Software Architectures and Design

ERROR HANDLING

by: Anil, Chashara, Kasun

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.

Anil

Importance of Error Handling

- Ensures software reliability and resilience.
- Helps handle unexpected scenarios gracefully.
- Critical for both runtime and validation errors.

Anil

Factory Pattern and Error Handling

- **Creation Design pattern**
- **Encapsulation of object creation**
- **Flexibility and Decoupling**

Errors handling are done within the factory method to centralize the error handling logic

Anil

Factory Pattern Code Example

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

    public abstract Burger createBurger();
}
```

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

```java
public class BeefBurger
        implements Burger {

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

}
```

ConcreteProductA

ConcreteCreatorA

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

```java
public class VeggieBurger
        implements Burger {

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

}
```

ConcreteProductB

ConcreteCreatorB

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

```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

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

Throwing Exceptions

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

Using Optional

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

Returning a Fallback Object

```
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
    b. Strategy Interface
    c. Concrete Strategies

  Error Handling with Strategy Pattern:

  - The Strategy Pattern allows handling various errors by defining different error-handling strategies (e.g., logging, retries, exception suppression). Multiple strategies can be created and applied 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
    }
}
```

Benefits of the Strategy Pattern:

- Flexibility
- Encapsulation
- Scalability

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.

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

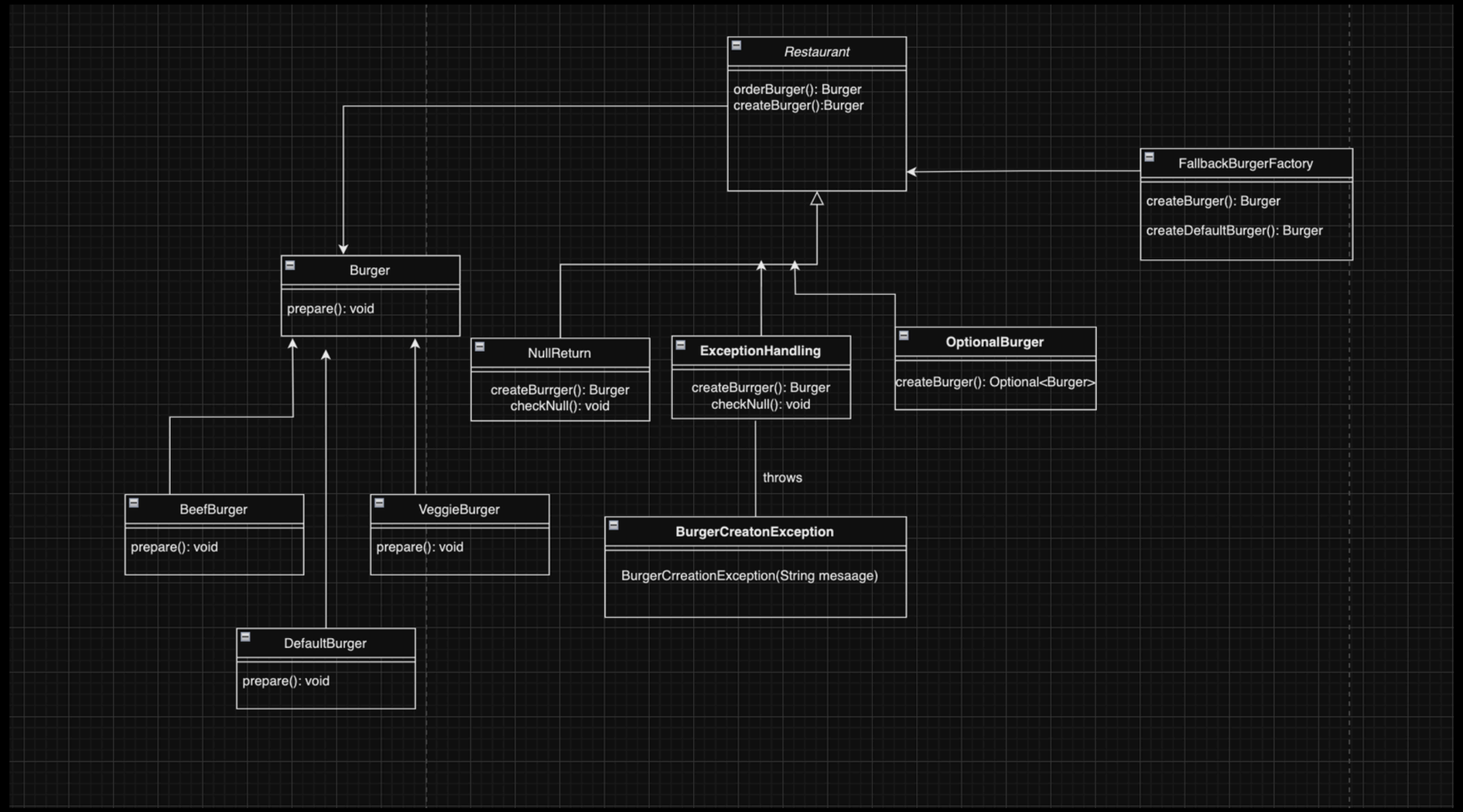
- Centralized handling of error acrpss the entire application
- Typically involves a single point of control, like a global error handler.
- Commonly used for logging, user notifications or fallball mechanisms
- Easier maintenence, consistent behaviour.
- Suitable for non critical errors

Local

- 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.

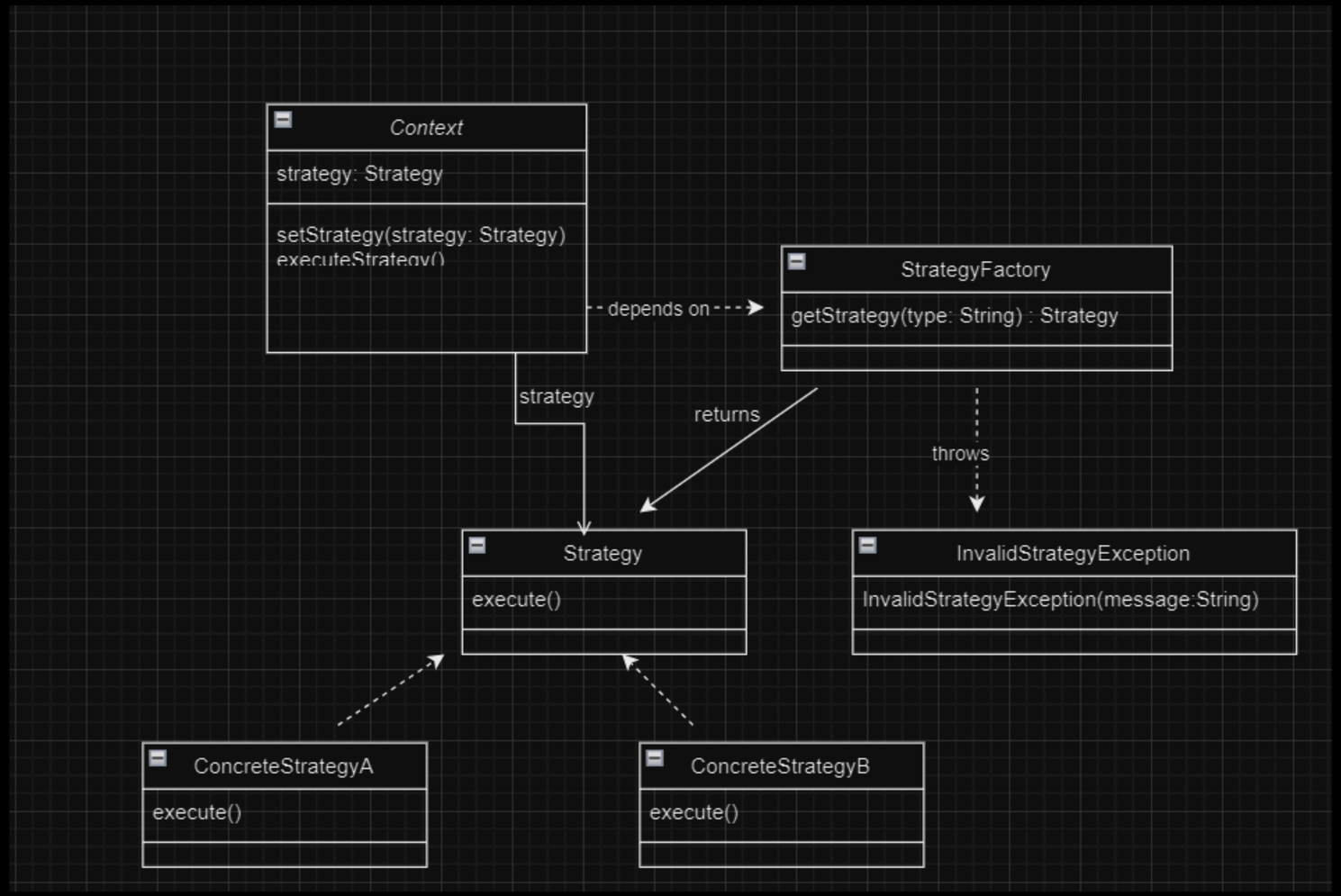
CHASHARA

Factory pattern and Error Handling



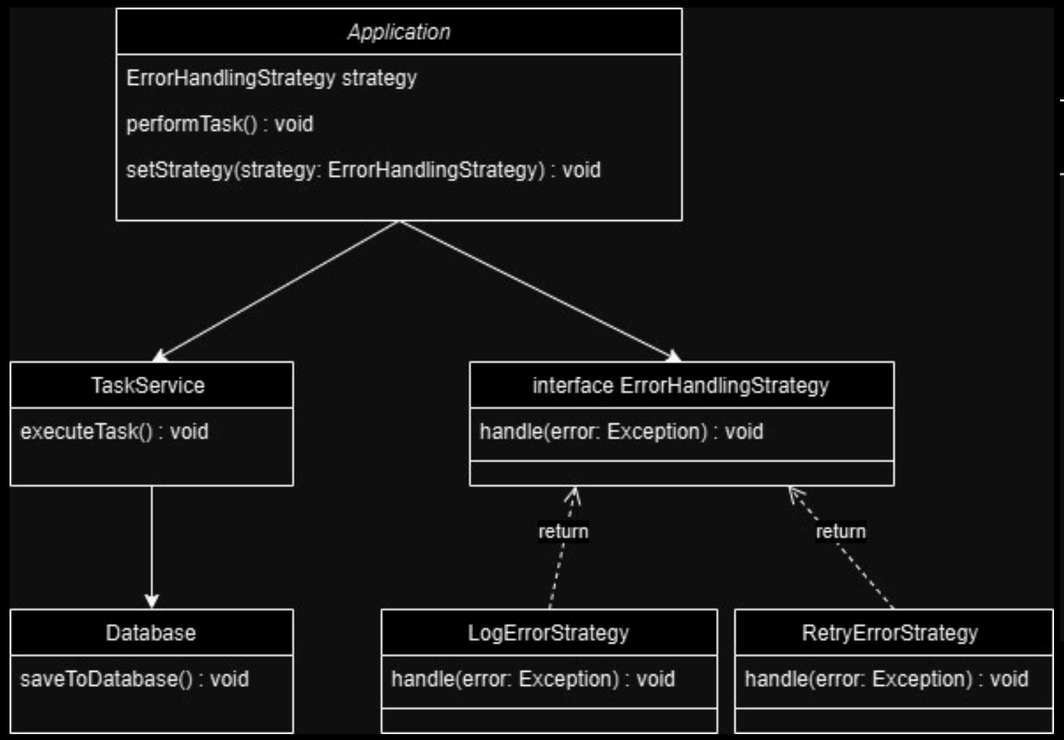
Anil

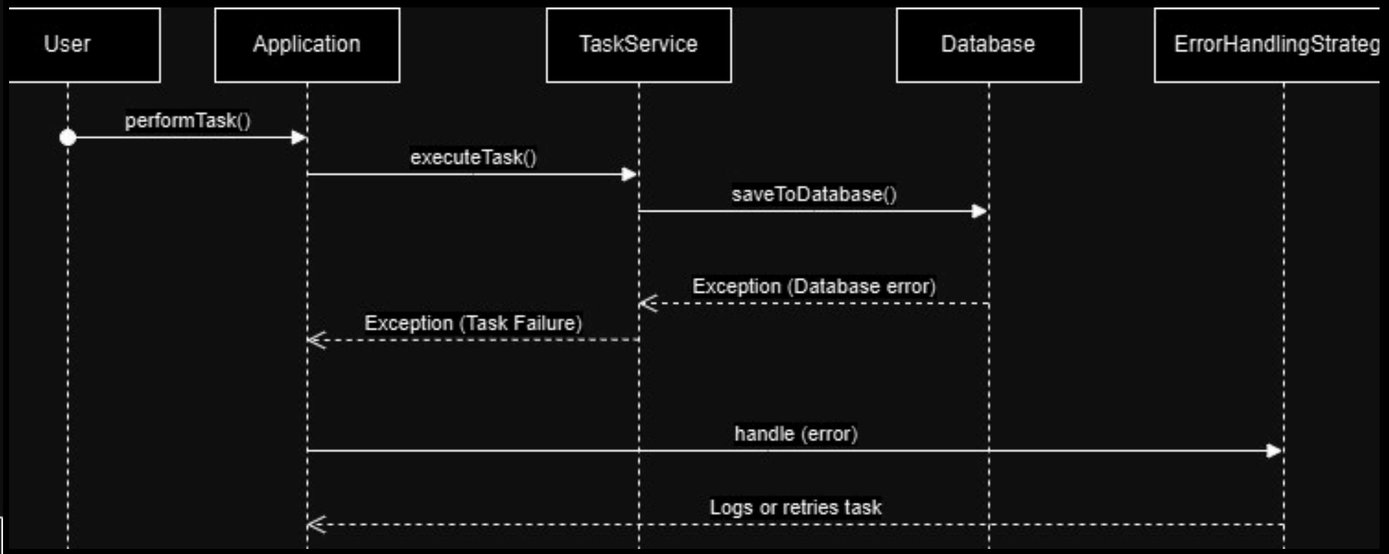
UML diagram: Strategy Pattern handles invalid algorithm selections.



CHASHARA

UML Diagram - Flow of exceptions visualized in the diagram.





Kasun

Best Practices for Error Handling

- Use only for exceptional conditions.
- Catch specific exceptions, 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.

Conclusion

- Error handling is crucial for maintaining stability and user trust.
- Structured error handling improves reliability and reduces downtime, while clear feedback enhances the user experience.
- Proper logging aids quick issue resolution.
- Design patterns ensure consistent error management, and custom exceptions allow more meaningful reporting.
- Retries for transient errors enhance resilience, and defensive programming prevents invalid inputs.
- Using focused try-catch blocks simplifies debugging, and following best practices leads to cleaner, maintainable code.

QUESTIONS?

Software Architectures and Design

THANK YOU!

by: Anil, Chashara, Kasun