Introduction

Design patterns are solutions to recurring problems; **guidelines on how to tackle certain problems**. They are not classes, packages or libraries that you can plug into your application and wait for the magic to happen. These are, rather, guidelines on how to tackle certain problems in certain situations.

> Design patterns are solutions to recurring problems; guidelines on how to tackle certain problems

Wikipedia describes them as

> 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.

 Be Careful

- Design patterns are not a silver bullet to all your problems.
- Do not try to force them; bad things are supposed to happen, if done so.
- Keep in mind that design patterns are solutions **to** problems, not solutions **finding** problems; so don't overthink.
- If used in a correct place in a correct manner, they can prove to be a savior; or else they can result in a horrible mess of a code.

Types of Design Patterns

- Creational
- Structural
- Behavioral

Creational Design Patterns

In plain words

> Creational patterns are focused towards how to instantiate an object or group of related objects.

Wikipedia says

> In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural Design Patterns

In plain words

> Structural patterns are mostly concerned with object composition or in other words how the entities can use each other. Or yet another explanation would be, they help in answering "How to build a software component?"

Wikipedia says

> In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Design Patterns

In plain words

> It is concerned with assignment of responsibilities between the objects. What makes them different from structural patterns is they don't just specify the structure but also outline the patterns for message passing/communication between them. Or in other words, they assist in answering "How to run a behavior in software component?"

Wikipedia says

> In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

- Chain of Responsibility
- Command
- Iterator

- Mediator
- Memento
- Observer
- Visitor
- Strategy
- State
- Template Method

Designing a Java/J2EE application

Design should be specific to a problem but also should be general enough to address future requirements. Designing reusable object oriented software involves decomposing the business use cases into relevant objects and converting objects into classes.

Create a tiered architecture

Client tier, business tier and data tier. Each tier can be further logically divided into layers. Use MVC (Model View Controller) architecture for the J2EE and Java based GUI applications.

Create a data model

A data model is a detailed specification of data oriented structures. This is different from the class modeling because it focuses solely on data whereas class models allow you to define both data and behavior. Conceptual data models (aka domain models) are used to explore domain concepts with project stakeholders. Logical data models are used to explore the domain concepts, and their relationships. Logical data models depict entity types, data attributes and entity relationships (with Entity Relationship (ER) diagrams). Physical data models are used to design the internal schema of a database depicting the tables, columns, and the relationships between the tables. Data models can be created by performing the tasks:

- Identify entity types, attributes and relationships: use entity relationship (E-R) diagrams.
- Apply naming conventions (e.g. for tables, attributes, indices, constraints etc): Your organization should have standards and guidelines applicable to data modeling.
- Assign keys: surrogate keys (e.g. assigned by the database like Oracle sequences, Sybase identity columns, max()+1, universally unique identifiers UUIDs, etc), natural keys (e.g. Tax File Numbers, Social Security Numbers etc), and composite keys.
- Normalize to reduce data redundancy and denormalize to improve performance: Normalized data have the advantage of information being stored in one place only, reducing the possibility of inconsistent data. Furthermore, highly normalized data are loosely coupled. But normalization comes at a performance cost because to determine a piece of information you have to join multiple tables whereas in a denormalized approach the same piece of information can be retrieved from a single row of a table. Denormalization should be used only when performance testing shows that you need to improve database access time for some of your tables.

Create a design model

A design model is a detailed specification of the objects and relationships between the objects as well as their behavior.

- Class diagram: contains the implementation view of the entities in the design model. The design model also contains core business classes and non-core business classes like persistent storage, security management, utility classes etc. The class diagrams also describe the structural relationships between the objects.
- Use case realizations: are described in sequence and collaboration diagrams.

Design considerations when decomposing business use cases into relevant classes

Designing reusable and flexible design models requires the considerations:

- Granularity of the objects (fine-grained versus coarse-grained): Can we minimize the network trip by passing a coarse-grained value object instead of making 4 network trips with fine-grained parameters?. Should we use method level (coarse-grained) or code level (fine-grained) thread synchronization?. Should we use a page level access security or a fine-grained programmatic security?
- Coupling between objects (loosely coupled versus tightly coupled). Should we use business delegate pattern to loosely couple client and business tier?Should we use dependency injection (e.g. using Spring) or factory design pattern to loosely couple the caller from the callee?
- Network overheads for remote objects like EJB, RMI etc: Should we use the session façade, value object patterns?
- Definition of class interfaces and inheritance hierarchy: Should we use an abstract class or an interface? Is there any common functionality that we can move to the super class (i.e. parent class)? Should we use interface inheritance with object composition for code reuse as opposed to implementation inheritance? Etc.
- Establishing key relationships (aggregation, composition, association etc): Should we use aggregation or composition? (composition may require cascade delete). Should we use an “is a” (generalization) relationship or a “has a” (composition) relationship?
- Applying polymorphism and encapsulation: Should we hide the member variables to improve integrity and security?. Can we get a polymorphic behavior so that we can easily add new classes in the future?.
- Applying well-proven design patterns (like Gang of four design patterns, J2EE design patterns, EJB design patterns etc) help designers to base new designs on prior experience. Design patterns also help you to choose design alternatives.
- Scalability of the system: Vertical scaling is achieved by increasing the number of servers running on a single machine. Horizontal scaling is achieved by increasing the number of machines in the cluster. Horizontal scaling is more reliable than the vertical scaling because there are multiple machines involved in the cluster. In vertical scaling the number of server instances that can be run on one machine are determined by the CPU usage and the JVM heap memory.

- How do we replicate the session state? Should we use stateful session beans or HTTP session? Should we serialize this object so that it can be replicated?
- Internationalization requirements for multi-language support: Should we support other languages? Should we support multi-byte characters in the database?

Vertical slicing

Getting the reusable and flexible design the first time is impossible. By developing the initial vertical slice of your design you eliminate any nasty integration issues later in your project. Also get the design patterns right early on by building the vertical slice. It will give you experience with what does work and what does not work with Java/J2EE. Once you are happy with the initial vertical slice then you can apply it across the application. The initial vertical slice should be based on a typical business use case.

Ensure the system is configurable

Ensure the system is configurable through property files, xml descriptor files, and annotations. This will improve flexibility and maintainability. Avoid hard coding any values. Use a constant class and/or enums for values, which rarely change and use property files (e.g. MyApp.properties file containing name/value pairs), xml descriptor files and/or annotations for values, which can change more frequently like application process flow steps etc. Use property (e.g. MyApp.properties) or xml (e.g. MyApp.xml) files for environment related configurations like server name, server port number, LDAP server location etc.

Design considerations during design, development and deployment phases

Designing a fast, secured, reliable, robust, reusable and flexible system require considerations in the key areas:

- Performance issues (network overheads, quality of the code etc): Can I make a single coarse-grained network call to my remote object instead of 3 fine-grained calls?
- Concurrency issues (multi-threading): What if two threads access my object simultaneously will it corrupt the state of my object?
- Transactional issues (ACID properties): What if two clients access the same data simultaneously? What if one part of the transaction fails, do we rollback the whole transaction? Do we need a distributed (i.e. JTA) transaction?. What if the client resubmits the same transactional page again?.
- Security issues: Are there any potential security holes for SQL injection or URL injection by hackers?
- Memory issues: Is there any potential memory leak problems? Have we allocated enough heap size for the JVM? Have we got enough perm space allocated since we are using 3rd party libraries, which generate classes dynamically? (e.g. JAXB, XSLT, JasperReports etc).
- Scalability issues: Will this application scale vertically and horizontally if the load increases? Should this object be serializable? Does this object get stored in the HttpSession?

- Maintainability, reuse, extensibility etc: How can we make the software reusable, maintainable and extensible? What design patterns can we use? How often do we have to refactor our code?
- Logging and auditing if something goes wrong can we look at the logs to determine the root cause of the problem?
- Object life cycles: Can the objects within the server be created, destroyed, activated or passivated depending on the memory usage on the server? (e.g. EJB).
- Resource pooling: Creating and destroying valuable resources like database connections, threads etc can be expensive. So if a client is not using a resource can it be returned to a pool to be reused when other clients connect? What is the optimum pool size?
- Caching: can we save network trips by storing the data in the server's memory? How often do we have to clear the cache to prevent the in memory data from becoming stale?
- Load balancing: Can we redirect the users to a server with the lightest load if the other server is overloaded?
- Transparent fail over: If one server crashes can the clients be routed to another server without any interruptions?
- Clustering: What if the server maintains a state when it crashes? Is this state replicated across the other servers?
- Back-end integration: How do we connect to the databases and/or legacy systems?
- Clean shutdown: Can we shut down the server without affecting the clients who are currently using the system?
- Systems management: In the event of a catastrophic system failure who is monitoring the system? Any alerts or alarms? Should we use JMX? Should we use any performance monitoring tools like Tivoli?
- Dynamic redeployment: How do we perform the software deployment while the site is running? (Mainly for mission critical applications 24hrs X 7days).
- Portability issues: Can I port this application to a different server 2 years from now?

Identifying performance and/or memory issues in your Java/J2EE application

Profiling can be used to identify any performance issues or memory leaks. Profiling can identify what lines of code the program is spending the most time in? What call or invocation paths are used to reach at these lines? What kinds of objects are sitting in the heap? Where is the memory leak? Etc.

- There are many tools available for the optimization of Java code like JProfiler, Borland OptimizeIt etc. These tools are very powerful and easy to use. They also produce various reports with graphs. Optimizeit Request Analyzer provides advanced profiling techniques that allow developers to analyze the performance behavior of code across J2EE application tiers. Developers can efficiently prioritize the performance of Web requests, JDBC, JMS, JNDI, JSP, RMI, and EJB so that trouble spots can be proactively isolated earlier in the development lifecycle. Thread Debugger tools can be used to identify threading issues like thread starvation and contention issues that can lead to system crash. Code coverage tools can assist developers with identifying and removing any dead code from the applications.

- Hprof which comes with JDK for free Simple tool.

```
Java –Xprof myClass
java -Xrunhprof:[help]|[<option>=<value>]
java -Xrunhprof:cpu=samples, depth=6, heap=sites
```

- Use operating system process monitors like NT/XP Task Manager on PCs and commands like ps, iostat, netstat, vmstat, uptime, nfsstat etc on UNIX machines.
- Write your own wrapper MemoryLogger and/or PerformanceLogger utility classes with the help of totalMemory() and freeMemory() methods in the Java Runtime class for memory usage and System.currentTimeMillis() method for performance. You can place these MemoryLogger and PerformanceLogger calls strategically in your code. Even better approach than utility classes is using Aspect Oriented Programming (AOP – e.g. Spring AOP Refer Q3 – Q5 in Emerging Technologies/Frameworks section) or dynamic proxies (Refer proxy design pattern in Q11 in How would you go about...? section) for pre and post memory and/or performance recording where you have the control of activating memory/performance measurement only when needed.