# DESIGN PATTERN (CST324)
# UNIT 1
# THEORY QUESTION BANK

**What is a design pattern?**
Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".
In object-oriented design, patterns offer solutions to problems using objects and interfaces, analogous to Alexander's use of patterns in architecture (walls, doors, etc.).
**Essential Elements of a Pattern**
a. Pattern Name:
- A pattern name acts as a handle to describe a design problem, its solution, and its consequences.
- It enhances the design vocabulary, allowing designers to work at a higher level of abstraction and communicate more effectively.
- Naming patterns is crucial, as it simplifies thinking and discussing designs.

b. Problem:
- This defines when to apply the pattern, explaining the context and specific design problems.
- It might describe common issues such as how to structure objects or algorithms.
- The problem can also include conditions that must be fulfilled to apply the pattern effectively.

c. Solution:
- The solution explains the components, their roles, and their interactions.
- Rather than prescribing a specific design or implementation, a pattern provides a general template that can be adapted to different situations.
- It focuses on abstract relationships between classes and objects to solve the problem.

d. Consequences:
- These are the results and trade-offs of applying the pattern.
- Consequences are essential for evaluating design alternatives, including space-time efficiency, language constraints, and the flexibility or extensibility of the system.
- Listing the consequences helps in understanding the impact of the pattern on system architecture.

These four elements help designers choose the appropriate pattern, apply it effectively, and evaluate its implications.

**Describing Design Patterns**
Design patterns are described using a uniform template to make them easier to learn, compare, and apply.
Sections of the Design Pattern Template:

a. Pattern Name and Classification:
- The name succinctly conveys the essence of the pattern and becomes part of the design vocabulary.
- The classification organizes the pattern within the broader scheme of design patterns.

b. Intent:
- A brief statement that explains what the pattern does, its rationale, and the design issue it addresses.

c. Also Known As:
- Lists other names the pattern might be known by.

d. Motivation:
- A real-world scenario demonstrating the design problem and how the pattern solves it.
- Helps in understanding the abstract description that follows.

e. Applicability:
- Describes situations where the pattern can be applied.
- Highlights poor design examples that the pattern addresses and how to recognize these situations.

f. Structure:
- Graphical representations of the classes involved, usually in the form of class diagrams (based on OMT) or interaction diagrams to show object collaborations.

g. Participants:
- Lists the classes or objects involved in the pattern and their responsibilities.

h. Collaborations:
- Describes how the participants collaborate to achieve the pattern's objectives.

i. Consequences:
- Discusses how the pattern supports its objectives, and the trade-offs involved in using it.
- Explains which aspects of the system structure can be varied independently through the pattern.

j. Implementation:
- Provides implementation tips, common pitfalls, and any language-specific considerations.

k. Sample Code:
- Code snippets, typically in languages like C++ or Smalltalk, to illustrate how to implement the pattern.

l. Known Uses:
- Real-world examples where the pattern is used, typically from different domains.

m. Related Patterns:
- Discusses related design patterns, highlighting differences and suggesting when they can be used together.

This consistent format helps in understanding, applying, and comparing design patterns systematically.

**The Catalog of Design Patterns**

The catalog beginning on page 79 contains 23 design patterns. Their names and intents are listed next to give you an overview. The number in parentheses after each pattern name gives the page number for the pattern (a convention we follow throughout the book).

Abstract Factory (87)
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Adapter (139)
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge (151)
Decouple an abstraction from its implementation so that the two can vary independently.

Builder (97)

Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Chain of Responsibility (223)
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command (233)
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Composite (163)
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator (175)
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade (185)
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Factory Method (107)
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Flyweight (195)
Use sharing to support large numbers of fine-grained objects efficiently.
Interpreter (243) 20
Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator (257)
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator (273)
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento (283)
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer (293)
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Prototype (117)
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Proxy (207)
Provide a surrogate or placeholder for another object to control access to it.
Singleton (127)
Ensure a class only has one instance, and provide a global point of access to it.
State (305)

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy (315)
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template Method (325)
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor (331)
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Organizing the Design Patterns Catalog**

1. Classification of Design Patterns:
   - Granularity and Abstraction: Design patterns vary in their level of detail and abstraction, necessitating an organized classification system to make them easier to learn and apply.
   - Purpose and Scope: Design patterns are classified based on:
     - Purpose: Defines what a pattern does.
       - Creational: Concerned with object creation.
       - Structural: Focus on the composition of classes or objects.
       - Behavioral: Describe interactions and responsibilities among objects or classes.
     - Scope: Specifies whether the pattern applies to:
       - Class patterns: Deal with static relationships between classes and their subclasses, typically established through inheritance.
       - Object patterns: Handle dynamic object relationships that can be modified at runtime.

2. Creational Patterns:
   - Class Patterns: Defer object creation to subclasses.
   - Object Patterns: Delegate object creation to another object.

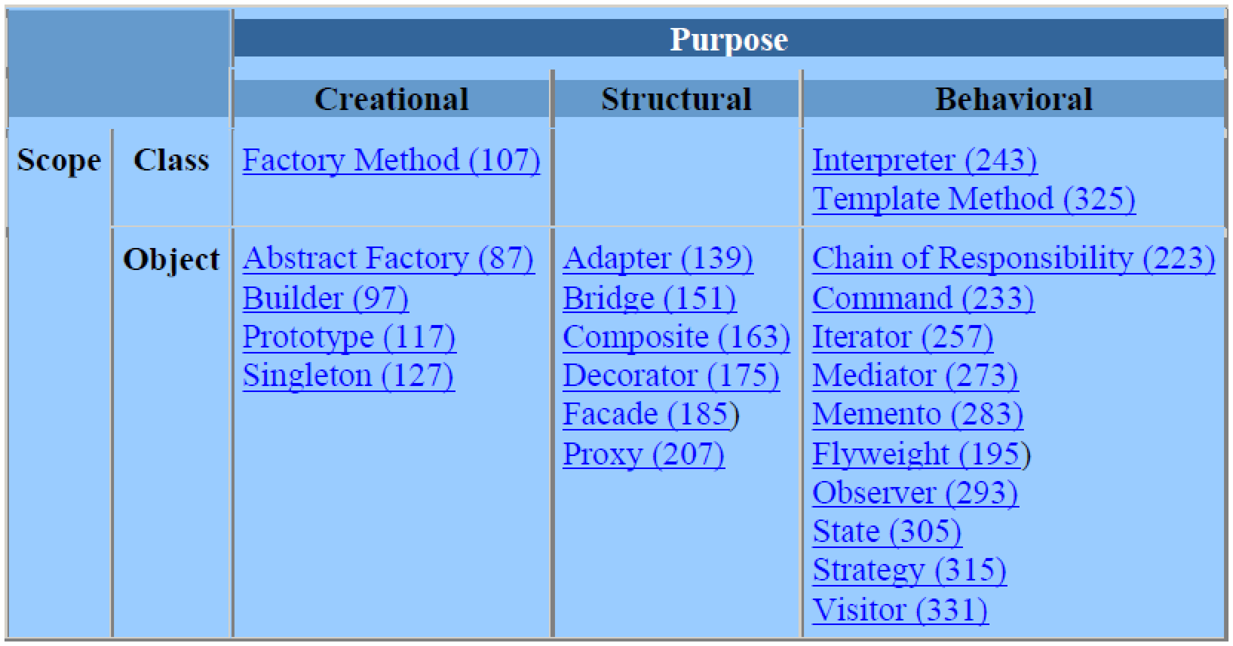
3. Structural Patterns:
   - Class Patterns: Use inheritance to compose classes.
   - Object Patterns: Assemble objects in flexible ways.

4. Behavioral Patterns:
   - Class Patterns: Use inheritance to define algorithms and control flow.
   - Object Patterns: Focus on how groups of objects work together to complete tasks that one object cannot perform alone.

5. Relationships Between Patterns:
   - Some patterns are frequently used together. For example:
     - Composite often works with Iterator or Visitor.
   - Some patterns serve as alternatives:
     - Prototype can be an alternative to Abstract Factory.
   - Patterns may result in similar designs even with different intents:

- The structure diagrams of Composite and Decorator are similar, despite their different goals.

This organized approach helps in understanding the catalog and applying patterns in various design situations.



| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method (107) | | Interpreter (243)<br>Template Method (325) |
| | Object | Abstract Factory (87)<br>Builder (97)<br>Prototype (117)<br>Singleton (127) | Adapter (139)<br>Bridge (151)<br>Composite (163)<br>Decorator (175)<br>Facade (185)<br>Proxy (207) | Chain of Responsibility (223)<br>Command (233)<br>Iterator (257)<br>Mediator (273)<br>Memento (283)<br>Flyweight (195)<br>Observer (293)<br>State (305)<br>Strategy (315)<br>Visitor (331) |

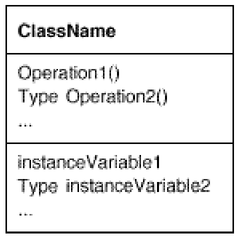
Table 1.1: Design pattern space

**How Design Patterns Solve Design Problems**

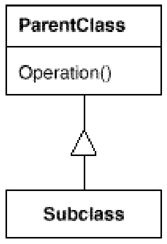
1. Finding Appropriate Objects:
  - Design patterns help identify abstract objects that may not exist in real-world models, like objects that represent processes or algorithms.
  - Patterns like Strategy and State provide structures for implementing flexible designs using interchangeable algorithms or states.

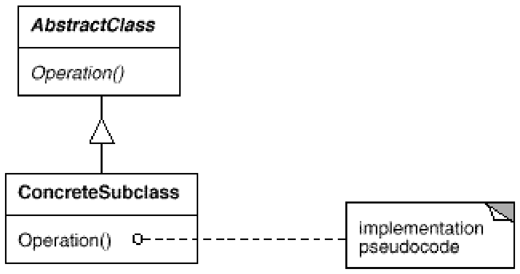
2. Determining Object Granularity:
  - Design patterns assist in deciding how big or small objects should be.
  - For example, Facade represents entire subsystems, while Flyweight supports many fine-grained objects efficiently.

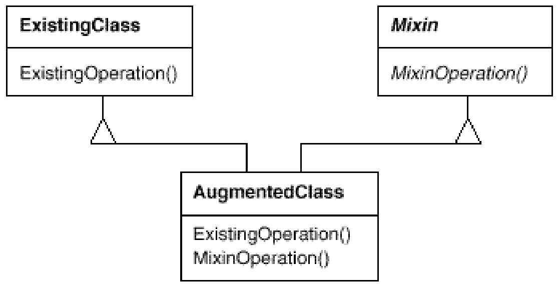
3. Specifying Object Interfaces:
  - An object's interface defines the set of operations it can perform.
  - Design patterns like Memento and Visitor help define and manage interfaces and relationships between them.
  - Patterns also enforce constraints on interfaces, like in Decorator and Proxy, where the interfaces of the decorator or proxy must match the original object.



4. Specifying Object Implementations:
- Classes define how objects are implemented. Abstract classes help define common interfaces, while subclasses refine or override behavior.
- Design patterns like Template Method and Command provide mechanisms to structure object behaviors through inheritance or delegation.







5. Class vs. Object Inheritance:
- Class Inheritance: Inheritance allows extending an application's functionality by reusing parent class features. It's a mechanism for code sharing.
- Object Composition: Achieved dynamically at runtime, allowing more flexibility and reducing implementation dependencies.
- Design patterns encourage favoring object composition over inheritance for flexibility, as seen in Decorator and Strategy.

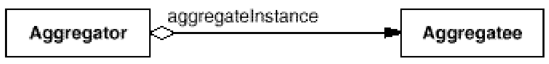
6. Programming to an Interface, not an Implementation:
  - Design patterns emphasize the importance of programming to interfaces, not specific implementations, enhancing flexibility and reducing dependencies.
  - Abstract Factory, Builder, and Factory Method allow associating interfaces with implementations without hard-coding them.

7. Inheritance vs. Composition:
  - Inheritance: Extends functionality but can introduce tight coupling and limit flexibility.
  - Composition: Encourages loosely coupled systems, making them easier to modify and extend. Patterns like Decorator and Composite rely heavily on composition.

8. Delegation:
  - Delegation enables objects to handle requests by passing them to other objects, providing flexibility at runtime.
  - Patterns like State, Strategy, and Visitor use delegation to alter object behavior dynamically.

9. Designing for Change:
  - Design patterns ensure systems can adapt to future changes, such as:
    - Abstract Factory, Prototype: Create objects without specifying the class.
    - Chain of Responsibility, Command: Avoid dependence on specific operations.
    - Observer, Strategy: Isolate algorithms for flexibility.

10. Relating Run-Time and Compile-Time Structures:
  - Design patterns help manage the dynamic, changing relationships between objects at runtime, which are often independent of the static class structures.
  - Patterns like Composite, Decorator, and Observer capture these complex runtime relationships effectively.



This systematic approach ensures that design patterns solve common design problems in a reusable, flexible, and maintainable way.

**How to Select a Design Pattern**

1. Consider How Design Patterns Solve Design Problems:
  - Review how design patterns address common issues, such as finding appropriate objects, determining object granularity, specifying interfaces, and more.
  - This understanding helps guide you toward the right pattern for your design problem.

2. Scan the Intent Sections:
  - Each pattern has an Intent section that summarizes what the pattern does.
  - Scanning these intents quickly helps identify patterns relevant to your problem.
  - Use the classification scheme (Creational, Structural, Behavioral) to narrow your search.

3. Study Pattern Relationships:

- Patterns are interrelated. Exploring these relationships helps you find connected patterns or groups of patterns.
- Some patterns are often used together or solve similar problems from different angles.

4. Study Patterns of Similar Purpose:
   - Design patterns are grouped into Creational, Structural, and Behavioral categories. Study the similarities and differences between patterns in these groups.
   - This comparison helps you choose the most suitable one.

5. Examine Causes of Redesign:
   - Look at common causes of redesign (e.g., hard dependencies, platform issues, algorithmic changes).
   - Select patterns that address these causes to make your design more adaptable.

6. Consider What Should Be Variable:
   - Focus on what aspects of your design need to vary without redesign.
   - Many patterns encapsulate the concept that varies, allowing for flexibility and adaptation without restructuring the entire system.

**How to Use a Design Pattern**

1. Read the Pattern Overview:
   - Start by reading the pattern to get a general understanding.
   - Focus on the Applicability and Consequences sections to ensure the pattern is a good fit for your problem.

2. Study Structure and Collaborations:
   - Review the Structure, Participants, and Collaborations to understand the relationships between the classes and objects in the pattern.

3. Examine Sample Code:
   - Look at the Sample Code section to see a practical example of the pattern in action, which helps you understand its implementation.

4. Choose Meaningful Names:
   - Use application-specific names for the pattern participants (e.g., rename abstract participant names to reflect their roles in your system).

5. Define Classes and Interfaces:
   - Define the classes, declare their interfaces, and establish inheritance relationships. Update existing classes to fit the pattern.

6. Name Operations Appropriately:
   - Use clear, application-relevant names for operations. Consistently apply naming conventions (e.g., "Create-" for factory methods).

7. Implement the Pattern:

- Implement the operations as outlined in the pattern's responsibilities and collaborations.
- Use hints from the Implementation section and refer to the sample code.

8. Avoid Overuse:
   - Do not apply design patterns indiscriminately. Patterns can add complexity and reduce performance.
   - Only use a pattern when the flexibility it provides is truly necessary, and review the Consequences to evaluate the trade-offs.

**Common Causes of Redesign and Related Design Patterns**

1. Creating an Object by Specifying a Class Explicitly:
   - Specifying a class name binds you to a particular implementation, making future changes difficult.
   - Solution: Create objects indirectly using patterns like:
     - Abstract Factory
     - Factory Method
     - Prototype

2. Dependence on Specific Operations:
   - Hard-coding specific operations limits flexibility and makes future changes difficult.
   - Solution: Avoid hard-coded operations using patterns like:
     - Chain of Responsibility
     - Command

3. Dependence on Hardware and Software Platform:
   - Relying on specific platform APIs makes it difficult to port the software across platforms.
   - Solution: Use patterns to abstract platform dependencies:
     - Abstract Factory
     - Bridge

4. Dependence on Object Representations or Implementations:
   - Exposing object representations or implementation details can cause cascading changes.
   - Solution: Hide object details using patterns like:
     - Abstract Factory
     - Bridge
     - Memento
     - Proxy

5. Algorithmic Dependencies:
   - Algorithms are often optimized or replaced. Coupling to specific algorithms makes changes difficult.
   - Solution: Isolate algorithms using patterns like:
     - Builder
     - Iterator
     - Strategy
     - Template Method

- Visitor

6. Tight Coupling:
   - Tight coupling between classes makes reuse difficult and leads to monolithic systems that are hard to modify.
   - Solution: Promote loose coupling using patterns like:
     - Abstract Factory
     - Bridge
     - Chain of Responsibility
     - Command
     - Facade
     - Mediator
     - Observer

7. Extending Functionality by Subclassing:
   - Subclassing can lead to complex dependencies and class explosion.
   - Solution: Use object composition and delegation with patterns like:
     - Bridge
     - Chain of Responsibility
     - Composite
     - Decorator
     - Observer
     - Strategy

8. Inability to Alter Classes Conveniently:
   - Sometimes modifying a class is not possible due to lack of access to the source code or complexity.
   - Solution: Modify behavior without changing classes using patterns like:
     - Adapter
     - Decorator
     - Visitor

These patterns help build flexibility into software, ensuring that changes and extensions can be made more easily in the future. The importance of these patterns depends on the specific nature of the software being developed.

---

Q. Illustrate the Single Responsibility Principle of object-oriented programming with an example code.
Q. Illustrate the Open Close Principle of object-oriented programming with an example.
Q. Illustrate the Liskov Substitution Principle of object-oriented programming with an example.
Q. Illustrate the Interface Segregation Principle of object-oriented programming with an example code.
Q. Illustrate the Dependency Inversion Principle of object-oriented programming with an example.
Q. Illustrate the "Favor Composition Over Inheritance" Principle of object-oriented programming with an example.

Q. Illustrate the “Program to an interface not an implementation” Principle of object-oriented programming with an example.
Q. Illustrate the “Separate aspects of the systems that vary from what remains same” Principle of object-oriented programming with an example.