



SE2030

Software Engineering

Year 2 – Semester 1

Lab Report – 06

Web based Blood Donation System

Group 1.1 - (MLB-B1G1-03)

Submitted to
Sri Lanka Institute of Information Technology

Student Name	Student ID
Samaraweera P.A.P. B	IT24103858
Manage M.D.N.S.	IT24104013
Wijayagunawardana P.M.P.	IT24103902
Subasinghe I.N.	IT24103980
Gunarathna H.M.S. S	IT24103989
Dilanka P.H.	IT24103939

Design Patterns Used

Our Blood Bank Management System project incorporates the following three design patterns:

1. Singleton Pattern (Creational)
2. Factory Pattern (Creational)
3. Strategy Pattern (Behavioral)

Justifications

A. Singleton Pattern

The Singleton pattern was implemented to manage the application's database connection. The primary objective was to ensure that the `DatabaseConnection` class would have only one instance throughout the application's lifecycle. This is critical for resource management, as creating multiple database connection objects is inefficient and can lead to inconsistent state management and potential data integrity issues.

By providing a single, global point of access (`getInstance()`), the Singleton pattern guarantees that all components of the system utilize the same connection object. This centralizes control over the database resource, promoting stability and predictability in data handling operations.

B. Factory Pattern

We utilized the Factory pattern to abstract and centralize the object creation process for `BloodRequest` entities. The `BloodRequestFactory` class encapsulates the instantiation logic, thereby decoupling the client (`BloodBankApp`) from the concrete implementation of the `BloodRequest` class.

This approach enhances the system's maintainability. If the construction process for `BloodRequest` objects were to change in the future, for instance, by requiring additional parameters or introducing subclasses—the modifications would be confined to the factory alone. The client code, which requests objects from the factory, would remain unaffected. This separation of concerns makes the codebase cleaner and more adaptable to future changes.

C. Strategy Pattern

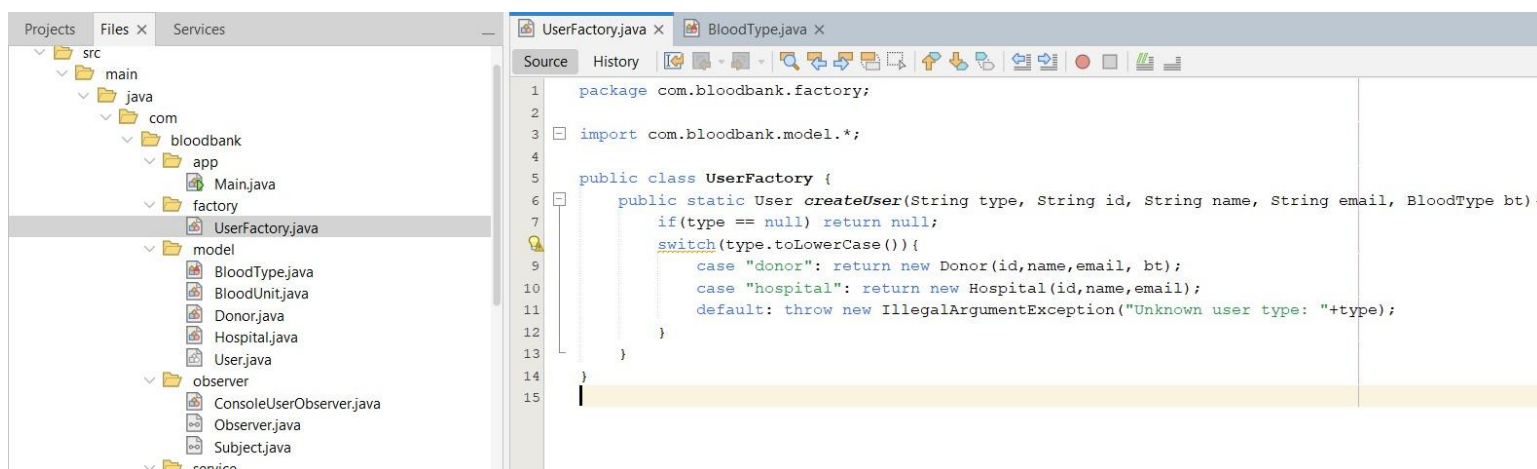
The Strategy pattern was selected to manage the different algorithms for searching for blood donors. The system required multiple search methodologies: a `StrictMatchStrategy` for finding donors with the exact blood type and a `CompatibleMatchStrategy` for finding donors based on compatibility rules.

This pattern allows us to encapsulate each algorithm into a separate class, making them

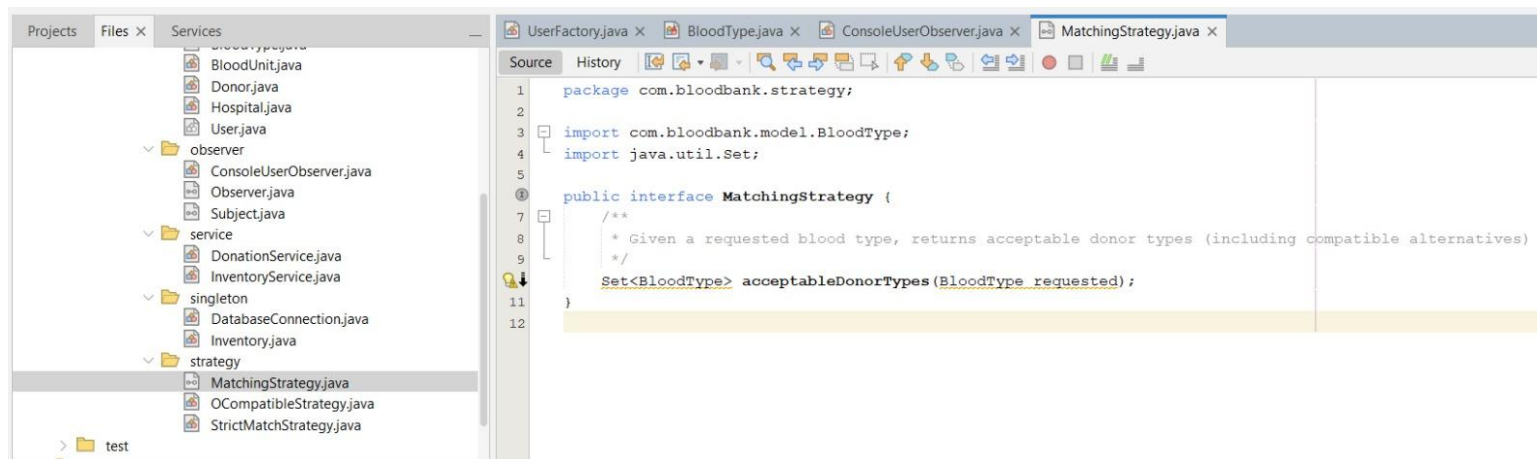
interchangeable at runtime. The BloodBank class can be configured with any SearchStrategy object, delegating the search task to it without being coupled to the specific implementation of the algorithm. This design avoids conditional logic (e.g., if-else blocks) for selecting the search method, adhering to the Open/Closed Principle. As a result, the system is more flexible, and new search strategies can be introduced with minimal modification to the existing codebase.

Screenshots

Factory pattern



Strategy pattern



Singleton pattern

The screenshot shows an IDE with the following components:

- Projects Panel:** Displays a project structure with packages: `observer` (containing `ConsoleUserObserver.java`, `Observer.java`, `Subject.java`), `service` (containing `DonationService.java`, `InventoryService.java`), `singleton` (containing `DatabaseConnection.java`, `Inventory.java`), and `strategy` (containing `MatchingStrategy.java`, `OCompatibleStrategy.java`, `StrictMatchStrategy.java`).
- Instance - Navigator:** Shows the `Inventory` class as a `Subject`. It lists methods: `addUnits(BloodType type, int qty)`, `getInstance(): Inventory`, `getQuantity(BloodType type): int`, `notifyObservers(String message)`, `registerObserver(Observer o)`, `removeObserver(Observer o)`, `requestUnits(BloodType type, int qty): boolean`, and `snapshot(): Map<BloodType, Integer>`. It also shows attributes: `instance: Inventory` and `observers: List<Observer>`.
- Source Editor:** Displays the `Inventory.java` file. The code implements the Singleton pattern and the Observer interface. It includes imports for `com.bloodbank.model.*`, `com.bloodbank.observer.Observer`, `com.bloodbank.observer.Subject`, `java.util.*`, and `java.util.concurrent.ConcurrentHashMap`. The class `Inventory` implements `Subject` and contains a private static `instance`, a private static `stock` (a `ConcurrentHashMap`), and a private static `observers` (a `synchronizedList`). The `getInstance()` method ensures only one instance exists. The `registerObserver`, `removeObserver`, and `notifyObservers` methods manage the observer list. The `addUnits` and `requestUnits` methods manage the stock, and `snapshot` returns a copy of the stock map.

This screenshot continues the view of the `Inventory.java` file from the previous image, showing lines 40 to 61. The `requestUnits` method is implemented, checking for available stock and notifying observers if a low stock alert is triggered. The `getQuantity` method returns the current stock for a given blood type. The `snapshot` method returns a new `HashMap` containing the current stock data.

Console output

The screenshot displays the Apache NetBeans IDE 27 interface. The top menu bar includes File, Edit, View, Navigate, Source, Refactor, Run, Debug, Profile, Team, Tools, Window, and Help. The toolbar contains various icons for file operations, running, and debugging. The Projects view on the left shows a project named 'BloodDonatingSystem' with a package structure including 'observer', 'service', 'singleton', and 'strategy'. The Source editor in the center shows the 'Inventory.java' file. The Console output window at the bottom shows the execution of the application, including resource copying, compilation, and runtime output.

```
Output - Run (BloodDonatingSystem) ...X
[ jar ]-----
--- resources:3.3.1:resources (default-resources) @ BloodDonatingSystem ---
Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
skip non existing resourceDirectory C:\Users\MSI\Documents\NetBeansProjects\BloodDonatingSystem\src\main\resources

--- compiler:3.13.0:compile (default-compile) @ BloodDonatingSystem ---
Recompiling the module because of changed source code.
File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!
Compiling 17 source files with javac [debug target 17] to target\classes
location of system modules is not set in conjunction with -source 17
not setting the location of system modules may lead to class files that cannot run on JDK 17
--release 17 is recommended instead of -source 17 -target 17 because it sets the location of system modules automatically

--- exec:3.1.0:exec (default-cli) @ BloodDonatingSystem ---
? Database connected successfully!
=== Blood Bank System ===

1. Add Donation
2. Request Blood
3. Show Inventory
4. Exit
Enter choice: 2
Enter Blood Type (e.g. A+, O-, etc): A+
Enter Quantity: 1
? Blood type A+ not found in inventory.

1. Add Donation
2. Request Blood
3. Show Inventory
4. Exit
Enter choice: 2
Enter Blood Type (e.g. A+, O-, etc): O-
Enter Quantity: 1
? Request processed: 1 units of O- issued.

1. Add Donation
2. Request Blood
3. Show Inventory
4. Exit
Enter choice:
```