After all ,It's necessary to define what is a design pattern before seeing its solutions.

What is a Design Pattern?

In a simple sentence, a design pattern is a way to design our software/application. Each design pattern addresses a problem or set of problems and describes a solution for it. So these design patterns help us understand those problems and solve them effectively.

- **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*.

Describing Design Patterns

- **Pattern Name and Classification**
- **Intent**

What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

- **Also Known As**
- **Motivation**

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.

- **Applicability**

What are the situations in which the design pattern can be applied?

- **Structure**

A graphical representation of the classes in the pattern using a notation based on OMT or UML.

- **Participants**

The classes and/or objects participating in the design pattern and their responsibilities.

- **Collaborations**

How the participants collaborate to carry out their responsibilities.

- **Consequences**

How does the pattern support its objectives? What are the trade-offs and results of using the pattern?

- **Implementation**

What should you be aware of when implementing the pattern? Are there language-specific issues?

- **Sample Code**

Code fragments that illustrate how you might implement the pattern in particular object-oriented programming languages.

- **Known Uses**

Examples of the pattern found in real systems.

- **Related Patterns**

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.

Organizing the Catalog

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

How Design Patterns Solve Design Problems?

1. Finding Appropriate Objects
2. Determining Object Granularity
3. Specifying Object Interfaces
4. Specifying Object Implementations
5. Putting Reuse Mechanisms to Work
6. Relating Run-Time and Compile-Time Structures
7. Designing for Change

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 the cause of redesign.*
6. *Consider what should be variable in your design.*

	Purpose	
Design Pattern	**Aspect(s) That Can Vary**	
Creational	**Abstract Factory**	**families of product objects**
	Builder	**how a composite object gets created**

	Factory Method	**subclass of object that is instantiated**
	Prototype	**class of object that is instantiated**
	Singleton	**the sole instance of a class**
Structural	**Adapter**	**interface to an object**
	Bridge	**implementation of an object**
	Composite	**structure and composition of an object**
	Decorator	**responsibilities of an object without subclassing**
	Facade	**interface to a subsystem**

	Flyweight	**storage costs of objects**
	Proxy	**how an object is accessed; its location**
Behavioral	**Chain of Responsibility**	**object that can fulfill a request**
	Command	**when and how a request is fulfilled**
	Interpreter	**grammar and interpretation of a language**
	Iterator	**how an aggregate's elements are accessed, traversed**
	Mediator	**how and which objects interact with each other**

	Memento	**what private information is stored outside an object, and when**
	Observer	**number of objects that depend on another object; how the dependent objects stay up to date**
	State	**states of an object**
	Strategy	**an algorithm**
	Template Method	**steps of an algorithm**
	Visitor	**operations that can be applied to object(s) without changing their class(es)**

How to Use a Design 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.**