ERROR HANDLING

Anil
Kasun
Chashara

What are Software Design Patterns?

- Reusable solutions to common software problems.
- Provide a structured approach to building applications.
- Help create scalable, maintainable, and flexible systems.

Importance of Error Handling

- Ensures software reliability and resilience.
- Helps handle unexpected scenarios gracefully.
- Java provides try-catch, checked, and unchecked exceptions.
- Critical for both runtime and validation errors.

Anil

Factory Pattern and Error Handling

- Factory Pattern: Encapsulates object creation.
- Error Handling: Deal with unsupported or invalid object requests.

Anil

Factory Pattern Code Example

```java
public abstract class Restaurant {
    public Burger orderBurger() {
        Burger burger = createBurger();
        burger.prepare();
        return burger;
    }

    public abstract Burger createBurger();
}
```

ConcreteProductB

```java
public class BeefBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new BeefBurger();
    }
}
```

ConcreteCreatorA

```java
public class VeggieBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new VeggieBurger();
    }
}
```

```java
public interface Burger {
    void prepare();
}
```

ConcreteProductA

```java
public class BeefBurger
        implements Burger {

    @Override
    void prepare() {
        // prepare beef
        // burger code
    }
}
```

ConcreteProductB

```java
public class VeggieBurger
        implements Burger {

    @Override
    void prepare() {
        // prepare veggie
        // burger code
    }
}
```

```java
public static void main(String[] args) {

    Restaurant beefResto = new BeefBurgerRestaurant();
    Burger beefBurger = beefResto.orderBurger();

    Restaurant veggieResto = new VeggieBurgerRestaurant();
    Burger veggieBurger = veggieResto.orderBurger();

}
```

Error Handling Strategies

Returning null

```java
if (burger == null) {
    System.out.println("Error: Failed to create burger.");
}
```

Throwing Exceptions

```java
try {
    restaurant.orderBurger();
} catch (BurgerCreationException e) {
    System.out.println("Error: " + e.getMessage());
}
```

Using Optional

```java
burgerOpt.ifPresentOrElse(
    burger -> burger.prepare(),
    () -> System.out.println("Error: No burger available.")
);
```

Returning a Fallback Object

```java
public Burger createBurger() {
    return new DefaultBurger();  // Fallback object
}
```

Strategy Pattern and Error Handling

- The Strategy Pattern is a behavioral design pattern that allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. The pattern lets the algorithm vary independently from clients that use it.

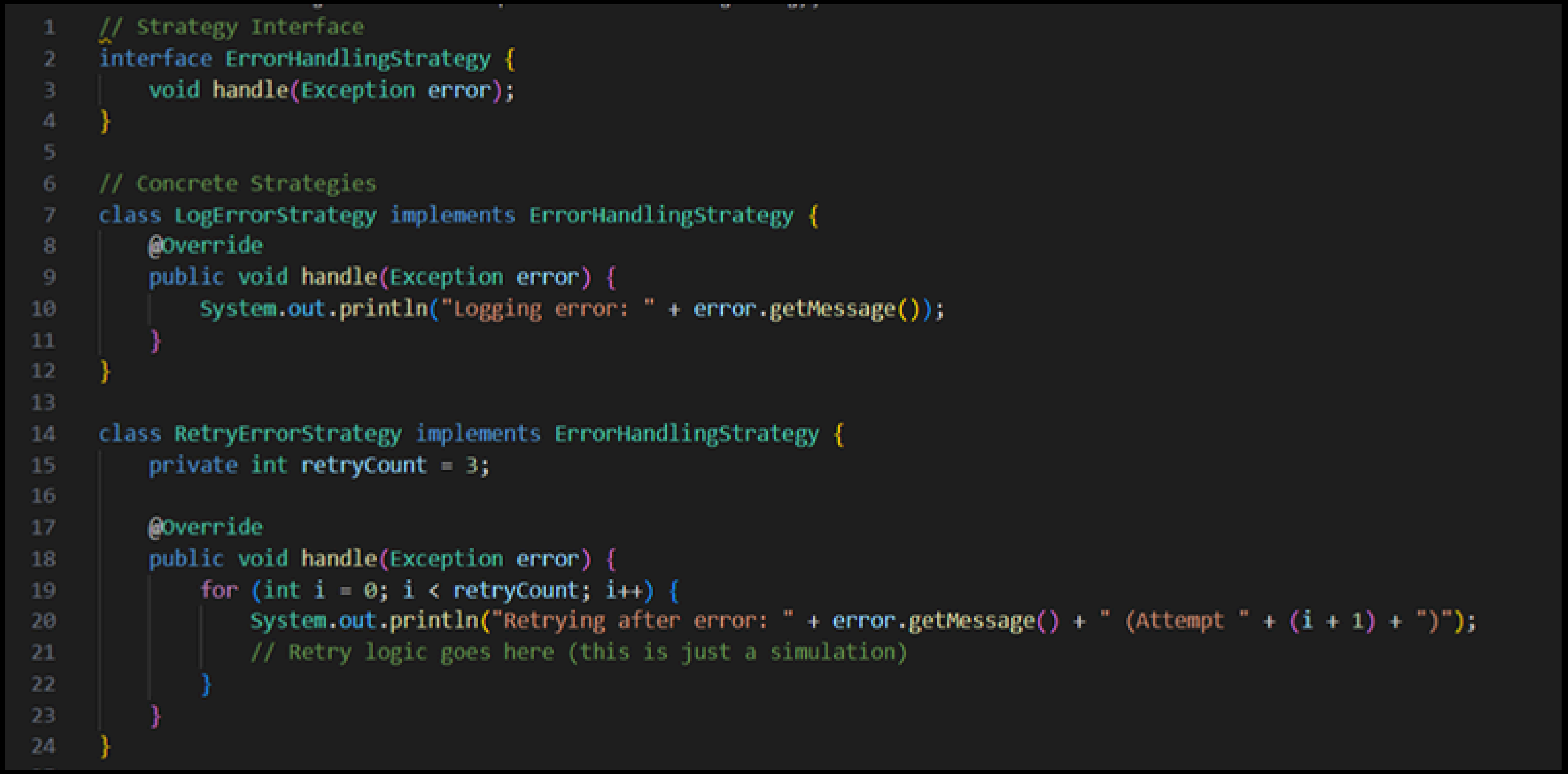
  How It Works:

  - Key Components:
    a. Context: Holds a reference to a strategy object.
    b. Strategy Interface: Declares operations common to all strategies.
    c. Concrete Strategies: Implement the algorithm defined by the strategy interface.

  Error Handling with Strategy Pattern:

  - The Strategy Pattern can be used to handle different types of errors by defining various error-handling strategies (e.g., logging errors, retry mechanisms, exception suppression).
  - You can create multiple error-handling strategies and apply them dynamically based on the situation (e.g., retry for network errors, fallback for data issues).

Strategy Pattern and Error Handling - Example Code

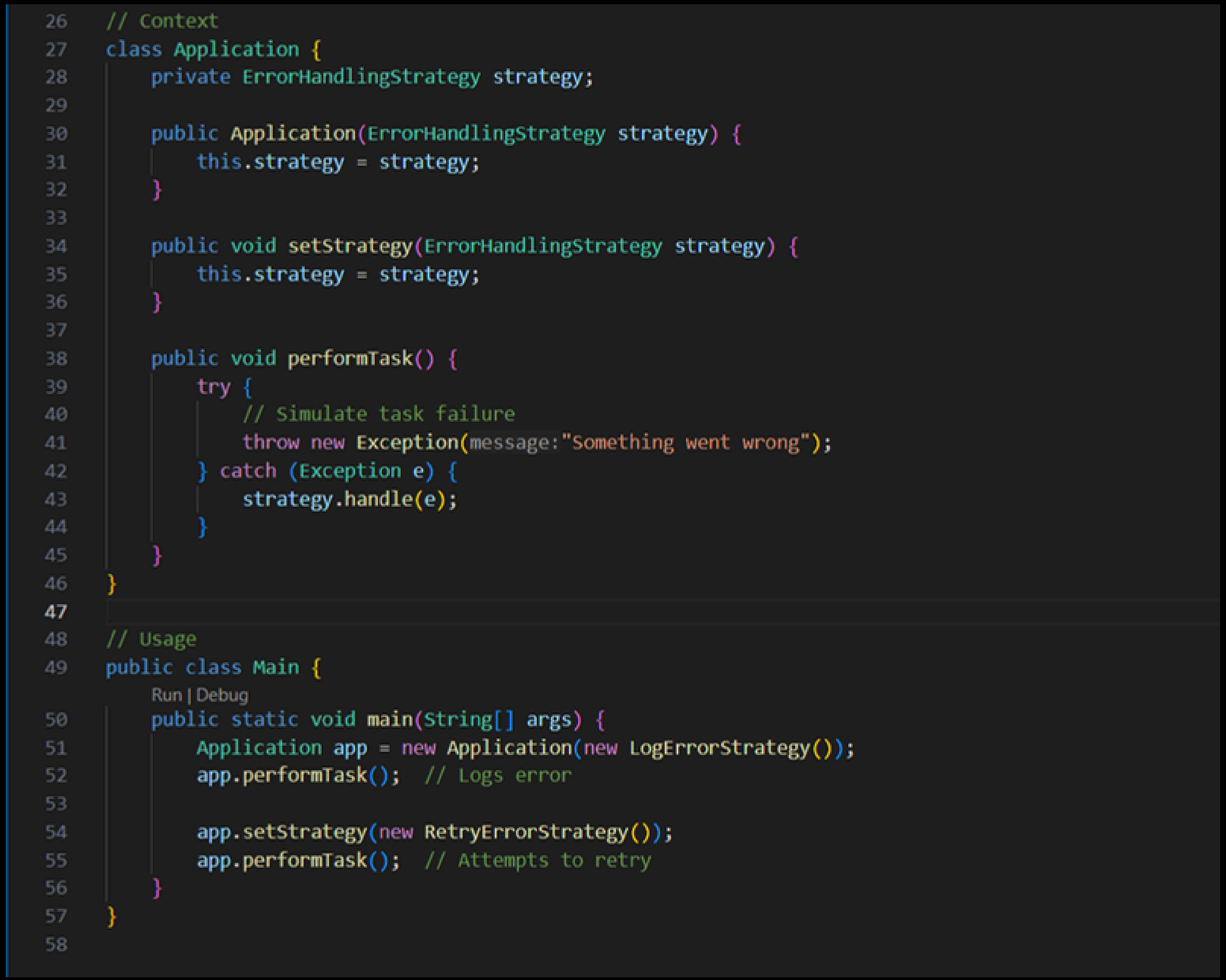


```java
// Strategy Interface
interface ErrorHandlingStrategy {
    void handle(Exception error);
}

// Concrete Strategies
class LogErrorStrategy implements ErrorHandlingStrategy {
    @Override
    public void handle(Exception error) {
        System.out.println("Logging error: " + error.getMessage());
    }
}

class RetryErrorStrategy implements ErrorHandlingStrategy {
    private int retryCount = 3;

    @Override
    public void handle(Exception error) {
        for (int i = 0; i < retryCount; i++) {
            System.out.println("Retrying after error: " + error.getMessage() + " (Attempt " + (i + 1) + ")");
            // Retry logic goes here (this is just a simulation)
        }
    }
}
```



```java
// Context
class Application {
    private ErrorHandlingStrategy strategy;

    public Application(ErrorHandlingStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(ErrorHandlingStrategy strategy) {
        this.strategy = strategy;
    }

    public void performTask() {
        try {
            // Simulate task failure
            throw new Exception(message:"Something went wrong");
        } catch (Exception e) {
            strategy.handle(e);
        }
    }
}

// Usage
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Application app = new Application(new LogErrorStrategy());
        app.performTask();  // Logs error

        app.setStrategy(new RetryErrorStrategy());
        app.performTask();  // Attempts to retry
    }
}
```

Explanation:

- Strategy Interface (ErrorHandlingStrategy): Declares a method handle() that will be implemented by different error-handling strategies.
- Concrete Strategies (LogErrorStrategy, RetryErrorStrategy): Provide specific implementations for handling errors. LogErrorStrategy logs the error, while RetryErrorStrategy simulates retrying the task multiple times.
- Context (Application): Uses the strategy to handle errors. It can switch between different strategies using setStrategy().
- In the Main class, the application initially uses the LogErrorStrategy to log errors, and then it switches to RetryErrorStrategy to handle the same error differently by retrying.

Benefits of the Strategy Pattern:

- Flexibility: You can swap out different strategies at runtime, making the system more adaptable to changing requirements.
- Encapsulation: It keeps the algorithm’s logic separate from the client, making the code more maintainable.
- Scalability: You can easily add new strategies without changing the client’s code, following the Open/Closed Principle (OCP).

Error Handling and Strategy Pattern:

- Modular Error Handling: Different errors can be handled with different strategies, such as retrying, logging, or even raising custom exceptions.

Real-World Use Cases:

- Retry mechanisms for failed network requests.
- Fallback strategies in case of data processing errors.
- Custom error-handling strategies for specific environments (e.g., production vs. development).

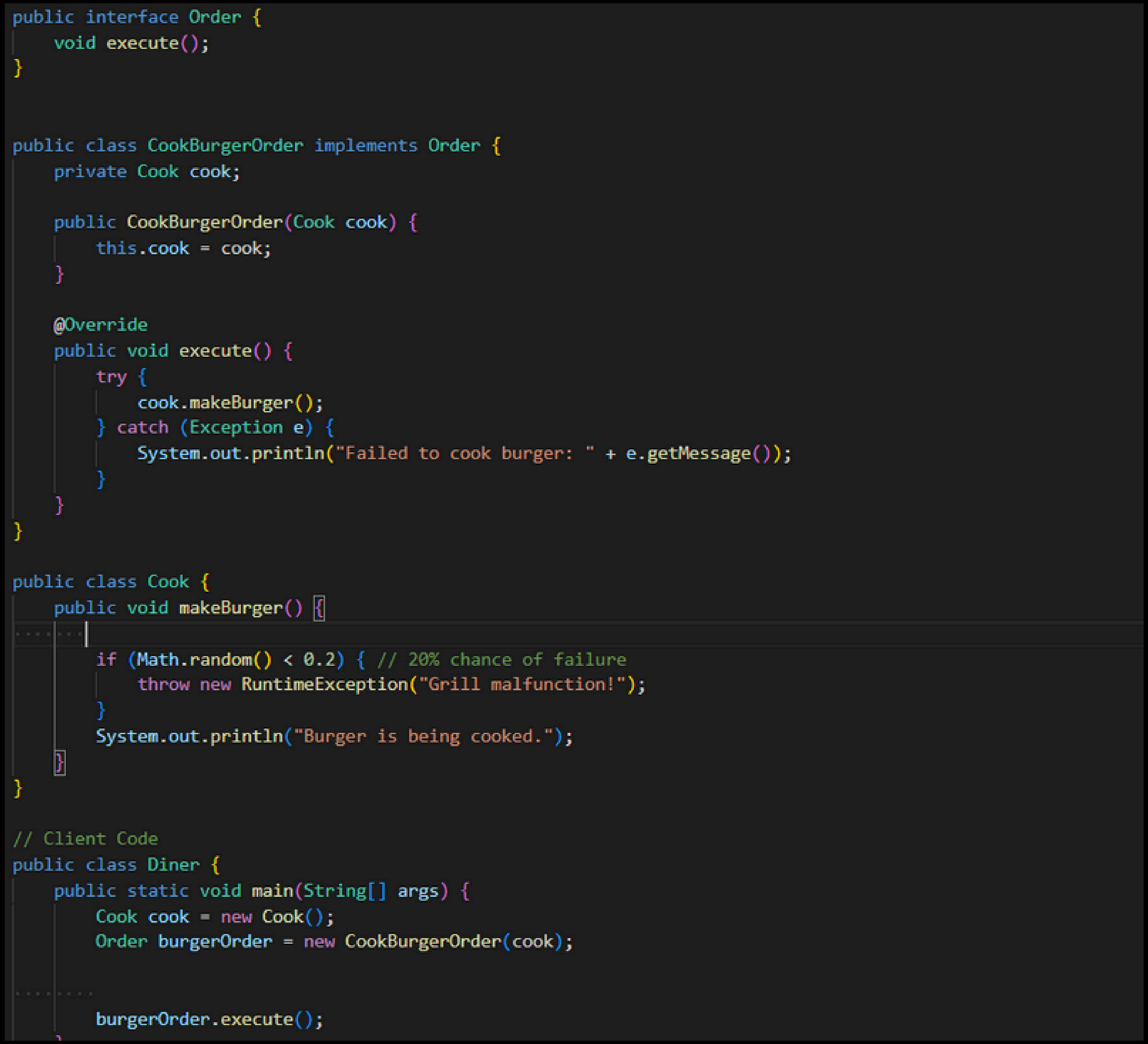
Command Pattern and Error Handling

HOW DOES COMMAND PATTERN WORK?
The command pattern encapulates a request as an object, allowing for parameterization of clients with queues, requests or operations. It decouples the object that involves the operation from the one that knows how to execute it.

ERROR HANDLING WITH COMMAND PATTERN
Involves encapsulating not only actions but also error handling logic within the command object. It enhances flexibility by allowing commands to implement different error recovery mechanisms or fallback operations.
Makes it possible to log, retry or rollback operations if an error occuers during command execution.

CHASHARA

Command Pattern and Error Handling - example code



```java
public interface Order {
    void execute();
}


public class CookBurgerOrder implements Order {
    private Cook cook;

    public CookBurgerOrder(Cook cook) {
        this.cook = cook;
    }

    @Override
    public void execute() {
        try {
            cook.makeBurger();
        } catch (Exception e) {
            System.out.println("Failed to cook burger: " + e.getMessage());
        }
    }
}

public class Cook {
    public void makeBurger() {

        if (Math.random() < 0.2) { // 20% chance of failure
            throw new RuntimeException("Grill malfunction!");
        }
        System.out.println("Burger is being cooked.");
    }
}

// Client Code
public class Diner {
    public static void main(String[] args) {
        Cook cook = new Cook();
        Order burgerOrder = new CookBurgerOrder(cook);


        burgerOrder.execute();
```

CHASHARA

Global vs Local Error handling

Global	Local
• Centralized handling of error acrpss the entire application • Typically involves a single point of control, like a global error handler. • Commonly used for logging, user notifications or fallball mechanisms • Easier maintenence, consistent behaviour. • Suitable for non critical errors	• Error handling is specific to a particular module or function • Allows for custom responses based on the context where the error occurs • Useful for specific recovery actions or detailed error information • More flexibility, context aware handling, better suited for critical operations.

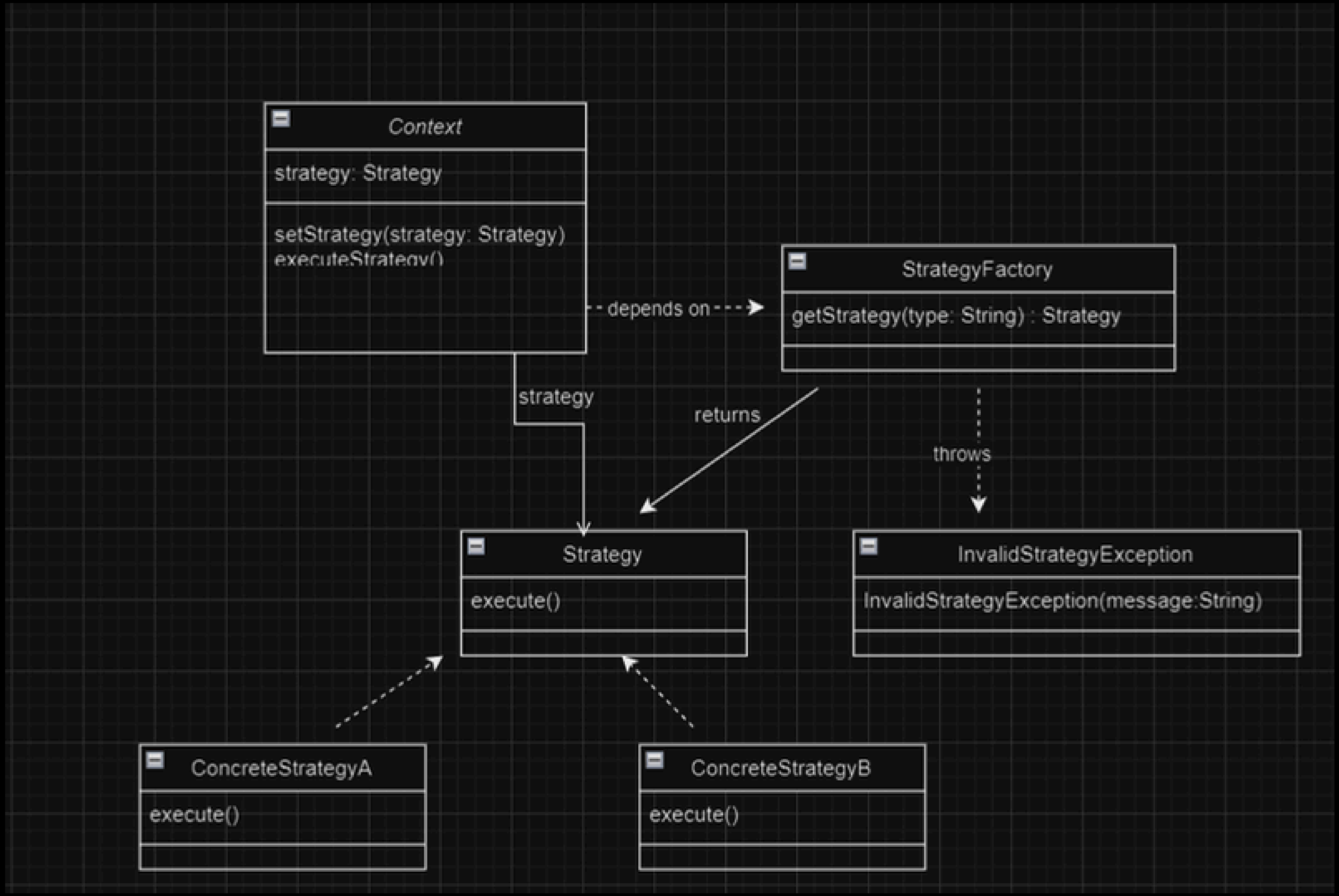
CHASHARA

UML

- Factory Pattern handles invalid object creation requests.

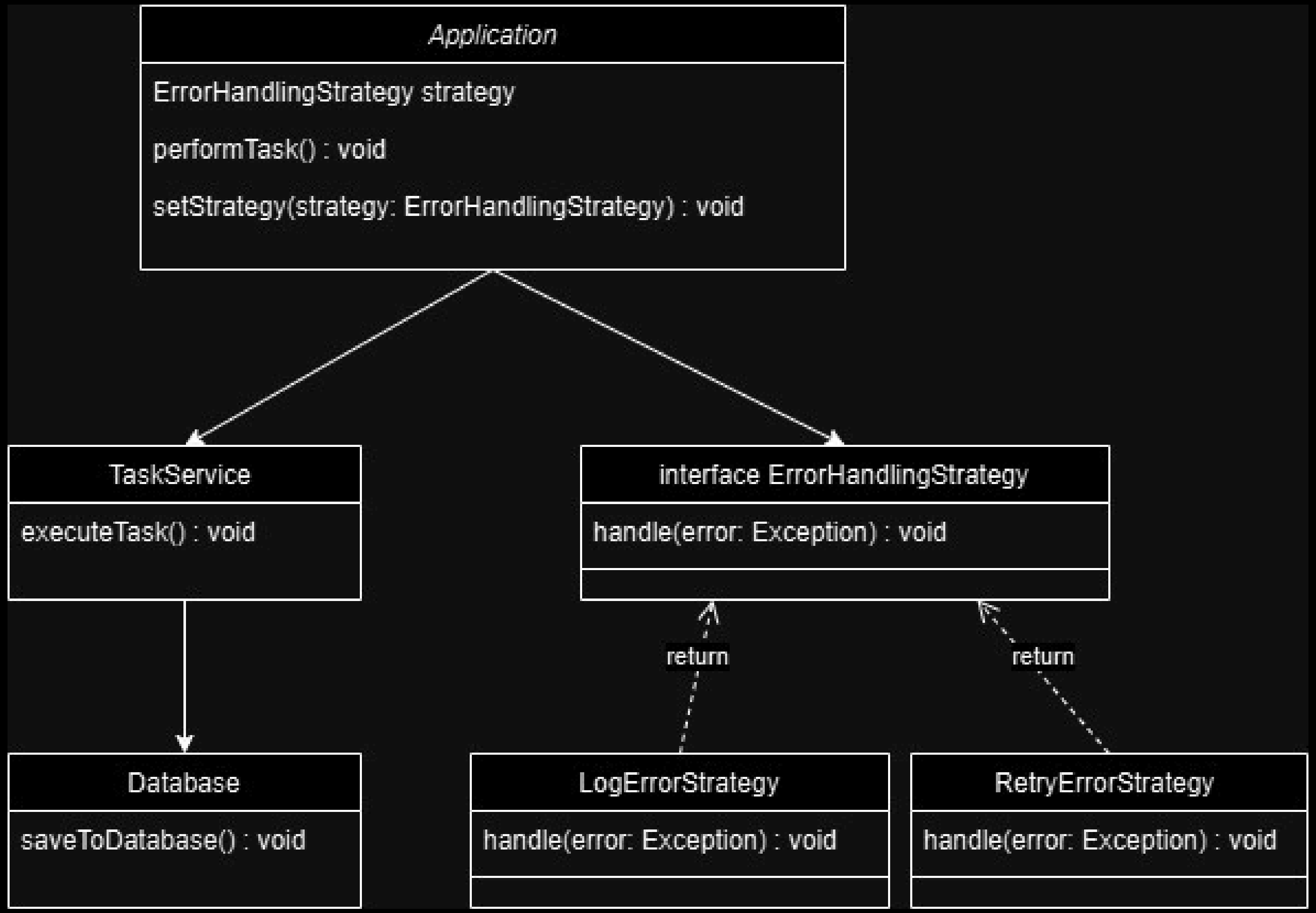
Anil

UML diagram: Strategy Pattern handles invalid algorithm selections.

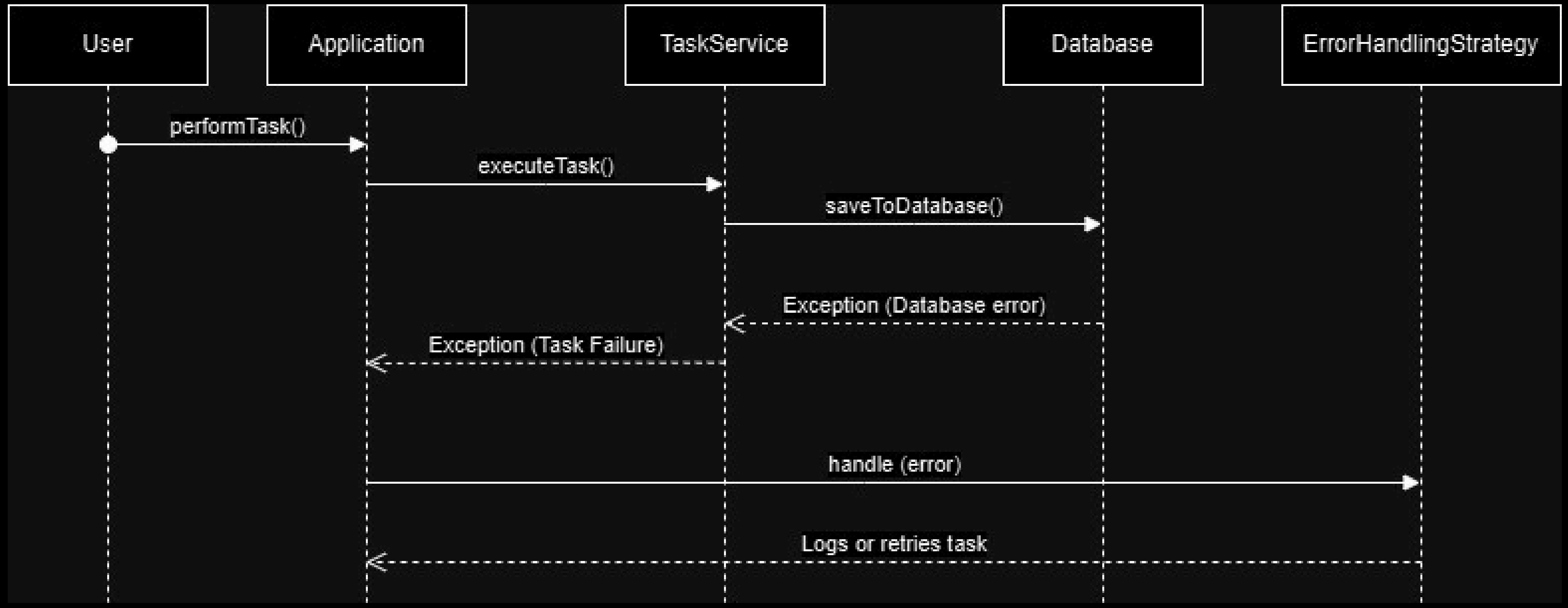


CHASHARA

UML Diagram - Flow of exceptions visualized in the diagram.



Application
ErrorHandlingStrategy strategy
performTask() : void
setStrategy(strategy: ErrorHandlingStrategy) : void
TaskService
executeTask() : void
interface ErrorHandlingStrategy
handle(error: Exception) : void
return
return
Database
saveToDatabase() : void
LogErrorStrategy
handle(error: Exception) : void
RetryErrorStrategy
handle(error: Exception) : void



Kasun

Best Practices for Error Handling

- Use exceptions only for exceptional conditions.
- Catch specific exceptions, and avoid catching too broadly.
- Log exceptions for debugging purposes.
- Provide clear, user-friendly error messages.
- Use custom exceptions to make error handling meaningful.
- Clean up resources using finally blocks or try-with-resources.
- Implement retry logic where appropriate.

Following these best practices will help you build more robust, maintainable, and predictable error-handling mechanisms in your software systems.

Conclusion

- Error handling is vital for maintaining application stability and user trust.
- Implementing structured error handling improves system reliability and reduces downtime.
- Effective error handling enhances user experience by providing clear feedback on issues.
- Proper logging of errors is essential for diagnosing and resolving issues quickly.
- Using design patterns facilitates consistent and effective error management across applications.
- Custom exceptions allow for more meaningful error reporting tailored to specific application needs.
- Incorporating retries for transient errors minimizes disruption and improves resilience.
- Defensive programming practices prevent invalid inputs and enhance overall software integrity.
- Narrowing try-catch blocks helps isolate issues and makes debugging more straightforward.
- Adhering to best practices in error handling contributes to cleaner, more maintainable code.

QUESTIONS?