Threading

Various problems that might arise in threaded applications.

In computing applications, the producer and consumer problem also known as the bounded/buffer problem.

It is old example of a multi-process synchronization problem. It creates two problem processes, the producer and the consumer, which share a general, fixed-size buffer used as a queue.

The producer works to create data, put it into the buffer, and start again.

- At the same time, the consumer consumes the data (removing it from the buffer), one piece at a time.

Two types of problems arise when multiple threads try to read and write shared data concurrently -

- Thread interference errors.
- Memory consistency errors.

In addition, the threading problems occurs in system calls (fork (), exec ()), cancellation asynchronous, deferred) of thread, signal handling, thread pool and thread-specific data.

Major Problems that might arise in threaded applications:

- Increased Complexity.
- Complications due to Concurrency.
- Difficult to Identify Errors.
- Testing Complications.
- Unpredictable results.
- Complications for Porting Existing Code.

Design Patterns

Describe a design pattern you have used recently and its application.

In a software industry, design pattern are typical solutions to generally happening problems/tasks in software design. We can create pre-made blueprints that we can customize to solve a recurring design problem in our code. The pattern is not a specific piece of code, but a common concept for solving a certain problem. We can follow the pattern details and implement a solution that suits the realities of our own program or application.

Patterns are sometimes confused with algorithms, because both concepts define typical solutions to some known problems. Whereas, an algorithm always describes a clear set of actions that can achieve some goal. The pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different. Algorithm is a cooking recipe, both have clear steps to achieve a goal. Whereas, pattern is more like a blueprint, we can see what the outcome and its features are. The pattern consists of the following:

Intent – The pattern shortly describes both the problem and the solution.
Motivation – It more describes the problem and the solution the pattern makes possible.
Structure – It describes classes that displays each phase of the pattern and how it is related.
Code example – It describes one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Design Patterns Applications

- Design patterns help in evaluating the solution of a complex problem.
- We can develop our code loosely-coupled.
- Using design patterns, we have the option of reusable code, which reduce the total development cost of the application.
- Moreover, future developers feel the code more user-friendly.
- It has all the standard approaches to figure out solutions to the general problem of a software.
- We can utilize the same pattern repeatedly in multiple projects or applications.
- It also helps in refactoring our code in a better way.

Three types of design patterns

1. **Creational design patterns**, it provides solution to instantiate an object in the best possible way for specific conditions. Creational design patterns are including:

- ***Singleton pattern***, a class of which only a single instance can exist
- **Factory pattern**, creates an instance of several derived classes
- **Abstract factory pattern**, creates an instance of several families of classes
- **Builder pattern**, separates object construction form its representation
- **Prototype pattern**, a fully initialized instance to be copied or clone.

2. **Structural design patterns**, it provides different ways to generate a class structure, for instance using inheritance and composition to generate a large object from small objects. Structural design patterns are including:

- **Adapter pattern**, match interfaces of different classes
- **Composite pattern**, a tree structure of simple and composite objects
- **Proxy pattern**, an object representing another object
- **Flyweight pattern**, a fine-grained instance used for efficient sharing
- **Façade pattern**, a single class that represent an entire subsystem
- **Bridge pattern**, separates an objects interface from its implementation
- **Decorator pattern**, add responsibilities to objects dynamically

3. **Behavioral Design Patterns,** it provides solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily. Behavioral design patterns are including:

- **Template Method Pattern,** defer the exact steps of an algorithm to a subclass
- **Mediator Pattern**, defines simplified communication between classes
- **Chain of Responsibility Pattern,** a way of passing a request between chain of objects
- **Observer Pattern**, a way of notifying change to a number of classes
- **Strategy Pattern**, encapsulates an algorithm inside a class
- **Command Pattern,** encapsulate a command request as an object
- **State Pattern**, alter an objects behavior when its state changes
- **Visitor Pattern**, defines a new operation to a class without change
- **Iterator Pattern**, sequentially access the elements of a collection
- **Interpreter Pattern**, a way to include language element in a program
- **Memento Pattern**, capture and restore an objects internal state

Describe the Abstract Factory design pattern.

Abstract Factory is a **creational design pattern**, which solves the problem of generating of product things or families without specifying their concrete classes.

It creates an interface for making all distinct products but leaves the actual product development to concreate factory classes. Each factory corresponds to a certain product diversity.

The client code calls the development methods of a factory object instead of generating products directly with a constructor call (new operator). Since a factory corresponds to single product variant, all its products will be compatible.

The client code defines with factories and products only through their abstract interfaces. The client code works with any product variants, generated by the factory object. Also, we can make a new concrete factory class and pass it to the client code. Examples of core java libraries:

- Javax.xml.parsers.DocumentBuilderFactory#newInstance()
- Javax.xml.transform.TransformerFactory#newInstance()
- Javax.xml.xpath.XPathFactory#newInstance()

Describe the Decorator pattern.

Wrapper also known as decorator is a **Structural pattern** that allows adding new behaviors to objects dynamically by placing it inside special wrapper objects.

The decorator is nice standard in java code, particularly in code related to streams. Examples of decorator in core java libraries are:

- All subclasses of java.io.InputStream, OutputSream, Reader and Writer have constructors that accept objects of their own type.
- Java.util.Collections, methods checkedXXX(), synchronizedXXX() and unmodifiableXXX().
- Javax.servlet.http.HttServletRequestWrapper and HttpSerletResponseWrapper