Agenda - Design Patterns

- What is a design pattern
- Motivation for patterns
- Pattern Categories
- Pattern Examples

Patterns Overview

- Patterns support reuse of software architecture and design.
- Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications.

Motivation for Patterns

- Developing software is hard
- Developing reusable software is even harder
- Patterns provide proven solutions
- Patterns can be reused in design

Becoming a software designer

- First learn the rules
  - Algorithms, data structures and languages
- Then learn the principles
  - Structured design, OO design
- However to truly master software design, you must study the design of masters
  - These designs have patterns to be understood, remembered and re-used

Design Pattern – example stock quote service
**Observer Pattern**

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- **Key forces**
  - There may be many observers
  - Each observer may react differently to same notification
  - Subjects should be decoupled as much as possible from the observer to allow observers to change independently.

---

**Observer Pattern – Class diagram**

- **Subject**
  - attach(Observer)
  - detach(Observer)
  - notify()

- **Observer**
  - update()

- **ConcreteSubject**
  - state
  - getStatus()
  - setStatus()
  - notify()

- **ConcreteObserver**
  - state
  - update()

---

**Observer Pattern – sequence diagram**

- **Sequence Diagram**
  - setStatus()
  - notify()
  - update()
  - getStatus()
  - update()
  - getStatus()

---

**Pattern Template**

- **Pattern Name and Classification**
  - A good, concise name for the pattern and the pattern’s type
- **Intent**
  - Short statement about what the pattern does
- **Also Known As**
  - Other names for the pattern
- **Motivation**
  - A scenario that illustrates where the pattern would be useful
- **Applicability**
  - Situations where the pattern can be used

---

**Pattern Template (cont’d)**

- **Structure**
  - A graphical representation of the pattern
- **Participants**
  - The classes and objects participating in the pattern
- **Collaborations**
  - How to do the participants interact to carry out their responsibilities?
- **Consequences**
  - What are the pros and cons of using the pattern?
- **Implementation**
  - Hints and techniques for implementing the pattern

---

**Pattern Template (cont’d)**

- **Sample Code**
  - Code fragments for a sample implementation
- **Known Uses**
  - Examples of the pattern in real systems
- **Related Patterns**
  - Other patterns that are closely related to the pattern
Pattern Types

- **Creational Patterns**
  - Deal with initializing and configuring classes and objects.
- **Structural Patterns**
  - Deal with decoupling interface and implementation of classes and objects.
- **Behavioural Patterns**
  - Deal with dynamic interactions among societies of classes and objects.

Creational Patterns

- **Singleton**
  - Factory for a singular (sole) instance
- **Factory Method**
  - Method in a derived class creates associates
- **Builder**
  - Factory for building complex objects incrementally.
- **Prototype**
  - Factory for cloning new instances from a prototype

Singleton

- **Pattern Name and Classification**
  - Singleton
- **Intent**
  - Ensure a class only has one instance, and provide a global point of access to it.
- **Motivation**
  - There are times when a class can only have one instance.

**Class Example**

```java
class Singleton {
    private static Singleton _instance = null;
    private Singleton() {
        //fill in the blank
    }

    public static Singleton getInstance() {
        if (_instance == null)
            _instance = new Singleton();
        return _instance;
    }

    public void otherOperations() {
    }
}
```

Singleton

- **Applicability**
  - There must be only one instance of a class, and it must be accessible to clients from a well-known access point
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

```java
class TestSingleton {
    public void method1() {
        X = Singleton.getInstance();
    }

    public void method2() {
        Y = Singleton.getInstance();
    }
}
```
**Structural Patterns**

- **Adapter**
  - Translator adapts a server interface for a client
- **Bridge**
  - Abstraction for binding one of many implementations
- **Composite**
  - Structure for building recursive aggregations
- **Decorator**
  - Decorator extends an object transparently

**Bridge**

- **Pattern Name and Classification**
  - Bridge
- **Intent**
  - Decoupling the interface from implementation
- **Motivation**
  - Used to hide the implementation from the client. Avoid permanent binding between the client and implementation.

**Bridge**

- **Applicability**
  - Avoid a permanent binding between an abstraction and its implementation
  - Both the abstractions and their implementations should be independently extensible by subclassing
  - Changes in the implementation of an abstraction should have no impact on the clients; that is, their code should not have to be recompiled
  - You want to hide the implementation of an abstraction completely from clients (users)

**Behavioural Patterns**

- **State**
  - An object whose behaviour depends on state
- **Observer**
  - Dependents update automatically when a subject changes
- **Iterator**
  - Aggregate elements are accessed sequentially

**When to use patterns**

- **Solutions to problems that recur with variations.**
  - No need to reuse if the problem occurs only once.
- **Solutions that require several steps.**
  - Not all problems need all steps
  - Patterns can be an overkill if problems have simple solutions.