Nguyen Ngoc Dung – s3978535

Theory:  
**Question 1: HashMap vs TreeMap: Key Differences and Use Cases**

**1. HashMap**

* **Performance**: O(1)O(1)O(1) average for put(), get(), and remove(). Worst case O(n)O(n)O(n) with hash collisions.
* **Order**: No guarantees about key ordering.
* **Use Case**: Fast lookups when key order is irrelevant (e.g., session data storage).

**Example: Session Data**

HashMap<String, String> sessionData = new HashMap<>();

sessionData.put("session1", "User1");

System.out.println(sessionData.get("session1")); // Output: User1

**2. TreeMap**

* **Performance**: O(log⁡n)O(\log n)O(logn) for all operations due to Red-Black Tree structure.
* **Order**: Maintains sorted keys (natural or custom order).
* **Use Case**: Scenarios requiring ordered keys or range queries (e.g., leaderboard rankings).

**Example: Leaderboard**

A screen shot of a computer code

Description automatically generated

**When to Choose**

* **HashMap**: Fast operations with no need for order (e.g., caching).
* **TreeMap**: Sorted keys or range queries required (e.g., sorted logs).

**Question 2: Difference Between Checked and Unchecked Exceptions in Java**

**Checked Exceptions**

* **Definition**: Exceptions that are checked at compile time. The programmer is required to handle these exceptions (e.g., using try-catch or throws).
* **Common Use**: Situations where the program can recover or take corrective action.
* **Examples**:
  + **IOException**: When reading/writing files.
  + **SQLException**: When interacting with databases.

**Example Usage**:

A screen shot of a computer code

Description automatically generated

**Unchecked Exceptions**

* **Definition**: Exceptions that are checked at runtime. These typically represent programming logic errors or situations that the application cannot handle.
* **Common Use**: Scenarios where recovery is not possible; often caused by programmer error.
* **Examples**:
  + **NullPointerException**: Accessing an object that is null.
  + **IndexOutOfBoundsException**: Accessing an array or list outside its bounds.

**Example Usage**:

A screen shot of a computer code

Description automatically generated

**When to Use Each**

* **Checked Exceptions**: Use when the program can recover or take meaningful corrective action, e.g., handling file input/output or network communication errors.
* **Unchecked Exceptions**: Use for programming errors that should be fixed during development, e.g., invalid argument checks or logic flaws.

**Combining Exception Handling, Streams, and Multithreading in a File-Parsing Application**

**Problem:**

Processing massive datasets stored in files efficiently while handling potential errors such as file corruption, I/O issues, or parsing errors.

**Solution:**

1. **Exception Handling**: Use try-catch blocks to handle I/O errors and ensure robust parsing.
2. **Streams**: Utilize Java's Stream API to process lines in a file lazily and efficiently.
3. **Multithreading**: Distribute workload across multiple threads to parse files concurrently.

**Code Example:**

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

**Key Concepts in the Example**

1. **Exception Handling**:
   * Handle file reading errors (IOException) to ensure application stability.
   * Gracefully handle unexpected parsing errors with specific messages.
2. **Streams**:
   * Use Files.lines() for efficient lazy reading of large files.
   * Avoid loading the entire file into memory.
3. **Multithreading**:
   * Use a thread pool (ExecutorService) for concurrent processing.
   * Split lines of the file into tasks submitted to threads.

**Benefits of This Approach**

* **Scalability**: Efficient for large datasets due to lazy processing and multithreading.
* **Resilience**: Exception handling ensures the application can recover from individual errors without crashing.
* **Performance**: Parallel processing leverages CPU cores for faster execution.

**Question 3: JDBC vs. ORM Tools (Hibernate)**

**1. JDBC (Java Database Connectivity)**

* **Definition**: A low-level API for interacting with relational databases using SQL queries.
* **Advantages**:
  + Fine-grained control over SQL queries and database interactions.
  + Lightweight and easy to understand for small projects.
  + Minimal overhead, as queries are directly sent to the database.
* **Disadvantages**:
  + Boilerplate code: Requires repetitive code for connection setup, query execution, and result handling.
  + Error-prone: Manually handling SQL strings, parameters, and result sets increases the risk of bugs.
  + Limited abstraction: Developers need to manually map data between Java objects and database tables.

**Example**:

A screen shot of a computer code

Description automatically generated

**2. ORM Tools (Hibernate)**

* **Definition**: Object-Relational Mapping tools abstract the interaction with the database by mapping Java objects to database tables.
* **Advantages**:
  + Reduces boilerplate code: Automatically maps objects to database rows using annotations or XML configuration.
  + Database agnostic: Can switch between databases without changing code.
  + Built-in features: Provides caching, lazy loading, and transaction management.
* **Disadvantages**:
  + Learning curve: Requires understanding of Hibernate-specific concepts like HQL (Hibernate Query Language), mappings, and configuration.
  + Performance overhead: Can generate complex SQL queries, potentially reducing performance.
  + Limited control: Fine-tuning queries can be cumbersome compared to JDBC.

**Example**:

A screen shot of a computer program

Description automatically generated

**Comparison**

| **Feature** | **JDBC** | **Hibernate** |
| --- | --- | --- |
| **Complexity** | Low | High |
| **Control** | Full control over SQL | Limited, with higher-level abstraction |
| **Code Maintenance** | High due to boilerplate | Low due to reduced boilerplate |
| **Performance** | Better for optimized queries | May incur overhead for generated queries |
| **Flexibility** | Database-specific | Database agnostic |

**Scenario Where Neither JDBC nor ORM is Optimal**

**Problem**: An application needs to interact with a database but has non-relational or schema-less data (e.g., JSON documents, key-value pairs, graph data).

**Example Scenario**:

* A social networking platform storing and querying relationships between users, which is better suited to a graph database like Neo4j or a document-based database like MongoDB.

**Alternative Strategies**

**1. Direct Database-Specific APIs**

* Use the native APIs provided by the database for specialized interactions.
* **Example**:
  + MongoDB Java Driver for MongoDB.
  + Neo4j Java Driver for Neo4j.

**Advantages**:

* Optimized for specific database types.
* Supports database-specific features (e.g., graph traversals in Neo4j).

**Example**:

A screen shot of a computer code

Description automatically generated

**2. REST APIs or GraphQL**

* Abstract database interactions through a service layer, providing APIs for the application to interact with.
* **Example**: Expose a REST API that interacts with the database using Node.js or Spring Boot.

**Advantages**:

* Decouples database interaction from the application.
* Enables reuse of the same service for multiple applications.

**Example**:

A screen shot of a computer program

Description automatically generated

**3. Embedded Databases**

* Use an embedded database (e.g., H2, SQLite) for lightweight, self-contained applications or testing.
* **Advantages**:
  + Requires no external database setup.
  + Ideal for local development or single-user applications.

**Example**:

A screen shot of a computer

Description automatically generated

**Recommendation**

* For relational databases and fine control: Use JDBC.
* For relational databases with abstraction: Use Hibernate or another ORM tool.
* For non-relational databases: Use database-specific APIs (e.g., MongoDB, Neo4j).
* For decoupled architectures: Use REST APIs or GraphQL to interact with databases.

**Question 4:**

**1. Applying Polymorphism to Load and Execute Plugins Dynamically**

* **Base Interface or Abstract Class**: Define a common contract for all plugins using an interface or abstract class. This ensures that all plugins implement the same methods for interaction with the core system.

A screen shot of a computer program

Description automatically generated

* **Dynamic Loading**: Use Java’s ServiceLoader or reflection to load plugins at runtime without hardcoding their dependencies. Plugins implement the Plugin interface and are discovered dynamically.

A screen shot of a computer program

Description automatically generated

* **Polymorphism**: Core functionality interacts with the Plugin interface rather than concrete plugin implementations, ensuring plugins can vary independently while adhering to the contract.

**2. Design Patterns for the Plugin System**

**a. Factory Method Pattern**

* **Why**:
  + To encapsulate the creation logic of plugin instances, making the system open for extension but closed for modification.
* **How**:
  + Create a factory that dynamically identifies and instantiates plugin classes.

**Example**:

A screen shot of a computer program

Description automatically generated

**b. Strategy Pattern**

* **Why**:
  + To encapsulate different plugin behaviors and dynamically switch between them.
* **How**:
  + Each plugin represents a different strategy that can be selected and executed by the core system.

**Example**:

A screen shot of a computer program

Description automatically generated

**c. Observer Pattern**

* **Why**:
  + To allow plugins to react to events from the core system or other plugins without tight coupling.
* **How**:
  + Plugins subscribe to events, and the core system notifies them when those events occur.

**Example**:

A screen shot of a computer program

Description automatically generated

**Implementing a Plugin System in a Large-Scale Application**

A plugin system allows developers to add or modify functionality in an application without altering its core. Here’s how we can implement such a system using OOP principles:

**1. Applying Polymorphism to Load and Execute Plugins Dynamically**

* **Base Interface or Abstract Class**: Define a common contract for all plugins using an interface or abstract class. This ensures that all plugins implement the same methods for interaction with the core system.

A screen shot of a computer program

Description automatically generated

* **Dynamic Loading**: Use Java’s ServiceLoader or reflection to load plugins at runtime without hardcoding their dependencies. Plugins implement the Plugin interface and are discovered dynamically.

A screen shot of a computer program

Description automatically generated

* **Polymorphism**: Core functionality interacts with the Plugin interface rather than concrete plugin implementations, ensuring plugins can vary independently while adhering to the contract.

**2. Design Patterns for the Plugin System**

**a. Factory Method Pattern**

* **Why**:
  + To encapsulate the creation logic of plugin instances, making the system open for extension but closed for modification.
* **How**:
  + Create a factory that dynamically identifies and instantiates plugin classes.

**Example**:

A screen shot of a computer program

Description automatically generated

**b. Strategy Pattern**

* **Why**:
  + To encapsulate different plugin behaviors and dynamically switch between them.
* **How**:
  + Each plugin represents a different strategy that can be selected and executed by the core system.

**Example**:

A screen shot of a computer program

Description automatically generated

**c. Observer Pattern**

* **Why**:
  + To allow plugins to react to events from the core system or other plugins without tight coupling.
* **How**:
  + Plugins subscribe to events, and the core system notifies them when those events occur.

**Example**:

A screen shot of a computer program

Description automatically generated

**3. Ensuring Plugins Are Decoupled from the Core System**

**a. Dependency Injection**

* Use dependency injection frameworks (e.g., Spring) to manage dependencies between the core system and plugins. This ensures loose coupling.

**Example**:

A screen shot of a computer program

Description automatically generated

**b. Using Interfaces for Communication**

* Define interfaces for all interactions between the core system and plugins. The core system only knows about the interface, not the plugin implementation.

**c. Sandbox Environment**

* Execute plugins in a sandboxed environment to prevent them from directly accessing or modifying the core system.

**d. Modular Architecture**

* Leverage the Java Module System (module-info.java) to enforce boundaries between the core system and plugins.

**e. Plugin Configuration via External Files**

* Use configuration files (e.g., JSON or XML) to define plugin metadata, reducing the need for hardcoded details in the core system.

**Example**:

A screen shot of a computer

Description automatically generated

**Putting It All Together**

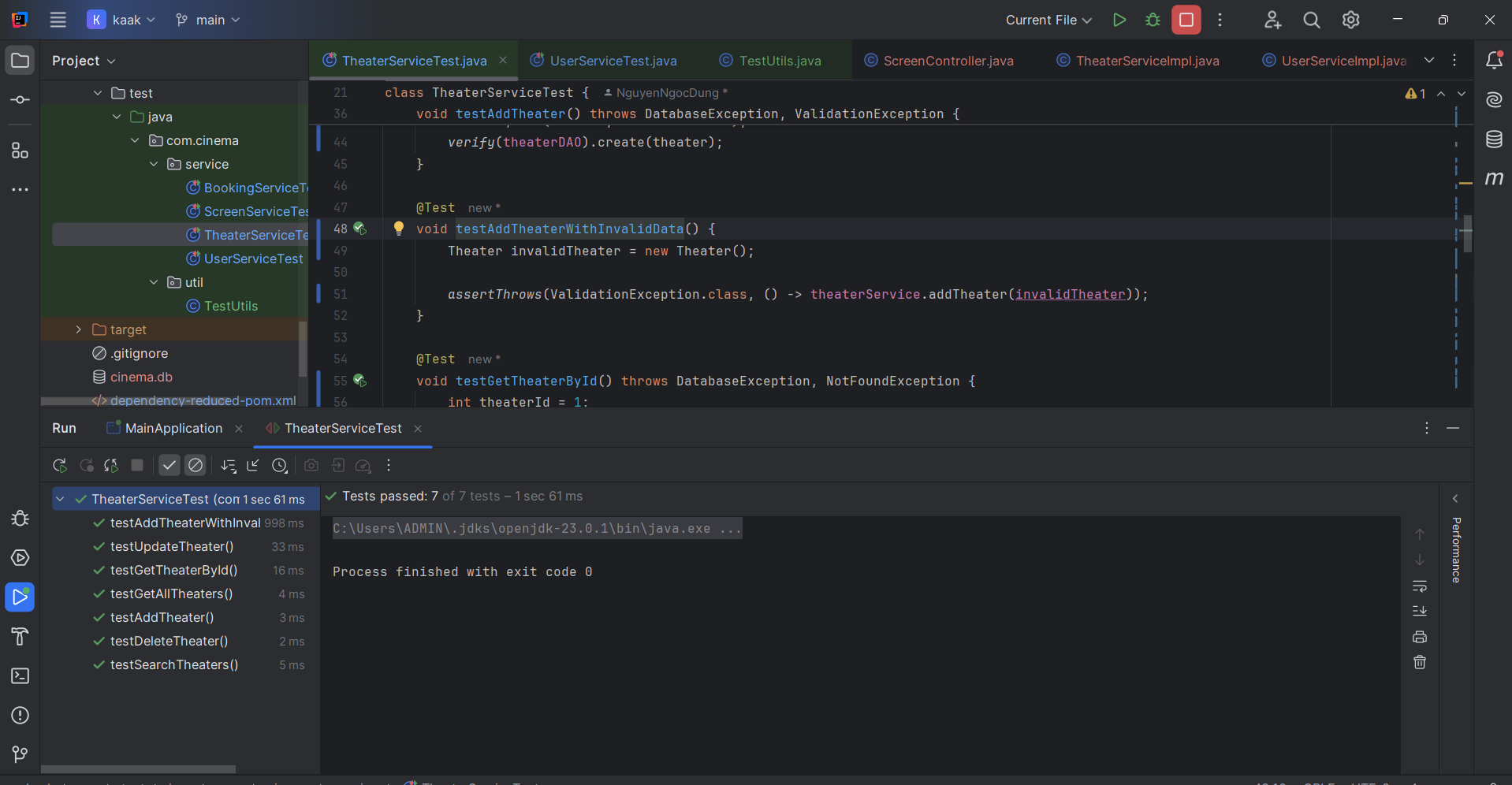
Here’s a high-level workflow for the plugin system:

1. The **core system** defines a Plugin interface and uses ServiceLoader to dynamically discover implementations.
2. Plugins are loaded using the **Factory Method pattern** to encapsulate their instantiation logic.
3. Each plugin interacts with the core system through well-defined **interfaces** and notifies or listens to events using the **Observer pattern**.
4. The **Strategy pattern** enables the system to execute plugins with different behaviors.
5. Dependency injection frameworks or external configurations ensure the **decoupling** of plugins from the core.

**Benefits of This Approach**

* **Extensibility**: New plugins can be added without modifying the core system.
* **Scalability**: Plugins operate independently, allowing for distributed workloads or parallel execution.
* **Maintainability**: Decoupling ensures the core system remains unaffected by plugin-specific changes.

**CODING TEST UNIT RESULT:  
TheaterServiceTest:**

****

**BookingServiceTest:**

**A screenshot of a computer program

Description automatically generated**

**ScreenServiceTest:  
A screenshot of a computer

Description automatically generated**

**UserServiceTest:**

**A screenshot of a computer

Description automatically generated**