Software Engineering - I

An Introduction to Software Construction Techniques for Industrial Strength Software

Chapter 9 – Introduction to Design Patterns

Design Patterns

Christopher Alexander says, "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." A Pattern Language: Towns/Buildings/Construction, 1977

Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but the core of both kinds of patterns is a solution to a problem in a context.

"Description of communicating objects and classes that are customized to solve a general design in a particular context."

Patterns are devices that allow programs to share knowledge about their design. In our daily programming, we encounter many problems that have occurred, and will occur again. The question we must ask our self is how we are going to solve it *this* time. Documenting patterns is one way that you can reuse and possibly share the information that you have learned about how it is best to solve a specific program design problem.

Essay writing is usually done in a fairly well defined form, and so is documenting design patterns. The general form for documenting patterns is to define items such as:

- The motivation or context that this pattern applies to.
- Prerequisites that should be satisfied before deciding to use a pattern.
- A description of the program structure that the pattern will define.
- A list of the participants needed to complete a pattern.
- Consequences of using the pattern...both positive and negative.
- Examples!

The origin of design patterns lies in work done by an architect named *Christopher Alexander* during the late 1970s. He began by writing two books, *A Pattern Language*[Alex77] and *A Timeless Way of Building* [Alex79] which, in addition to giving examples, described his rationalle for documenting patterns.

The pattern movement became very quiet until 1987 when patterns appeared again at an OOPSLA conference. Since then, many papers and presentations have appeared, authored by people such as Grady Booch, Richard Helm, and Erich Gamma, and Kent Beck. From then until 1995, many periodicals, featured articles directly or indirectly relating to patterns. In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published Design Patterns: Elements of Reusable Object-Oriented Software [Gamma95], which has been followed by more articles in trade journals.

Software Engineering-I (CS504)

The concept of design patterns is not new as we can find a number of similar pursuits in the history of program designing and writing. For instance, Standard Template Library (STL) is a library of reusable components provided by C++ compilers. Likewise, we use algorithms in data structures that implement typical operations of manipulating data in data structures. Another, similar effort was from Peter Coad whose patterns are known for object-oriented analysis and design.

Anti-patterns is another concept that corresponds to common mistakes in analysis and design. These are identified in order to prevent potential design and analysis defects from entering into the design. Another, similar concept is object-oriented framework that is a set of cooperative classes that make up reusable design of a system. Framework dictates the architecture of the software and describes the limitations and boundaries of architecture.

With this introduction, we now describe the format that has been adopted in GoF book for describing various design patterns.

The basic template includes ten things as described below

- Works as idiom
- Name has to be meaningful
- A statement of the problem which describes its intent
- The goals and objectives it wants to reach within the given context
- Preconditions under which the problem and its solutions seem to occur
- Result or consequence
- State or configuration after the pattern has been applied
- Relevant forces and constraints and their interactions and conflicts.
- motivational scenario for the pattern.
- Static and dynamic relationships describing how to realize the pattern.
- Instructions on how to construct the work products.
- Pictures, diagrams, prose which highlight the pattern's structure, participants, and collaborations.
- One or more sample applications to illustrate
 - o a specific context
 - o how the pattern is applied

•

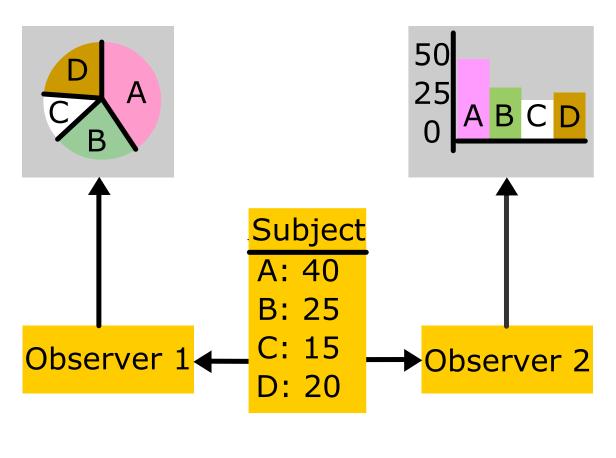
- the state or configuration after the pattern has been applied
- consequences (good and bad) of applying the pattern

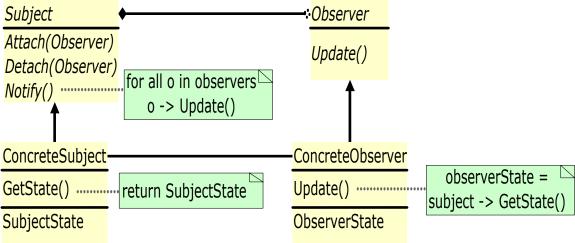
- justification of the steps or rules in the pattern
- how and why it resolves the forces to achieve the desired goals, principles, and philosophies
- how are the forces orchestrated to achieve harmony
- how does the pattern actually work
- the static and dynamic relationships between this pattern and other patterns
- to demonstrate that this is a proven solution to a recurring problem
- How to create and instantiate
- Abstract the instantiation process and make the system independent of its creational process.
- Class creational rules
- Object creational rules
- Abstract factory and factory method
- Abstract the instantiation process
- Make a system independent to its realization
- Class Creational use inheritance to vary the instantiated classes
- Object Creational delegate instantiation to an another object
- Deals with object's structure
- Class structural patterns concern the aggregation of classes to form the largest classes.
- Object structural patterns concerns the aggregation of objects to form the largest classes
- Class Structural patterns concern the aggregation of classes to form largest structures
- Object Structural pattern concern the aggregation of objects to form largest structures
- Describe the patterns of communication between classes and objects
- How objects are communicating with each other
- Behavioral class patterns
- Behavioral object patterns
- Use object composition to distribute behavior between classes
- Help in distributing object's intelligence
- Concern with algorithms and assignment of responsibilities between objects
- Describe the patterns of communication between classes or objects
- Behavioral class pattern use inheritance to distribute behavior between classes
- Behavioral object pattern use object composition to distribute behavior between classes

In the following, one pattern from each of the above mentioned categories of design patterns is explained on GoF format.

- Observer
- It is intended to define a many to many relationship between objects so that when one object changes state all its dependants are notified and updated automatically.
- Dependence/publish-subscribe mechanism in programming language
 - Smalltalk being the first pure Object Oriented language in which observer pattern was used in implementing its Model View Controller (MVC) pattern. It was a publish-subscribe mechanism in which views (GUIs) were linked with their models (containers of information) through controller objects. Therefore, whenever underlying data changes in the model objects, the controller would notify the view objects to refresh themselves and vice versa.
 - o MVC pattern was based on the observer pattern.
- It provides a common side effect of partitioning a system into a collection of cooperating classes that are in the need to maintain consistency between related objects.
- This can be used when multiple displays of state are needed.
- Optimizations to enhance display performance are impractical.

Many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.





Subject

- Knows its observers. Any number of Observer objects may observe a subject.
- Provides an interface for attaching and detaching Observer objects.

Observer

 Defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject

• Stores state of interest to concreteObserver objects.

• Sends a notification to its observers when its state changes.

ConcreteObserver

- Maintains a reference to a ConcreteSubject object.
- Stores state state that should stay consistent with the subject's.
- Implements the Observer updating interface to keep its state consistent with the subject's.
- It ensures that a class only has one instance and provides a global point of access to it.
- Singleton pattern should be used when there must be exactly one instance of a class and it must be accessible to clients from a well-known access point.
- Singleton pattern should be used when controlling the total number of instances that would be created for a particular class.
- Singleton pattern should be used when the sole instance should be extensible by sub classing and clients should be able to use an extended instance without modifying their code.

Singleton

static instance()........... SingletonOperation() GetSingletonData()

static uniqueInstance singletonData

static instance().....return uniqueInstance

Singleton

- Defines an instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
- May be responsible for creating its own unique instance.

The Singleton class is declared as

```
class Singleton { public:
```

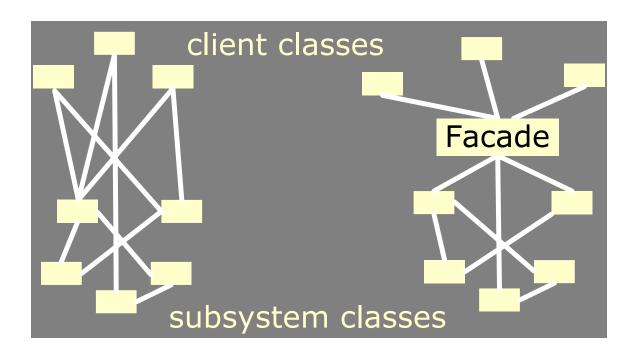
```
static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};

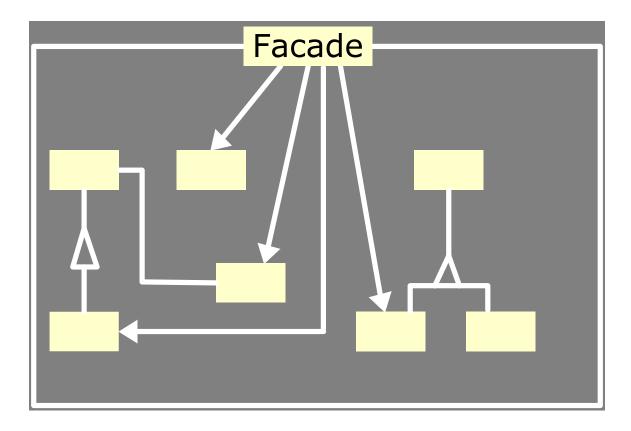
The corresponding implementation is
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance(){
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Clients access the singleton exclusively through the Instance member function. The variable _instance is initialized to 0, and the static member function Instance returns its value, initializing it with the unique instance if it is 0.

- It provides a unified interface to a set of interfaces in a sub-system.
- Façade defines a higher level interface that makes a subsystem easier to use
- You would use façade when you want to provide a simple interface to a complex sub-system.
- You would use façade pattern when there are many dependencies between clients and the implementation classes of an abstraction.
- You should introduce a façade to decouple the system from clients and other subsystems.
- You want to layer your subsystem.
- You would use façade when you want to provide a simple interface to a complex sub-system
- You would use façade pattern when there are many dependencies between clients and the implementation classes of an abstraction
- You should introduce a façade to decouple the system from clients and other subsystems
- You want to layer the subsystem

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.





Façade

- Knows which subsystem classes are responsible for a request.
- Delegates client requests to appropriate subsystem objects.

Subsystem classes

Software Engineering-I (CS504)

- Implement subsystem functionality.
- Handle work assigned by the Façade object.
- Have no knowledge of the façade, that is, they keep no references to it.