Software Design And Architecture

Property	
Tags	SWE 316

Lecture 11 - Basic Concepts & Styles - I

- ▼ Process is not as important as architecture

  True

- ▼ The architecture of the Web is wholly separate from the code

  True

- ▼ What is software design?

  It as an activity that creates part of a system's architecture

- ▼ What are the main concerns in design phase?

  1. Define the system's structure
  2. Identification of its primary components
  3. The components interconnections

- ▼ What is an architecture?

  It denotes the set of principal design decisions about a system

- ▼ What is the point behind the design phase?

  Is to translate the requirements into algorithms, so programmer can implement them

- ▼ What is architecture-centric design method?

  It is a method that ensures the following points

1. Avoid stakeholder issues
2. Decision about use of COTS component
3. Develop package and primary class structure

Lecture 12 - Basic Concepts & Styles - II

▼ What is software architecture?

Is the set of principal design decisions about the system

"Principal" → Important decision

▼ Why defining the software architecture is important?

Because it discuss key points on the system

1. The description of elements from which systems are build
2. Interactions among those elements
3. Patterns that guide their compositions
4. Constraints on these patterns

▼ What is the meaning of temporal aspect?

Architecture has a temporal aspect which means at any give point in time the system has only one architecture and it can change over time

▼ What is the difference between prescriptive and descriptive architecture?

Prescriptive → The architecture before the system is constructed (as planned)

Descriptive → The architecture after the system has been build (as implemented)

▼ What are software components?

Elements that encapsulate processing and data in a system's architecture

▼ What are the characteristics of a software component?

1. Encapsulates a subset of the system's functionality and/or data
2. Restricts access to that subset via interface
3. Has explicitly defined dependencies

▼ What is a software connector?

It is an architectural building block tasked with effecting and regulating interactions among components

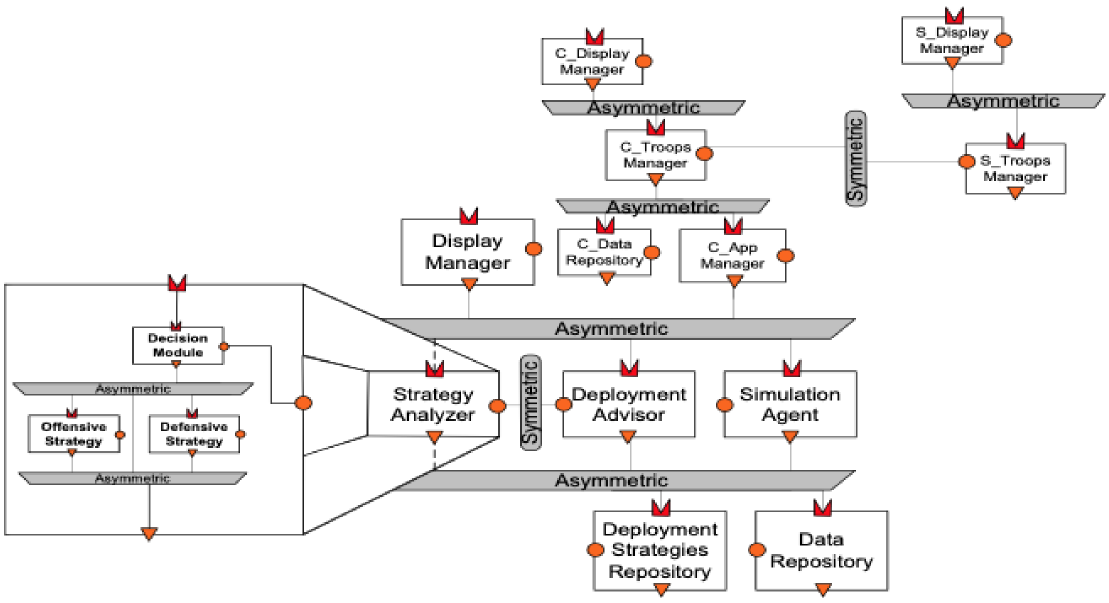
▼ Give an example of a software connector

1. Procedure call
2. Shared data accesses
3. APIs
4. Event

▼ What is the meaning of architectural configuration (topology)?

Is a set of specific associations between the components and connectors of a software system's architecture

▼ Give an example of an architecture configuration



Lecture 13 - Basic Concepts & Styles - III

- What are architectural styles?

  A collection of architectural design decisions

- What are the characteristics of architectural styles?

  1. They are applicable in a given development context (specific scenarios)
  2. Constrain architectural design decisions that are specific to a particular system within that context
  3. Acquire beneficial qualities in each resulting system

- List some common architectural styles

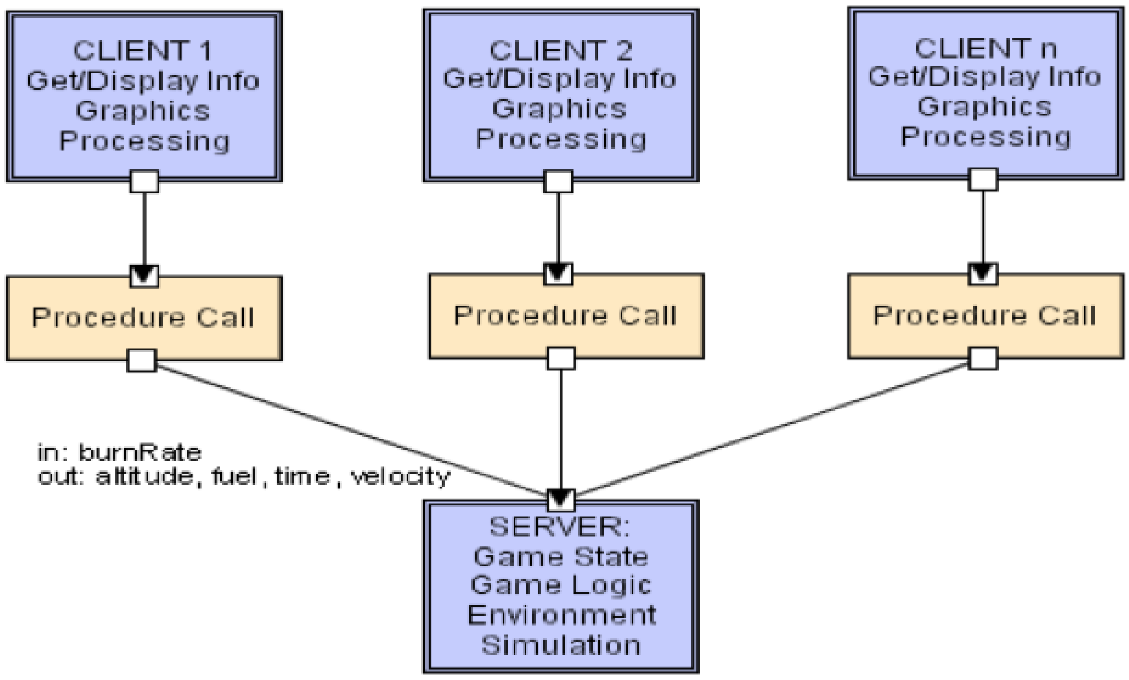
  1. Client-server
  2. Pipe and filter

3. Blackboard
4. Interpreter
5. Event-based
6. Publish-subscribe
7. Peer-to-peer

▼ Describe the client-server style

1. Components → Clients and servers
2. Connectors → Remote procedure calls, network protocols
3. Data elements → Parameters and return values as send by connectors
4. Typical use → Applications where centralization of data is required and business applications

Topology

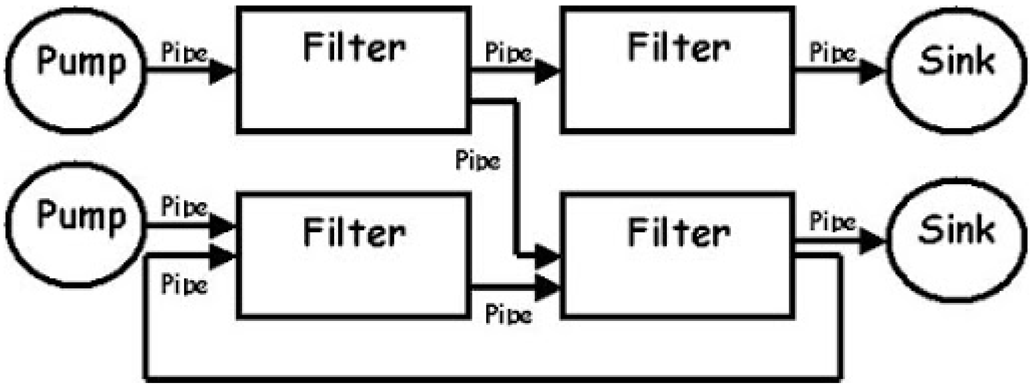


▼ Describe the pipe and filter style

Separate programs are executed, potentially at the same time; data is passed as a stream from on program to the other

1. Components → Independant programs (Filters)
2. Connectors → Explicit routers of data streams (Pipes)
3. Data elements → No specific form, but it must be linear data stream
4. Typical use → Operating system application

Topology



▼ What are the advantages of pipe and filter style?

1. System behavior is a succession of component behaviors
2. Filter addition, replacement, and reuse is easy

▼ What are the disadvantages of pipe and filter style?

1. Batch organization of processing
2. Interactive applications
3. Lowest common denominator on data transimission

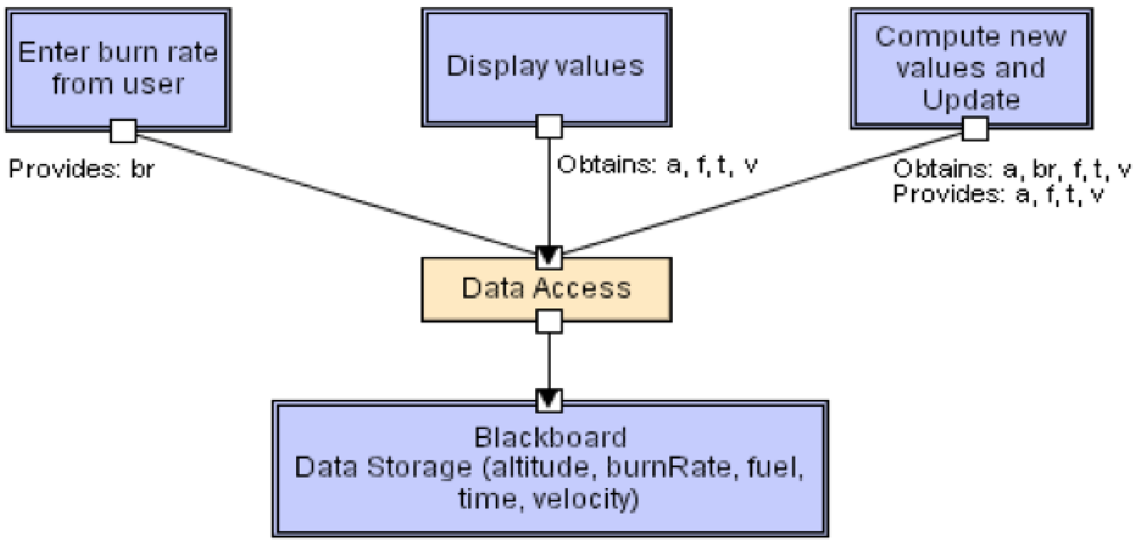
▼ Describe the blackboard style

Independent programs access and communicate exclusively though a global data repository, known as a blackboard

1. Components → Independent programs (Knowledge sources)

2. Connectors → Access to the blackboard may be by direct memory reference or can be through a procedure call or a database query
3. Data elements → Data stored in the blackboard
4. Typical use → Heuristic problem solving in artificial intelligence applications

Topology


Enter burn rate from user
Display values
Compute new values and Update
Provides: br
Obtains: a, f, t, v
Obtains: a, br, f, t, v
Provides: a, f, t, v
Data Access
Blackboard
Data Storage (altitude, burnRate, fuel, time, velocity)

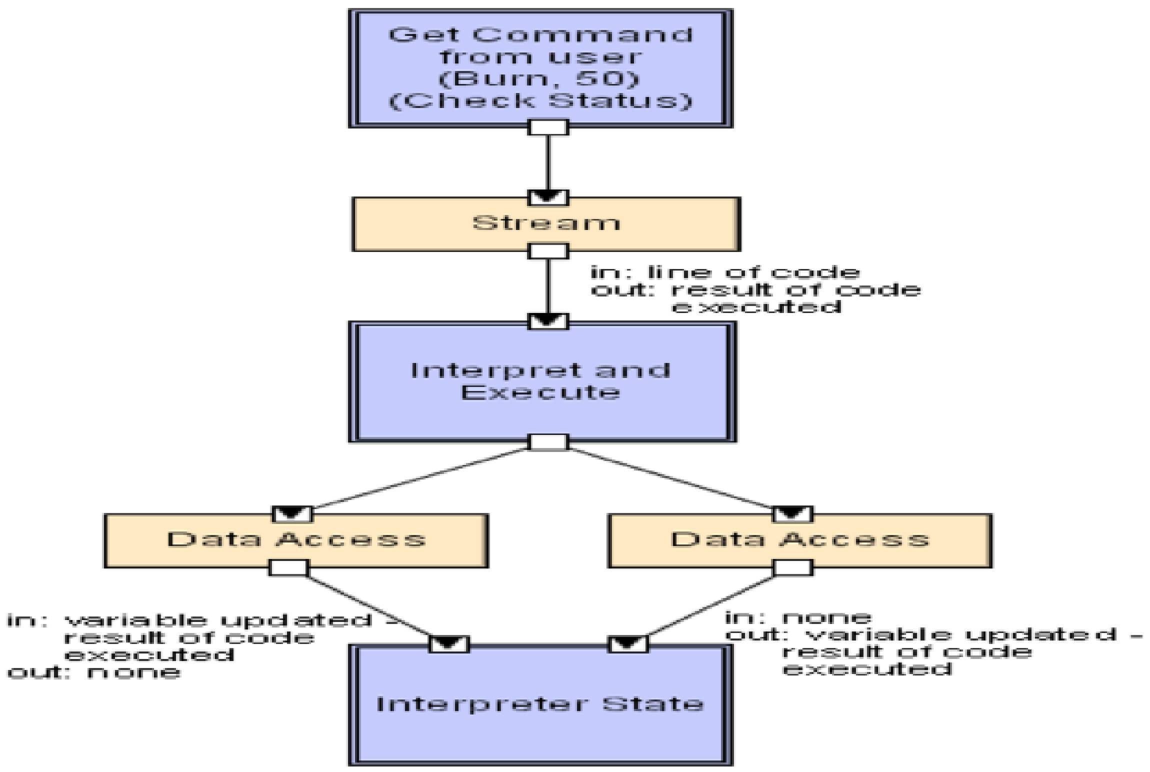
▼ Describe the interpreter style

Interpreter parses and executes input commands, updating the state maintained by the interpreter

1. Components → Command interpreter (Program, UI)
2. Connectors → Closely bound with direct procedure calls and shared state
3. Data elements → Commands
4. Typical use → End-user programmability; supports dynamically changing set of capabilities

Example → Lisp, Word/Excel macros

Topology



Get Command from user (Burn, 50) (Check Status)
Stream
in: line of code
out: result of code executed
Interpret and Execute
Data Access
Data Access
in: variable updated - result of code executed
out: none
in: none
out: variable updated - result of code executed
Interpreter State

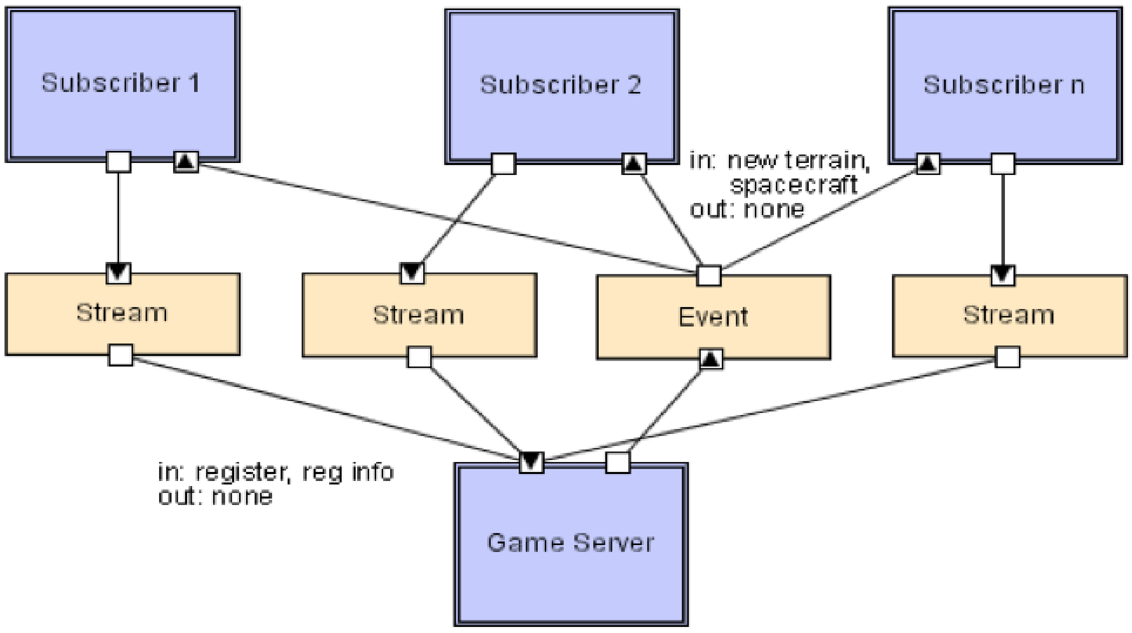
Lecture 14 - Basic Concepts & Styles - IV

▼ Describe the publish-subscribe style

Subscribers register/deregister to receive specific messages or specific content (sync and async)

1. Components → Publishers, subscribers, proxies
2. Connectors → Network protocol
3. Data elements → Subscription, notifications, published information
4. Typical use → Graphical user interface programming, multiplayer network based games

Topology



▼ What is a software middleware?

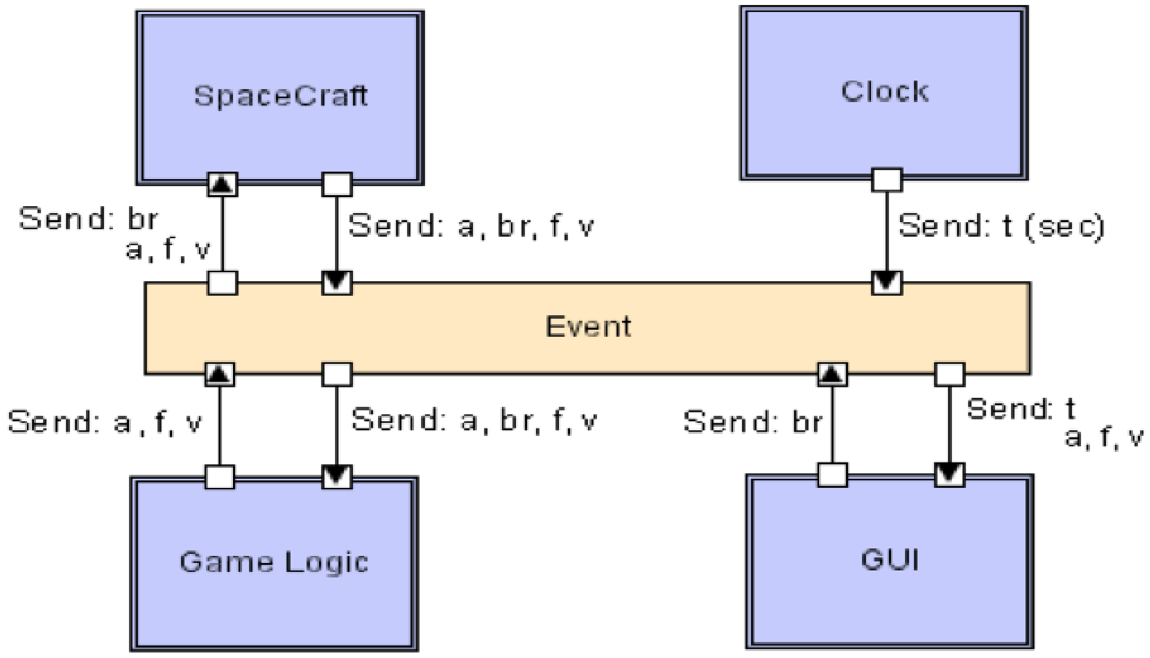
It is a software that provides services to software applications beyond those available from the operating system (Software glue)

▼ Describe the event-based style

Independent components asynchronously emit and receive events communicated over event buses

1. Components → Independent event generators and/or consumers
2. Connectors → Event buses (at least one)
3. Data elements → Events - data sent over the event bus
4. Typical use → Graphical user interface programming, mobile applications

Topology

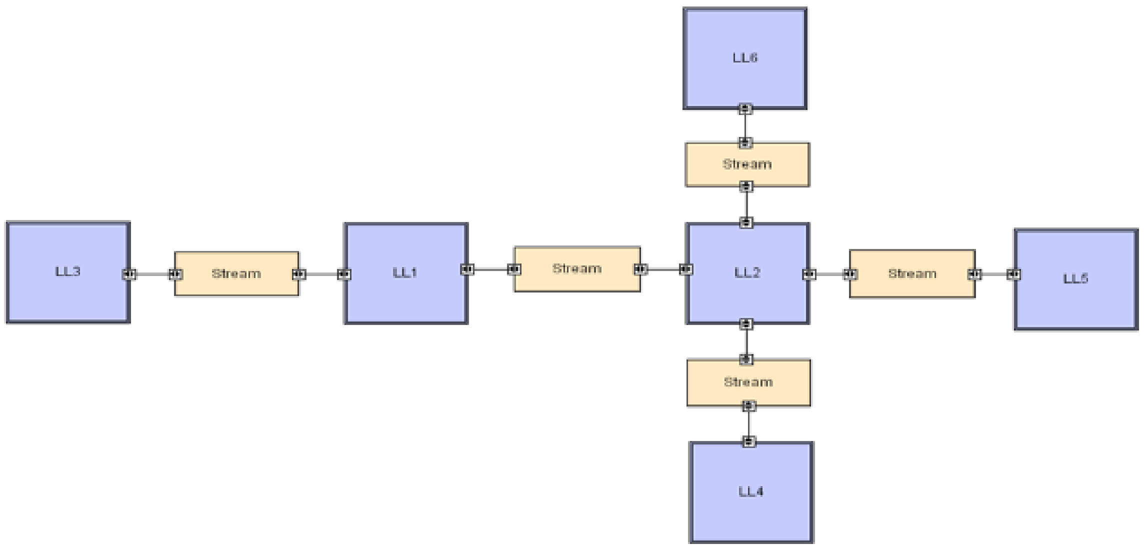


▼ Describe the peer-to-peer style

State and behavior are distributed among peers which can act as either clients or servers

1. Components → Nodes (Peers) (Independent from each other)
2. Connectors → Network protocol
3. Data elements → Network protocols
4. Typical use → Share resources and help computers and devices work collaboratively

Topology



LL1
Stream
LL5

▼ What is the difference between BitTorrent and Peer-to-peer?

BitTorrent is very different from P2P sharing protocols because it does not have any search functionality in the protocol and no content localization

▼ How BitTorrent works?

A peer is in the seed state if it has the complete file and is uploading to leechers (should be at least on seeder). The leecher state is when you download the file.

BitTorrent is a hybird of client-server and peer-to-peer

▼ Does BitTorrent have a centralized server?

Yes, and it is called "tracker". And its responsibility is to help peers find other peers

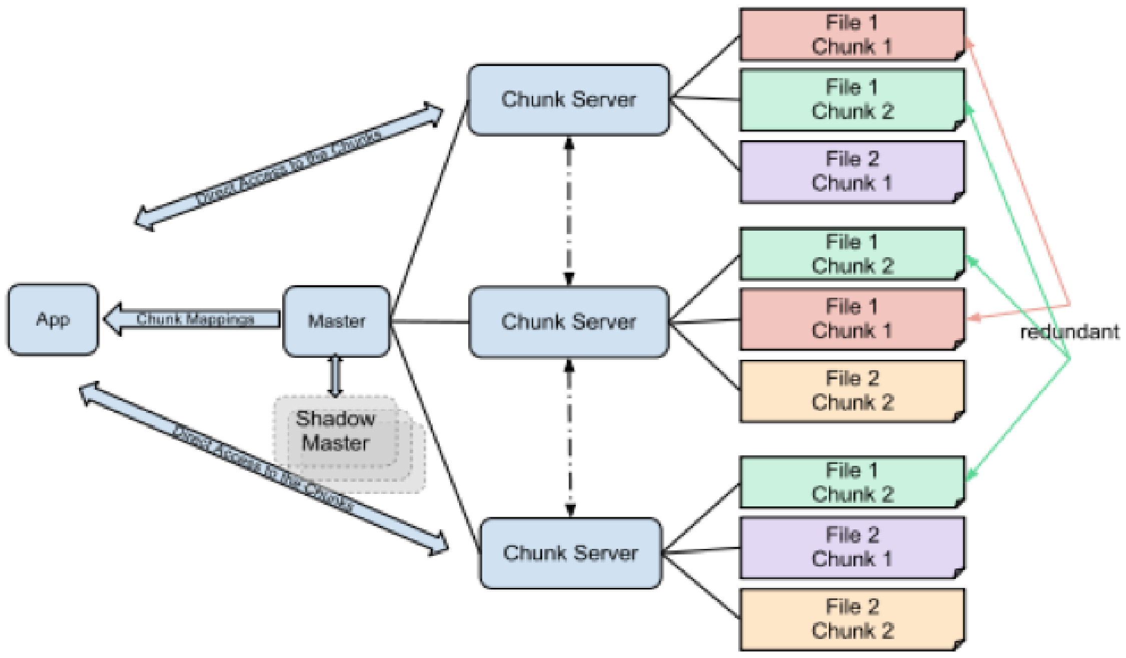
▼ What is google file system (GFS)?

It is a system designed to provide efficient reliable access to data using large clusters of commodity hardware

▼ What are the contents of google file system?

1. Chunk server → Split file into chunks
2. Master → Has access to chunks, stores metadata

3. Client library → Can talk to the master to find chunk servers



▼ What is map-reduce?

MapReduce is a framework which we can write applications to process huge amounts of data in parallel.

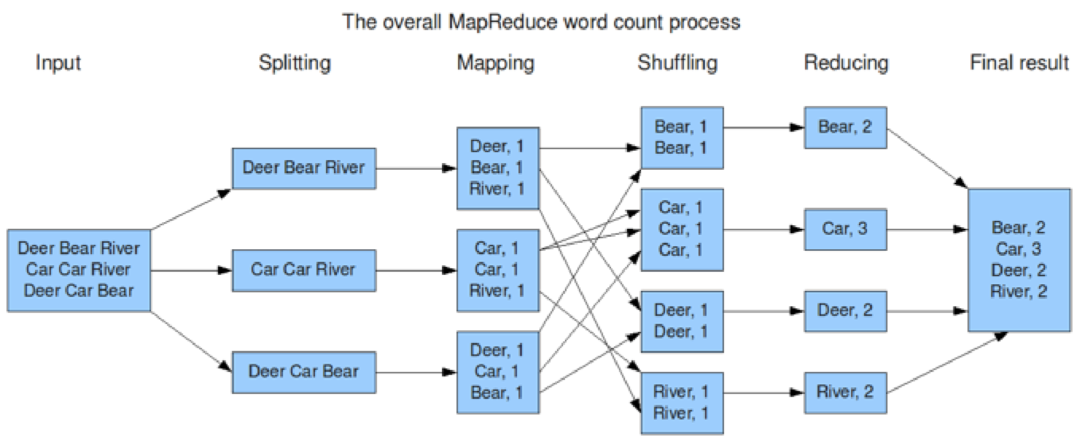
▼ When to use map-reduce?

1. To process big data
2. Index google search
3. Spam detection for e-mail
4. Data mining
5. Ad optimization

▼ How map-reduce works?

Hadoop map reduce operate in three main steeps

1. Mapping → Split the string into individual tokens
2. Shuffle → Values are sorted in an alphabetical order
3. Reducer → Values of the keys are added up



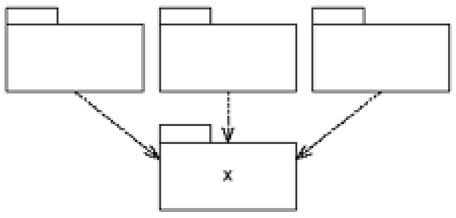
Lecture 15 - Principles of Package Design

▼ What are the principles of package cohesion?

1. The common closure principle (CCP) → Classes that change together should belong together
2. The common reuse principle (CRP) → Classes that aren't reused together should not be grouped together
3. Reuse-release equivalence principle (REP) → If you make something reusable, release the whole thing as a single item

▼ What does a stable packages means?

It means that the packages is hard to change



▼ How to calculate the stability of a package?

Ca → Incoming dependencies

Ce → Outgoing dependencies

1 = Stable

0 = Instable

$$S = \frac{Ca}{Ca + Ce}$$

▼ What are the principles of package coupling?

1. Acyclic dependencies principle (ADP) → The dependencies between packages must not form cycles
2. Stable dependencies principle (SDP) → The dependencies between packages should be in the direction of the stability of the packages (should depend on more stable packages)
3. Stable abstractions principle (SAP) → Stable packages should be abstract packages, while instable package should be concrete

▼ What is abstractness matrix?

It is the ratio of abstract classes in a package to the total number of classes in the package

0 = Has no abstract classes

1 = Has only abstract classes

Lecture 16 - Principles of Class Design - I

▼ What are the four principles of class design (SOLID) ?

1. Single-responsibility principle
2. Ope closed principle
3. Liskov substitution principle
4. Interface segregation principle
5. Dependency inversion principle

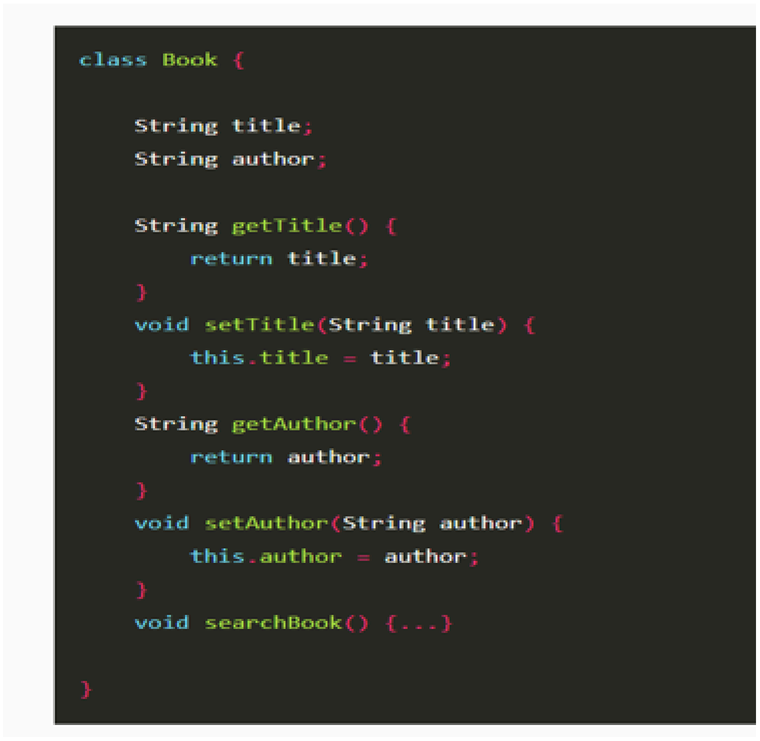
▼ What is the single-responsibility principle?

A class should have one and only one reason to change (should only have one job)

▼ Why it is important to have only one responsibility for a class?

Because it reduces the dependency to this class. Also, you can save a lot of testing time and create a more maintainable software

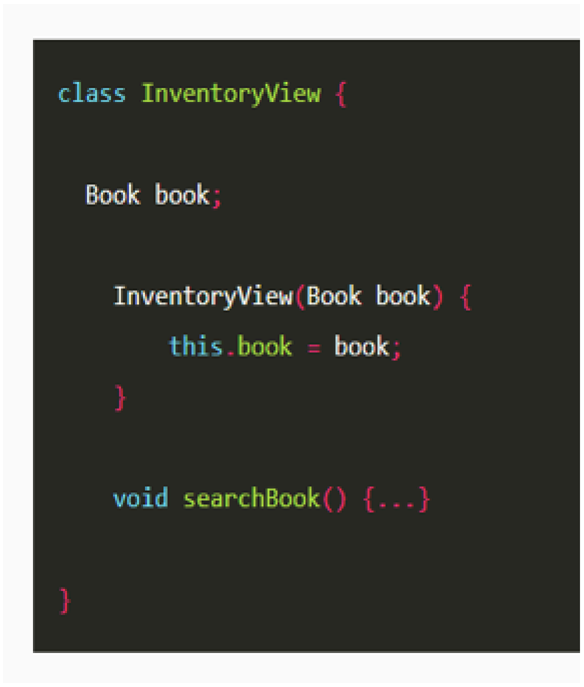
▼ Identify the issue in the given code



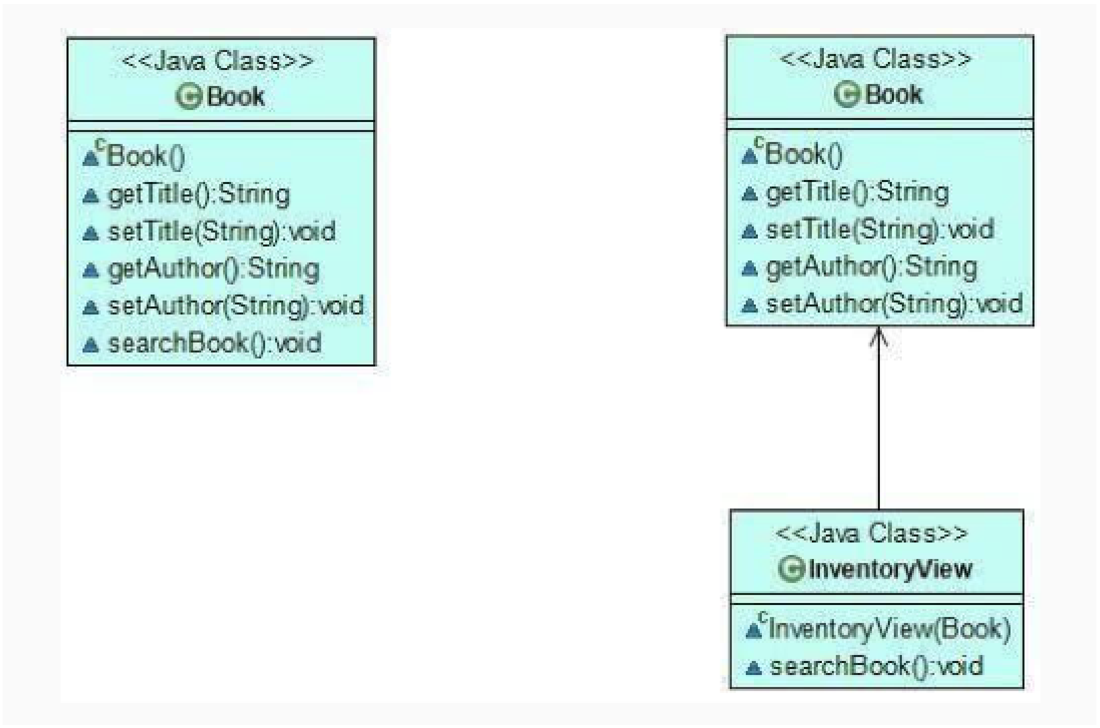
```
class Book {

    String title;
    String author;

    String getTitle() {
        return title;
    }
    void setTitle(String title) {
        this.title = title;
    }
    String getAuthor() {
        return author;
    }
    void setAuthor(String author) {
        this.author = author;
    }
    void searchBook() {...}

}
```

This class violates the single-responsibility principle because it has two responsibilities (initialize the object for a book, and searches for a book in the inventory)

We can solve this issue by creating another class which will be responsible for checking the inventory, and by that the "Book" class will have only one responsibility



```
class InventoryView {

  Book book;

    InventoryView(Book book) {
        this.book = book;
    }

    void searchBook() {...}

}
```



▼ What is open closed principle?

A module should be opened for extension but closed for modification. You should never modify a class that already exist, you can add new feature by extending your code rather than modifying the original class.

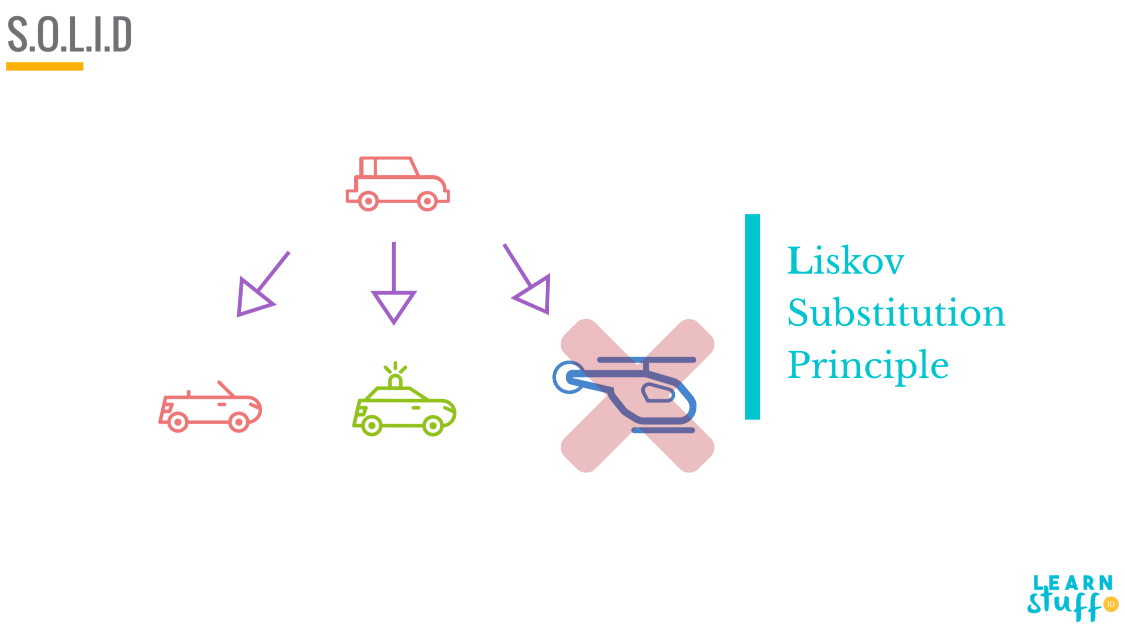
Lecture 17 - Principles of Class Design - II

▼ What is liskov substitution principle?

An object of a superclass should be replaceable by objects of its subclasses without causing issues in the application (A child class should never change the characteristics of its parent class)

▼ How can you implement liskov substitution principle?

By paying attention to the correct inheritance hierarchy



In this example all these objects are vehicle, but the issue is that all of them have wheels excluding the helicopter. Hence, if we have a method called "changeWheels()" we can't use it with the helicopter object that is why this example has a bad inheritance hierarchy. We can add additional layer with two classes one for vehicles with wheels and the other for vehicles without wheels

▼ What is interface segregation principle?

It state that interfaces shouldn't include too many functionalities so classes will not be forced to implement unnecessary methods

▼ How to implement interface segregation principle?

Try to create smaller interfaces with specific functionalities (Increase reusability)

▼ What is the goal of dependency inversion principle?

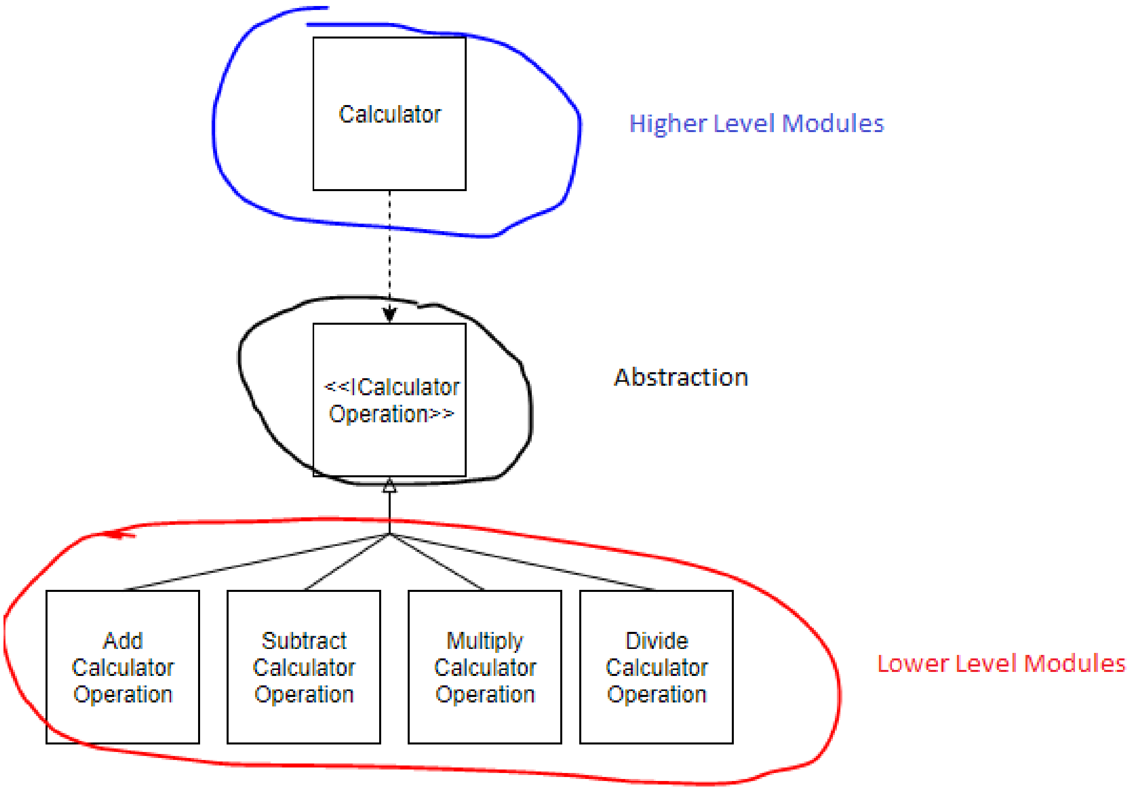
The goal is to avoid tightly coupled code (this principle is a combination of open/close and liskov substitution)

1. High-level modules should not depend on low-level modules both should depend on abstraction
2. Abstraction should not depend on details. Details should depend on abstraction

High-level modules →

▼ How to implement the dependency inversion principle?

You need to create and abstract layer for the low-level classes, so that high-level classes can depend on this abstract layer instead of depending on the low-level classes directly


Calculator
Higher Level Modules
<<ICalculator Operation>>
Abstraction
Add Calculator Operation
Subtract Calculator Operation
Multiply Calculator Operation
Divide Calculator Operation
Lower Level Modules

▼ What are the effect on the code when implementing SOLID principles?

It will increase the overall complexity of the code, but the design will be more flexible

Lecture 18 - Design Patterns - I

▼ What are design patterns?

Design patterns are description of communicating objects and classes that are customized to solve a general design problem in a particular context

▼ What are the types of design patterns?

1. Creational → Used to create objects in flexible or constrained ways
2. Structural → Used to describe the organization of objects and how classes and objects are composed to form larger structures
3. Behavioral → Capturing behavior among a collection of objects during execution

▼ List examples about creational design patterns

1. Factory
2. Abstract factory
3. Singleton
4. Prototype
5. Builder

▼ List examples about structural patterns

1. Adapter
2. Proxy
3. Bridge
4. Facade
5. Decorator

▼ List examples about behavioral patterns

1. State
2. Observer
3. Iterator
4. Mediator

▼ What is a singleton design pattern?

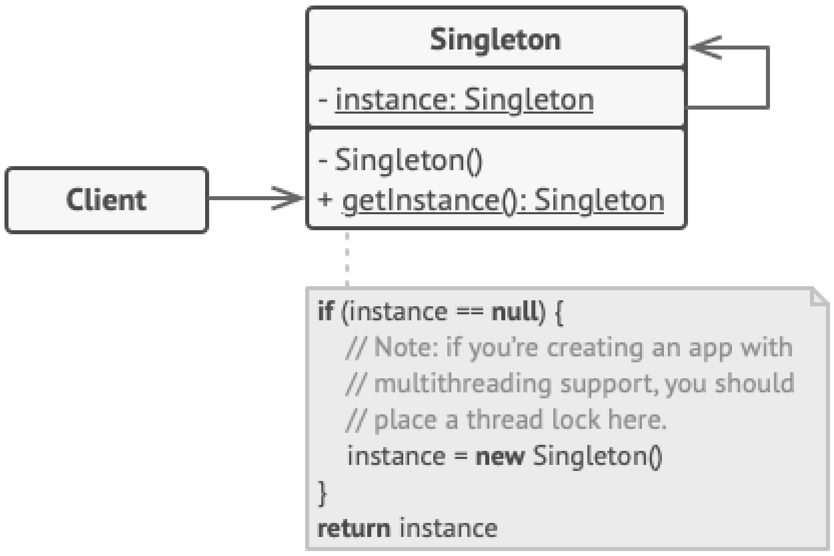
Design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

▼ What are the benefits of the singleton design pattern?

1. Controlled access to one instance
2. Permits a variable number of instances
3. More flexible than class operations

▼ How to implement singleton design pattern?

1. Make the default constructor private, to prevent other objects from using the `new` operator with the Singleton class.
2. Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.



```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

▼ When to use singleton design pattern?

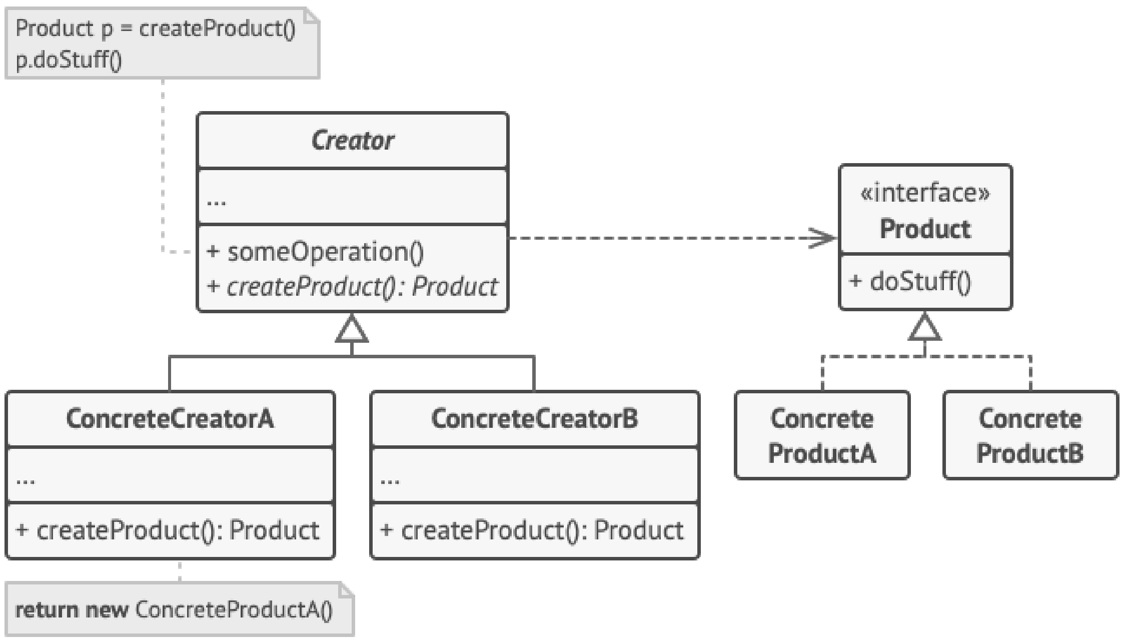
Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.

- ▼ What is a factory design pattern?

  Design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

- ▼ How to implement factory design pattern?

  The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method. Objects returned by a factory method are often referred to as products.



- ▼ When to use factory design pattern?

  Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.

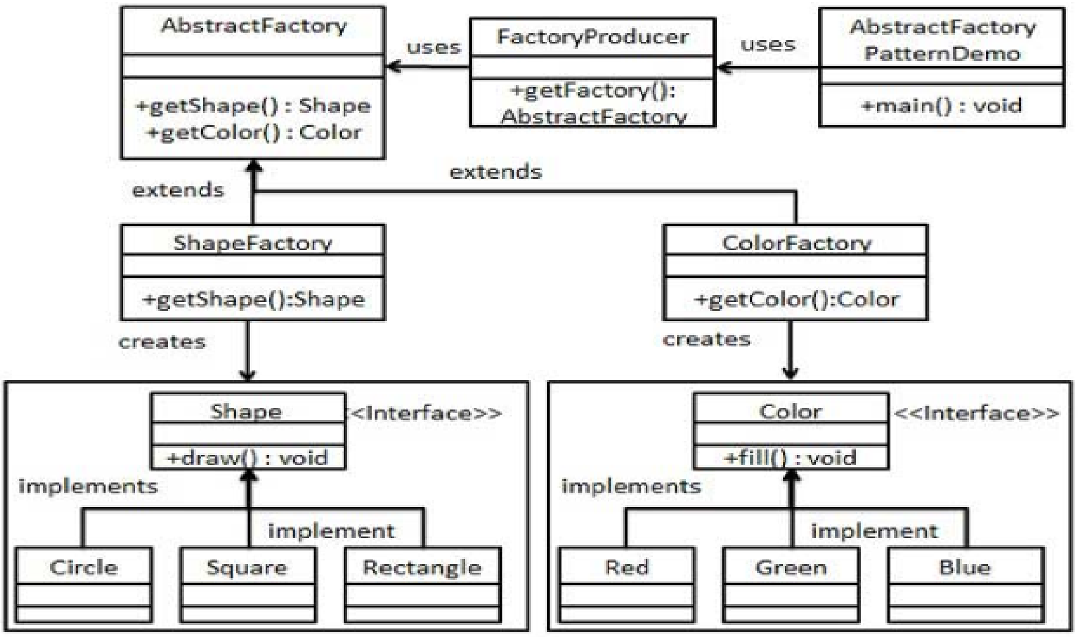
Lecture 19 - Design Patterns - II

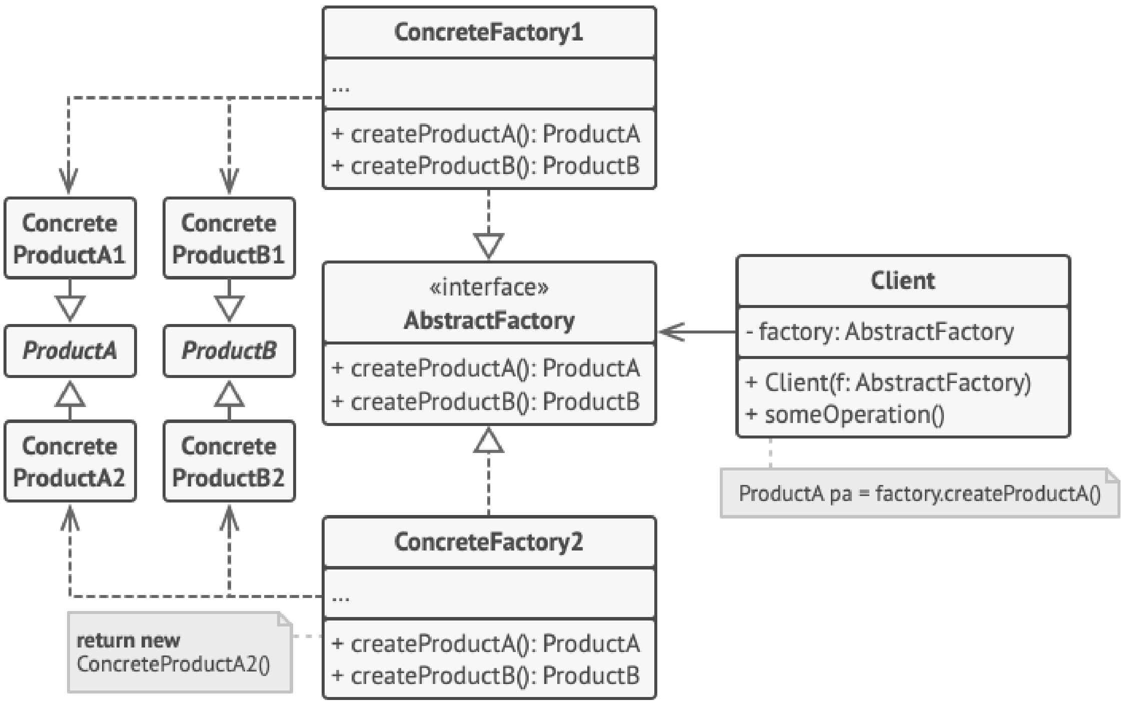
▼ What is an abstract factory pattern?

Design pattern that lets you produce families of related objects without specifying their concrete classes. (Factory of factories)

▼ How to implement an abstract factory?

Overall same as the factory but the difference is that abstract factory create multiple object types instead of one




ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
ProductA
ProductB
Concrete ProductA2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

▼ When to use abstract factory?

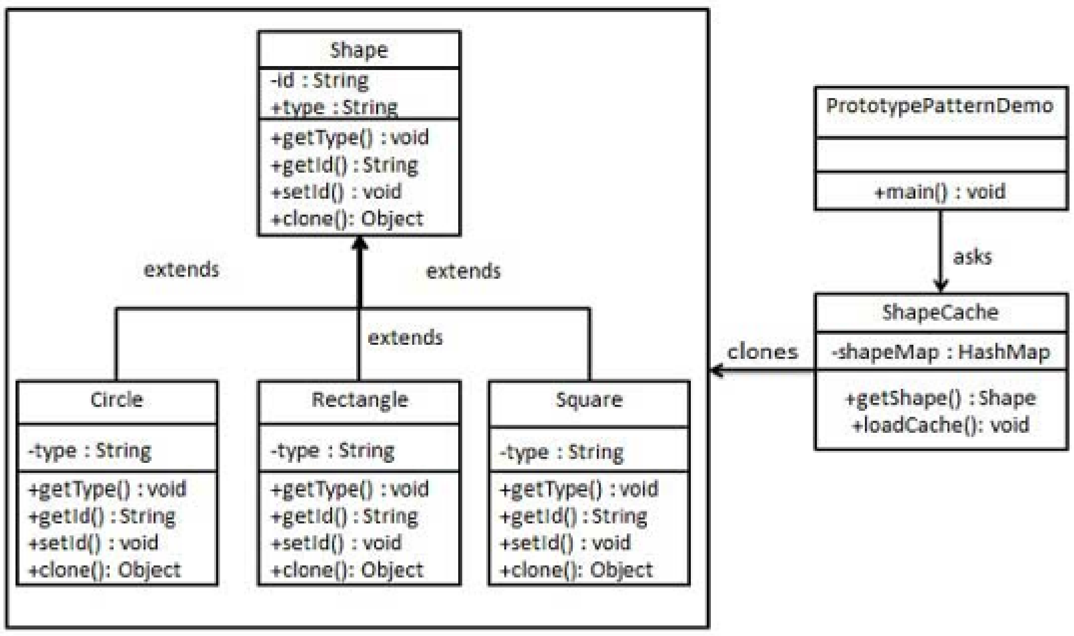
Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

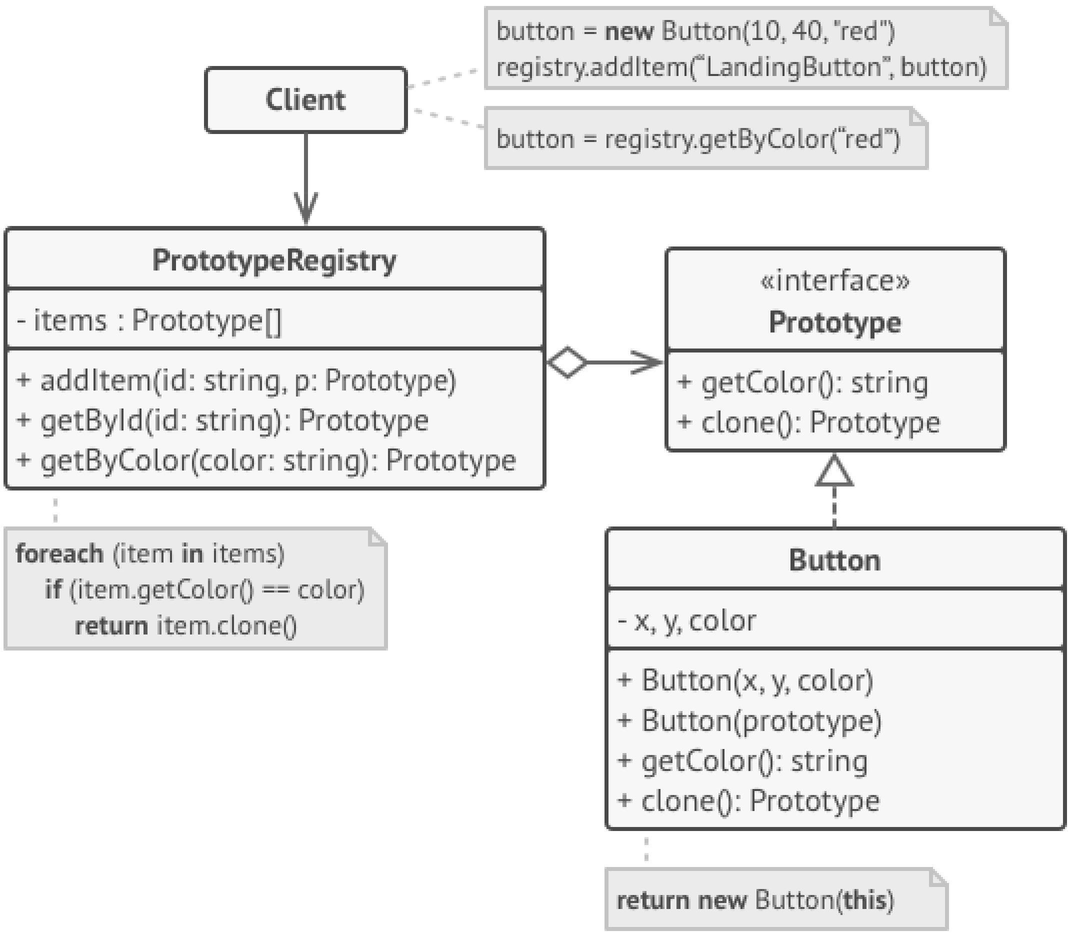
▼ What is a prototype design pattern?

Design pattern that lets you copy existing objects without making your code dependent on their classes.

▼ How to implement prototype design pattern?

The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single clone method.


Shape
-id : String
+type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
extends
extends
extends
Circle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Rectangle
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
Square
-type : String
+getType() : void
+getId() : String
+setId() : void
+clone(): Object
PrototypePatternDemo
+main() : void
asks
ShapeCache
-shapeMap : HashMap
+getShape() : Shape
+loadCache(): void
clones


Client
button = new Button(10, 40, "red")
registry.addItem("LandingButton", button)
button = registry.getByColor("red")
PrototypeRegistry
- items : Prototype[]
+ addItem(id: string, p: Prototype)
+ getById(id: string): Prototype
+ getByColor(color: string): Prototype
foreach (item in items)
if (item.getColor() == color)
return item.clone()
«interface»
Prototype
+ getColor(): string
+ clone(): Prototype
Button
- x, y, color
+ Button(x, y, color)
+ Button(prototype)
+ getColor(): string
+ clone(): Prototype
return new Button(this)

- ▼ When to use prototype design pattern?
    1. When creation of object is directly is costly
    2. Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.
    3. When you want to add or remove products at run-time

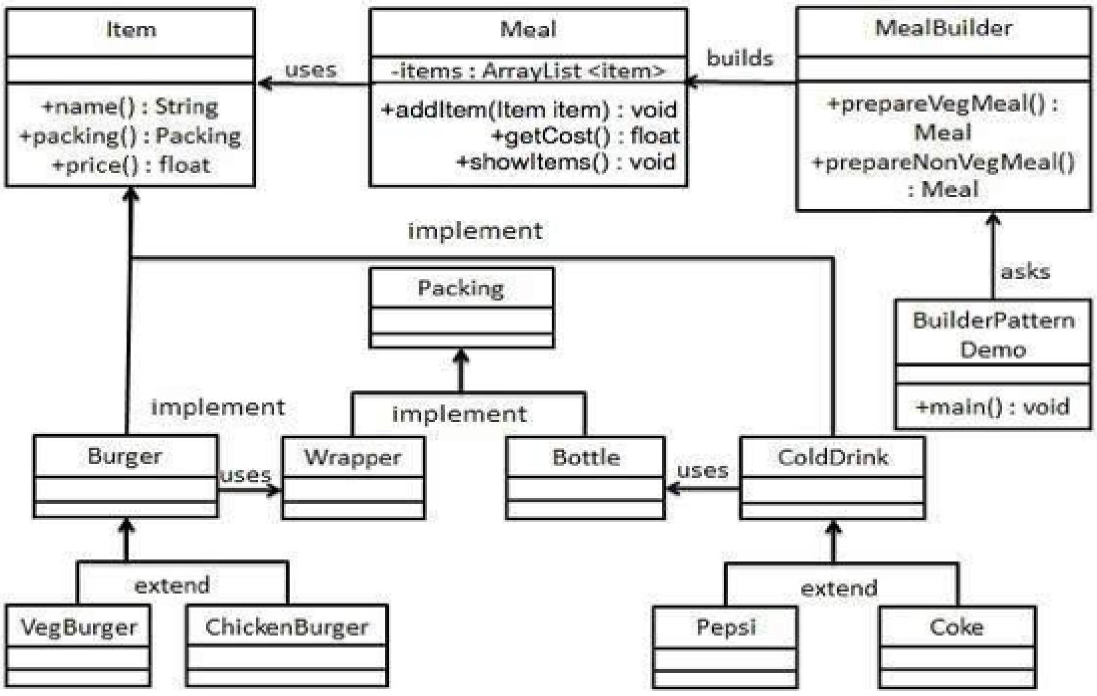
Lecture 20 - Design Patterns - III

- ▼ What is a builder design pattern?

  Design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

- ▼ How to implement a builder design pattern?

  The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.



- ▼ When to use builder design pattern?

Use the Builder pattern when you want your code to be able to create different representations of some product (for example, veg meal or non veg meal).

Lecture 21 - Structural Design Patterns - I

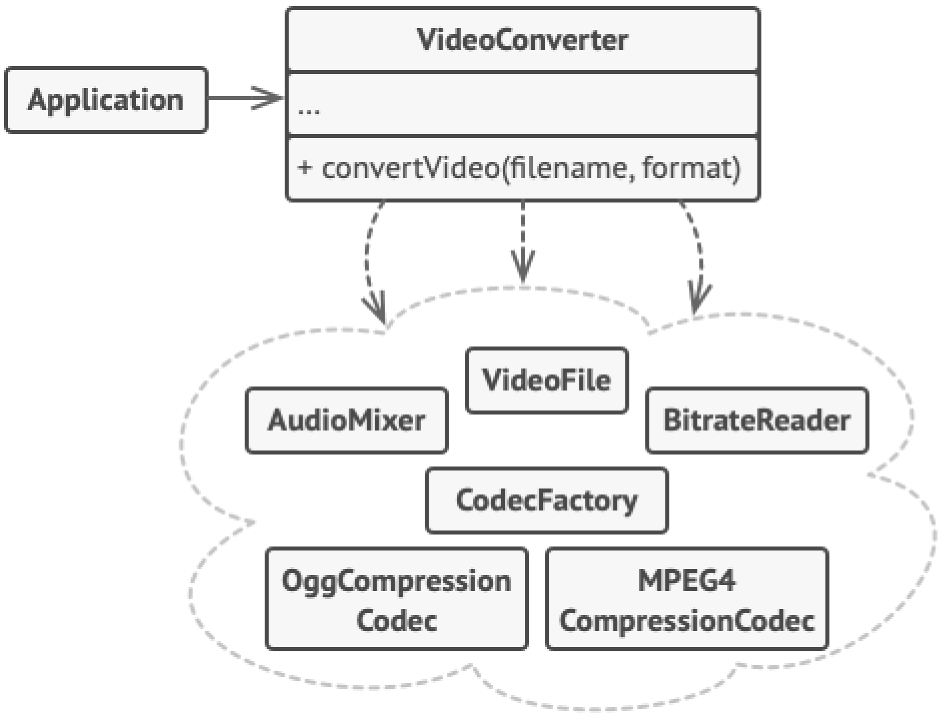
▼ What is a facade design pattern?

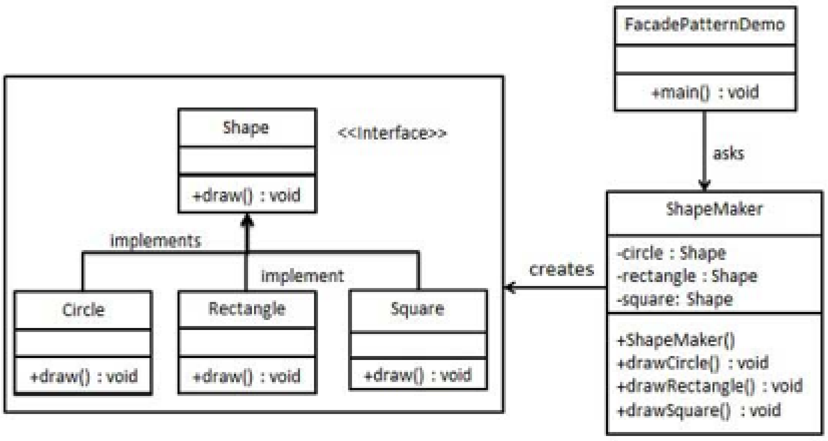
Is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

▼ How to implement a facade design pattern?

Declare and implement an interface. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.

(VideoConverter is a facade)





▼ When to use facade design pattern?

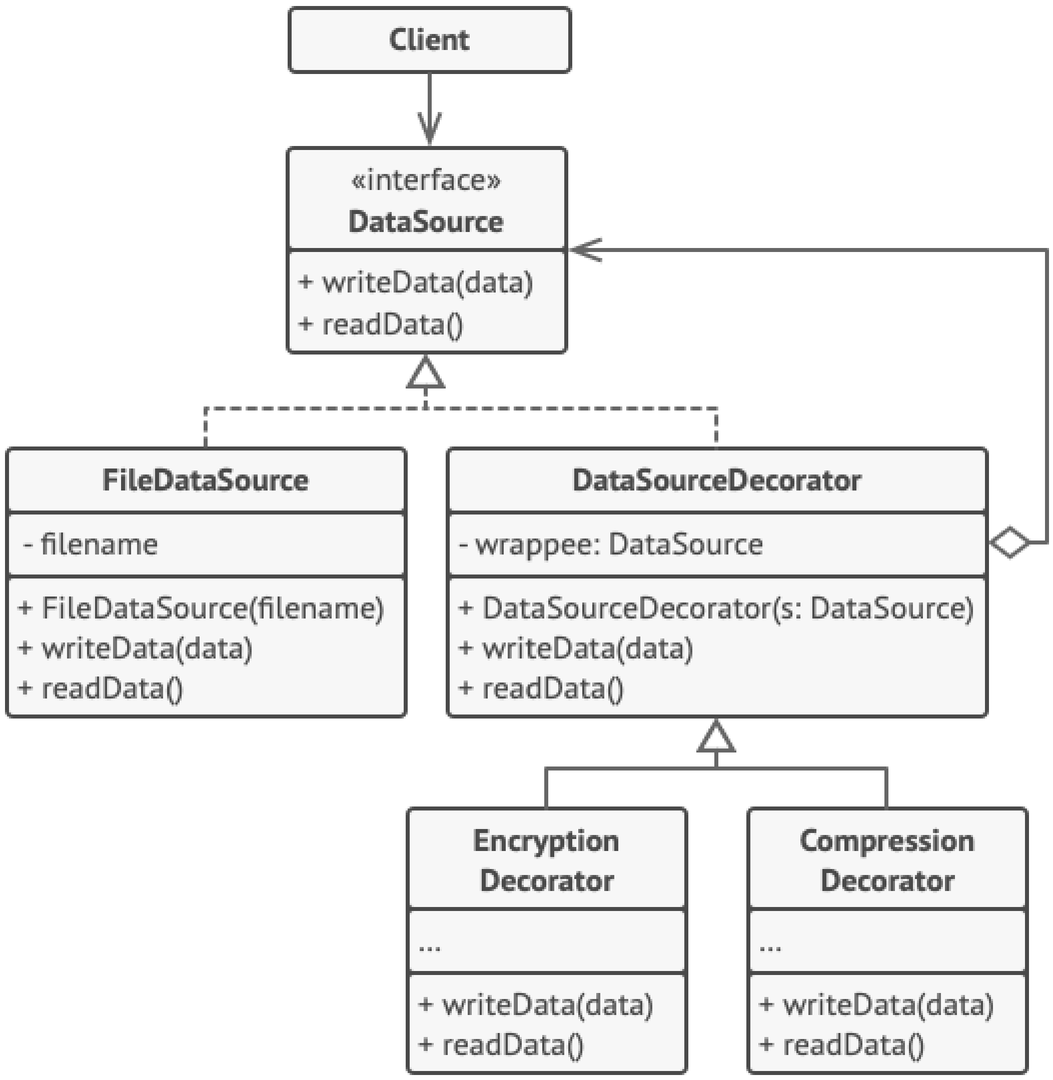
1. To hide the complexity of the system and provide easy interface to the client to use
2. To reduce dependencies between the client and the subsystem

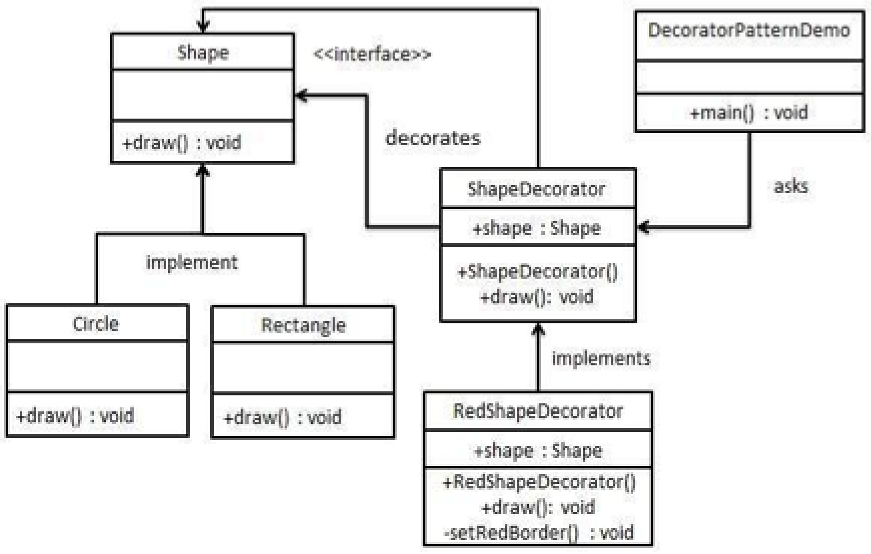
▼ What is a decorator design pattern?

Is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

▼ How to implement decorator design pattern?

Create a decorator class which wraps the original class and provides additional functionality





▼ When to use decorator design pattern?

1. Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
2. Use the pattern when it is not possible to extend an object's behavior using inheritance.

Lecture 22 - Structural Design Patterns - II

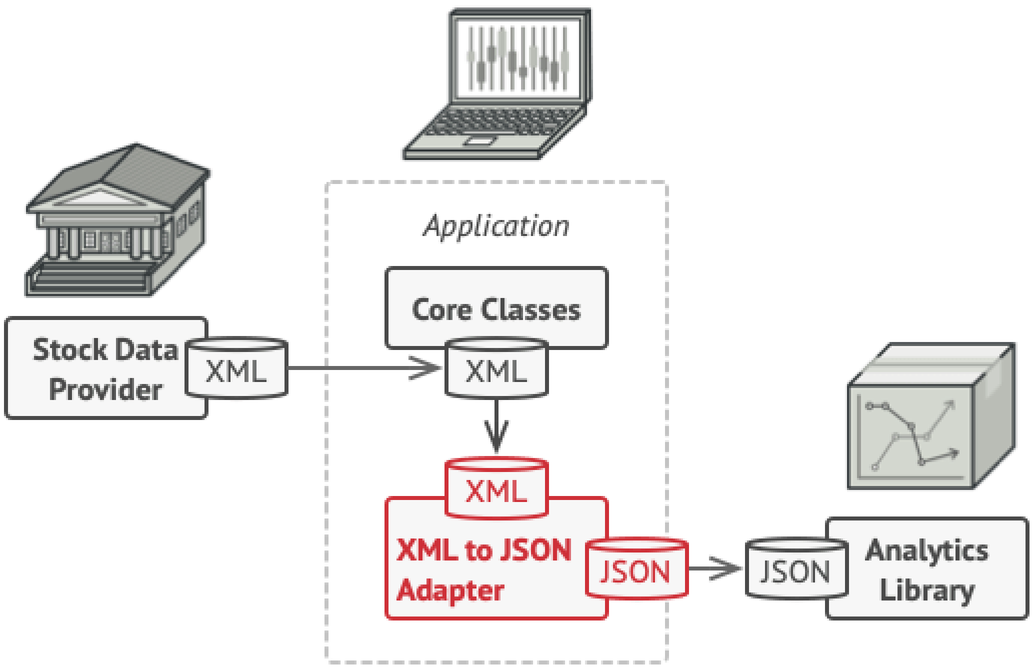
▼ What is an adapter design pattern?

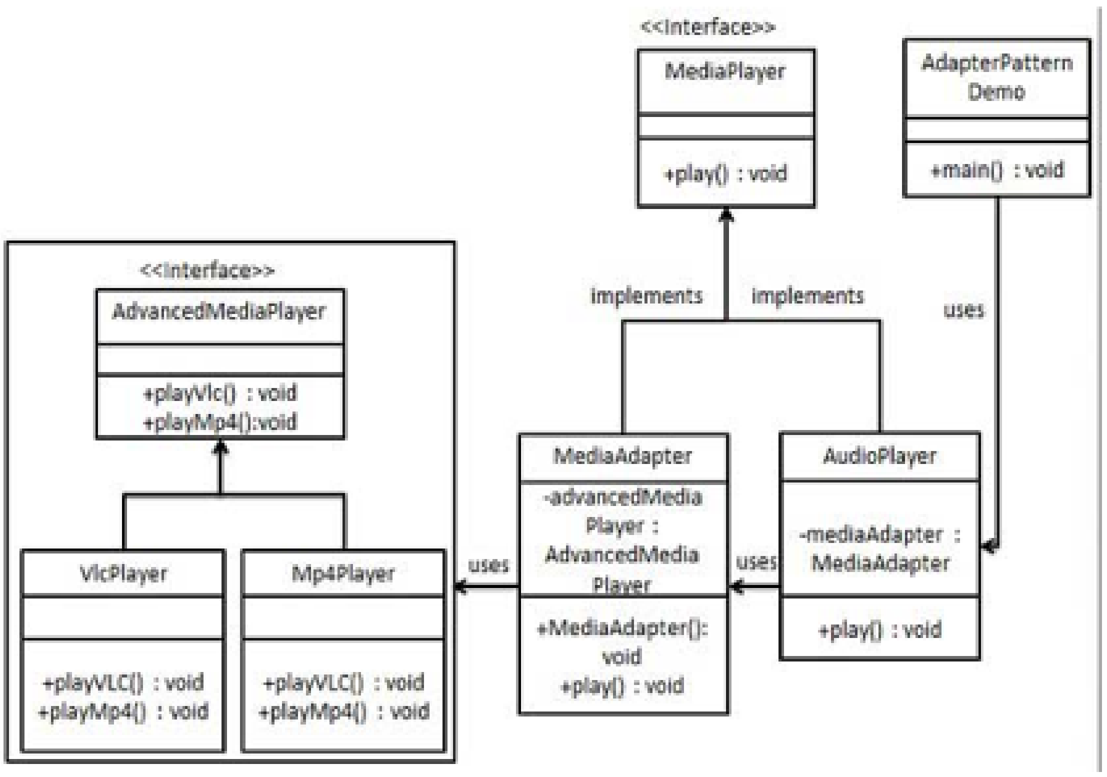
It is a structural design pattern that allows objects with incompatible interfaces to collaborate. (bridge between two interfaces)

▼ How to implement an adapter design pattern?

You can create an adapter. This is a special object that converts the interface of one object so that another object can understand it.

(The adapter link the stock data provider with the analytics library by changing the data format from XML to JSON)





▼ When to use adapter design pattern?

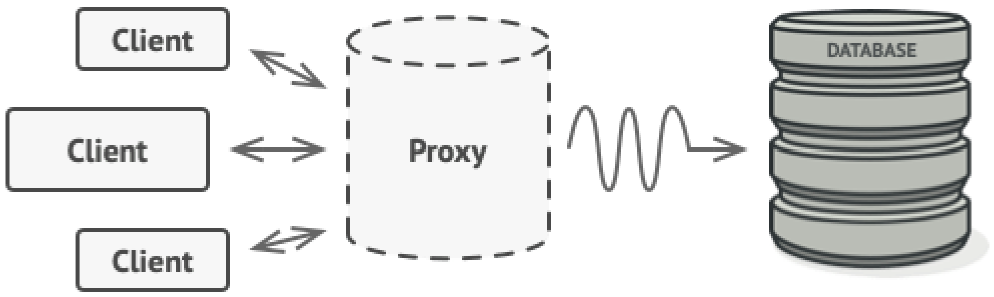
1. When you try to compatible two incompatible interfaces

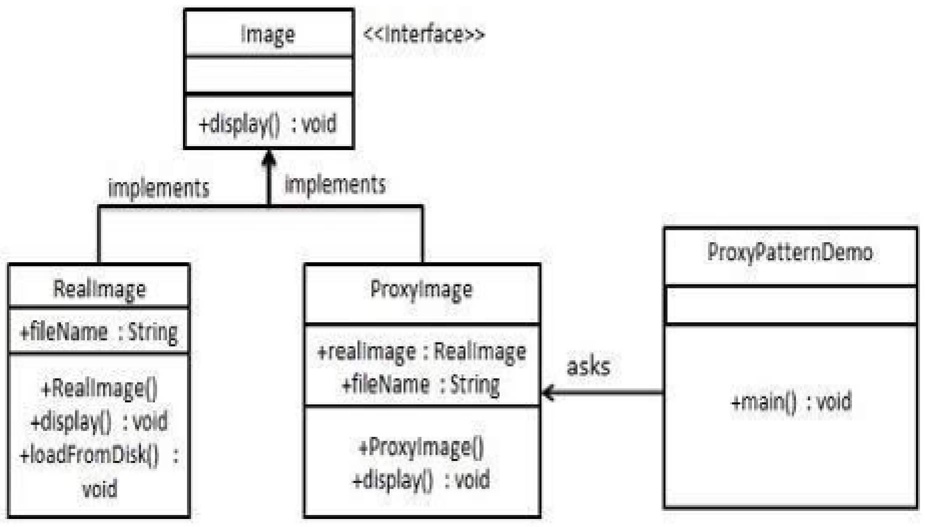
▼ What is a proxy design pattern?

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

▼ How to implement a proxy design pattern?

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.





▼ When to use proxy design pattern?

1. When you want to control access to an object

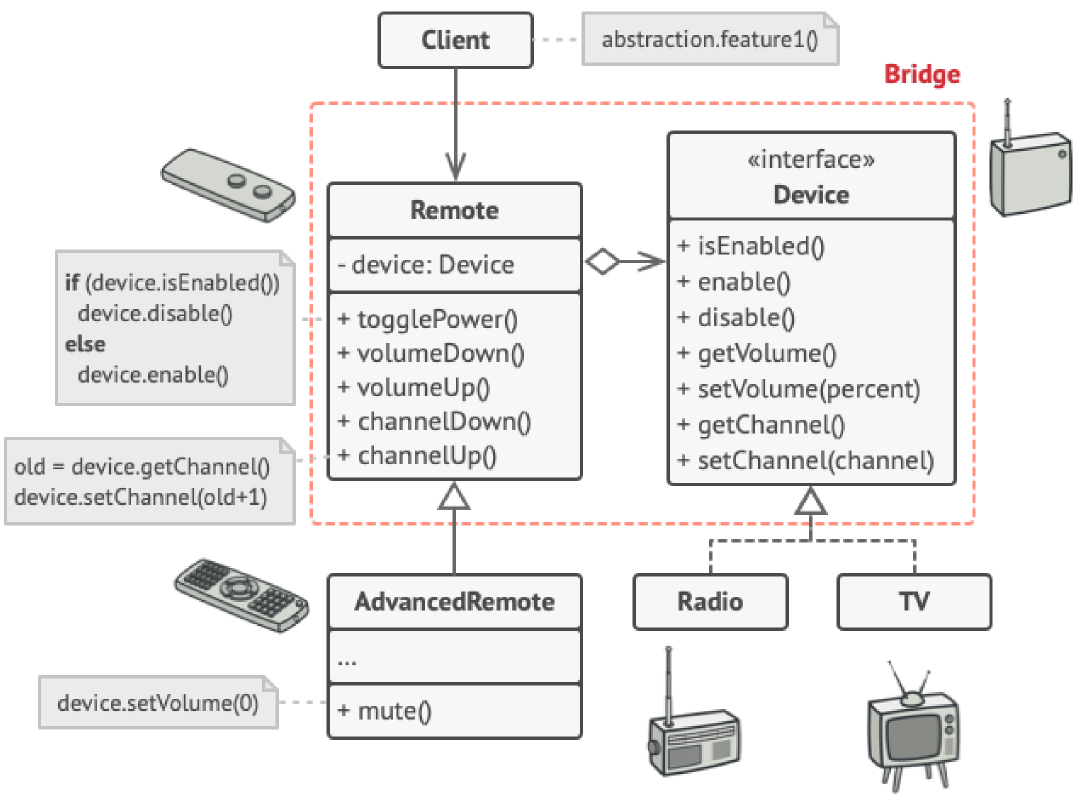
▼ What is bridge design pattern?

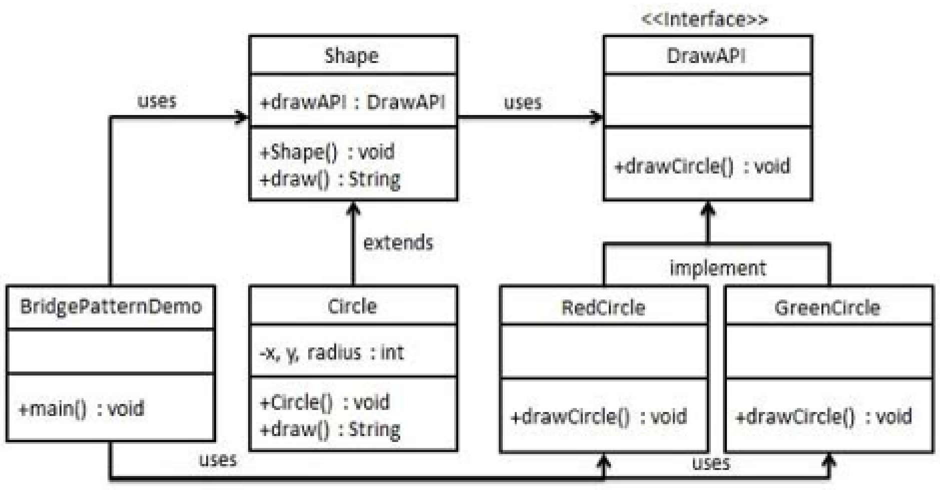
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

▼ How to implement bridge design pattern?

we divide the classes into two hierarchies:

- Abstraction (Device)
- Implementation (Remote)





▼ When to use bridge design pattern?

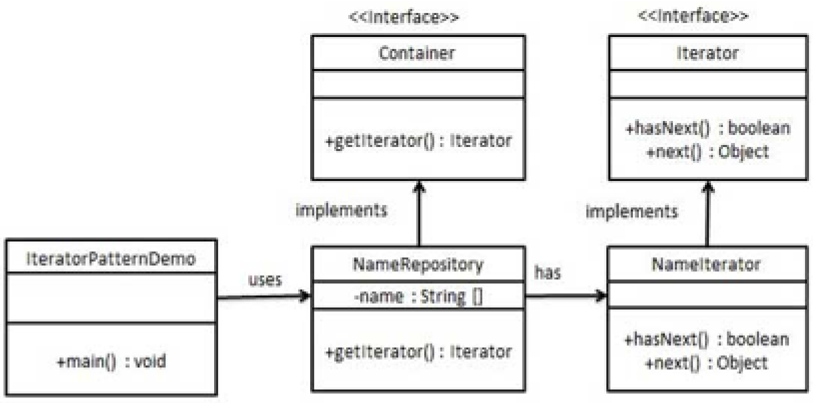
Lecture 23 - Behavioral Design Patterns - II

▼ What is iterator design pattern?

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

▼ How to implement iterator design pattern?

Create an iterator interface which declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc. Then implement it whenever you need to use it.



▼ When to use iterator design pattern?

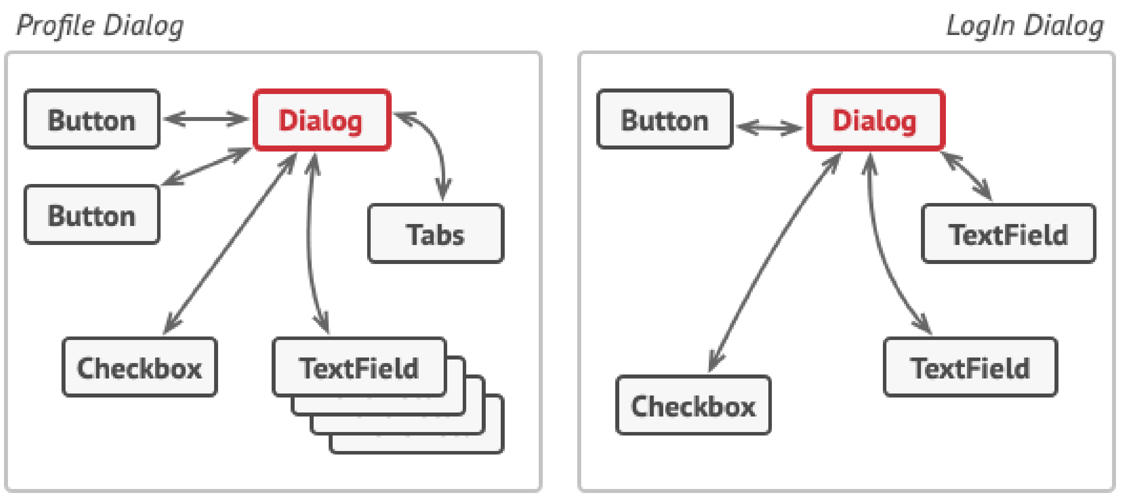
1. To access elements of a collection object in a sequential manner
2. Use the pattern to reduce duplication of the traversal code across your app.

▼ What is mediator design pattern?

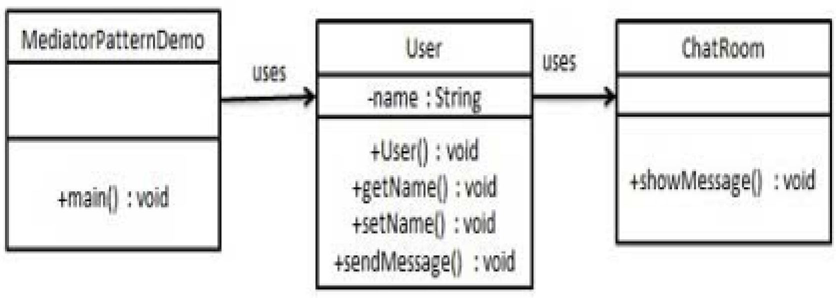
Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

▼ How to implement mediator design pattern?

The Mediator pattern suggests that you should manage all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components. As a result, the components depend only on a single mediator class



(Users do not have to have dependencies between each other, the chatRoom(mediator) will handle it)



▼ When to use mediator design pattern?

1. Use the Mediator pattern when it's hard to change some of the classes because they are tightly coupled to a bunch of other classes.
2. Reduce dependencies between components

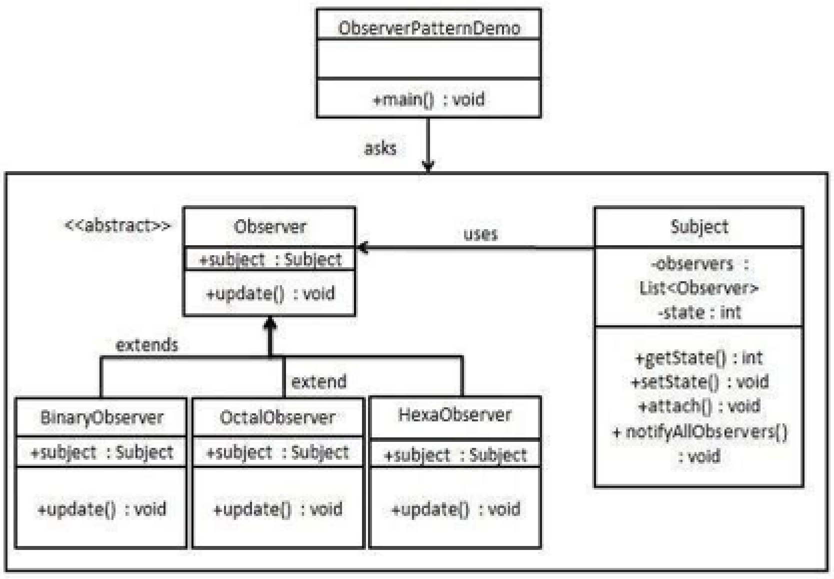
▼ What is the observer design pattern?

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing (when there is one-to-many relationship between objects)

▼ How to implement observer design pattern?

Observer design pattern has three classes

1. Subject → Object having methods to attach and detach observers to a client object
2. Observer → An object that can access and manipulate the state
3. Client → Whoever interact with the system



All the observers (Binary, Octal and Hexa) can access the state. And whenever the state value will change, it will change in the all observers