Introduction

What is a design pattern?

According to Christopher Alexander,
"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Definition (In Software Development):
Design patterns are repeatable or reusable solutions to commonly occurring problems in a certain context in software design.

Four essential elements of a design pattern:
The four essential elements of a design pattern are:

Pattern name: The pattern name provides a way to describe the design problem, its solution and consequences in a word or two. A pattern name provides vocabulary using which we can communicate with other people, document them and reference them in software designs. Finding a good name for a design pattern is a major issue.

Problem: The problem describes when to apply the pattern and in which context to apply the pattern. The problem may describe the class and object structures in a bad design context. Sometimes, the problem might also list out the conditions that must be pre satisfied to apply the pattern.

Solution: The solution describes the elements (classes and objects) that make up the design, their relationships, responsibilities and collaborations. The solution does not describe a particular concrete design or implementation (no code). A pattern is a template that can be applied in many different situations.

Consequences: These are the costs and benefits of applying the patterns. These consequences involve space and time trade-offs, language and implementation issues as well as the effect of pattern on system's flexibility, extensibility or portability. Listing these consequences explicitly enables us to understand and evaluate them.

Need for design patterns

Design of object oriented software is hard: People who are familiar with object oriented software development are familiar with the fact that ***designing reusable object oriented software is very hard***. So, what will make the development of reusable software easier?

Experienced Vs novice designers: The experienced designers use their knowledge they had gained from their experience in designing object oriented software. The novice designers who have no experience in designing are left with numerous ways to develop the designs which may lead them to develop ineffective designs. The experienced designers have documented their experience as design patterns. These ***design patterns helps the novice designers to develop the right and effective design***.

Flexible, elegant and reusable software: By applying design patterns we can develop software that is flexible: software that is easily adaptable, elegant: easy to understand and code, and reusable: designs or code can be reused for developing software in the future.

Documentation: As a software designer and developer we will be developing a variety of software for several problems. While designing, we will observe some patterns which will be encountered in other future designs. It is a human habit to forget what we do now. So, we can document the patterns we observe in our designs for future reference. ***Design patterns are well documented which provide us with necessary information when needed***.

Use of design patterns

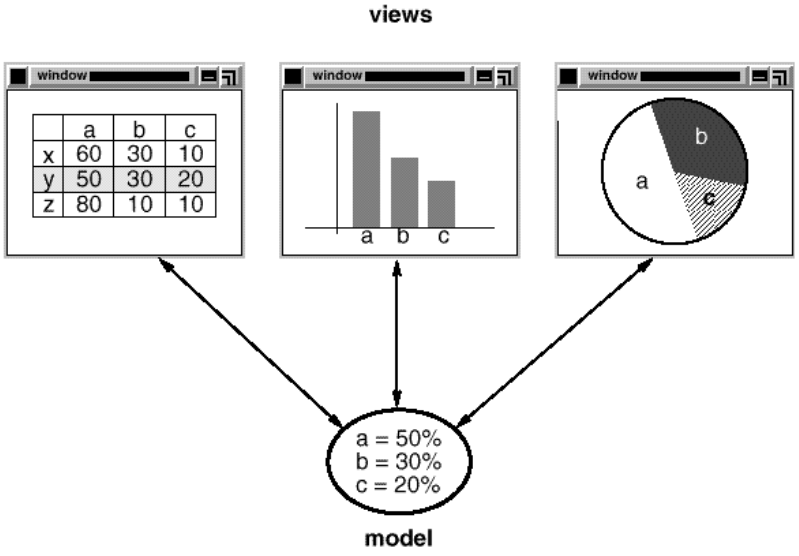
Following are some of the uses of design patterns:

- Make it easier to reuse successful designs and architectures.
- Make it easy for the new developers to design software.
- Allows us to choose different design alternatives to make the software reusable
- Helps in documenting and maintaining software.

The bottom-line of using design patterns is: ***"To get a design right faster"***.

Design Patterns in Smalltalk MVC

The Model/View/Controller set of classes are used to build user interfaces in Smalltalk-80. MVC consists of three kinds of objects. Model is the application object. View is the representation of the model and the Controller specifies how the user interface (View) reacts to the user input. Before MVC, these three objects (Model, View and Controller) used to be coupled together.



MVC decouples model and view objects using the subscribe/notify protocol. Whenever the data in the model changes, it has to notify the associated views. In response, each view gets a chance to update itself. This allows add or create views on a model without changing the model itself. In the figure we can see three views: a table display, a bar chart and a pie chart associated with the same

model. When the data in the model changes, the views are updated accordingly. This design that decouples views from models can be applied in a more general problem context like: decoupling objects where a change in one object causes change in all other associated objects. Such design pattern is the observer pattern.

Another feature of MVC is, views can be nested. For example, a control panel can contain a set of buttons as a button view along with other controls which are nested inside the main container view. In Smalltalk, nested views are supported by the class CompositeView which is a sub class of the View class. This design can be applied in a more general context like: to group a set of objects and that group can be treated as an individual object. This general design is described as the composite pattern.

MVC also lets us change how the view responds based on the user input. This is achieved through controller classes in Smalltalk. For example, we might want the view to respond based on key strokes or through mouse movement on the user interface. It is even possible to change a view's controller at run time. This design can be applied in a general context: where you have several variants of an algorithm and you want to apply a single algorithm either statically or dynamically. This View-Controller relationship is an example of Strategy pattern. A strategy is an object which represents an algorithm.

These three patterns: Observer, Composite and Strategy specify the main relationships in MVC although there are other design patterns like Factory Method and Decorator.

Describing design patterns

The information in a design pattern's documentation will be described as follows:

Pattern name and classification:
The pattern name specifies the essence of the pattern precisely. A good name is a key as it will become a part of our design vocabulary. Classification specifies the type of pattern.

Intent:
A short statement that tells us what the pattern does and which design issue or problem the pattern addresses.

Also known as:
Other well known names for the pattern, if any.

Motivation:
A scenario that illustrates a design problem and how the class and object structures solve that problem. The scenario will help us understand the abstract pattern definition.

Applicability:
This specifies in which situations a design pattern can be applied. What are the examples of poor designs that the pattern can address? How to recognize these situations?

Structure:
A graphical representation of the classes involved in the pattern following the notations of Object Modeling Technique (OMT). We also use interaction diagrams to illustrate sequences of the requests and the collaborations between objects.

Participants:
The classes and / or objects participating in the pattern and their responsibilities.

Collaborations:
How the classes and objects collaborate to carry out their responsibilities.

Consequences:
Tells us what are the costs and benefits of using the pattern. Also tells us which part of the system can be changed independently.

Implementation:
Specifies what techniques we should be aware of when implementing the pattern and also tells us if there are any language specific issues.

Sample code:
Code fragments that tell us how to implement the pattern in java.

Known uses:
Example usage of the patterns in the real world.

Related patterns:
Specifies which other patterns are related to this pattern and what are the differences between them.

Catalog of design patterns

The catalog of GOF patterns contains 23 design patterns. They are list below:

Abstract Factory:
Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Adapter:
Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge:
Decouple (separate) the abstraction from its implementation so that the two can vary independently.

Builder:
Separate (hide) the construction process from its representation so that the same construction process can create different representations.

Chain of Responsibility:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command:
Encapsulate a request as an object which allows us to parameterize the clients with different requests and support undoable operations.

Composite:
Combine objects into tree structures to form part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator:
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Façade:
Provide a uniform interface for a set of interfaces in a subsystem. Façade defines a higher level interface to make the subsystem easier to use.

Factory Method:
Defines an interface to create an object, but lets subclasses decide which class to instantiate. Factory Method lets a class differ instantiation to subclasses.

Flyweight:
Use sharing to support large numbers of fine-grained objects efficiently.

Interpreter:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator:
Provides a way to access the individual elements of an aggregate object without exposing its underlying representation.

Mediator:
Define an object that encapsulates how a set of objects interact.

Memento:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer:
Defines a one-to-many dependency between objects so that if one object changes its state, it will notify the change to all other objects and update automatically.

Prototype:
Specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Proxy:
Provide a placeholder for another object to control access to it.

Singleton:
Ensure a class has only one object and a global point of access to that object.

State:
Allow an object to change its behavior when its internal state changes. The object will appear to change its class.

Strategy:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method:
Define the skeleton of an algorithm in an operation, deferring some steps to sub classes.

Visitor:
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Organizing the catalog

Design patterns vary in their granularity and level of abstraction. As there are many design patterns, we can organize or classify patterns to learn them faster. Here we classify the patterns along two dimensions: one is purpose which reflects what a pattern does and the second one is scope which specifies whether the pattern applies to classes or objects.

Purpose / Scope		Purpose		
		Creational (5)	Structural (7)	Behavioral (11)
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Along the purpose dimension, patterns are classified into three types:
1) Creational patterns
2) Structural patterns and
3) Behavioral patterns.

The creational patterns focus on the process of object creation. The structural patterns concern is, the structure of classes and objects and the behavioral patterns focus on how the classes and objects collaborate with each other to carry out their responsibilities.

Along the scope dimension, patterns are classified into two types:
1) Class patterns and
2) Object patterns.

Class patterns deal with the relationships between classes which is through inheritance. So, these class relationships are static at compile time. Object patterns deal with object relationships which can be changed at run-time and these are dynamic. Almost all patterns use inheritance to some degree.

How to select a design pattern

As there are 23 patterns in the catalog it will be difficult to choose a design pattern. It might become hard to find the design pattern that solves our problem, especially if the catalog is new and unfamiliar to you. Below is a list of approaches we can use to choose the appropriate design pattern:

Consider how design patterns solve design problems:
Considering how design patterns help you find appropriate objects, determine object granularity, specify object interfaces and several other ways in which design patterns solve the problems will let you choose the appropriate design pattern.

Scan intent sections:
Looking at the intent section of each design pattern's specification lets us choose the appropriate design pattern.

Study how patterns interrelate:
The relationships between the patterns will direct us to choose the right patterns or group of patterns.

Study patterns of like purpose:
Each design pattern specification will conclude with a comparison of that pattern with other related patterns. This will give you an insight into the similarities and differences between patterns of like purpose.

Examine a cause of redesign:
Look at your problem and identify if there are any causes of redesign. Then look at the catalog of patterns that will help you avoid the causes of redesign.

Consider what should be variable in your design:
Consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies.

How to use a design pattern

Below steps will show you how to use a design pattern after selecting one:

Read the pattern once through for a overview:
Give importance to the applicability and consequences sections to ensure that the pattern is right for your problem.

Study the structure, participants and collaborations sections:
Make sure to understand the classes and objects in the pattern and how they relate to one another.

Look at the sample code section to see a concrete example of the pattern in action:
Studying the code helps us to implement the pattern.

Choose names for pattern participants that are meaningful in the application context:
The names in the design patterns are too abstract to be directly used in the application. Include the participant name into the name that appears in the application which makes the pattern implementation explicit. For example, if you use strategy pattern for developing UI, use UIStartegy or GridStrategy as class names.

Define the classes:
Declare their interfaces, inheritance relationships and instance variables, which represent the data and objet references. Identify the effected classes by the pattern and modify them accordingly.

Define application-specific names for the operations in the pattern:
Use the responsibilities and collaborations as a guide to name the operations. Be consistent with the naming conventions. For example you can always prefix an operation with “Create-” to denote a factory method.

Implement the operations to carry out the responsibilities and collaborations in the pattern:
The implementation section provides hints to guide us in the implementation. The examples in the sample code section can also help as well.