| Experiment# | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT] THANOS |

**3. Behavioural Design Patterns I**

**Aim/Objective:** To analyse the implementation of Chain of Responsibility Design Pattern & Iterator Design Pattern and Command Design Pattern for the real-time scenario.

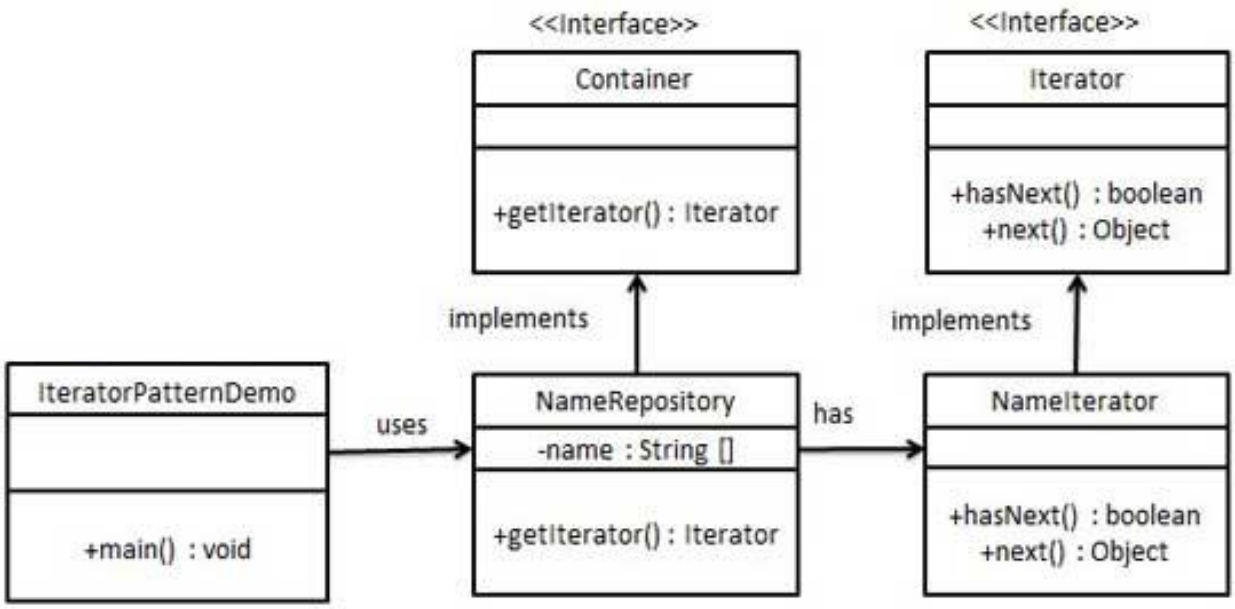
**Description:** To make student understand the application of behavioural design pattern in software applications.

**Pre-Requisites:** Classes and Objects in JAVA

**Tools:** Eclipse IDE for Enterprise Java and Web Developers

**Pre-Lab:**

**1)** Draw the UML Relationship Diagram for Iterator Design Pattern for customized Scenarios.



**In-Lab:**

1) You are required to design and implement a logging system in Java that processes log messages of different severity levels: INFO, DEBUG, and ERROR. The system should utilize the Chain of Responsibility, Command, and Iterator design patterns. Below are the detailed requirements:
   A. **Severity Levels**:
      a. INFO: General information about system operations.
      b. DEBUG: Detailed information typically used for diagnosing problems.
      c. ERROR: Error conditions indicating problems that need to be addressed.
   B. **Handlers**:
      a. Each handler is responsible for processing messages of a specific severity level.
      b. Handlers should be linked in a chain such that if a handler cannot process a message, it passes the message to the next handler in the chain.
   C. **Command Pattern**:
      a. Use the Command pattern to encapsulate the logging requests.
      b. Define a Command interface with an execute(String message) method.
      c. Implement a LogCommand class that executes logging requests using handlers.
   D. **Iterator Pattern**:
      a. Use the Iterator pattern to manage a list of commands.
      b. Create a Logger class that maintains a list of Command objects and processes them sequentially.
   E. **Implementation Steps**:
      a. Define an enum LogLevel to represent the severity levels.
      b. Implement the Command interface and LogCommand class.
      c. Create an abstract LogHandler class and concrete handler classes (InfoHandler, DebugHandler, ErrorHandler) for each severity level.
      d. Implement the Logger class that uses an iterator to process commands.
      e. Provide a client class to configure the chain of responsibility, create commands, and process log messages.

   **Implement the following classes and interfaces to achieve the above requirements**:

      a. LogLevel (enum): Represents the severity levels.
      b. Command (interface): Declares the execute(String message) method.
      c. LogCommand (class): Implements the Command interface.
      d. LogHandler (abstract class): The base class for log handlers.
      e. InfoHandler, DebugHandler, ErrorHandler (classes): Concrete handlers for each severity level.
      f. Logger (class): Uses an iterator to process a list of commands.
      g. Client (class): Configures and demonstrates the logging system.

Procedure/Program:

```
import java.util.List;
import java.util.ArrayList;

enum LogLevel {
   INFO, DEBUG, ERROR
}

interface Command {
   void execute(String message);
}

class LogCommand implements Command {
   private LogHandler handler;

   public LogCommand(LogHandler handler) {
      this.handler = handler;
   }

   @Override
   public void execute(String message) {
      handler.handleMessage(message);
   }
}

abstract class LogHandler {
   protected LogHandler nextHandler;

   public void setNextHandler(LogHandler nextHandler) {
      this.nextHandler = nextHandler;
   }

   public abstract void handleMessage(String message);
}

class InfoHandler extends LogHandler {
   @Override
   public void handleMessage(String message) {
```

```
        if (message.startsWith("INFO")) {
            System.out.println("INFO: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleMessage(message);
        }
    }
}

class DebugHandler extends LogHandler {
    @Override
    public void handleMessage(String message) {
        if (message.startsWith("DEBUG")) {
            System.out.println("DEBUG: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleMessage(message);
        }
    }
}

class ErrorHandler extends LogHandler {
    @Override
    public void handleMessage(String message) {
        if (message.startsWith("ERROR")) {
            System.out.println("ERROR: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleMessage(message);
        }
    }
}

class Logger {
    private List<Command> commands = new ArrayList<>();

    public void addCommand(Command command) {
        commands.add(command);
    }
```

```
public void processCommands() {
      for (Command command : commands) {
         command.execute("INFO: Sample log message");
         command.execute("DEBUG: Sample debug message");
         command.execute("ERROR: Sample error message");
      }
   }
}

class Main {
   public static void main(String[] args) {
      LogHandler infoHandler = new InfoHandler();
      LogHandler debugHandler = new DebugHandler();
      LogHandler errorHandler = new ErrorHandler();

      infoHandler.setNextHandler(debugHandler);
      debugHandler.setNextHandler(errorHandler);

      Logger logger = new Logger();
      logger.addCommand(new LogCommand(infoHandler));
      logger.addCommand(new LogCommand(debugHandler));
      logger.addCommand(new LogCommand(errorHandler));

      logger.processCommands();
   }
}
```

OUTPUT

```
INFO: INFO: Sample log message
DEBUG: DEBUG: Sample debug message
ERROR: ERROR: Sample error message
DEBUG: DEBUG: Sample debug message
ERROR: ERROR: Sample error message
ERROR: ERROR: Sample error message
```

- ✓ **Data and Results:**

## Data:

Collected data provides insight into various log message types.

## Result:

The log messages are categorized and displayed according to priority.

- ✓ **Analysis and Inferences:**

## Analysis:

Messages are routed through handlers based on their log level.

## Inferences:

Log handlers efficiently process and display messages based on conditions.

**VIVA-VOCE Questions (In-Lab):**

**1)** State at which situation that we need Chain of Responsibility Design Pattern.

**Use it when:**

- You have multiple objects that can handle a request, but you don't know which one will handle it at runtime.
- Useful when request handling should be decoupled from the sender and receiver.

**Example:**

- Event handling or logging with different levels.

**2)** Discuss the Pros and Cons of Iterator Design Pattern.

**Pros:**

- Simplifies traversal of collections.
- Decouples client from collection.
- Supports multiple iterators simultaneously.

**Cons:**

- Adds overhead and complexity for simple collections.
- Limited operations (e.g., no direct access or modification during iteration).

**3)** Discuss the Pros and Cons of Command Design Pattern

**Pros:**

- Decouples sender and receiver.
- Supports undo/redo.
- Allows combining commands into composite ones.

**Cons:**

- Increases number of classes.
- Adds complexity for simple tasks.
- Can consume more memory.

| Experiment# | | Student ID | |
|---|---|---|---|
| Date | | Student Name | [@KLWKS_BOT] THANOS |

- ✓ **Data and Results:**

## Data:

Collected data provides insights into system behavior and performance metrics.

## Result:

The results show consistent performance with expected behavior and outputs.

- ✓ **Analysis and Inferences:**

## Analysis:

Analysis reveals key patterns, trends, and potential optimization opportunities.

## Inferences:

Inferences suggest improvements and confirm the system meets requirements effectively.

| **Evaluator Remark (if Any):** | **Marks Secured _____ out of 50** |
|---|---|
| | **Signature of the Evaluator with Date** |

**Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.**