WEEK 2

SYSTEM DESIGN

LOW_LEVEL DESIGN

Signature Rule

1. Method Argument Rule / Overriding Rule

Broad: Animal is a broad class.

Narrow : Dog is a narrow class.


```cpp
class Parent {
public:
    virtual void solve(string s) {
        // Parent implementation
    }
};

class Child : public Parent {
public:
    void solve(string s) override {
        // Child implementation
    }
};
```

- The Signature Rule in object-oriented programming states that when a method is overridden in a subclass, its parameter data types must match the method signature in the parent class (same no of parameters, same data –types of parameters).
- In the example, Animal is a broad (parent) class and Dog is a narrow (child) class.
- This ensures polymorphism and type consistency across class hierarchies.

2. Covariant Return Type Rule

In method overriding, the return type of the overriding method in the child class must either be the same as or a subtype (narrower) of the return type declared in the parent class method.

```cpp
class Animal {
    // ...
};

class Dog : public Animal {
    // ...
};
```

```cpp
class Parent {
public:
    virtual Animal* getAnimal() {
        return new Animal();
    }
};

class Child : public Parent {
public:
    // ✅ Allowed: Dog is a subtype of Animal
    Dog* getAnimal() override {
        return new Dog();
    }
};
```

3. Exception Rule

When a child class overrides a method from a parent class, it can only throw:

- The same exception type as the parent method, or
- A subclass (narrower exception) of the exception thrown by the parent method.

It cannot throw:

- A broader exception, or
- A completely different unrelated exception.

TYPES OF ERRORS:

1. Syntax errors
2. Logical errors
3. Runtime errors
4. Time Limit exceeded error
5. User defined

```java
class Parent {
    void process() throws RuntimeException {
        // ...
    }
}
```

```java
class Child extends Parent {
    @Override
    void process() throws NullPointerException { // ✅ allowed (subclass of RuntimeException)
        // ...
    }
}
```

```java
class Child extends Parent {
    @Override
    void process() throws Exception { // ❌ broader than RuntimeException
        // ...
    }
}
```

❌ Invalid

These three rules (Signature Rule, Return Type Rule, and Exception Rule) are **fundamental to adhering to the Liskov Substitution Principle (LSP)**, one of the SOLID principles of object-oriented design.

These rules enforce consistency so that child classes can truly replace their parent classes without introducing: Compilation errors, Unexpected exceptions, Logic failures due to incompatible method contracts

Property Rule

1. Class Invariant
   - A class invariant is a condition that must always hold true for an object after construction and after any public method finishes execution. It defines the essential valid state of an object.
   - In LSP: A subclass must preserve the parent's class invariants (the **rules or conditions** that define what it means for an object of the parent class to be in a valid, correct state.). This ensures that substituting the subclass for the parent does not violate the expected behavior or state constraints.
   - Example: If a BankAccount class guarantees that the balance is never negative, then any subclass (e.g., SavingsAccount) must also maintain that invariant.
   - 🧠 Why it matters: Violating invariants in a subclass can break assumptions made by the rest of the system, making the behavior unpredictable.
2. History Constraint
   - The history constraint (also called the immutable property rule) states that a subclass should not allow changes to the observable behavior of a superclass in a way that violates expectations based on the parent's history.
   - In LSP: The subclass should not alter how the object evolves over time in a way that breaks client expectations set by the parent class.
   - Example: If a ReadOnlyList guarantees its contents don't change, a subclass MutableList that adds a push_back() method would break LSP — because the object's observable behavior has changed in a way that breaks the parent's contract.
   - 🧠 Why it matters: Clients relying on the original behavior (e.g., immutability) might now be affected by unexpected side effects, breaking substitutability.

These 2 rules are also necessary for LSP

Method Rule

1. Pre Condition

A precondition is a condition that must be true before a method is called, in order for the method to work correctly.

```
int divide(int a, int b) {
    // Precondition: b must not be 0
    return a / b;
}
```

- If someone passes b = 0, the method fails.
- Subclasses must not strengthen preconditions — they should expect the same or fewer assumptions as the parent method.

🔑 Why Important in LSP: If a subclass requires stricter inputs, it breaks substitutability — the subclass can't be used in place of the parent.

2. Post Condition

A postcondition is a condition that must be true after a method finishes running, assuming the preconditions were met.

```
int increment(int x) {
    // Postcondition: result must be x + 1
    return x + 1;
}
```

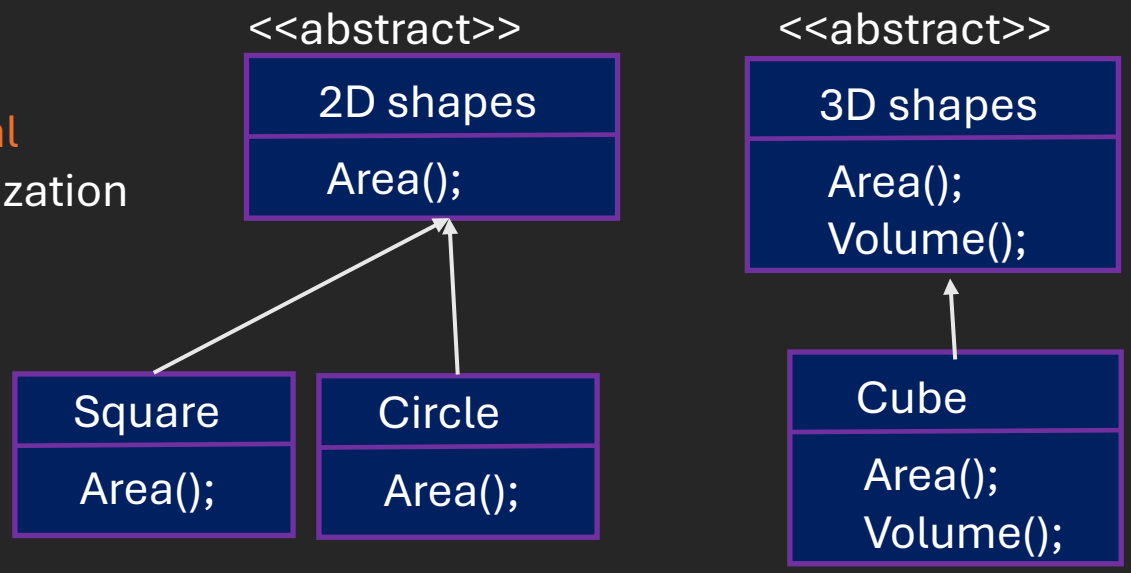
- After this method runs, the result must be one more than the input.
- Subclasses must not weaken postconditions — they should guarantee the same or stronger results than the parent.

🔑 Why Important in LSP: If a subclass method does less than expected, it breaks the promise of the parent class — violating substitutability.

These 3 rules are also necessary for LSP

: Interface Segregation Principle

Many client specific interface are better than one general purpose interface. Specialization is better than Generalization
Client should not be forced to implement methods they don't need.



Imagine we create a single Shape interface with methods like area() and volume(). At first glance, this seems convenient, but problems arise when we try to implement 2D shapes like circles or rectangles. These shapes only have an area, not a volume. Yet, under this general interface, they are forced to implement a volume() method, which might either throw an exception or return a meaningless value. This is a violation of the Interface Segregation Principle, which states that a class should not be forced to depend on methods it does not use.
To correct this, we can split the interface into two: one for 2D shapes with just the area() method, and another for 3D shapes with the volume() method. Now, 2D classes like Circle or Square only implement the interface relevant to them, while 3D classes like Cube or Sphere implement the one suited for their behavior. This separation ensures each class is only concerned with the operations it supports, resulting in cleaner, more maintainable, and logically consistent code. Such design aligns with the Interface Segregation Principle, making the system more robust and easier to scale.

D: Dependency Inversion Principle

High level module should not depend on low level module but rather both should depend on abstraction.
Meaning, there should be a layer of abstraction between high level module and low level module, so that both don't interact directly.

In simpler terms:

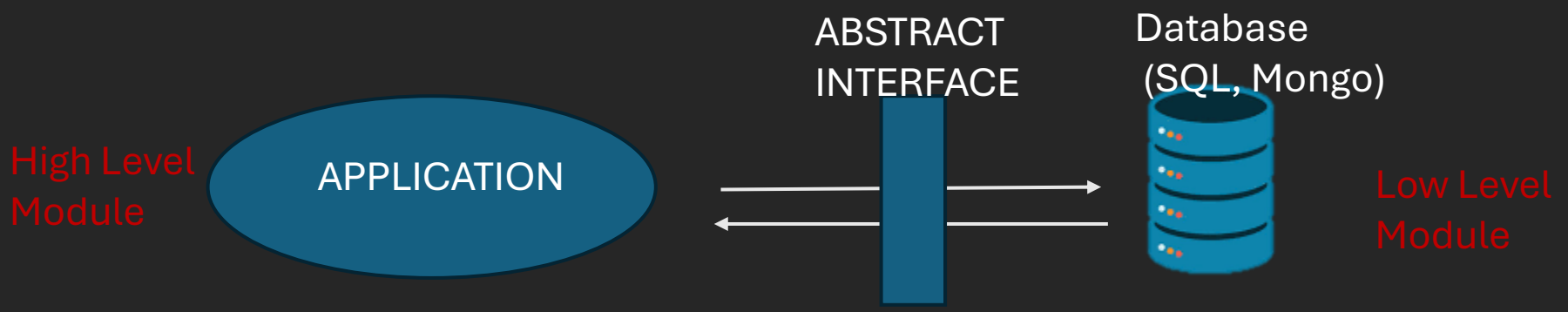
- High-level modules (like business logic) shouldn't directly rely on low-level modules (like database access or file systems).
- Instead, both should interact through abstract interfaces, so that implementation details can change without affecting the overall system.

Without DIP:

- Changes in low-level modules can ripple up and break high-level logic.
- Testing becomes harder due to tight coupling.

With DIP:

- You can easily swap implementations (e.g., switch a database or mock a service).
- The system is more flexible, maintainable, and testable.



The image illustrates the Dependency Inversion Principle (DIP) in a software system, where the high-level module (the Application) does not directly interact with the low-level module (the Database such as SQL or MongoDB). Instead, both modules communicate through an Abstract Interface. This abstraction acts as a contract that defines how the application can store or retrieve data without knowing the exact implementation. By depending on this interface, the application becomes loosely coupled and more flexible. For instance, switching from SQL to MongoDB does not require changes in the application logic—only the implementation of the interface changes—making the system easier to maintain and extend.

The Dependency Inversion Principle is necessary in the given example—and in general—because it promotes loose coupling between components, making systems more flexible, testable, and maintainable. In the example, if the application directly depended on a specific database like SQL, any switch to MongoDB would require rewriting core parts of the application. By introducing an abstract interface, the application can remain agnostic to the underlying database technology. This abstraction ensures that high-level business logic isn't tightly bound to low-level implementation details, which allows independent development, easier upgrades, and seamless integration of new technologies without breaking existing functionality.

IMPORTANT

The idea that
“if the Open/Closed Principle (OCP) is the goal, then the Dependency Inversion Principle (DIP) is the solution”
is widely attributed to Robert C. Martin (Uncle Bob). Uncle Bob emphasizes that DIP enables OCP.

Here's why:

- Open/Closed Principle (OCP): Classes should be open for extension but closed for modification.
- Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions.

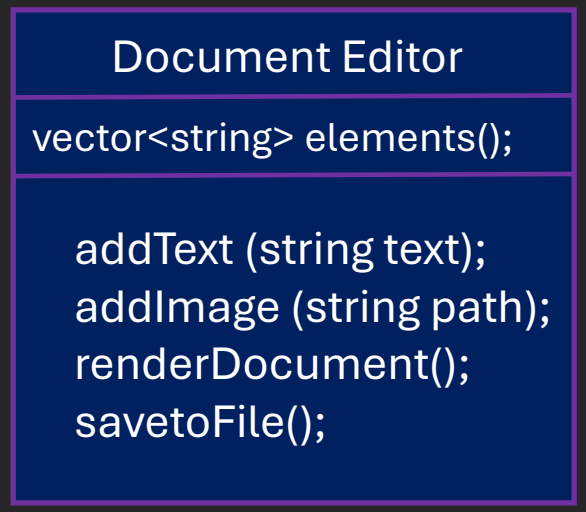
By depending on abstractions (interfaces), rather than concrete implementations, modules can be extended with new behavior without modifying existing code—thus achieving OCP.

LLD PROJECT – Document Editor

There are 2 approaches to tackle LLD problems:

1. Top-down approach
2. Bottom-up approach

Initial Design



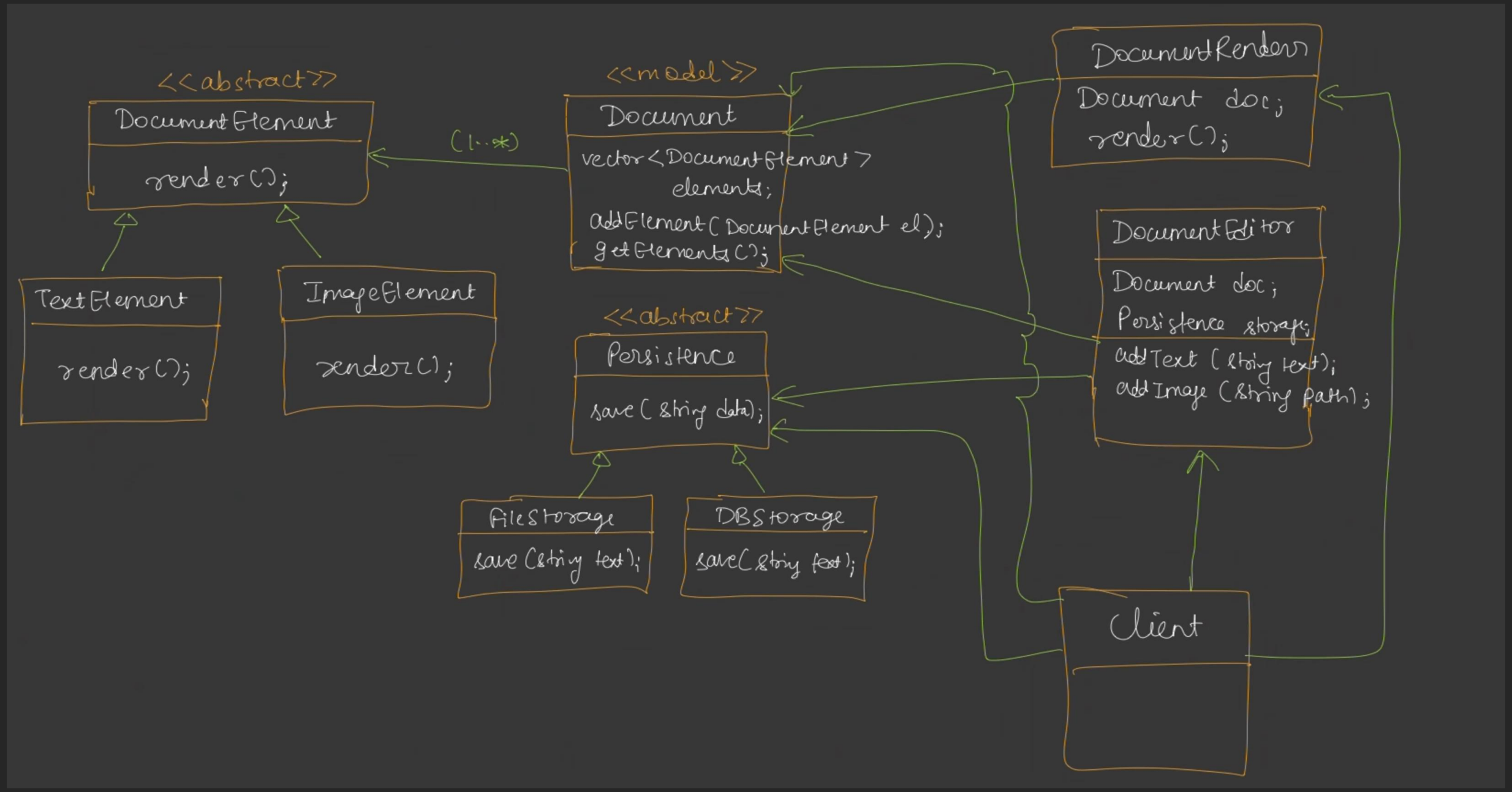
Problems in this design:

1. Breaks Single responsibility principle (one class is doing many tasks)
2. Breaks Open Close principle because suppose in future we have to add more features like adding videos, different font colors, etc.. then we would have to make changes in the same class.

It doesn't have LIP, DIP, ISP because there is only 1 class (not a problem)

We will use Polymorphism

Final Design



The final diagram follows :

1. SIP : as each class has 1 task.
2. OCP: as no need to change code of an existing class. If we want new feature we can create a class for it and attach to any of the 2 abstract class.
3. LSP: because Document, TextElement, and ImageElement all override render() from DocumentElement. Any code that works with DocumentElement can substitute these derived classes without breaking behavior. Sub-types are replaceable by the parent class.
4. ISP: There is no extra method. Only required methods are present.
5. DIP: Let's say if TextElement and ImageElement are low level module and Document is high-level module, they are not interacting directly, they have a DocumentElement interface between them.

But it has a problem. It breaks principle of Least Knowledge. The Principle of Least Knowledge, also known as the Law of Demeter, is a software design principle aimed at reducing the coupling between components in a system. This helps make your code easier to maintain, test, and evolve. It prevents tight coupling.
"An object should only talk to its immediate friends — not to strangers."

An object should only call methods on:

- itself,
- its fields,
- parameters passed into it,
- objects it creates.

Here, Document Render to call render() method, talks via :
Document Render → Document → Document Element

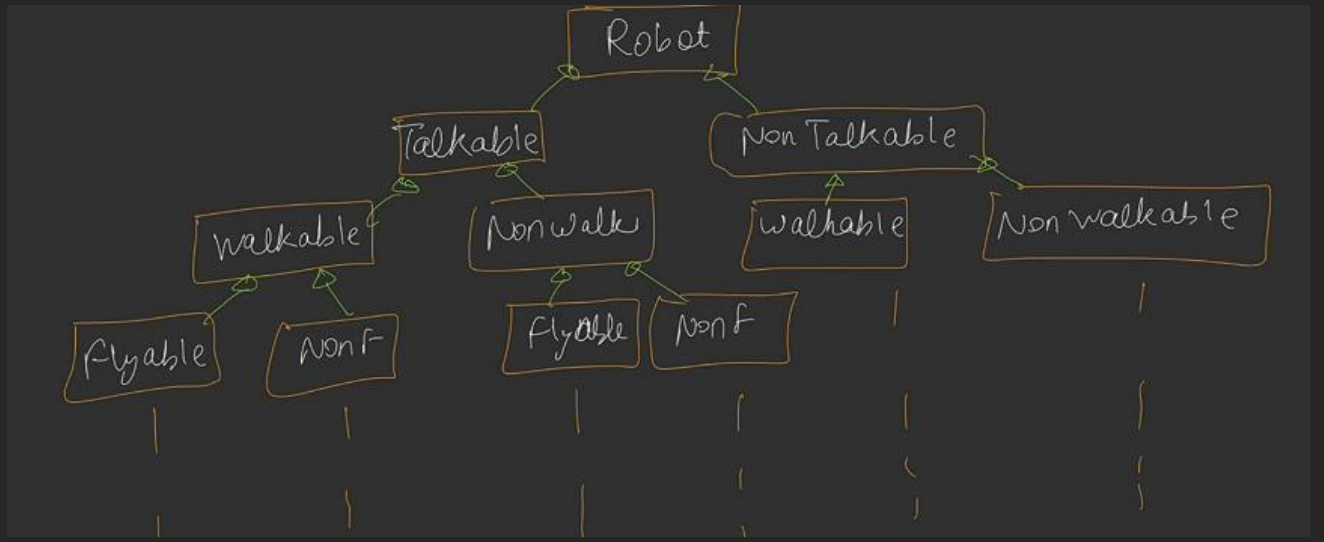
Strategy Design Pattern

A design pattern is a proven solution to common software design problems, and one of its core goals is to manage the balance between stability and flexibility in code.
In any system, there are always two parts:
the static part, which remains constant and forms the backbone of the system,
and the dynamic part, which changes frequently due to evolving requirements, features, or contexts.

Design patterns aim to separate the dynamic parts from the static parts, so that changes can be made in one area without disrupting the rest of the system. This separation of concerns improves modularity, enhances maintainability, and allows developers to adapt and extend software with minimal risk of introducing bugs into the stable foundation.

There are several types of Design patterns, and Strategy design pattern is one of those.

We can see in this picture, that inheriting each type of robot makes the hierarchy complicated!
We will need a solution to simplify this.



In short, favour Composition over Inheritance. !

Strategy Design Pattern

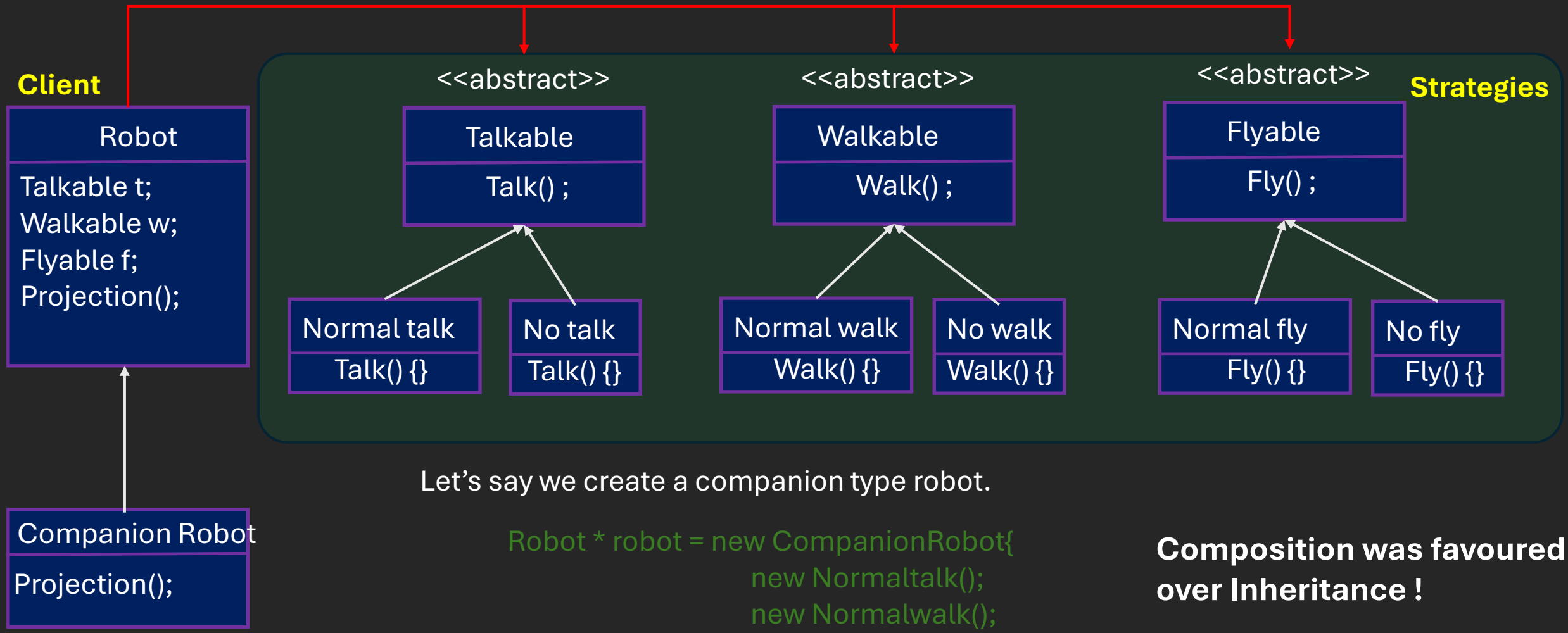
Defines a family of algorithms, encapsulates each one in a separate class, and makes them interchangeable at runtime.

Robot
Talk() {} Walk() {} Fly() {} Projection();

- Talk, Walk, Fly are dynamic and change from robot to robot.
- Way of talking, walking or flying.
- We should separate these dynamic parts from the static part.
- Here, talk, walk and fly are family of algorithms and should be in separate class as per Strategy Design Principle.

We can see the Strategy Design pattern for this in the next slide.

Here, the reference of each abstract interface is stored in Robot. Robot has-a relationship (composition).



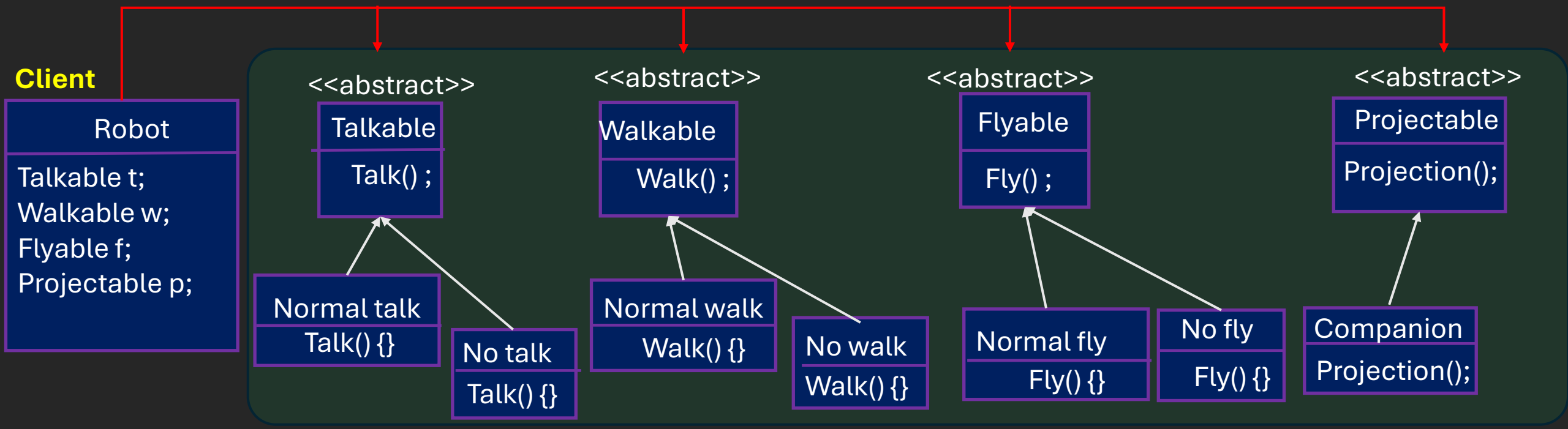
Let's say we create a companion type robot.

```
Robot * robot = new CompanionRobot{
                    new Normaltalk();
                    new Normalwalk();
                    new nofly();
}
```

Composition was favoured over Inheritance !

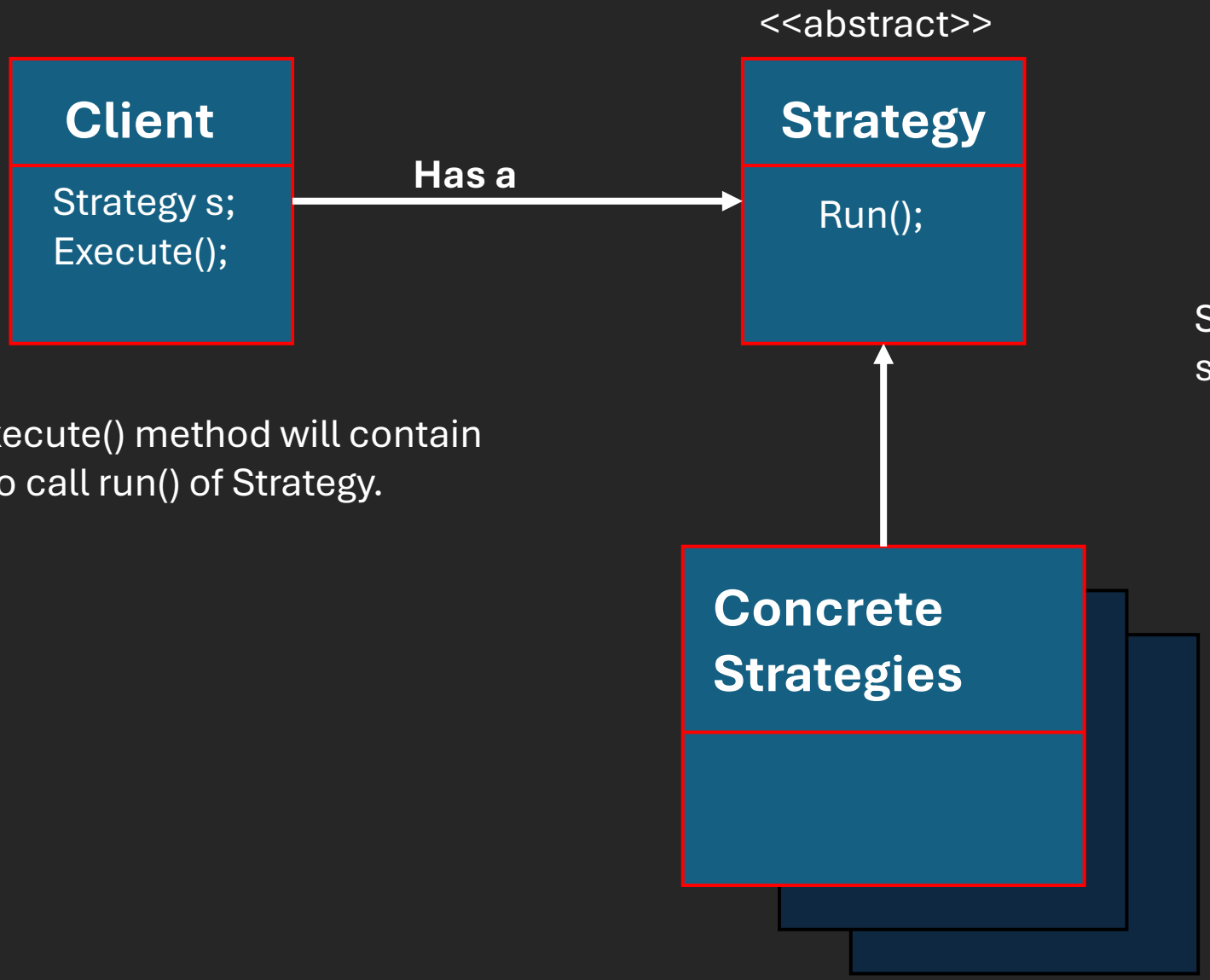
- Talkable, Walkable and Flyable are the family of algorithms.
- Also, it can be changed at runtime, we can pass notalk() instead of Normaltalk() and so on...

Note, here we can have several types of projection() robots like (companion, worker) as well. Again, we can convert those inheritance to composition so that it is easy to modify code if needed in future.



Now to create a new robot, we can:

```
Robot * robot = new Robot{
                    new Normaltalk();
                    new Normalwalk();
                    new nofly();
                    new Companion();
}
```



The Execute() method will contain logic to call run() of Strategy.

Since, strategy is abstract, we need something concrete.

Real life examples could be Payment System and different strategies like : UPI, Net banking, Credit/ Debit cards each having their strategy for Paying (concrete strategy)

Sorting : Bubble sort, merge sort, quick sort are strategies. Quick sort can be normal or randomized quick sort (concrete)

Factory Design Pattern

Imagine ordering a car. You tell the factory, “I want a sedan.” The factory knows the exact steps, parts, and configuration required to build it, but you don’t have to worry about that. You just get your sedan.

The **Factory Design Pattern** in system design is inspired by the concept of factories in the real world — places where products are created without the client needing to know the exact steps or components involved in the creation process.

We have 3 types of Factories:

1. Simple Factory
2. Factory Method
3. Abstract Factory Method

Real world application of Factory design pattern:

1. Notification system

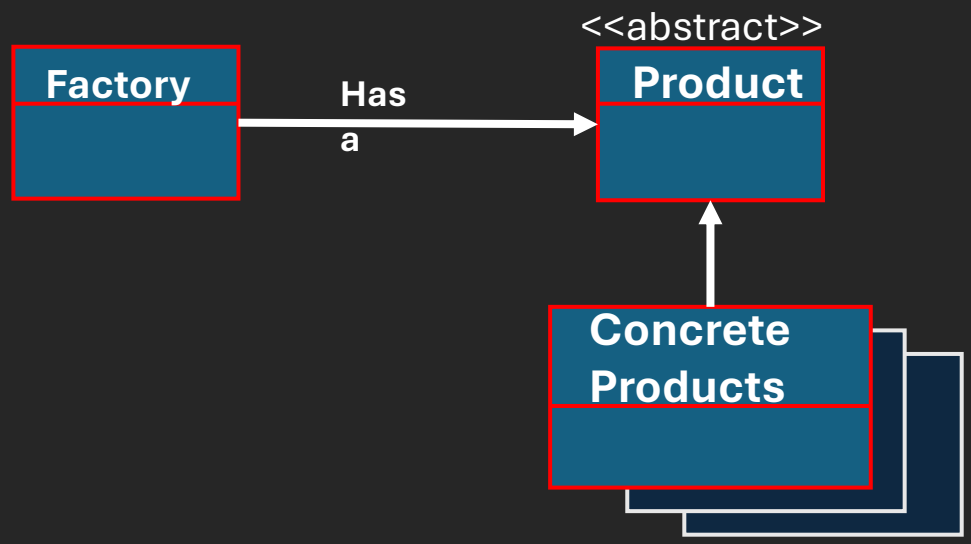
Imagine you have an abstract class or interface called Notification, which defines a common method like send(). The concrete implementations of this interface could be EmailNotification, SMSNotification, and PushNotification, each overriding the send() method with its specific logic.

Now, you introduce a NotificationFactory class that contains a method like getNotification(String type). Based on the type passed (e.g., "EMAIL", "SMS", or "PUSH"), the factory returns an instance of the corresponding notification class.

Simple Factory

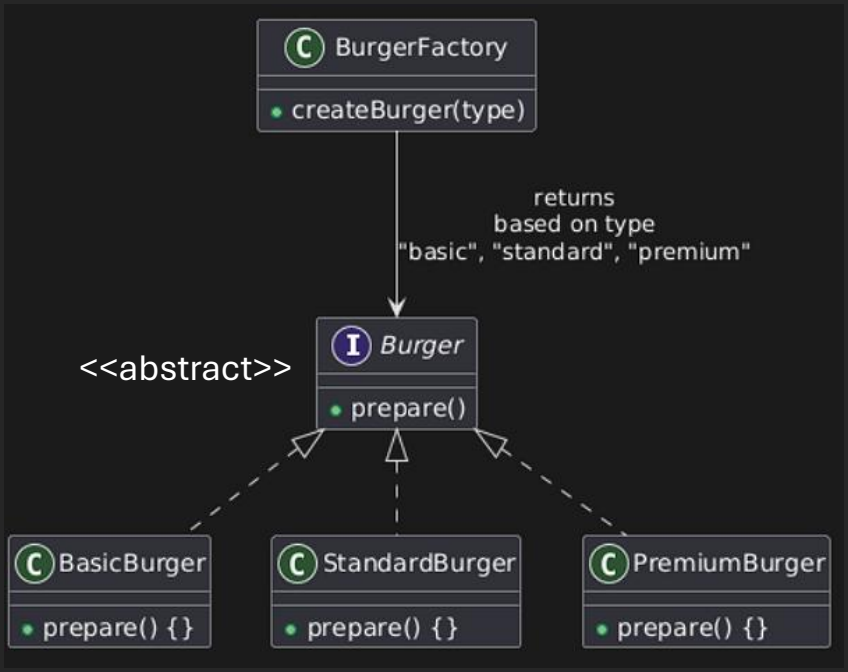
A factory class that decide which concrete class to instantiate.

The Simple Factory Pattern is a design pattern where a separate factory class is used to create objects, hiding the creation logic from the client. It provides a static method that returns different types of objects based on input, making object creation easy and centralized.



In the diagram we provided, the BurgerFactory class has a method called createBurger(type). Based on the type value ("basic", "standard", or "premium"), it returns an instance of either BasicBurger, StandardBurger, or PremiumBurger. All these burger types inherit from a common abstract class Burger, which has a method prepare().

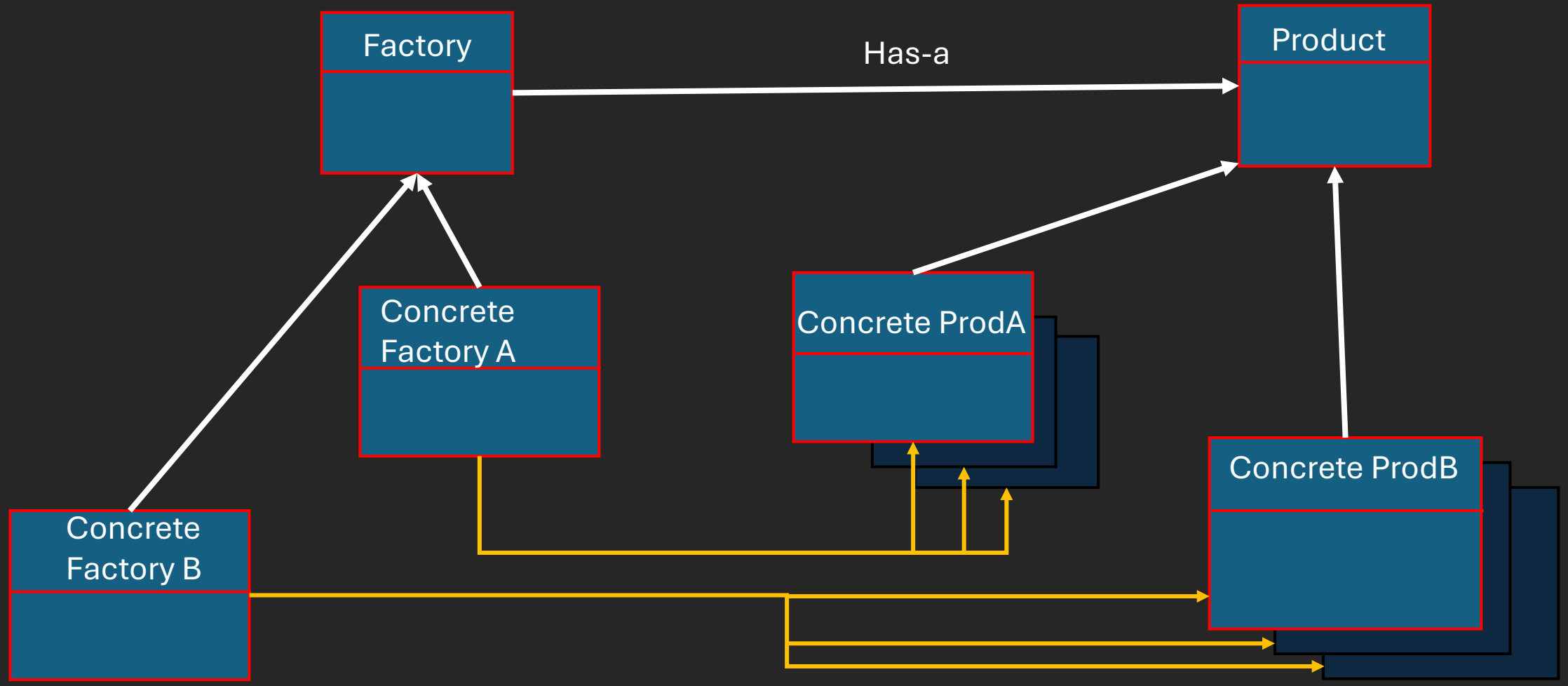
This way, the client doesn't need to know which class to instantiate — they simply call the factory with a type, and the appropriate burger is created and returned. This simplifies object creation and keeps the client code clean.



Factory Method

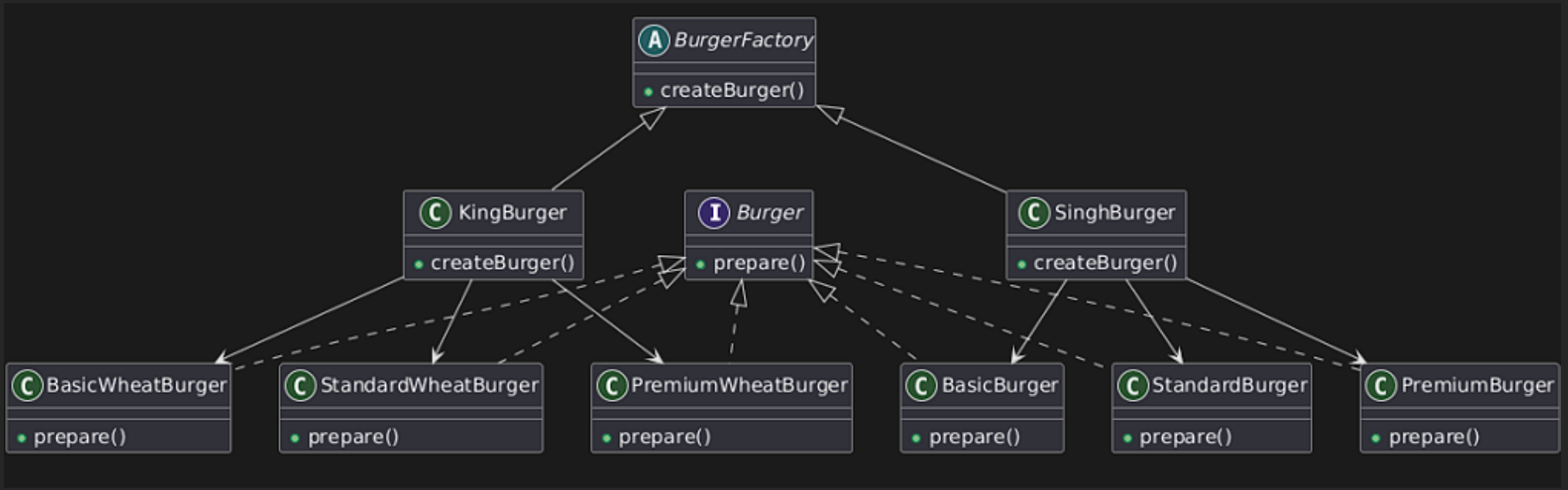
A factory base class defines an interface for creating an object, but lets subclasses decide which concrete class to instantiate.

The Factory Method Pattern delegates the instantiation logic to derived classes, promoting loose coupling and adherence to the Open/Closed Principle. Each subclass (concrete factory) overrides the factory method to return its own specific type of product, allowing flexibility and scalability when creating families of related objects.



This diagram is a clear example of the Factory Method Pattern, where the process of object creation is delegated to subclasses. At the core, we have an abstract class BurgerFactory that defines a method createBurger(), but it does not implement the actual logic of which burger to create. Instead, two concrete factories — SinghBurger and KingBurger — extend this abstract class and provide their own implementations of createBurger(). SinghBurger is responsible for creating burgers with regular buns (BasicBurger, StandardBurger, and PremiumBurger), while KingBurger produces wheat-based variants (BasicWheatBurger, StandardWheatBurger, and PremiumWheatBurger). All these burger classes implement a common interface Burger, which defines a prepare() method.

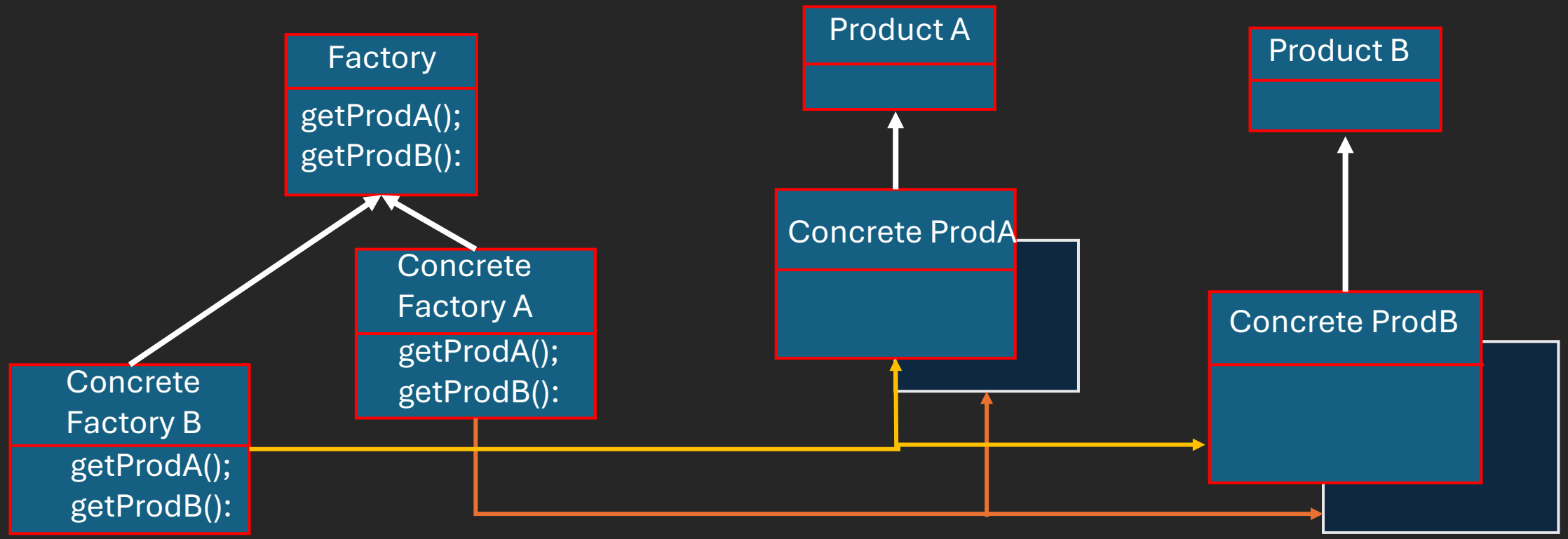
This structure allows each factory to create a family of related products while keeping the creation logic encapsulated and decoupled from the client, which only interacts with the factory and the Burger interface — a hallmark of the factory method pattern.



Abstract Factory Method

An abstract factory base class defines an interface for creating families of related or dependent objects without specifying their concrete classes.

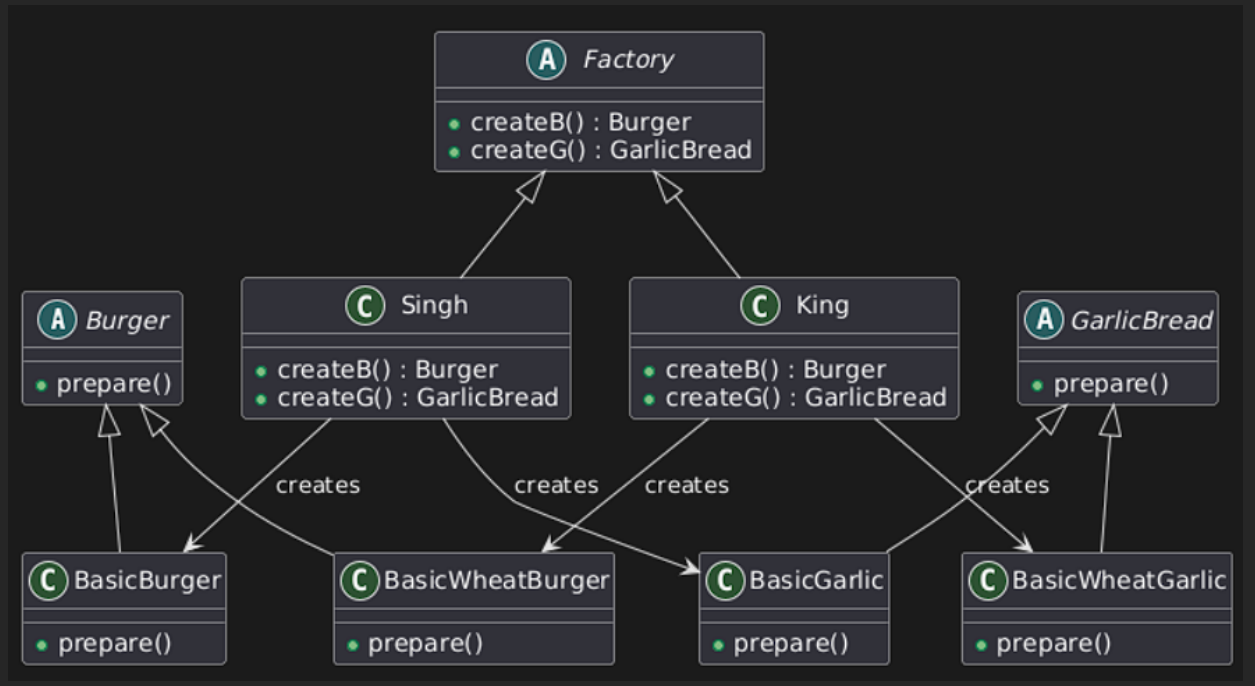
The Abstract Factory Pattern delegates the responsibility of creating these product families to its derived classes, where each concrete factory produces a set of related products that are designed to work together. This promotes loose coupling between client code and specific product implementations, making it easier to switch between product families. By encapsulating the creation logic, the pattern supports the Open/Closed Principle, allowing new product families to be introduced with minimal changes to existing code. This design enhances scalability and consistency when building systems that require interchangeable components with compatible interfaces.



The given UML diagram represents the Abstract Factory design pattern, which is used to create families of related or dependent objects without specifying their concrete classes. In this diagram, Burger and GarlicBread are abstract product classes that define a common interface (prepare()) for all types of burgers and garlic bread, respectively. Their concrete implementations include BasicBurger and BasicWheatBurger for burgers, and BasicGarlic and BasicWheatGarlic for garlic bread. These concrete classes form two distinct product families.

At the core of the pattern is the abstract class Factory, which declares methods createB() and createG() for creating burger and garlic bread objects. Two concrete factories, Singh and King, extend the Factory class and implement these methods to produce specific product families. The Singh factory creates a BasicBurger and a BasicGarlic, while the King factory creates a BasicWheatBurger and a BasicWheatGarlic. Each factory ensures that the products it creates are compatible with each other, maintaining consistency within the product family.

This structure promotes loose coupling between client code and the actual product classes, as the client interacts only with the abstract interfaces. It also adheres to the Open/Closed Principle, allowing new product families to be introduced by simply creating new factory subclasses without altering existing code. Overall, the diagram illustrates a clean and effective implementation of the Abstract Factory pattern for managing related objects in a scalable and maintainable way.



Singleton Design Pattern

A singleton class is a design pattern that ensures a class has only one instance throughout the lifetime of a program and provides a global point of access to that instance.

Singleton Design Pattern restricts the instantiation of a class to a single object and provides a way to access that single instance globally.

```cpp
class Singleton {
private:
    static Singleton* instance;

    Singleton() {
        cout << "Singleton Constructor called" << endl;
    }

public:
    static Singleton* getInstance() {
        if(instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};

// Initialize static member
Singleton* Singleton::instance = nullptr;

int main() {
    Singleton* s1 = Singleton::getInstance();
    Singleton* s2 = Singleton::getInstance();

    cout << (s1 == s2) << endl;
}
```

In this code, the Singleton class ensures that the instance is created only once. The first call to getInstance() checks if the instance pointer is nullptr, and if so, creates a new Singleton object.
On subsequent calls, since the instance is no longer nullptr, it simply returns the already created instance.
This guarantees that only one object of the class is used throughout the program.

However, this implementation is not thread-safe.
In a multithreaded environment, if two threads call getInstance() at the same time when instance is still nullptr, both might enter the if block and create multiple instances, violating the Singleton principle.
To make it thread-safe, synchronization mechanisms like mutexes, double-checked locking, or using a static local variable (in C++11 and later) should be used.

MUTEX : for thread-safety

This version of the Singleton pattern ensures that only one instance of the class is ever created, and it is done in a thread-safe way using a mutex.

```
class Singleton {
private:
    static Singleton* instance;
    static mutex mtx;

    Singleton() {
        cout << "Singleton Constructor Called!" << endl;
    }

public:
    static Singleton* getInstance() {
        lock_guard<mutex> lock(mtx); // Lock for thread safety
        if (instance == nullptr) {
            instance = new Singleton();
        }
         return instance;
    }
};

// Initialize static members
Singleton* Singleton::instance = nullptr;
mutex Singleton::mtx;
```

- static Singleton* instance; Stores the only instance of the Singleton class.
- static mutex mtx; Ensures mutual exclusion so only one thread can create the instance.
- Private Constructor: Prevents creation of objects from outside the class.
- getInstance(): Public static method to access the singleton instance.

1. Lock Acquired Immediately: As soon as getInstance() is called, the thread locks the mutex using lock_guard.
This ensures only one thread at a time can enter the critical section.
2. Check and Create Instance: Once inside the locked section, it checks if instance is nullptr.
If it is, the singleton object is created.
Otherwise, it just returns the existing instance.
3. Automatic Unlock: When the lock_guard goes out of scope, the mutex is automatically released.

DOUBLE-CHECKED LOCKING : for thread-safety

Problem with the previous approach: Lock is an expensive operation. So, we should try to avoid it. To make it less expensive we will again put an if-condition to check if instance ==nullptr, only then run this locking for mutual exclusion. Else, it will just return the instance. This is why we used the “First Check”.

```
public:
    // Double check locking..
    static Singleton* getInstance() {
        if (instance == nullptr) {  // First check (no locking)
            lock_guard<mutex> lock(mtx);  // Lock only if needed
            if (instance == nullptr) {  // Second check (after acquiring lock)
                instance = new Singleton();
            }
        }
        return instance;
    }
};
```

This implementation ensures that the Singleton class will create only one instance, even in a multi-threaded environment, by using a technique called double-checked locking.

1. First Check (if (instance == nullptr)): Quick, non-locking check. Most calls to getInstance() will skip the lock for performance if the instance is already created.
2. Locking (lock_guard<mutex> lock(mtx)): If the instance is still nullptr, a thread acquires the mutex lock. lock_guard is RAII-based: it automatically releases the lock when it goes out of scope.
3. Second Check (if (instance == nullptr)): Ensures that no other thread has created the instance while the current thread was waiting for the lock. Only the first thread that enters this block creates the instance.
4. Subsequent Calls: Once the instance is created, all threads will pass the first if check and avoid locking.

EAGER INITIALIZATION: for thread-safety

Eager Initialization is a Singleton pattern approach where the instance is created at the time of class loading, rather than when it's first requested. It is better because we didn't need any if-else condition or had to use locking (which would have been expensive).

```
class Singleton {
private:
    static Singleton* instance;

    Singleton() {
        cout << "Singleton Constructor Called!" << endl;
    }

public:
    static Singleton* getInstance() {
        return instance;
    }
};

// Initialize static members
Singleton* Singleton::instance = new Singleton();

int main() {
    Singleton* s1 = Singleton::getInstance();
    Singleton* s2 = Singleton::getInstance();

    cout << (s1 == s2) << endl;
}
```

- This means the object is constructed before main() even starts — ensuring thread safety without any need for locking, since the instance is created before multiple threads can access it.
- This ensures that the object is ready before any thread accesses it, and eliminates the need for synchronization (locks), since the instance is created by the time any thread can call getInstance().

```
// Initialize static members
Singleton* Singleton::instance = new Singleton();
```

This ensures that the singleton instance is created once and only once, during program startup. Since this happens before any threads begin execution, it eliminates race conditions — thus making locking unnecessary. It's simple, thread-safe by default, and ideal when the Singleton is lightweight, but it creates the instance regardless of whether it's ever used, which can be wasteful in some cases and can take up memory.
Both s1 and s2 just return the same instance.

❖ **So, we only use Eager Initialization only when object is lightweight.**

Conclusion:

Singleton Design:

1. Create a private constructor.
2. Create a static instance (getInstance()) that returns the same instance every time.

Real World Usage:

1. Logging System

   In most applications, logs need to be written to a single shared output — like a file, console, or monitoring system. If multiple logger instances existed, they could create file access conflicts or duplicate entries. Ensures only one Logger instance manages all logging activity across the app, avoiding sync issues and keeping log format consistent.

2. Database Connection

   Applications typically use a single point of access to a database to control resources, manage connection pooling, and maintain consistent state. Only one DB connection manager (or pool) is needed. Singleton ensures all database requests are handled through a shared, centralized connection, improving performance and resource usage.

3. Configuration Manager

   Applications often need to read configuration (e.g., settings from a file or environment) that shouldn't change at runtime. A single instance of a configuration manager ensures that all parts of the application use the same settings without reloading or redefining them, preserving consistency and avoiding overhead.