Design Patterns: In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

The basic idea is to re-use well-designed and documented OO solutions instead of re-inventing the wheel.

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Catalog of Patterns: First recognized by **GOF** = "***Gang of four***" (1995)
* **Erich Gamma** * **Richard Helm** * **Ralph Johnson** * **John Vlissides**

Published a book with 23 patterns. Others have identified additional patterns

What is Gang of Four (GOF)?
In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.

These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

* Program to an interface not an implementation
* Favor object composition over inheritance

* Usage of Design Pattern
Design Patterns have two main usages in software development.

* Common platform for developers
Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

* Best Practices
Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

Essential Elements
Pattern Name: Describes the pattern
Problem: What is being solved, the context of the problem. When to apply the pattern
Solution: The elements that make up the design, relationships in terms of classes and objects, responsibilities and collaborations.
Consequences: Results, benefits, and trade-offs

Documenting Design Patterns – Essential Information
Pattern Name: Describes the essence of the pattern in a short, but expressive, name.
Intent: Describes what the pattern does
Also Known As: List any synonyms for the pattern
Motivation: Provides an example of a problem and how the pattern solves that problem.
Applicability: Lists the situations where the pattern is applicable.
Structure: Set of diagrams of the classes and objects that depict the pattern.
Participants: Describes the classes and objects that participate in the design pattern and their responsibilities.
Collaborations: Describes how the participants collaborate to carry out their responsibilities.
Consequences: Describes the forces that exist with the pattern and the benefits, trade-offs, and the variable that is isolated by the pattern.

Design patterns are divided into three fundamental groups:
Behavioral, Creational and Structural.

Behavioral patterns:
Behavioral patterns describe interactions between objects and focus on how objects communicate with each other. They can reduce complex flow charts to mere interconnections between objects of various classes. Behavioral patterns are also used to make the algorithm that a class uses simply another parameter that is adjustable at runtime.

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

These patterns characterize complex control flow that is difficult to follow at run-time. They shift your focus away from the flow of control to let you concentrate just on the way objects are interconnected. Behavioral class patterns use inheritance to distribute behavior between classes.
The Template Method is the simpler and more common of the two. A template method is an abstract definition of an algorithm. It defines the algorithm step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations. The other behavioral class pattern is Interpreter pattern, which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Chain of responsibility: A way of passing a request between a chain of objects
Command: Encapsulate a command request as an object
Interpreter: A way to include language elements in a program
Iterator: Sequentially access the elements of a collection
Mediator: Defines simplified communication between classes
Memento: Capture and restore an object's internal state
Null Object: Designed to act as a default value of an object
Observer: A way of notifying change to a number of classes
State: Alter an object's behavior when its state changes
Strategy: Encapsulates an algorithm inside a class

Template method: Defer the exact steps of an algorithm to a subclass
Visitor: Defines a new operation to a class without change

Creational Patterns:
Creational patterns are used to create objects for a suitable class that serves as a solution for a problem. Generally, when instances of several different classes are available. They are particularly useful when you are taking advantage of polymorphism and need to choose between different classes at runtime rather than compile time.

Creational patterns support the creation of objects in a system. Creational patterns allow objects to be created in a system without having to identify a specific class type in the code, so you do not have to write large, complex code to instantiate an object. It does this by having the subclass of the class create the objects. However, this can limit the type or number of objects that can be created within a system.
PHP Objects, Patterns, and Practice

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Abstract Factory: Creates an instance of several families of classes
Builder: Separates object construction from its representation
Factory Method: Creates an instance of several derived classes
Object Pool: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
Prototype: A fully initialized instance to be copied or cloned
Singleton: A class of which only a single instance can exist

Structural Patterns:
Structural patterns form larger structures from individual parts, generally of different classes.
Structural patterns vary a great deal depending on what sort of structure is being created for what purpose.

Structural patterns are concerned with how classes and objects are composed to form larger structures.
Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

Another example is the class form of the Adapter Pattern. In general, an adapter makes one interface (the adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces. A class adapter accomplishes this by inheriting privately from an adaptee class. The adapter then expresses its interface in terms of the adaptee's.

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Adapter: Match interfaces of different classes

Bridge: Separates an object's interface from its implementation
Composite: A tree structure of simple and composite objects
Decorator: Add responsibilities to objects dynamically
Façade: A single class that represents an entire subsystem
Flyweight: A fine-grained instance used for efficient sharing
Private Class Data: Restricts accessor/mutator access
Proxy: An object representing another object

Classification of Design Patterns
Two criteria are used to classify:

- Purpose (What the pattern does)
  - Creational (The process of object creation)
  - Structural (Composition of classes or objects/ways assemble objects)
  - Behavioral (The way classes and objects interact/ use inheritance to assemble solutions and control flow)
- Scope (Does the pattern apply to classes or objects?)
  - Class Patterns (Relationships between classes and subclasses through inheritance)
  - Object Patterns (Object Relationships that can be changed at run-time)

Benefits

- Focus on specific issues, not implementation
- Reduce development time. At the beginning, it may have a learning curve. The more familiar you get with the pattern, the development time should decrease gradually
- Patterns are language independent as long as it's Object Oriented.
- A design pattern is well documented by experienced programmers. Consequently, it will be easier to understand and apply a solution. Communication of development teams is easier/ You establish a common vocabulary.
- Reduces the technical risk in new development by applying new/untested designs.
- Patterns are flexible and can be applied to any type of application or domain.
- Improves code/ Design for change

How to Select a Design Pattern

- Consider how design patterns solve design problems
- Scan Intent sections
- Study how patterns interrelate and patterns of like purpose
- Consider what should be variable in your design

How to use a pattern

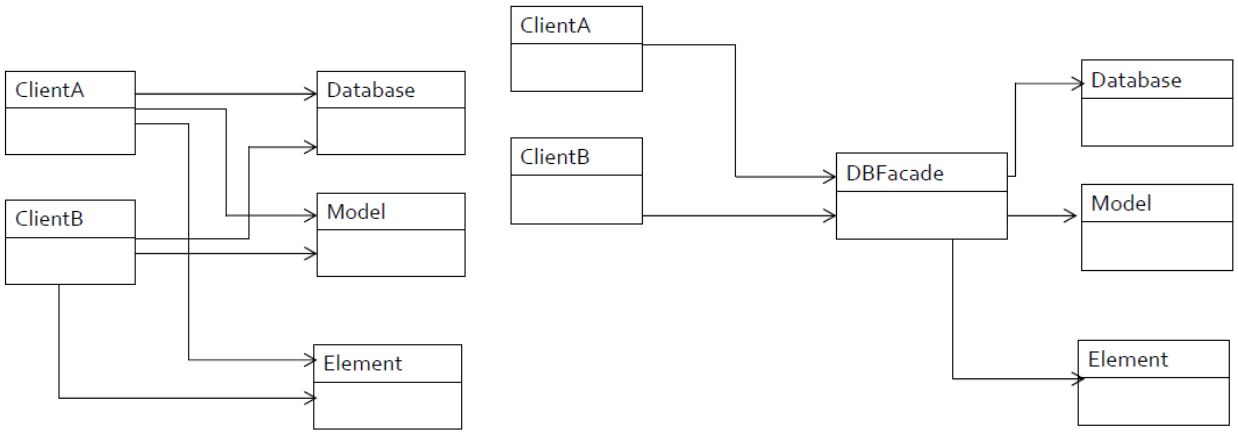
- Read the pattern paying attention to Applicability and Consequences
- Study the Structure, Participants, and Collaborations
- Look at Sample Code
- Choose good names for participants
- Define the classes, interfaces, and operations
- Implement the operations to carry out the responsibilities and collaborations

*** Facade (Structural)**
to simplify the use of an existing system. Need to define an interface that is a subset of existing one.

Solution: The façade presents a new interface
Participants: presents a specialized interface to the client make it easier use
Consequences:
Simplifies the use of subsystem.
Some functionality may not be available to the client.
Implementation:
Define a new class (or classes) that has the required interface.
Have this class use the existing system.

Façade Example:

- Huge database interface, but we only need a few of the functions.
- Rather than everyone learning the database interface, create the interface you need, and build a façade.



Remote control for your house which controls different equipment. You just interact with the remote control, and the remote control figures out which device should respond and what signal to send.

*** Mediator (Behavioral)**
to define an object that encapsulates how a set objects interact.
("An Intermediary")

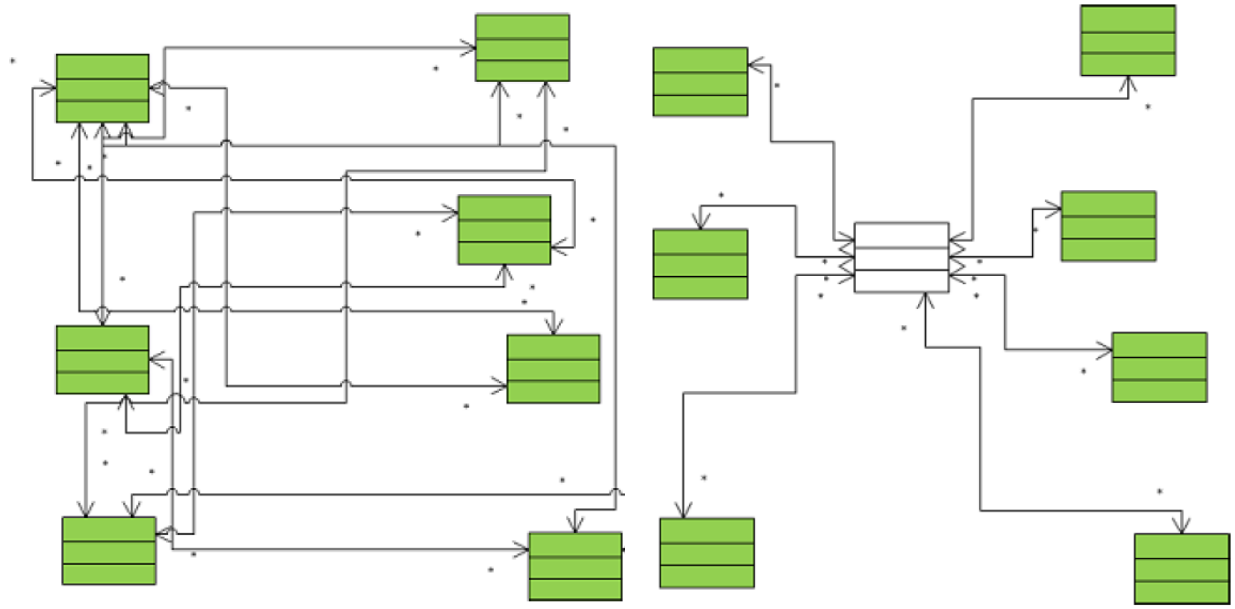
Solution: Define an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly.
Participants:
-Mediator: An interface for communication with the objects interacting with each other (Colleague)
-Concrete Mediator: Coordination of objects.
-Colleague classes: Each class knows and communicates with its mediator object instead of another colleague.
Consequences:
-Limits subclassing: Subclassing the mediator ONLY, not several objects. Colleague classes are reused.
-Decouples colleagues. You can manage/reuse Colleague classes and mediator independently of each other.
-Simplifies object protocols. Replaces "many -to -many" interactions with "one -to -many" interactions.
-Abstracts objects cooperation: You focus on how objects interact apart from their individual behavior.

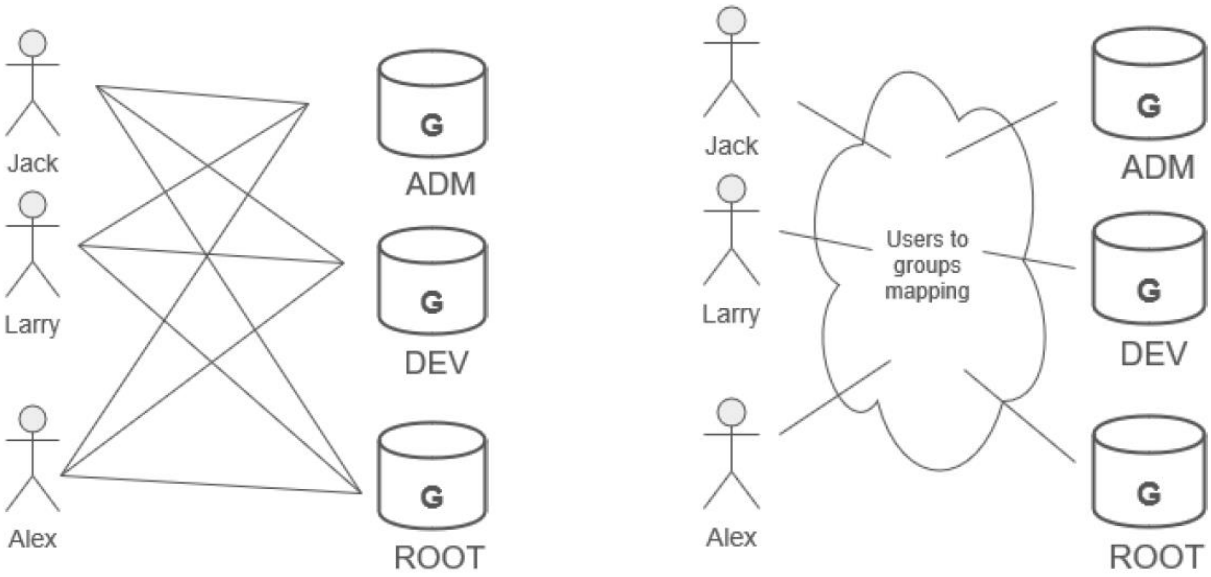
-Centralizes control: Complexity of interaction vs complexity in the mediator. Mediator can become so complex that it is hard to maintain.

Implementation:

-Colleagues can work with different mediator subclasses. No need for an abstract mediator class when colleagues work with only one mediator.

-Communication: Colleagues communicate with their mediator when an event occurs. You could combine with the Observer pattern. A colleague class sends notifications to the mediator when there is a change of state. The mediator responds by propagating the effects of change to other colleagues.



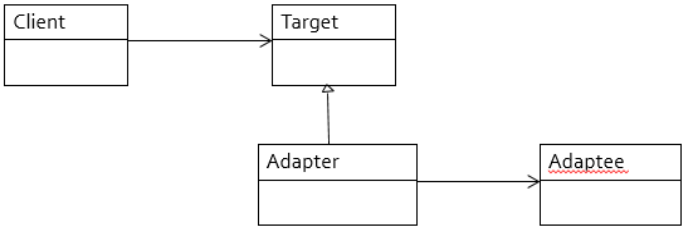


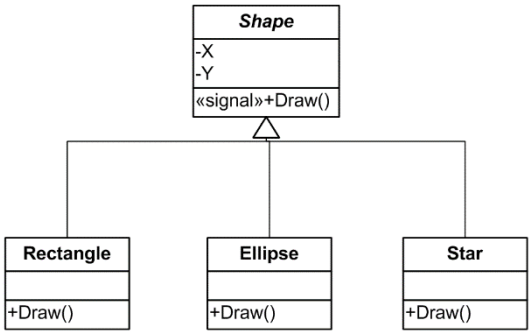
The control tower at a controlled airport, the pilots of the planes approaching or departing the terminal area, communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. The tower does not control the whole flight. It exists only to enforce constraints in the terminal area.

*** Adapter (Structural)**
Intent: Convert the interface of a class into another interface clients expect
Also Known As: Wrapper
Problem:

- A system has the right data and behavior, but the wrong interface.
- Use this when you have to make an existing class a subclass of a new class structure.

Solution: Adapter provides a wrapper with the desired interface.
Participants: The adapter adapts the interface of an adaptee to match that of the adapter's target. This allows the client to use the adaptee as if it were a type of target.
Consequences: the adapter pattern allows for pre-existing objects to fit into a new class structure without being limited by their interfaces.
Implementation: encapsulate an existing class in another class. The containing class matches the required interface.

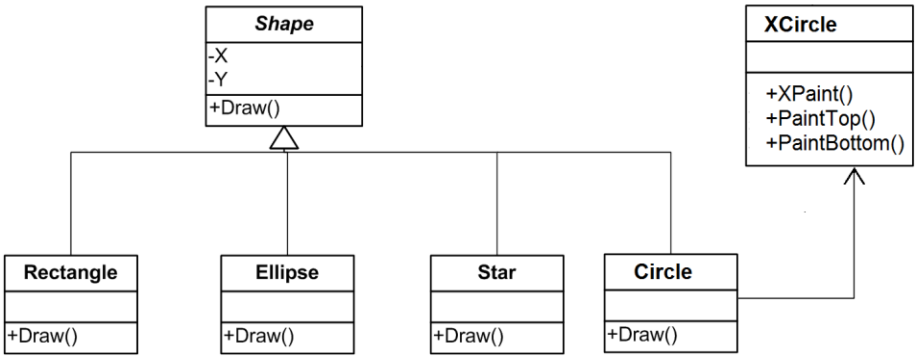




- Drawing program
  - Superclass called Shape
  - Subclasses
    - Rectangle
    - Ellipse
    - Star
- Need to add circle
  - Have circle class implemented in a class called xcircle
  - But the circle I need must be of type Shape.
  - Xcircle isn't. The methods are named wrong.

Adapter:

- Create a class circle
- It encapsulates xcircle
- Circle class has the correctly named methods
- Only need to call some of the xcircle methods.



Types of Design Patterns
As per the design pattern reference book Design Patterns - Elements of Reusable Object-Oriented Software, there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

Creational Patterns:
These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

Structural Patterns:
These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

Behavioral Patterns:
These design patterns are specifically concerned with communication between objects.

J2EE Patterns:
These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

References:
https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
https://sourcemaking.com/design_patterns
https://sourcemaking.com/design_patterns/facade
https://sourcemaking.com/design_patterns/adapter
https://sourcemaking.com/design_patterns/mediator