SE301 Project

1. Yue Zheng Ting
2. Owen Goh Heng Yi
3. Tan Kai Xuan

Problem Definition

Automated Warehouse with Robotic Pickers and Inventory System

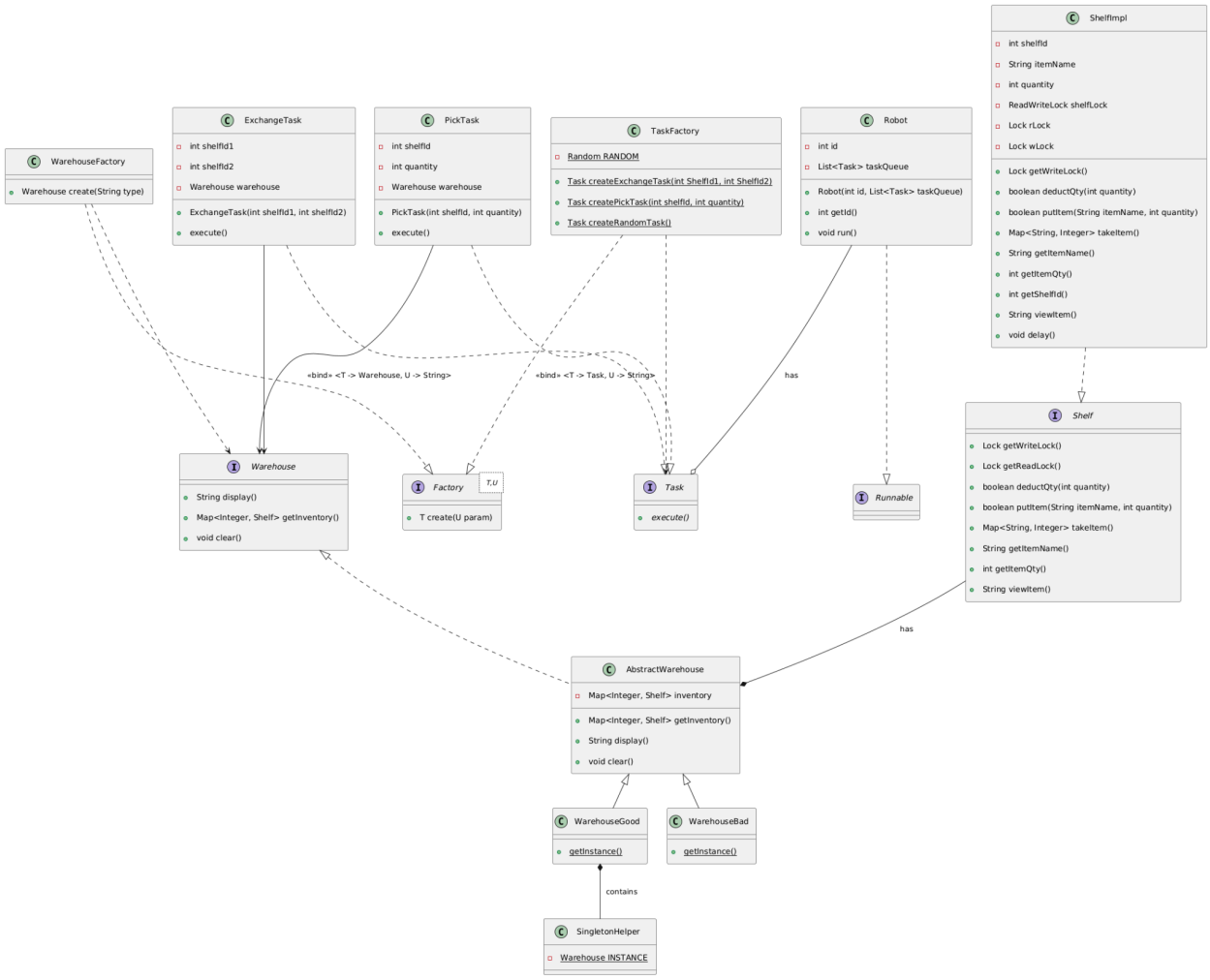
Scenario:

- Imagine an automated warehouse where multiple robotic pickers (threads) navigate the warehouse to pick items from shelves.
- These robots share several resources:
    1. **Shelves:** Robots need access to the same shelves to pick items, and only one robot can access a shelf at a time.
    2. **Inventory System**: When a robot picks an item, it updates the shelf to reflect the quantity of the item picked. This inventory is stored in a single Warehouse object.

Concurrency Challenges:

1. **Race Condition:**
    - The shelf is updated when a robot picks an item. If two robots pick the same item simultaneously, both might try to update the shelf's item at the same time, causing incorrect or inconsistent data (e.g., the quantity of that item being deducted twice).
    - This race condition can occur if multiple robots access and modify the inventory without synchronisation, leading to incorrect inventory counts.
2. **Deadlock:**
    - Robots might need to lock access to two shelves in order to perform a swap.
    - Each robot might lock access to a certain shelf and wait for the other to release the resource. This can create a circular wait situation, leading to a deadlock.
    - For instance, there can be a case where 3 robots perform swaps.
    - Robot A locks shelves 1 and 2. Robot B locks shelves 2 and 3. Robot C locks shelves 3 and 1. Without any concurrency measures, this would cause a deadlock where each needs access to the other resource, but neither can proceed.

Design and Implementation



Good coding and design principles

1. **Use of interfaces**
    - Interfaces are used as contracts between different components, abstracting specific behaviours and promoting loose couplings between each component.
    - Implementation details are hidden, enabling flexibility in how tasks are executed.
    - Using interfaces promotes modularity and testability as implementations can be swapped out seamlessly, promoting adaptability and conformance to Liskov's Substitution Principle (LSP).
2. **Abstract classes**
    - An AbstractWarehouse class was created to centralise shared functionalities without the ability for it to be instantiated on its own.
    - Concrete Warehouse classes were created extending the AbstractWarehouse class, only having to implement its own instance getter methods.
    - This approach conforms well to SOLID principles, specifically the Open-Closed Principle (OCP) and LSP through abstraction, as well as to good coding practices such as Don't Repeat Yourself (DRY).
3. **Thread-safe Singleton pattern**
    - Eager-loaded Singletons are not ideal as they consume resources even when they are not used during the application's lifetime, whereas lazy-loaded Singletons are not thread-safe without synchronisation, which has additional overhead.
    - The Bill Pugh Singleton pattern ensures that only one instance of the Singleton object gets created by using a static inner class which is loaded only when the inner class is accessed.
    - This allows us to achieve lazy-loading without the overhead of synchronisation by taking advantage of Java's Class Loader, which only loads each class once.
4. **Command pattern**
    - The Command interface (Task) was used to encapsulate method calls of the robot to command objects, decoupling the invoker from the client.
    - This design allows for easy queuing and management of robot tasks, adhering to the SRP as it ensures each class has a clear and distinct role.
    - Conformance to OCP is also present as it allows new commands to be added as new classes without breaking existing client code.
5. **Factory pattern with generic interface**
    - Factories were used to decouple the instantiation from the business logic, promoting loose coupling between client and task creation.
    - The concrete factories implement a generic factory interface that requires developers to provide both the return type and parameter type.
    - This allows clients to create complex objects – such as random tasks used for testing in the main method – since the creation logic is encapsulated in the factory class while being able to substitute superclasses for subclasses without breaking the application.

- This pattern helps promote SRP by allowing classes to focus on their tasks rather than how to construct the dependencies, OCP through implementation of the interfaces, and LSP by depending on the interface rather than the actual implementation..

6. **Constructor dependency injection**
    - This technique was used to decouple creation of object dependencies from their usage, promoting flexibility and reusability by enabling objects to depend on interfaces or abstractions rather than implementations.
    - This approach strongly adheres to the Dependency Inversion Principle (DIP) by depending on abstractions rather than concrete implementations.
    - It also conforms to SRP by allowing classes to focus on primary tasks rather than managing their dependencies, as well as OCP through allowing new implementations to be introduced to the dependencies with minimal changes to the existing code.
7. **Use of Lombok for common boilerplate code**
    - The Lombok library was used to reduce boilerplate code and improve code readability and maintainability through automatic generation of common methods such as getters and constructors, allowing code to remain fully functional while remaining concise.
    - This allows us to focus on core business logic rather than writing repetitive boilerplate code, allowing us to achieve better productivity overall.

Concurrency Control

1. **Use of Read and Write Locks**
    - Read and Write locks are used to manage concurrent access to shared resources. They allow for multiple threads to read from the resource simultaneously, while providing exclusive access for a single thread to write to the resource. This will help improve the liveness of our application while maintaining data consistency.
    - Read and Write Locks are mainly used in the Shelf class to prevent the race conditions as stated above.
2. **Lock Ordering**
    - In order to prevent deadlocks in our application due to the circular dependencies among the Shelves' locks, we utilised locking ordering technique to acquire locks in a predetermined and consistent order. By establishing strict order in which locks are acquired, threads will avoid the risk of cyclic waiting as locks are acquired in the same sequence.
    - The Lock Ordering technique is used in the ExchangeTask class where each ExchangeTask being executed requires the locks of 2 shelves.

Guide to run

1. Cloning the Project
    - To clone the project, run the following command:
    - `git clone https://github.com/kaixuantan/se301-proj.git`
    - cd se301-project
2. Building the Project
    - To build the project, run the following command in the root directory of the project:
    - `mvn clean install`
3. Running the Project
    - To run the project, execute the `Main` class. You can do this from the command line with the following command:
    - ```
      mvn exec:java -Dexec.mainClass="se301.project.Main"
      -Dexec.args="good 50 5"
      ```
4. Running Tests
    - To run the tests, run the following command in the root directory of the project:
    - `mvn clean test`
      ```