DAVE JHARED G. PADUADA
BSCpE - III - GF
CPE - 18 - Final Term Assignment

1. Define and discuss the following:

a. Software Design Pattern

- are like ready-made templates or blueprints that help solve common problems faced when designing software. Imagine you're building a house, you could figure out every detail from scratch, but it's easier to use proven ideas like placing windows for light or designing a roof that doesn't leak. Similarly, in software development, programmers often run into the same issues, like managing object creation, organizing code for flexibility, or ensuring components communicate smoothly. Design patterns offer standardized, tested solutions to these recurring problems. They aren't chunks of code you copy-paste but general strategies or best practices that you adapt to fit your specific situation. For example, the "Singleton" pattern ensures only one instance of a class exists, w/c is handy for things like database connections, while the "Observer" pattern helps objects notify each other about changes, like updating a user interfaces when data changes.

b. Four Essential Features of Design Pattern

- Design patterns - as defined by the Gang of Four are structured around four essential features that make them practical and effective tools for solving recurring software design problems. These features: Pattern Name, Problem, Solution, & Consequences, acts like a "user manual" for understanding and applying patterns correctly.

i. Pattern Name:

+ This is the label given to a pattern, like Singleton or Observer. The name acts a shared vocabulary for developers. Instead of explaining a complex idea from scratch, you can simply say, "Let's use the Factory pattern here," and others instantly grasp the general approach. Names make communication faster and help teams discuss solutions without getting lost in technical jargon. For example, saying "We need a Decorator" is quicker than explaining, "Let's dynamically add responsibilities to an object without subclassing."

ii. Problem:

+ This describes when to use the pattern. It outlines the specific design challenge the pattern addresses, along with the context where it applies. For instance, the Adapter pattern solves the problem of making incompatible interfaces work together, like connecting a new payment gateway to an old system. The "problem" sections helps you recognize situations where a pattern is useful, such as needing to decouple (using the Observer pattern) or manage object creation flexibly (using the Factory pattern).

iii. Solution:

+ This explains how the pattern solves the problem. It's not a step-by-step code recipe but a general blueprint of the structure, relationships, & responsibilities of the classes or objects involved. For example, the strategy pattern suggests creating interchangeable algorithms (like sorting methods) that can be swapped at runtime. The solution focuses on the "big picture" of the design, such as how classes interact or how data flows, so you can adapt it to your code w/out rigid rules.

iv.: Consequences:

+ Every design choice has tradeoffs, and this section highlights the pros and cons of using the pattern. For example, the Singleton pattern ensures only one instance exists (good for saving memory) but can make testing harder due to global state. Consequences might include performance costs, increased flexibility, or reduced code duplication. Understanding these helps you decide if a pattern is worth using. For instance, the Composite pattern simplifies working with tree-like structures but might add complexity for simple cases.

c. Three different major classes of patterns and it's 23 classic design patterns that fall under it.

i.: Creational Patterns - These Patterns focus on how objects are created, providing flexibility and reusability in object instantiation. Instead of hard-coding object creation, they decouple the process, making systems more adaptable.

* Abstract Factory - Creates families of related objects w/out specifying their concrete classes.
* Builder - Constructs complex objects step by step, separating construction from representation.
* Factory Method - Lets subclasses decide w/c class to instatiate (e.g., a logistics app where a "Ship" subclass creates ships, while a "Truck" subclass creates trucks).
* Prototype - Clones existing objects to create new ones, avoiding costly initialization.
* Singleton - Ensures only one instance of a class exists.

Why they matter: Creational patterns reduce dependencies in code, making it easier to swap components or scale systems w/out rewriting logic.

ii.: Structural Patterns - These patterns deal with how classes and objects are organized into larger structures, simplifying interactions between components. They focus on composition and relationships to ensure parts work together seamlessly.

* Adapter - Bridges incompatible interfaces (e.g., converting data from an XML format to JSON for a modern API).
* Bridge - Separates an abstraction from it's implementation (e.g., splitting a drawing tool's interface from it's rendering engine, so it works on both desktop and mobile).
* Composite - Treats individual objects & groups uniformly (e.g., managing nested folders and files in a file system as a single structure).
* Decorator - Adds responsibilities to objects dynamically (e.g., adding encryption or compression features to a file uploader w/out altering it's core code).
* Façade - Provides a simplified interface to a complex subsystem (e.g., a one-click "order" button that handles payment, inventory, & shipping behind the scenes).
* Flyweight - Shares a small, reusable objects to save memory (e.g., reusing identical character styles in a word processor instead of creating new ones).
* Proxy - Controls access to an object (e.g., a placeholder for a large image that loads only when needed).

Why they matter: Structural patterns help manage complexity by organizing code into logical, reusable structures, improving readability and performance.

iii.: Behavioral Patterns - these patterns define how objects communicate & share responsibilities, ensuring flexible and efficient interactions. They focus on algorithms, delegation, and managing workflows.

* Chain of responsibility - Passes requests through a chain of handlers (e.g., a customer support ticket moving from junior to senior staff until resolved).
* Command - Encapsulates requests as objects (e.g., undo/redo actions in a text editor.
* Interpreter - Implements a language grammar (e.g., evaluating simple math expressions like "5+3").
* Iterator - Accesses elements of a collection sequentially (e.g. looping through a playlist without exposing it's internal structure).
* Mediator - Reduces direct communication between objects by introducing a middleman.
* Memento - Saves and restores an object's state.
* Observer - Notifies objects of changes
* State - Allows an object to alter behavior when its internal state changes.
* Strategy - Swaps algorithms at runtime
* Template Method - Defines a skeleton of an algorithm in a base class.
* Visitor - Adds operations to objects without changing their classes.

Why this matter: Behavioral patterns streamline collaboration between objects, making systems more modular & easier to extend.