Assignment 3

DESIGN PATTERNS

Software Engineering

COMSATS University Islamabad
Sahiwal Campus

Usama Sarwar

FA17-BS(CS)-090-B

Ahmad Shaf

Software Design Methodology

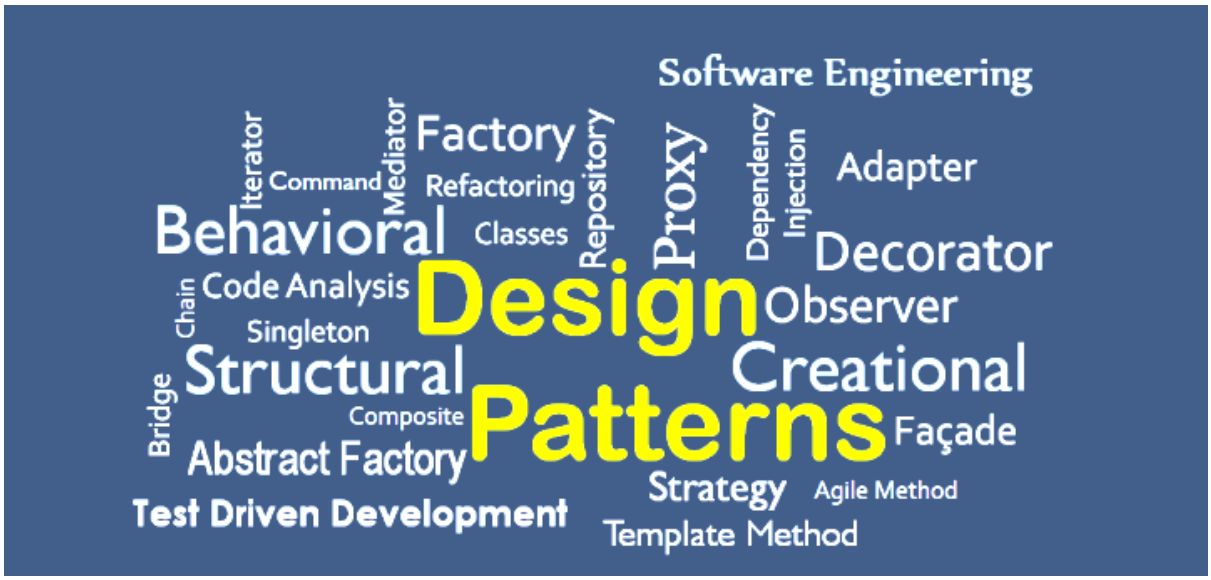
November 05, 2019

Table of Contents

List of Figures

Design Patterns in Software Engineering

A general and Reusable Solution



1. Design Patterns

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

1.1 Singleton

The singleton pattern is used to limit creation of a class to only one object. This is beneficial when one (and only one) object is needed to coordinate actions across the system. There are several examples of where only a single instance of a class should exist, including caches, thread pools, and registries.

1.1.1 Important consideration:

It's possible to subclass a singleton by making the constructor protected instead of private. This might be suitable under some circumstances. One approach taken in these scenarios is to create a **register of singletons** of the subclasses and the **getInstance** method can take in a parameter or use an environment variable to return the desired singleton. The registry then maintains a mapping of string names to singleton objects, which can be accessed as needed.

1.2 Factory Method

A normal factory produces goods; a software factory produces objects. And not just that — it does so without specifying the exact class of the object to be created. To accomplish this, objects are created by calling a factory method instead of calling a constructor.

Usually, object creation in Java takes place like so:

SomeClass someClassObject = new SomeClass();

The problem with the above approach is that the code using the **SomeClass**'s object, suddenly now becomes dependent on the concrete implementation of **SomeClass**. There's nothing wrong with using **new** to create objects but it comes with the baggage of tightly coupling our code to the concrete implementation class, which can occasionally be problematic.

1.3 Strategy

The strategy pattern allows grouping related algorithms under an abstraction, which allows switching out one algorithm or policy for another without modifying the client. Instead of directly implementing a single algorithm, the code receives runtime instructions specifying which of the group of algorithms to run.

1.4 Observer

This pattern is a one-to-many dependency between objects so that when one object changes state, all its dependents are notified. This is typically done by calling one of their methods.

For the sake of simplicity, think about what happens when you follow someone on Twitter. You are essentially asking Twitter to send you (**the observer**) tweet updates of the person (**the**

subject) you followed. The pattern consists of two actors, *the observer* who is interested in the updates and *the subject* who generates the updates.

A subject can have many observers and is a one to many relationship. However, an observer is free to subscribe to updates from other subjects too. You can subscribe to news feed from a Facebook page, which would be the subject and whenever the page has a new post, the subscriber would see the new post.

1.5 Builder

As the name implies, a builder pattern is used to build objects. Sometimes, the objects we create can be complex, made up of several sub-objects or require an elaborate construction process. The exercise of creating complex types can be simplified by using the builder pattern. A *composite* or an *aggregate* object is what a builder generally builds.

Key consideration: The builder pattern might seem similar to the 'abstract factory' pattern but one difference is that the builder pattern creates an object step by step whereas the abstract factory pattern returns the object in one go.

1.6 Adapter

This allows incompatible classes to work together by converting the interface of one class into another. Think of it as a sort of translator: when two heads of states who don't speak a common language meet, usually an interpreter sits between the two and translates the conversation, thus enabling communication. If you have two applications, with one spitting out output as XML with the other requiring JSON input, then you'll need an adapter between the two to make them work seamlessly.

1.7 State

The state pattern encapsulates the various states a machine can be in, and allows an object to alter its behavior when its internal state changes. The machine or the ***context***, as it is called in pattern-speak, can have actions taken on it that propel it into different states. Without the use of the pattern, the code becomes inflexible and littered with if-else conditionals.

2. Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software**, there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

2.1 Creational Patterns

These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

2.2 Structural Patterns

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

2.3 Behavioral Patterns

These design patterns are specifically concerned with communication between objects.

2.4 J2EE Patterns

These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

3. Software User Interface Design

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.

UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interfacing screens

UI is broadly divided into two categories:

1. Command Line Interface
2. Graphical User Interface

3.1 Command Line Interface (CLI)

CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate. CLI uses less amount of computer resource as compared to GUI.

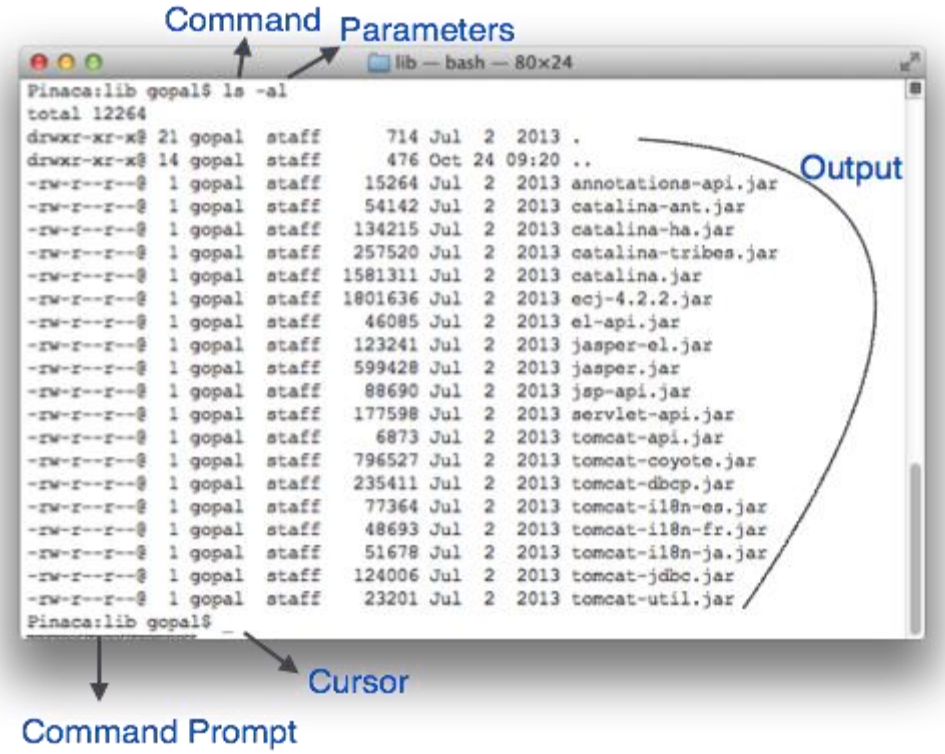


Figure 1. Command Line Interface (CLI)

3.2 Graphical User Interface

Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.

Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

3.2.1 GUI Elements

GUI provides a set of components to interact with software or hardware. Every graphical component provides a way to work with the system. A GUI system has following elements such as:

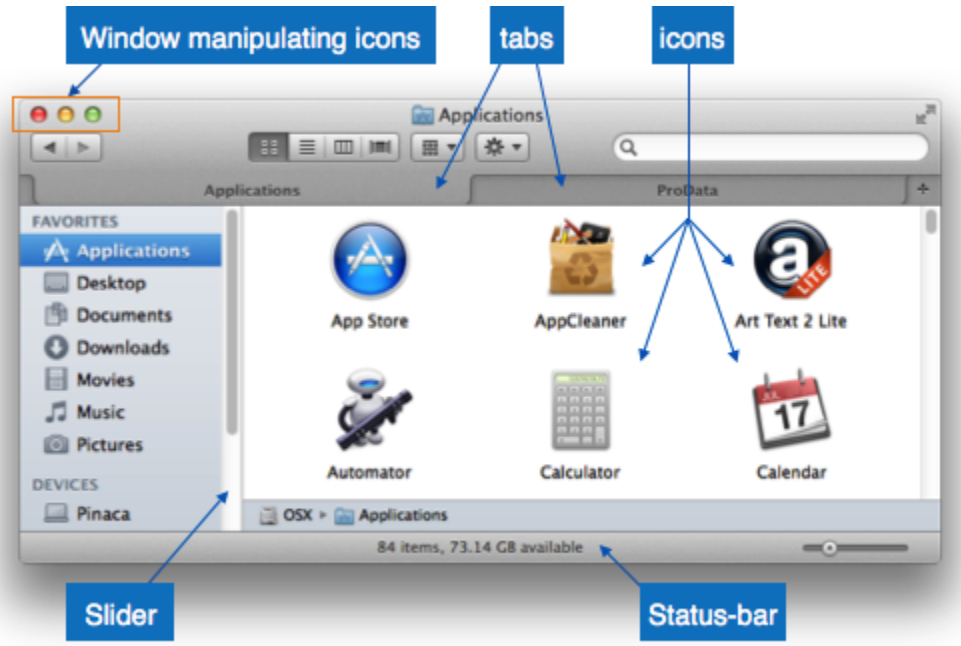


Figure 2. GUI Elements

Window: An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure

Tabs: If an application allows executing multiple instances of itself, they appear on the screen as separate windows. **Tabbed Document Interface** has come up to open multiple documents in the same window.

Menu: Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window.

Icon: An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened

Cursor: Interacting devices such as mouse, touch pad, digital pen is represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.