

Design Patterns

Origins....

- Christopher Alexander, an architect, studied ways to improve the process of designing buildings and urban areas
- "Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution."
 - Christopher Alexander , The Timeless Way of Building
- Reusing solutions that have worked leads to more productivity and offers other advantages of re-use such as flexibility, efficiency etc
- Object oriented systems that have been well designed have been seen to have recurring patterns of classes and objects

Pattern and Types of patterns

- A pattern is a solution to a problem in a given context. As it becomes reliable it can be followed over and over.
- Conceptual Pattern

“A conceptual pattern is a pattern whose form is described by means of the terms and concepts from an application domain.”
- Design Pattern

“A design pattern is a pattern whose form is described by means of software design constructs, for example objects, classes inheritance, aggregation and use-relationship.”
- Programming Pattern (Programming Idiom)

“A programming pattern is a pattern whose form is described by means of programming language constructs.”

-- Dirk Riehle and Heinz Züllighoven “Understanding and Using Patterns in Software Development”

“GoF” Design Patterns

- Design Patterns: Elements of Reusable Object-Oriented Software by the “Gof (Gang of Four)”
 - Erich Gamma
 - Richard Helm
 - Ralph Johnson
 - John Vlissides

Advantages

- They are reusable in multiple projects.
- They provide the solutions that help to define the system architecture.
- They capture the software engineering experiences.
- They provide transparency to the design of an application.
- They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
- Design patterns don't guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

When should we use the design patterns?

- We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).
- Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.
- Categorization of design patterns:
- Basically, design patterns are categorized into two parts:
 - Core Java (or JSE) Design Patterns.
 - JEE Design Patterns.

Core Java Design Patterns

- Creational Patterns
 - Patterns relating to the creation of objects
 - Example: Singleton
- Structural Patterns
 - Patterns relating to the structural relationship between objects
 - Example: Façade
- Behavioral Patterns
 - Patterns relating to communication between objects
 - Example: Observer

Classification

- Creational Patterns
 - Factory Pattern
 - Abstract Factory Pattern
 - Singleton Pattern
 - Prototype Pattern
 - Builder Pattern

Classification

- Structural Design Pattern
 - Adapter Pattern
 - Bridge Pattern
 - Composite Pattern
 - Decorator Pattern
 - Facade Pattern
 - Flyweight Pattern
 - Proxy Pattern

Classification

- Behavioural Design Pattern
 - Chain Of Responsibility Pattern
 - Command Pattern
 - Interpreter Pattern
 - Iterator Pattern
 - Mediator Pattern
 - Memento Pattern
 - Observer Pattern
 - State Pattern
 - Strategy Pattern
 - Template Pattern
 - Visitor Pattern

Singleton Pattern

- “Ensure a class only has one instance, and provide a global point of access to it.”
 - According to Design Patterns: Elements of Reusable Object-Oriented Software by the “Gof (Gang of Four)”
- In other words there should be only one instance of a class accessible by multiple users or multiple parts of application
- When this class is subclassed, the subclasses instances should not modify the base class code and make it NOT singleton.

Singleton Pattern: usage

- Situation where there is only one resource like print spooler or a database engine or the object with registry settings or network connection. In such case only one instance of the resource is desired.
- A stateless object that has only utility methods with parameters can also be designed as singleton.

- Use in JEE :

Most servlets run as Singletons in their servlet engines

Tell me why?

- We can get Singleton-like behavior with static fields and methods as well? Then why singleton classes?
- One of the advantages of creating singleton is lazy initialization that is you can initialize only when it is first used.
- In case design changes then Singleton can be easily changed to allow multiple instances without affecting other parts of the code.
- In case multiple instance are desired some implementations of the Singleton, allow the subclasses to override methods such that multiple instances of subclasses can be created (something not possible with static methods)

Tell me why?

- There are two forms of singleton design pattern
 - **Early Instantiation:** creation of instance at load time.
 - **Lazy Instantiation:** creation of instance when required.

Structure

Singleton
<code>private Singleton uniqueinstance</code>
<code>public static Singleton getInstance singletonOperation()</code>

Code skeleton

```
public class Singleton {
    private static uniqueinstance;

    private Singleton() {
        // initialize data
    }

    // Lazy initialization
    public static synchronized Singleton getInstance() {
        if (uniqueinstance ==null) {
            uniqueinstance = new Singleton();
        }
        return uniqueinstance ;
    }
    // rest of the code
}
```

Note that this implementation does not allow Singleton to be subclassed

Some pitfalls of Singleton

- If the singleton is allowed to be Serialized, then at some point we will end up getting 2 or more instances of singleton.
- If you allow subclasses to be created then, subclasses if not coded properly may end up creating multiple instances of base class through conversion.

Prototype Design Pattern

- Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement.**
- This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

Prototype Design Pattern

- Advantage of Prototype Pattern
 - The main advantages of prototype pattern are as follows:
 - It reduces the need of sub-classing.
 - It hides complexities of creating objects.
 - The clients can get new objects without knowing which type of object it will be.
 - It lets you add or remove objects at runtime.

Prototype Design Pattern

- Usage of Prototype Pattern
 - When the classes are instantiated at runtime.
 - When the cost of creating an object is expensive or complicated.
 - When you want to keep the number of classes in an application minimum.
 - When the client application needs to be unaware of object creation and representation.

Prototype Design Pattern



Factory Design Pattern

- A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.**
- In other words, subclasses are responsible to create the instance of the class.
- The Factory Method Pattern is also known as **Virtual Constructor.**

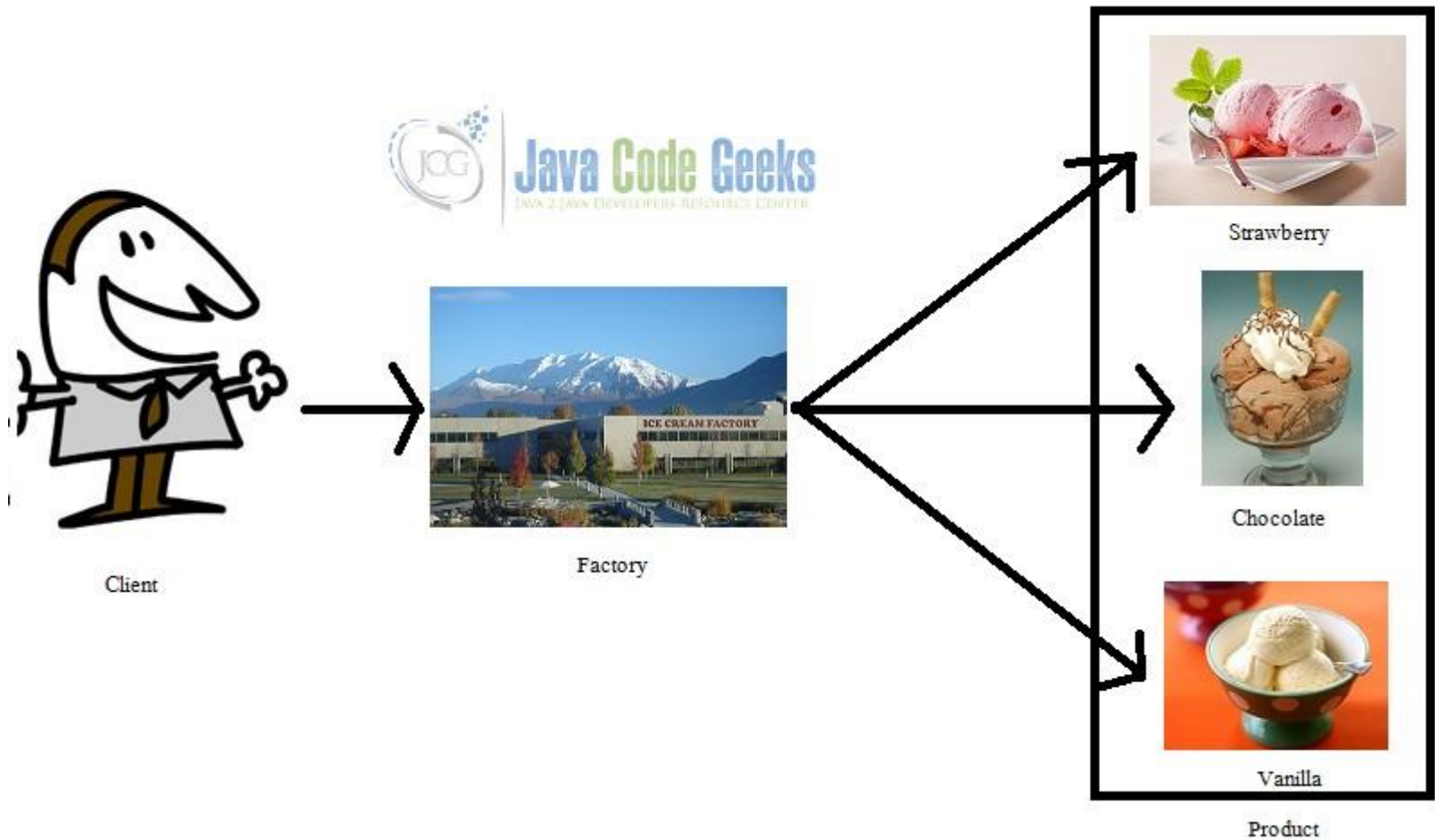
Factory Design Pattern

- Advantage of Factory Design Pattern
 - Factory Method Pattern allows the sub-classes to choose the type of objects to create.
 - It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Factory Design Pattern

- Usage of Factory Design Pattern
 - When a class doesn't know what sub-classes will be required to create
 - When a class wants that its sub-classes specify the objects to be created.
 - When the parent classes choose the creation of objects to its sub-classes.

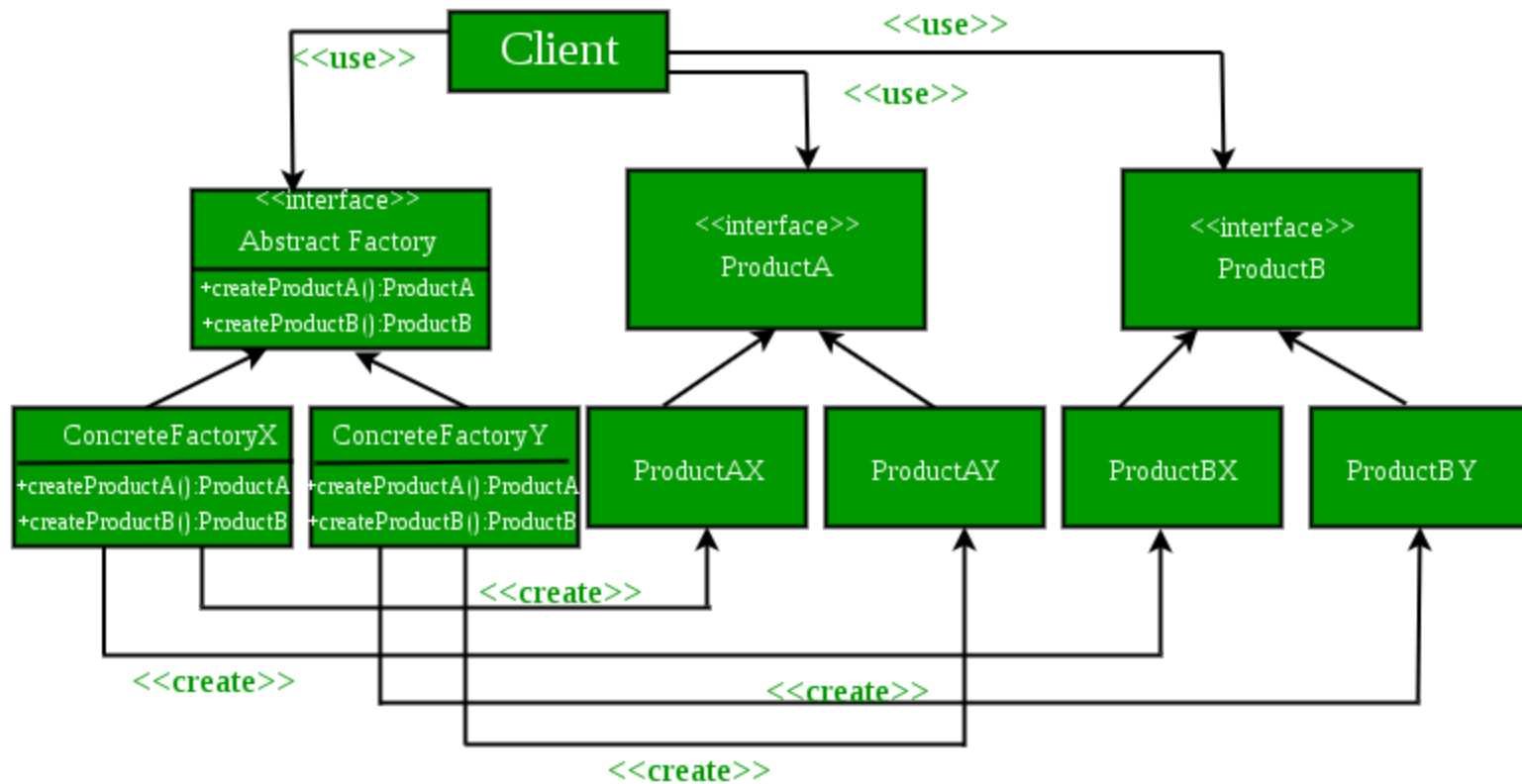
Factory Design Pattern



Abstract Factory Pattern

- Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**
- That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.
- An Abstract Factory Pattern is also known as **Kit.**

Abstract Factory Pattern



Abstract Factory Pattern

- Advantage of Abstract Factory Pattern
 - Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
 - It eases the exchanging of object families.
 - It promotes consistency among objects.

Abstract Factory Pattern

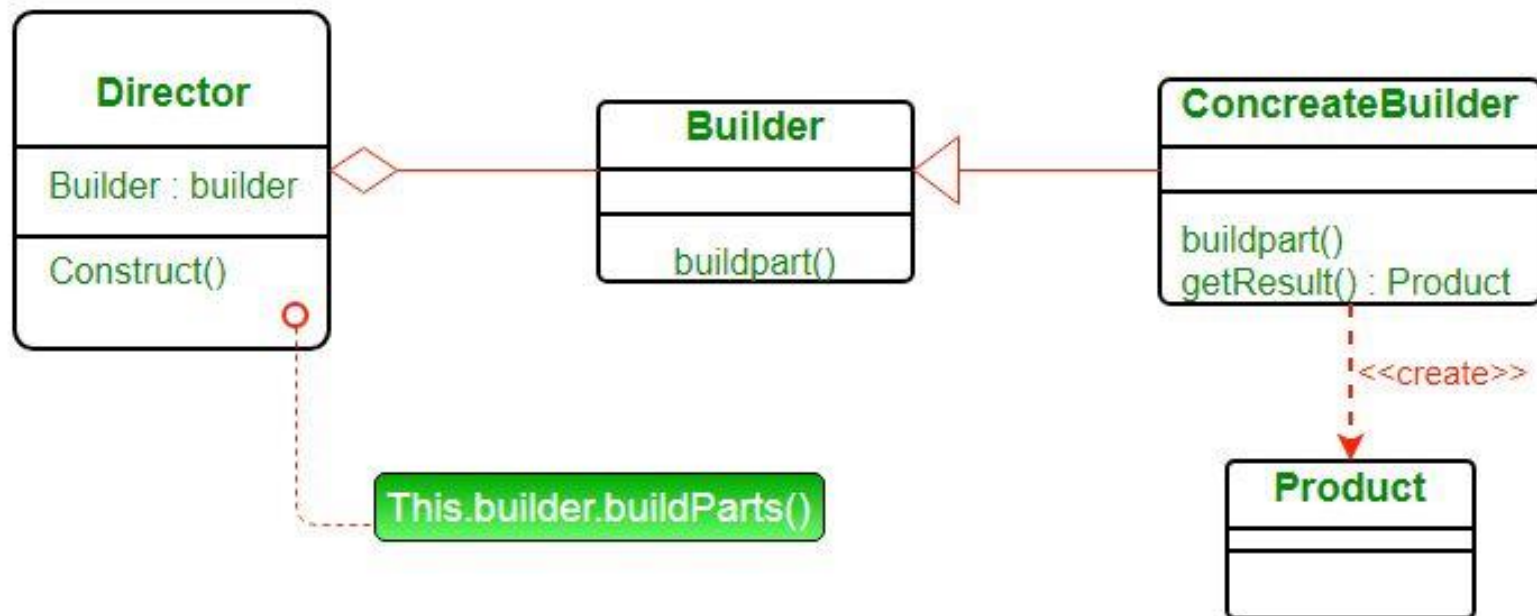
- Usage of Abstract Factory Pattern
 - When the system needs to be independent of how its object are created, composed, and represented.
 - When the family of related objects has to be used together, then this constraint needs to be enforced.
 - When you want to provide a library of objects that does not show implementations and only reveals interfaces.
 - When the system needs to be configured with one of a multiple family of objects.

Builder Design Pattern

- Builder Pattern says that "**construct a complex object from simple objects using step-by-step approach**"
- It is mostly used when object can't be created in single step like in the de-serialization of a complex object.
- The main advantages of Builder Pattern are as follows:
 - It provides clear separation between the construction and representation of an object.
 - It provides better control over construction process.
 - It supports to change the internal representation of objects.

Builder Design Pattern

UML diagram of Builder Design pattern



Builder Design Pattern

- **Product** – The product class defines the type of the complex object that is to be generated by the builder pattern.
- **Builder** – This abstract base class defines all of the steps that must be taken in order to correctly create a product.
- Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses.
- The GetProduct method is used to return the final product. The builder class is often replaced with a simple interface.

Builder Design Pattern

- **ConcreteBuilder** – There may be any number of concrete builder classes inheriting from Builder. These classes contain the functionality to create a particular complex product.
- **Director** – The director class controls the algorithm that generates the final product object. A director object is instantiated and its Construct method is called. The method includes a parameter to capture the specific concrete builder object that is to be used to generate the product.

Builder Design Pattern

- The director then calls methods of the concrete builder in the correct order to generate the product object. On completion of the process, the GetProduct method of the builder object can be used to return the product.
- **Disadvantages of Builder Design Pattern**
 - The number of lines of code increase at least to double in builder pattern, but the effort pays off in terms of design flexibility and much more readable code.
 - Requires creating a separate ConcreteBuilder for each different type of Product.

Object Pool Pattern

- Mostly, performance is the key issue during the software development and the object creation, which may be a costly step.
- Object Pool Pattern says that " **to reuse the object that are expensive to create**".
- Basically, an Object pool is a container which contains a specified amount of objects. When an object is taken from the pool, it is not available in the pool until it is put back. **Objects in the pool have a lifecycle: creation, validation and destroy.**
- A pool helps to manage available resources in a better way. There are many using examples: especially in application servers there are data source pools, thread pools etc.

Object Pool Pattern

- Advantage of Object Pool design pattern
- It boosts the performance of the application significantly.
- It is most effective in a situation where the rate of initializing a class instance is high.
- It manages the connections and provides a way to reuse and share them.
- It can also provide the limit for the maximum number of objects that can be created.

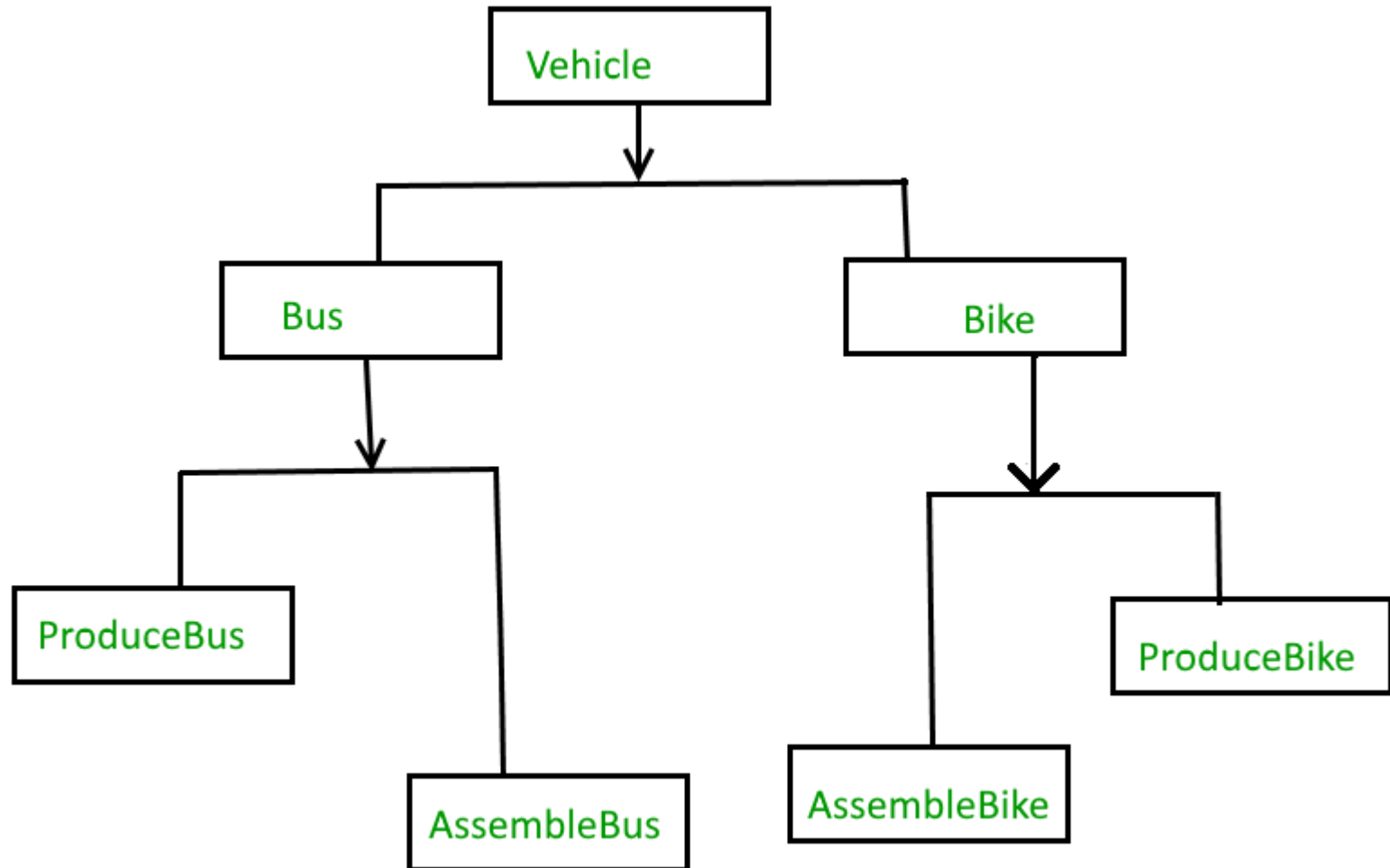
Object Pool Pattern

- Usage:
- When an application requires objects which are expensive to create.
Eg: there is a need of opening too many connections for the database then it takes too longer to create a new one and the database server will be overloaded.
- When there are several clients who need the same resource at different times.

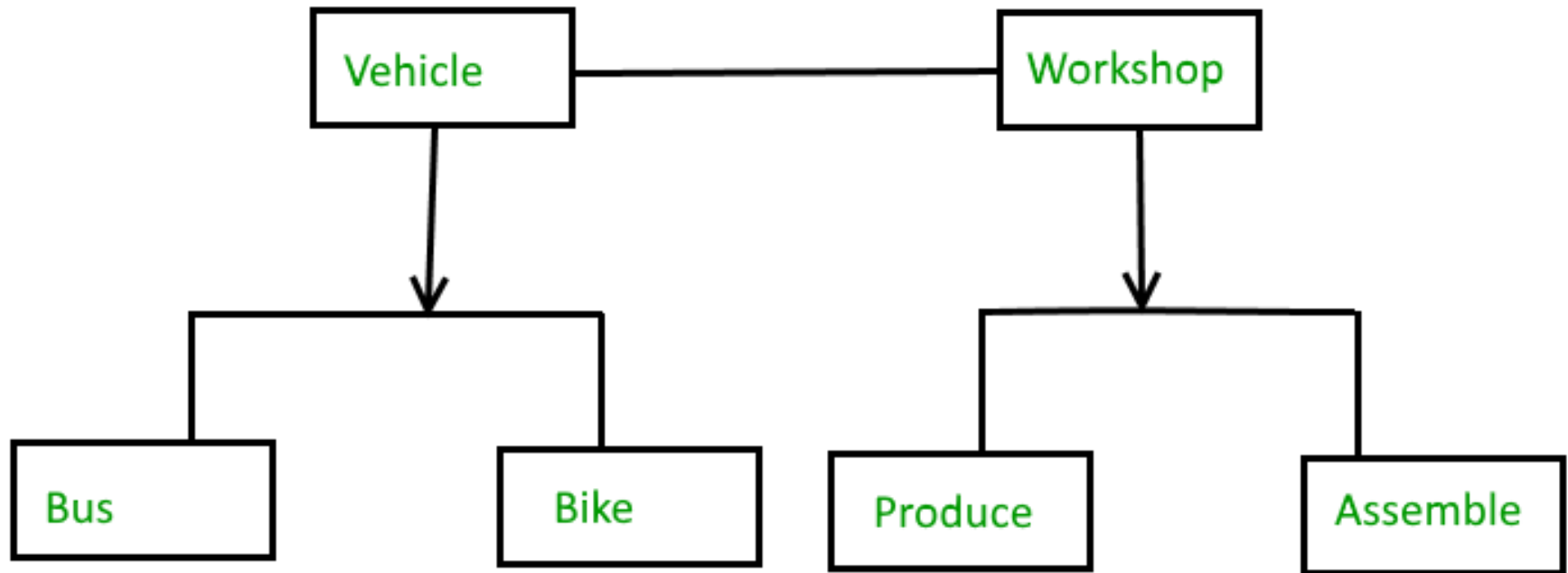
Bridge Pattern

- The Bridge design pattern allows you to separate the abstraction from the implementation. It is a structural design pattern.
- There are 2 parts in Bridge design pattern :
 - Abstraction
 - Implementation

Bridge Pattern(Without Bridge)



Bridge Pattern(With Bridge)

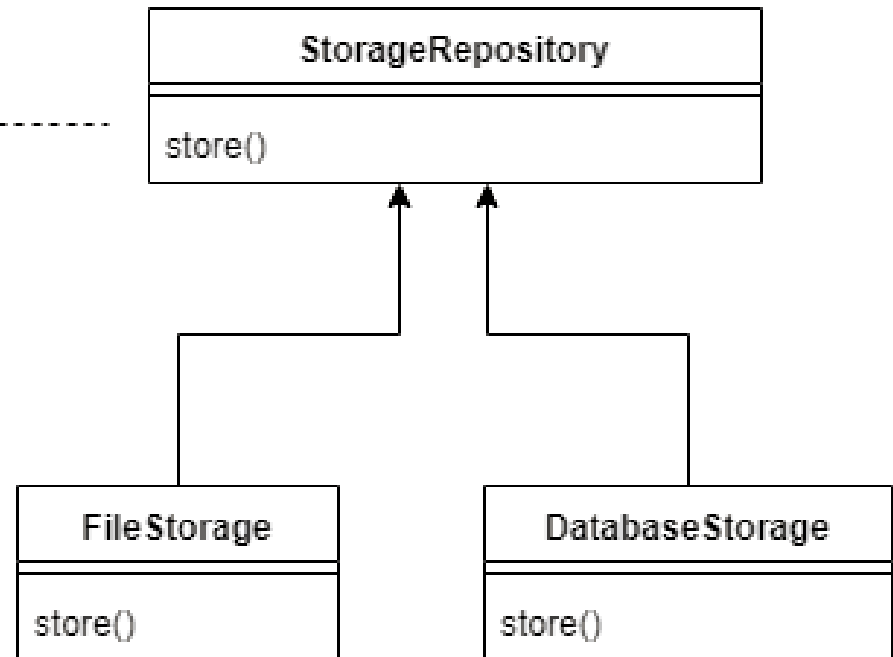


Bridge Pattern(With Bridge)

Abstraction



Implementation



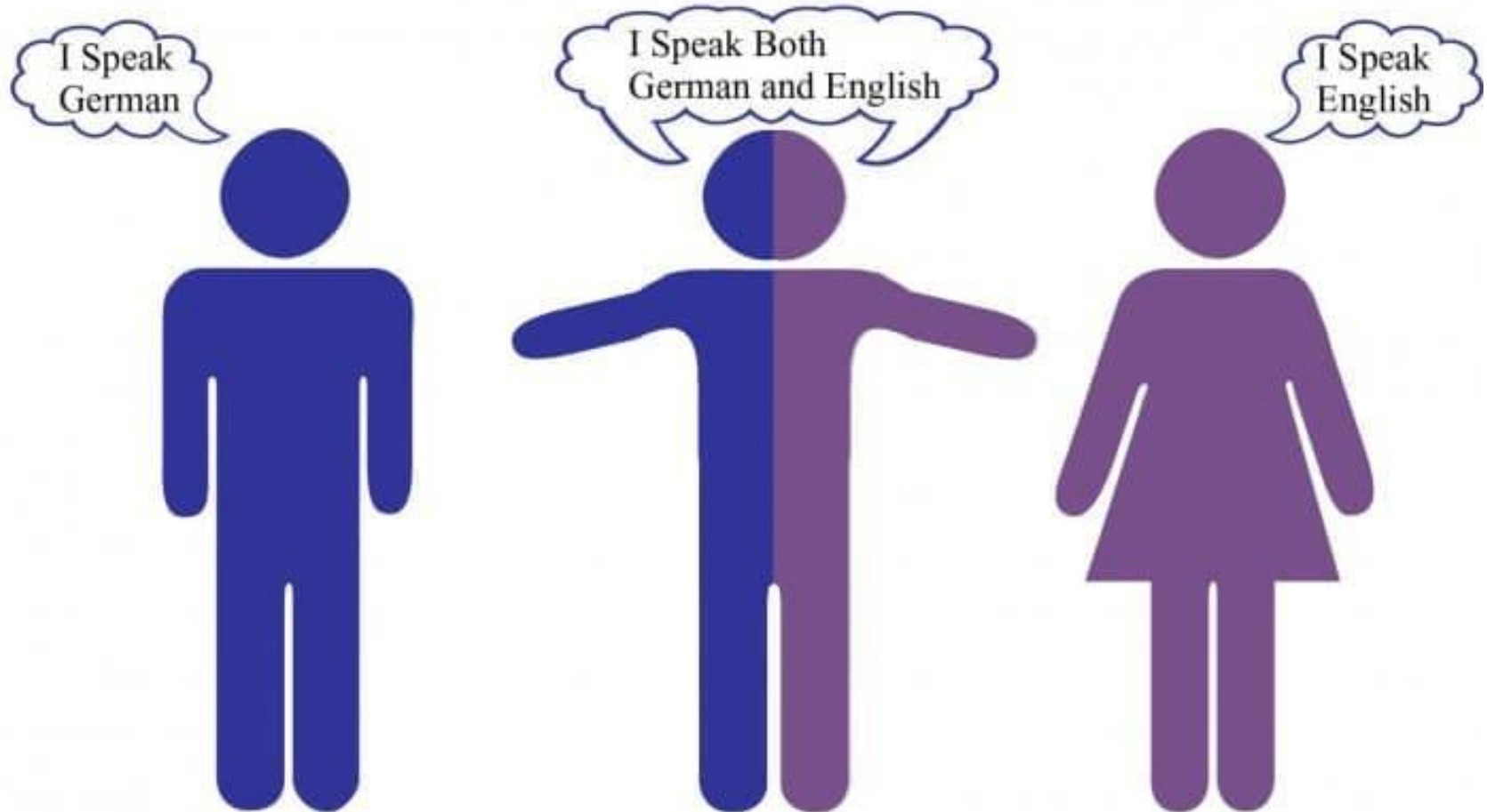
Bridge Pattern

- Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
- It is used mainly for implementing platform independence feature.
- It adds one more method level redirection to achieve the objective.
- Publish abstraction interface in a separate inheritance hierarchy, and put the implementation in its own inheritance hierarchy.
- Use bridge pattern to run-time binding of the implementation.
- Use bridge pattern to map orthogonal class hierarchies
- Bridge is designed up-front to let the abstraction and the implementation vary independently.

Adapter Pattern

- Adapter design pattern in java is a structural design pattern.
- It provides solution for helping incompatible things to communicate with each other.
- It works as an inter-mediator who takes output from one client and gives it to other after converting in the expected format.
- Adapter pattern is also known as wrapper.

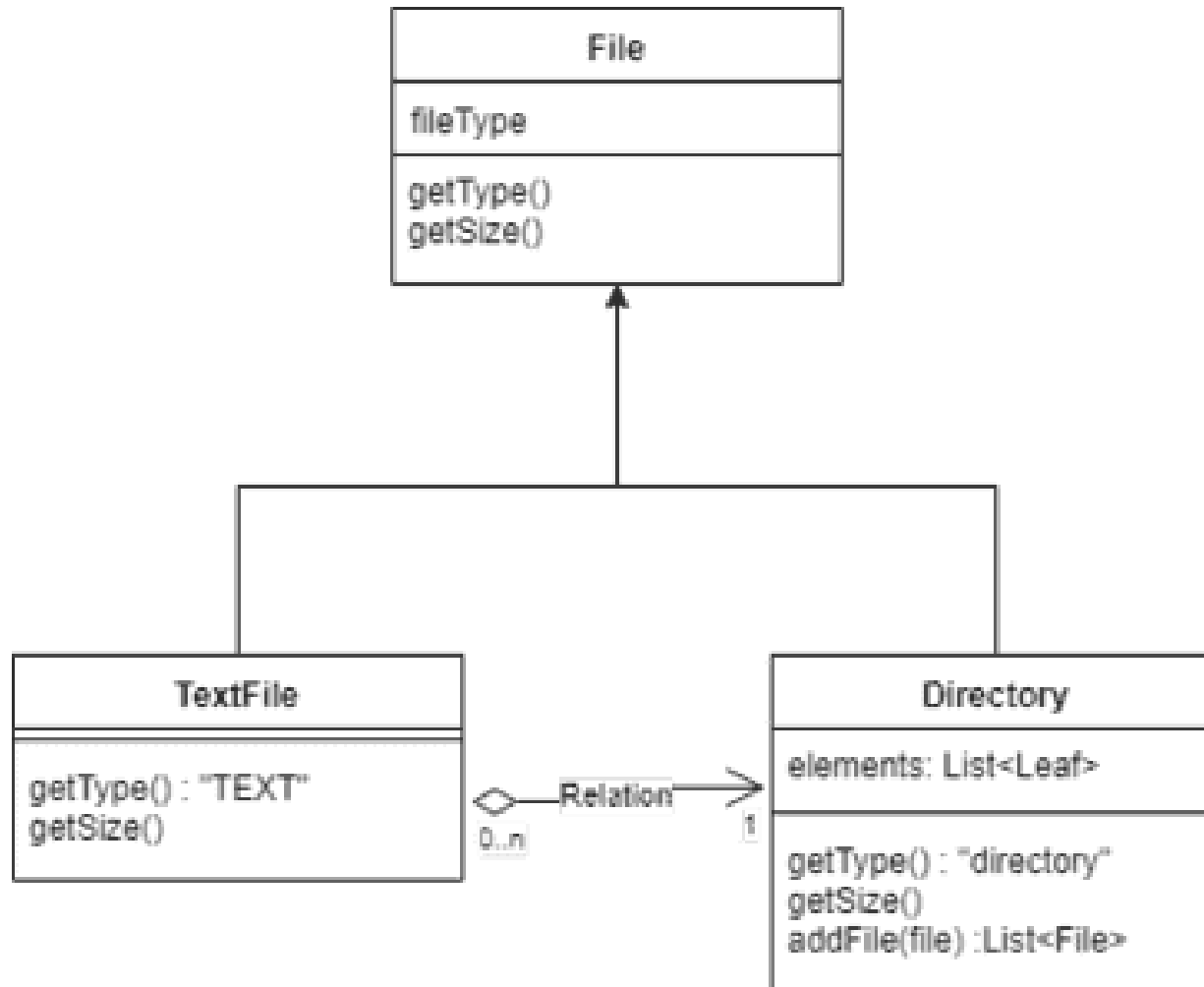
Adapter Pattern



Composite Design Pattern

- From the name, “Composite” means the combination or made from different parts. So this pattern provides solution to operate group of objects and single object in similar way.
- The two most important uses of the composite design pattern are,
- Objects can be represented in a tree structure hierarchy
- Composite and individual objects are treated uniformly

Composite Design Pattern



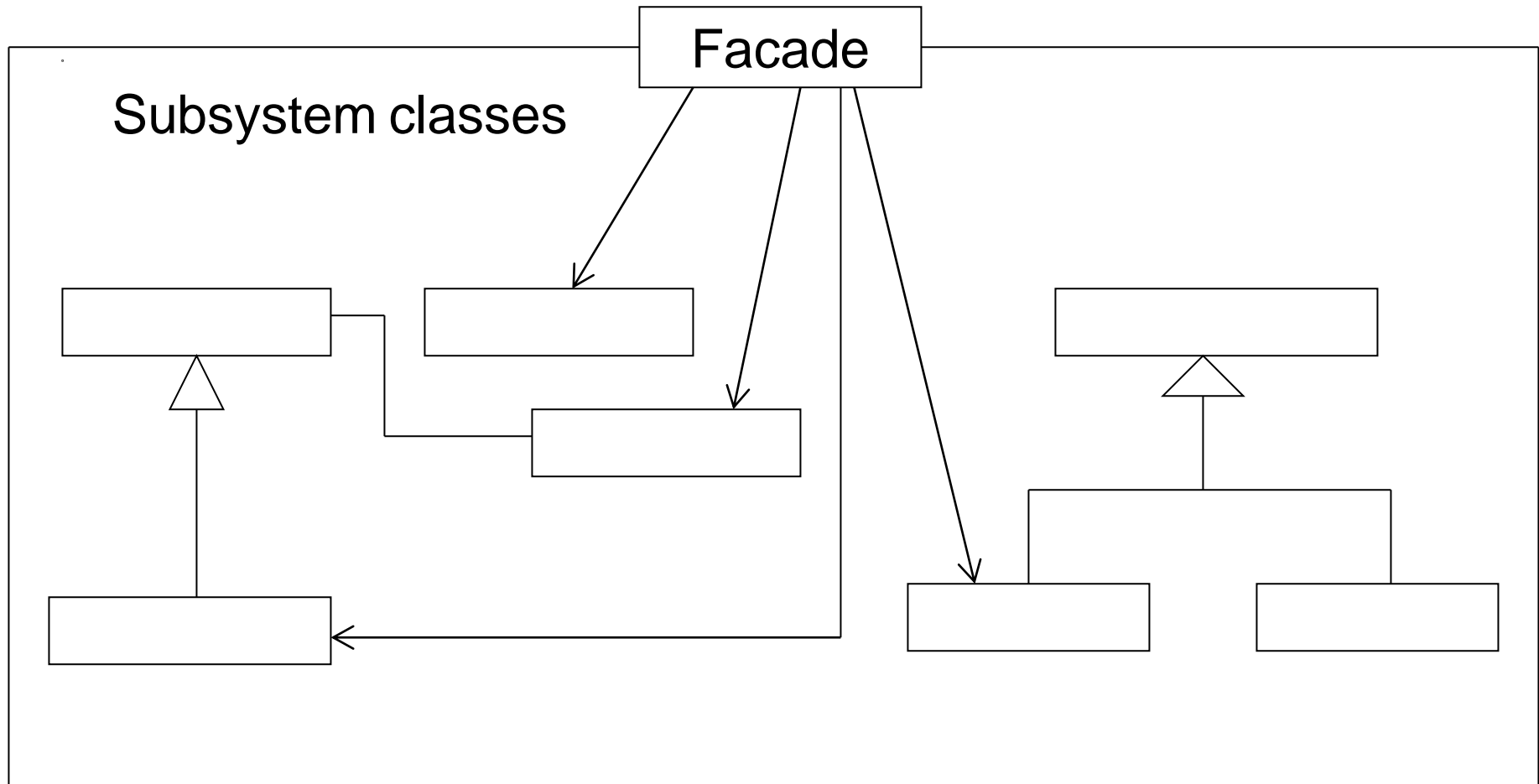
Façade Pattern

- “Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.”
 - Design Patterns: Elements of Reusable Object-Oriented Software by the “Gof (Gang of Four)”
- In other words, façade pattern creates a simplified view of complex sub-system system to the other sub-systems by providing an interface through which the other subsystem interact.
- It helps to layer and decouple subsystems.

Facade Pattern: usage

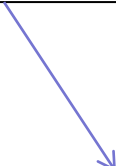
- Situation where a simple interface need to be provided to a complex subsystem like say Employee Management subsystem can provide an interface through which Payroll subsystem can access Employee details.
- Another commonly used pattern which is a derivation of Façade pattern is Session Façade
- Session Facade defines a higher-level business logic that in turn calls the lower-level business logic implemented in the business components.
- Clients interact with Session Façade only and this decouples lower-level business components from one another, making designs more flexible and comprehensible
- In JEE, a Session Facade is implemented as a session enterprise bean.

Structure



Code skeleton

```
class Payroll{  
public void calculatePay(){  
Employee e=getEmployee();  
...  
int i=e.getEmpID(); }  
}
```



```
interface Employee { public int getEmpID();}  
class IndiaEmployees implements Employee {  
public int getEmpID(){ ...}  
//other methods  
}  
class ChinaEmployees implements Employee {  
public int getEmpID(){ ...}  
//other methods  
}
```

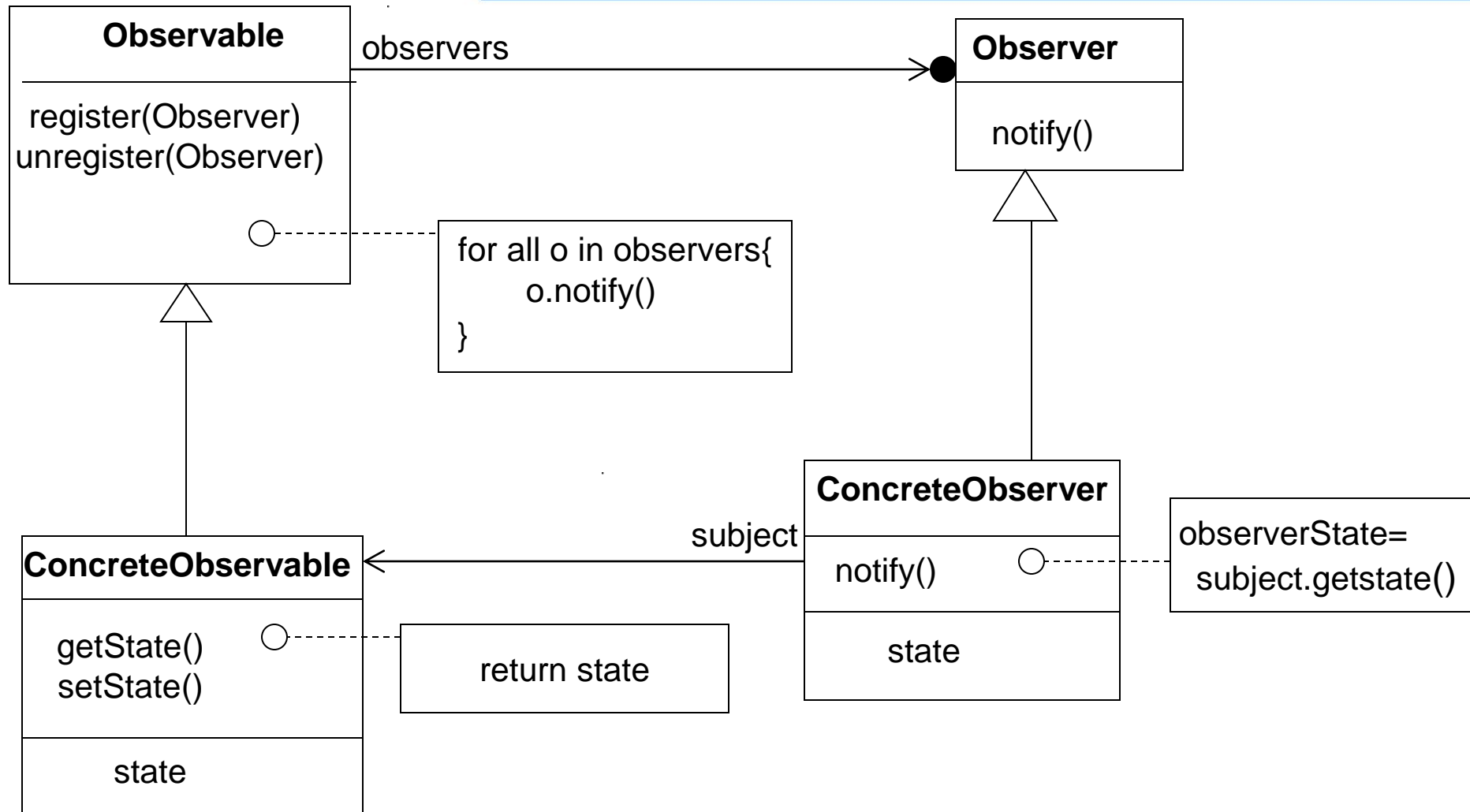
Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”
 - Design Patterns: Elements of Reusable Object-Oriented Software by the “Gof (Gang of Four)”
- Idea is for an object to notify the interested objects about the change(s) that happen in it.
- Also known as Dependents / Publish-Subscribe

Observer Pattern: usage

- When a change to one object requires changing others and we don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, we don't want these objects tightly coupled.
- The JSE has many examples that implement this pattern
 - Inter-thread communication using `wait()`, `notify()` methods of `Object` class
 - AWT Event handling mechanism
 - SAX Parser

Structure



Explanation

- The Observer is the object interested in changes that happen in Observable object.
- The Observer must call register method of Observable to register itself. Observable object must maintain a list of Observer objects.
- When change happens in Observable, it calls the notify() method of the Observer.
- Observer then implements the required changes with respect to the change that happened in the Observable.

Revisiting SaxParser

```
import java.io.*;
import org.xml.sax.Attributes;
import javax.xml.parsers.SAXParser;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;

public class CountSax extends DefaultHandler{
public static void main(String s[]) throws Exception{

SAXParserFactory factory=SAXParserFactory.newInstance();

SAXParser saxParser=factory.newSAXParser();
File f= new File("persons.xml");
if(f.exists())

saxParser.parse(f,new CountSax());
else
System.out.println("unknown file");}
```

Observer

Observable


register

```
static private int ele=0;
```

```
public void startDocument(){ele=0;}
```

```
public void  
startElement(String uri, String localName, String  
qName, Attributes attrs)  
{ ele++;}
```

```
public void endDocument(){  
System.out.println("Number of elements :" +ele);  
}  
}
```



Many
notify()
type
methods

Tell me what

- Which type of design pattern does MVC belong to?
- MVC does not fall into any of the 3 patterns.
- Initially these 3 design patterns were introduced by GoF.
- Later in another classification, architectural pattern also came up.
- MVC is an architectural pattern .

Architectural pattern

- An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

1.Layered pattern

5.Broker pattern

pattern

2.Client-server pattern

6.Peer-to-peer pattern

9.Blackboard pattern

3.Master-slave pattern

7.Event-bus pattern

10.Interpreter pattern

4.Pipe-filter pattern

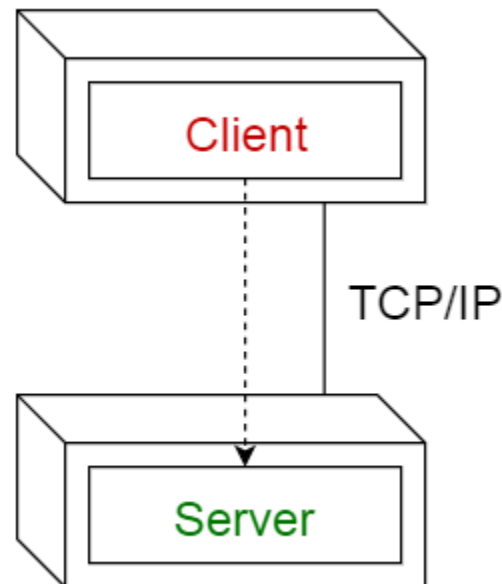
8.Model-view-controller

Layered Design Pattern

- Presentation layer (also known as UI layer)
- Application layer (also known as service layer)
- Business logic layer (also known as domain layer)
- Data access layer (also known as persistence layer)

Client-server pattern

- This pattern consists of two parties; a server and multiple clients. The server component will provide services to multiple client components.
- Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.



Master-slave pattern

- This pattern consists of two parties; **master** and **slaves**.
- The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.
- **Usage**
 - In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
 - Peripherals connected to a bus in a computer system (master and slave drives).

Pipe-filter pattern

- This pattern can be used to structure systems which produce and process a stream of data.
- Each processing step is enclosed within a filter component. Data to be processed is passed through pipes.
- These pipes can be used for buffering or for synchronization purposes.
- Usage
 - Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
 - Workflows in bioinformatics.

Broker pattern

- This pattern is used to structure distributed systems with decoupled components. These components can interact with each other by remote service invocations.
- A **broker** component is responsible for the coordination of communication among **components**.
- Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.
- Usage
- Message broker software such as Apache ActiveMQ, Apache Kafka, RabbitMQ and JBoss Messaging.

Peer-to-peer pattern

- In this pattern, individual components are known as **peers**. Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.
- **Usage**
- File-sharing networks such as **Gnutella** and **G2**)
- Multimedia protocols such as **P2PTV** and **PDTP**.

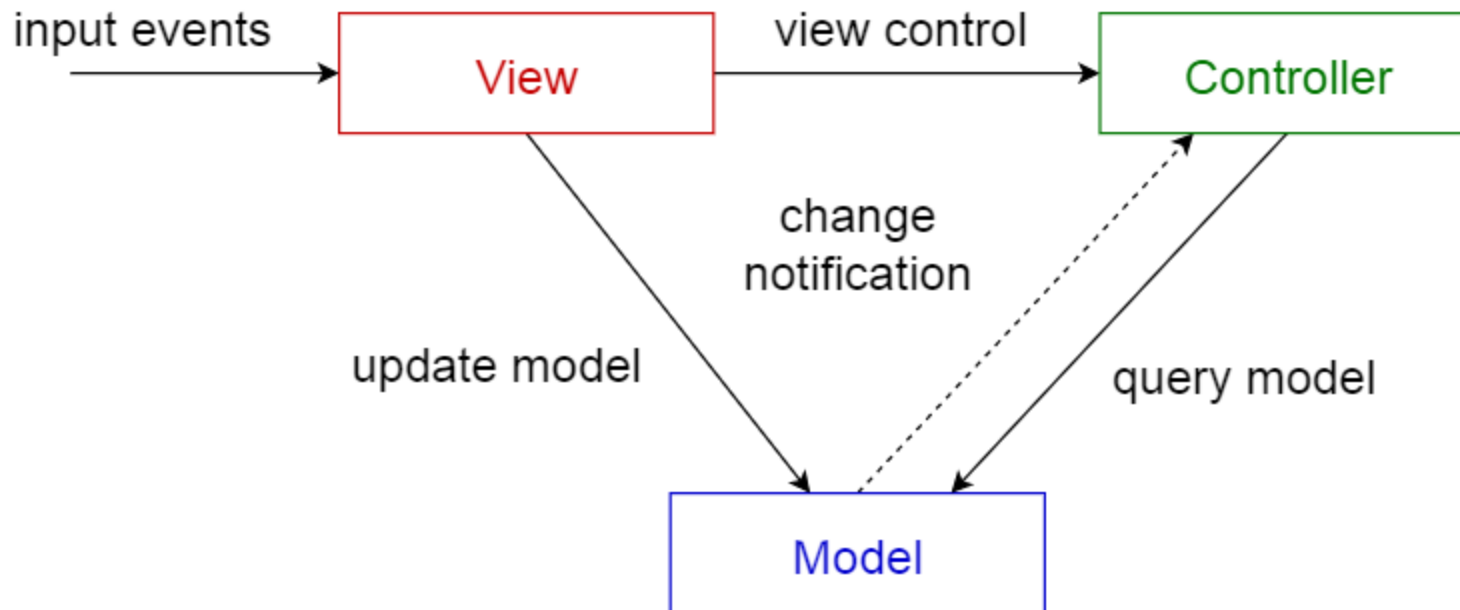
Event-bus pattern

- This pattern primarily deals with events and has 4 major components; **event source**, **event listener**, **channel** and **event bus**. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before.
- **Usage**
- Android development
- Notification services

Model-view-controller pattern

- This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,
- **model** — contains the core functionality and data
- **view** — displays the information to the user (more than one view may be defined)
- **controller** — handles the input from the user

Model-view-controller pattern



Blackboard pattern

- This pattern is useful for problems for which no deterministic solution strategies are known. The blackboard pattern consists of 3 main components.
- **blackboard** — a structured global memory containing objects from the solution space
- **knowledge source** — specialized modules with their own representation
- **control component** — selects, configures and executes modules.

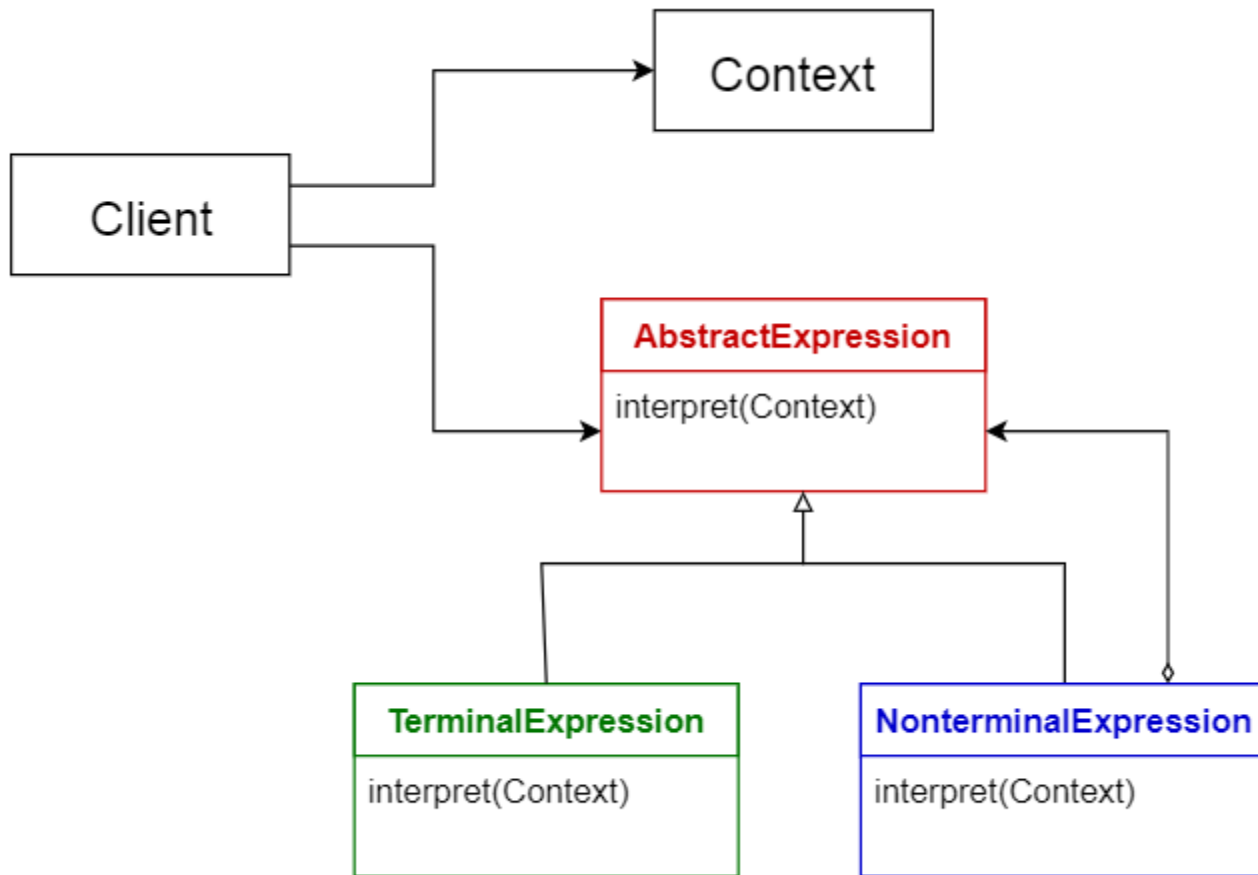
Blackboard pattern

- All the components have access to the blackboard. Components may produce new data objects that are added to the blackboard. Components look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source.
- **Usage**
- Speech recognition
- Vehicle identification and tracking
- Protein structure identification
- Sonar signals interpretation.

Interpreter pattern

- This pattern is used for designing a component that interprets programs written in a dedicated language. It mainly specifies how to evaluate lines of programs, known as sentences or expressions written in a particular language. The basic idea is to have a class for each symbol of the language.
- **Usage**
- Database query languages such as SQL.
- Languages used to describe communication protocols.

Interpreter pattern



Comparison of Architectural Patterns

Name	Advantages	Disadvantages
Layered	A lower layer can be used by different higher layers. Layers make standardization easier as we can clearly define levels. Changes can be made within the layer without affecting other layers.	Not universally applicable. Certain layers may have to be skipped in certain situations.
Client-server	Good to model a set of services where clients can request them.	Requests are typically handled in separate threads on the server. Inter-process communication causes overhead as different clients have different representations.
Master-slave	Accuracy - The execution of a service is delegated to different slaves, with different implementations.	The slaves are isolated: there is no shared state. The latency in the master-slave communication can be an issue, for instance in real-time systems. This pattern can only be applied to a problem that can be decomposed.
Pipe-filter	Exhibits concurrent processing. When input and output consist of streams, and filters start computing when they receive data. Easy to add filters. The system can be extended easily. Filters are reusable. Can build different pipelines by recombining a given set of filters	Efficiency is limited by the slowest filter process. Data-transformation overhead when moving from one filter to another.
Broker	Allows dynamic change, addition, deletion and relocation of objects, and it makes distribution transparent to the developer.	Requires standardization of service descriptions.

Comparison of Architectural Patterns

Peer-to-peer	Supports decentralized computing. Highly robust in the failure of any given node. Highly scalable in terms of resources and computing power.	There is no guarantee about quality of service, as nodes cooperate voluntarily. Security is difficult to be guaranteed. Performance depends on the number of nodes.
Event-bus	New publishers, subscribers and connections can be added easily. Effective for highly distributed applications.	Scalability may be a problem, as all messages travel through the same event bus
Model-view-controller	Makes it easy to have multiple views of the same model, which can be connected and disconnected at run-time.	Increases complexity. May lead to many unnecessary updates for user actions.
Blackboard	Easy to add new applications. Extending the structure of the data space is easy.	Modifying the structure of the data space is hard, as all applications are affected. May need synchronization and access control.
Interpreter	Highly dynamic behavior is possible. Good for end user programmability. Enhances flexibility, because replacing an interpreted program is easy.	Because an interpreted language is generally slower than a compiled one, performance may be an issue.