What problems do design patterns solve?

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".
- Four essential elements of a pattern::

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. The **problem** describes when to apply the pattern.

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations
4. The **consequences** are the results and trade-offs of applying the pattern.

- The design patterns are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*.

The Catalog of Design Patterns

1. **Abstract Factory** provides an interface for creating families of related or dependent objects without specifying their concrete classes.
2. **Adapte**r converts the interface of a class into another interface clients expect. Adapter lets classes work together that

couldn't otherwise because of incompatible interfaces.

3. **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
4. **Builder** separates the construction of a complex object from its representation so that the same construction process can create different representations.
5. **Chain of Responsibility** avoids 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.
6. **Command** encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
7. **Composite** composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat

individual objects and compositions of objects uniformly.

8. **Decorator** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
9. **Facade** provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
10. **Factory Method** defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
11. **Flyweight** uses sharing to support large numbers of fine-grained objects efficiently.
12. **Interpreter**,given a language, defines a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

13. **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
14. **Mediator** defines an object that encapsulates how a set of objects interact.
15. **Memento**, without violating encapsulation, captures and externalizes an object's internal state so that the object can be restored to this state later.
16. **Observer** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
17. **Prototype** specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
18. **Proxy** provides a surrogate or placeholder for another object to control access to it.

19. **Singleton** ensures a class only has one instance, and provide a global point of access to it.
20. **State** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
21. **Strategy** defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
22. **Template Method** defines 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.
23. **Visitor** represents 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.

How Design Patterns Solve Design Problems?

1. Finding Appropriate Objects

- Design patterns help you identify less-obvious abstractions and the objects that can capture them.

2. Determining Object Granularity

- **Facade** pattern describes how to represent complete subsystems as objects.
- **Flyweight** pattern describes how to support huge numbers of objects at the finest granularities.
- **Abstract Factory** and Builder yield objects whose only responsibilities are creating other objects.
- **Visitor and Command** yield objects whose only responsibilities are to implement a

request on another object or group of objects.

3. Specifying Object Interfaces

- The set of all signatures defined by an object's operations is called the **interface** to the object.
- A **type** is a name used to denote a particular interface.
- An object may have many types, and widely different objects can share a type.
- A type is a **subtype** of another if its interface contains the interface of its **supertype**, or a subtype *inheriting* the interface of its supertype.
- Objects are known only through their interfaces. An object's interface says nothing about its implementation
- When a request is sent to an object, the particular operation that's performed depends on *both* the request *and* the receiving object.

- **Dynamic binding**: the run-time association of a request to an object and one of its operations.
- **Polymorphism**: dynamic binding can substitute objects that have identical interfaces for each other at run-time
- Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface.
- Design patterns also specify relationships between interfaces.

4. Specifying Object Implementations

- An object's implementation is defined by its **class**.

- An **abstract class** is one whose main purpose is to define a common interface for its subclasses.

- A **mixin class** is a class that's intended to provide an optional interface or functionality to other classes.

- *Class versus Interface Inheritance*

  - An object's class defines how the object is implemented.
  - An object's type only refers to its interface.
  - An object can have many types.

   - Objects of different classes can have the same type.
   - Relationship between class and type: class as type (C++) vs. interface as type (Java).
   - Class inheritance: Sub-typing + Implementation inheritance
   - Interface inheritance: Sub-typing only (Polymorphism)
   - Pure abstract classes as interfaces.
   - Many of the design patterns depend on the distinction between class and interface inheritances

   .

- *First Principle of reusable object-oriented design: Programming to an Interface, not an Implementation*

   - Class inheritance-based implementation reuse is only half the story. Inheritance's ability to define families of objects with ***identical*** **interfaces** is also important, because polymorphism depends on it.

- Two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

- Don't declare variables to be instances of particular concrete classes.
- Creational patterns ensure that your system is written in terms of interfaces, not implementations.

5. Putting Reuse Mechanisms to Work

- *Inheritance versus Composition*

- **White-box reuse**: class inheritance.
- **Black-box reuse**: object composition
- Class inheritance:
  - Advantages
    - supported by programming languages, defined statically at compile-time and is straightforward to use
    - make it easier to modify the implementation being reused, when a subclass overrides some but not all operations.
  - Disadvantages
    - Cannot change the implementations/representation s inherited from parent classes at run-time
    - Implementation dependency between a subclass and its parent class.
- Object composition
  - Advantages

        - Defined dynamically at run-time by referencing interfaces of objects.
        - Access other objects though their interfaces only, not break encapsulation.
        - Fewer implementation dependencies.
        - Small class hierarchies
    - Disadvantages
        - More objects
        - The system's behavior will depend on their interrelationships instead of being defined in one class
    - The second principle of object-oriented design:

      ***Favor object composition over class inheritance*.**

- *Delegation*

- In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**.
- The receiver passes itself to the delegate to let the delegated operation refer to the receiver.

    - Advantage: it makes it easy to compose behaviors at run-time and to change the way they're composed.
    - Disadvantage: harder to understand than more static software, and run-time inefficiencies,
    - Delegation works best when it's used in standard patterns.
    - Design patterns that use delegation: State, Strategy, Visitor, Mediator, Chain of

Responsibility, and Bridge patterns.

6. Relating Run-Time and Compile-Time Structures

- An object-oriented program's run-time structure often bears little resemblance to its code structure.
- **Aggregation**
    - Manifested at run-times.
    - One object owns (having) or is responsible for another object (being part).
- **Acquaintance**
    - Manifested at compile-times.

    - An object merely *knows of* another object (association, using).
    - A weaker relationship than aggregation.
- In implementation or code, aggregation and acquaintance cannot be distinct.
- Many design patterns capture the distinction between compile-time and run-time structures explicitly..
- The run-time structures aren't clear from the code until you understand the patterns.

7. Designing for Change

- Common causes of redesign along with the design pattern(s) that address them:

1. *Creating an object by specifying a class explicitly.*

   Design patterns: Abstract Factory, Factory Method, Prototype.

2. *Dependence on specific operations.*.

Design patterns: Chain of Responsibility, Command.

3. *Dependence on hardware and software platform..*

Design patterns: Abstract Factory, Bridge.

4. *Dependence on object representations or implementations..*

Design patterns: Abstract Factory, Bridge, Memento, Proxy.

5. *Algorithmic dependencies.*

Design patterns: Builder, Iterator , Strategy, Template Method , Visitor.

6. *Tight coupling.*

Design patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.

Extending functionality by subclassing.

Design patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy.

7. *Inability to alter classes conveniently.*

   Design patterns: Adapter, Decorator, Visitor.

- The role design patterns play in the development of three broad classes of software: application programs, toolkits, and frameworks.
  - *Application Programs*
    - *Internal* reuse, maintainability, and extension are high priorities.
    - Design patterns that reduce dependencies can increase internal reuse.
    - Design patterns make an application more maintainable when they're used to limit platform dependencies and to layer a system.

        - Design patterns enhance extensibility.
- *Toolkits (class/component libraries)*
    - Code reuse
    - Application-general design
- *Frameworks*
    - A framework is a set of cooperating classes that make up a reusable design for a specific class of software
    - The framework dictates the architecture of your application.
    - Frameworks emphasize *design reuse* over code reuse.
    - Frameworks are implemented as class hierarchies..
    - Reuse on framework level leads to an inversion of control between the application and the software on which it's based.
    - Mature frameworks usually incorporate several design patterns
    - Design patterns vs. frameworks

1. *Design patterns are more abstract than frameworks.*
2. *Design patterns are smaller architectural elements than frameworks.*
3. *Design patterns are less specialized than frameworks.*

How to Select a Design Pattern

1. *Consider how design patterns solve design problems.*
2. *Scan Intent sections.*
3. *Study how patterns interrelate.*
4. *Study patterns of like purpose.*
5. *Examine a cause of redesign.*
6. *Consider what should be variable in your design.*

How to Use a Desgn Pattern

1. *Read the pattern once through for an overview.* Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.
2. *Go back and study the Structure, Participants, and Collaborations sections.* Make sure you understand the classes and objects in the pattern and how they relate to one another.
3. *Look at the Sample Code section to see a concrete example of the pattern in code.* Studying the code helps you learn how to implement the pattern.
4. *Choose names for pattern participants that are meaningful in the application context.*
5. *Define the classes.*
6. *Define application-specific names for operations in the pattern.*

7. *Implement the operations to carry out the responsibilities and collaborations in the pattern.* The Implementation section offers hints to guide you in the implementation.