CPE18 - Summary Finals

Chapter 11 Design Patterns

Design Patterns	are a set of rules and templates for code that you accumulate as you gain experience writing programs. They are solutions to recurring design problems. Learning them makes code more uniform, readable, and improves its quality over time (reusable). **Patterns -** deal with object creation, composition of classes or objects, or the ways in which objects interact and distribute responsibilities in the program.
Christopher Alexander	An architect who defined patterns for design in architecture, influencing software design patterns. He emphasized three key elements:
Recurring	The problem that **evokes** the design pattern must be a **common one**.
Core Solution	The pattern provides a **template for solution**, extracting it's essence.
Reuse	The pattern must be **easily reusable** in different domains.

Model-View-Controller(MVC)	One of the earliest published software design patterns, used with graphical user interfaces. It divides a program into three parts:
Model	Contains the processing rules of the program.
View	Presents the data and interface to the user.
Controller	Mediates communication between the Model and the View.

Gang of Four	Refers to **Gamma**, **Helm**, **Johnson**, and **Vlissides**, authors of the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software*. They define a design pattern as something that "names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design". **Four Essential Features:**
Pattern Name	A handle to describe a design problem, its solution, and consequences in a word or two, increasing design vocabulary.
Problem	Describes when to use the pattern, explaining the **problem and its context**.
Solution	Describes the elements that make up the design, their relationships, responsibilities, and collaborations, providing an abstract description of how a general arrangement of elements solves a problem.
Consequences	The **results and tradeoffs** of applying the pattern to a problem, including time and space tradeoffs, flexibility, extensibility, and portability.

Classification Criteria	
Scope - **Deals with relationships** between classes and objects. - Static relationships between classes are fixed at compile time - Dynamic relationships apply to objects and these relationships can change at run-time.	Purpose - deals with what the **pattern does** with respect to classes and objects. Patterns can deal with object **creation, composition** of classes or objects, or the ways in which objects **interact and distribute responsibilities** in the program.

Classic Design Patterns (GoF 23 Patterns)

Creational	Structural	Behavioral
Abstract Factory	**Adapter**	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	**Iterator**
Singleton	**Façade**	Mediator
	Flyweight	Memento
	Proxy	**Observer**
		State
		Strategy
		Template Method
		Visitor

Creational – object pattern creations; deals with how objects are **created; creating objects**

Structural – Object composition and relationships; describe how objects are **composed** into larger groups

Behavioral – object interaction and responsibility distribution; how **responsibilities are distributed** in the design

Creational Patterns – all have to do with **creating objects**	
The **Singleton** Pattern	easiest of the design patterns; you need **1 & only 1 instance of a class**.
The **Factory Method** Pattern	• Creates objects for you. Says that you **just define an interface or abstract class** for creating an object but let the sub-class themselves decide **which class to instantiate.** • Sub-classes – are responsible for creating **the instance of the class.** • **Promotes loose coupling** by eliminating the need to bind application-specific classes into the client code. • Client code interacts solely with the **resultant interface or abstract class**. • In order to test our factory, we create a **driver**.
Structural Patterns – Structural patterns help you **put objects together** so you can use them more easily. Remember, composition, aggregation, delegation, and inheritance are all about structure & coordination.	
The **Adapter** Pattern	2 Ways to implement adapters: • **Class adapters –** where adapter will **inherit from the target class** • **Object adapters –** that use **delegation** to create the adapter.
The **Façade** pattern	For a second example, we try **to simplify interfaces**. Say you have a set of classes that constitute a **subsystem**. They could be individual classes that make up a more complex system or part of a large class library.

Behavioral Pattern – creational patterns are all about how to **create objects,** and structural patterns are all about getting **objects to communicate and cooperate,** behavioral patterns are all about **getting objects to do things**. They examine how responsibilities are distributed in the design and how communication happens between objects.

The **Iterator** pattern	If you have a collection of elements, you can organize them in many different ways. They can be **arrays, linked lists, queues, hash tables, sets, and so on**. **traverse the entire collection** from beginning to end, one element at a time.
The **Observer** pattern	defines a **one-to-many dependency** between objects so that when one object changes state, all of its dependents are notified and updated automatically. With the Observer pattern, we need a **Subject interface** so that the **Subject** and the **Observer** and the **Client** all can tell what the **state interface** they're using is. We also need an Observer interface that just tells us **how to update an Observer**.
The **Strategy** Pattern	Sometimes you have an application where you have several ways of doing a single operation or you have several different behaviors, each with a different interface. One of the ways to implement something like this is using a **switch statement**, like so**.** The problem with this type of construct it that **if you add another behavior,** you need to **change this code** and potentially **all the other code** that has to select different behaviors. This is not good. The Strategy design pattern gets you around this. You need to select from **dynamically**, you should make sure they all **adhere to the same interface**—a Strategy interface—and then that they're selected dynamically via a driver, called the **Context,** that is told which to call by the client software.