- factor them into classes at the right granularity
- define class interfaces
- define inheritance hierarchies
- establish key relationships among classes
- the design should be specific to the problem at hand, but also general enough to address future problems and requirements
- you want to avoid redesign, or at least minimize it

New designers are overwhelmed by the options available and tend to fall back on non object oriented techniques they've used before. It takes a long time for novices to learn what good object oriented is all about.

One thing expert designers know what not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use again and again.

Consequently, you'll find recurring patterns of classes and communicating objects in many object oriented systems. These patterns solve specific design problems and make object oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience.

Design patterns advantages:

- make it easier to reuse successful designs and architectures
- expressing proven techniques as design patterns makes them more accessible to developers of new systems
- help you choose design alternatives that make a system reusable
- help you avoid alternatives that compromise reusability
- improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent

- help a designer get a design “right” faster

Design Patterns in Smalltalk MVC

The Model is the application object, the View is its screen presentation, and the Controller defines the way that user interface reacts to user input. The MVC decouples those objects to increase flexibility and reuse.

MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself. This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it.

The model communicates with its views when its values change, and the views communicate with the model to access these values.

This design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know the details of the others. This more general design is described by the **Observer** design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. MVC supports nested view with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

The design is applicable to a more general problem, which occurs whenever we want to group objects and treat the group like individual objects. This is described by the **Composite** design pattern.

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way its responds to the keyboard, for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.

A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller. It's even possible to change a view's controller at run-time to let the view change the way it responds to user input. For example, a view can be disabled so that it doesn't accept input simply by giving it a controller that ignores input events.

The View-Controller relationship is an example of the **Strategy** design pattern. A Strategy is an object that represents an algorithm. It's useful when you want to replace the algorithm either statically or dynamically, when you have a lot of variants of the algorithm, or when the algorithm has complex data structures that you want to encapsulate.

How Design Patterns Solve Design Problems

Design patterns solve many of the day to day problems object oriented designers face, and in many different ways:

1. Finding Appropriate Objects

The hard part about object oriented is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on. They all influence the decomposition, often conflicting ways. There are many different approaches, but there will always be disagreement on which approach is best.

Many objects in a design come from the analysis model. But object oriented designs often end up with classes that have no counterparts in the real world. Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.

Design patterns help you identify less-obvious abstractions and the objects that capture them.

2. Determining Object Granularity

Objects can vary tremendously in the size and number. They can represent everything down to the hardware or all the way up to entire applications.

Some design patterns describe how to represent complete subsystems as objects (**Facade**), how to support huge number of objects at the finest granularity (**Flyweight**), ways of decomposing an object into smaller objects, or how to yield objects whose only responsibilities are creating other objects (**Factory, Builder**).

3. Specifying Object Interfaces

Interfaces are fundamental in object oriented systems. Objects are known only through their interfaces. An object's interface characterizes the complete set of requests that can be sent to the object.

Design patterns help you define interfaces by identifying their key elements and the kinds of data that gets sent across the interface. A design pattern might also tell you what *not* to put in the interface.

Example: The **Memento** pattern describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later. It stipulates that objects must define two interfaces: a restricted one that lets client hold and copy mementos, and a privileged one that only the original object can use to store and retrieve state in the memento.

Design patterns also specify relationships between interfaces. They often, require some classes to have similar interfaces.

4. Specifying Object Implementations

A design pattern specifies class operations, instance variables, object instantiation, inheritance, abstract classes, and for some cases implementation pseudocode. Emphasis is placed on programming to an interface, versus an implementation.

There are two benefits to manipulating objects solely in terms of the interface defined by the abstract classes:

1. Clients remain unaware of the specific types of objects they use, as long as objects adhere to the interface that clients expect.
2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

5. Putting Reuse Mechanisms to Work

6. Designing for Change

The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.

Design patterns help you ensure that a system can change in specific ways. Each pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

Common causes of redesign:

1. Creating an object by specifying a class explicitly.
2. Dependence on specific operations.
3. Dependence on hardware and software platform.
4. Dependence on object representations or implementations.
5. Algorithmic dependencies.
6. Tight coupling.
7. Extending functionality by subclassing.
8. Inability to alter classes conveniently.

Role of Design Patterns in the development of three broad classes of software:

Application Programs

Internal reuse, maintainability, and extension are high priority. Internal reuse ensures that you don't design and implement any more than you have to. Design patterns that reduce dependencies can increase internal reuse. They also make an application more maintainable when they're used to limit platform dependencies and to layer a system. They enhance extensibility by showing you how to extend class hierarchies and how to exploit object composition. Reduced coupling also enhances extensibility.

Toolkits

A toolkit is a set of related and reusable classes designed to provide useful, general purpose functionality. Example: set of collection classes for lists, tables, or stacks. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do your job. They emphasize **code reuse**.

Toolkits have to work in many applications to be useful. That makes it more important to avoid assumptions and dependencies that can limit the toolkit's flexibility and consequently its applicability and effectiveness.

Frameworks

A framework is a set of cooperating class that make up a reusable design for a specific class of software. You customize a framework to a particular application by creating application specific subclasses of abstract classes from the framework.

The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control.

A framework predefines these design parameters so that the application designer can concentrate on the specifics of the application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize **design reuse** over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

When you use a toolkit (or any subroutine) you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code it calls.

As a result, you can build applications faster and with similar structure. They are easier to maintain, and they seem more consistent to their users.

If applications are hared to design, and toolkits are harder, then frameworks are hardest of all. A framework designer gambles that one architecture will work for all applications in the domain. Therefore it is imperative to design a framework with loose coupling and to be as flexible and extensible as possible.

The issues just discussed are most critical to framework design. A framework that addresses them using design patterns is far more likely to achieve high levels of design and code reuse than one that doesn't. Mature frameworks usually incorporate several design patterns.

Differences between patterns and frameworks:

1. Design patterns are more abstract than frameworks.

2. Design patterns are smaller architectural elements than frameworks.

3. Design patterns are less specialized than frameworks.

Model – View
Approach to Interface Design

Java encourages you to use an approach for GUI design that tells you to identify particular classes with one of two groups:

Observable group (models): classes that correspond to those domain entities that contain displayed info (subjects).

Observer group (views): classes that control your display (clients).

Division of Responsibility:

- Each model keeps track of the views that display the model's values.
- Whenever a model's value changes, all the dependent views (observers) are notified that a change has occurred.
- Each dependent view, in response to a change notification, interrogates the model's values to determine what should be done to the display for which the view is responsible.

Observable Class (util package)

- **addObserver()**: attaches views.
- **deleteObserver()**: detaches views.
- **setChanged()**: establishes that a change has occurred.
- **notifyObservers()**: announces to viewers that a change has occurred, provided that the setChanged method has occurred since the last notifyObserver.

Observer Interface (util package)

- **update()**: defines what is to be done when a view receives a change notification from a model.

Example:

```
public class ModelX extends Observable  {

   ??? someValues;

   public void setValues( . . .)  {
         . . .
         changed();
    }

   public void move( . . . )  {
         . . .
         changed();
    }

   public void changed()  {
         setChanged();
         notifyObservers();
    }

 . . .
}
```

```
public class ViewX  implements Observer {

   ModelX aModel;
   ???  viewValues;


    public void update (Observable o, Object x) {
         ModelX m = (ModelX) o;
         viewValues = m.getValues();
         repaint();
    }
```

```
  public void attachObservable(ModelX m)
  {
        if (m != aModel)
          if (aModel != null)
                { aModel.deleteObserver(this);}
          aModel = m;
          aModel.addObserver(this);
          aModel.changed();
   }
```

```
public class Example extends Applet  {

  ViewX aView = new ViewX();

  public void init()  {
        . . .
        ModelX aModel = new ModelX( . . .);
        aView.attachObservable(aModel);
   }
}
```

Actions:

1. ***attachObservable*** calls ***attachObserver***, which attaches the Model to the View.
2. ***attachObservable*** calls ***changed*** in the Model which calls ***setChanged*** and ***notifyObservers***.
3. ***notifyObservers*** tells Java to call the update method of the attached observers.
4. ***update*** calls ***getValues*** from the Model, then calls ***repaint*** to update the display.