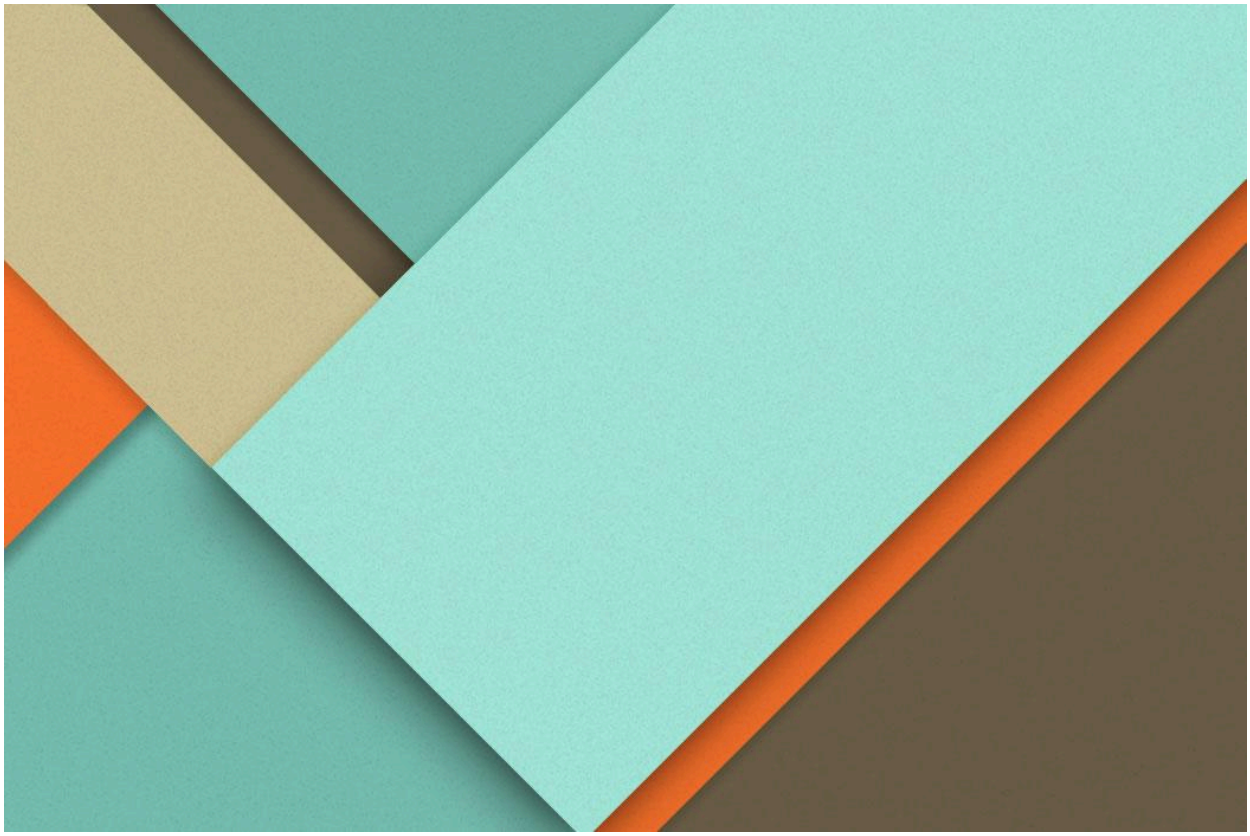

Volunteer Matching System

SWEN-383 – Final Report (Project Part 2)

Dijar Hashani, Fiona Kapllani, Benita Grainca, Olta Gjevukaj

1. Introduction

The Volunteer Matching System is designed to connect volunteers with organizations seeking help for various community-related opportunities. The system was initially developed in Part 1 with basic domain models, simple controller logic, and in-memory storage.

For **Project Part 2**, the goal was to refine and extend the system by applying **software design principles and patterns** covered in SWEN-383. The focus is on improving maintainability, extensibility, and reducing design risks through the use of well-established design patterns.

This report documents:

- The overall architecture of the application
- The implementation of **three key design patterns**:
  Strategy, **Observer**, and **State**
- How these patterns solve existing problems and mitigate future risks
- Additional improvements made to the design
- Refined UML diagrams reflecting the final implementation

2. System Architecture Overview

The final system follows a **layered MVC-inspired architecture**, ensuring separation of concerns and loose coupling between components.

2.1 Model Layer

The domain model includes:

- `User`, `Volunteer`, `Admin`, `Organization`
- `Opportunity`
- `Application` with integrated **State pattern**
- `ApplicationState` hierarchy (Pending, Approved, Rejected)

These classes represent the core business entities and behaviors.

2.2 Controllers

Controllers act as the system's application layer:

- `VolunteerController`
- `AdminController`
- `OrganizationController`

They delegate logic to services and repository instances, resulting in clean and testable components.

2.3 Service Layer

Key services include:

- `MatchingService` (**Strategy Pattern**) for flexible volunteer–opportunity matching

- `NotificationService` (**Observer Pattern**) for decoupled event notifications

Each service focuses on a single responsibility aligned with SRP (Single Responsibility Principle).

2.4 Repository Layer

`DataRepository` implements a simple, in-memory storage mechanism and follows the **Singleton Pattern** implicitly via shared instance usage.
It stores all model objects grouped by type and provides methods for saving, updating, and retrieving objects.

2.5 Composition Root

`main.js` wires dependencies together:

- Injects the repository into controllers
- Sets matching strategies
- Registers notification observers

This aligns with the **Dependency Inversion Principle (DIP)**.

3. Applied Design Patterns

Project Part 2 required demonstrating at least **three design patterns**. The system includes:

- **Strategy Pattern** (MatchingService)
- **Observer Pattern** (NotificationService)
- **State Pattern** (Application lifecycle)

Each pattern is implemented in a realistic and meaningful way, solving real design issues from Part 1.

3.1 Strategy Pattern – Matching Service

Problem Addressed

In Part 1, volunteer–opportunity matching was embedded directly inside the controller.
This created issues such as:

- Tight coupling between matching logic and UI logic
- Difficulty adding new matching rules
- Future scalability issues as the system grows

Pattern Implementation

To address this, a classic **Strategy Pattern** was implemented:

- `IMatchingStrategy` (interface)
- `SkillBasedMatchingStrategy` (default)
- `LocationAndSkillMatchingStrategy` (advanced)
- `MatchingService` (Context)

Controllers do not know how matching is performed.
They only call:

```
matchingService.findMatches(volunteer);
```

Benefits

- Easy to add new matching algorithms
- Controllers remain simple and independent

- Code follows the **Open/Closed Principle**
- Improves testability because strategies are isolated

Risk Reduction

If matching rules change frequently or become more complex, Strategy avoids repeated modifications to controller logic, preventing code smells like “God Classes” or “Spaghetti Code”.

3.2 Observer Pattern – Notification System

Problem Addressed

In earlier design versions, sending notifications directly from controllers risked tight coupling:

- Controllers must know *how* notifications are sent
- Adding new notification channels (SMS, push notifications) required modifying multiple files
- Violates SRP and DIP principles

Pattern Implementation

`NotificationService` acts as an **Observer Subject**, while:

- `EmailNotificationObserver`

- `InAppNotificationObserver`

act as listeners:

```
notifier.addObserver(new EmailNotificationObserver());
notifier.addObserver(new InAppNotificationObserver());
```

Whenever events occur (e.g., application submitted or approved), observers are notified automatically.

Benefits

- Zero coupling between controllers and notification methods
- Highly extendable (add SMSObserver, PushObserver, etc.)
- Clean separation of concerns
- Matches the Observer pattern taught in SWEN-383

Risk Reduction

Prevents a future “Golden Hammer” anti-pattern where all notification logic is forced into a single class.
Adding new channels will never break existing functionality.

3.3 State Pattern – Application Lifecycle

Problem Addressed

In Part 1, the application lifecycle used simple string fields:

```
"Pending", "Approved", "Rejected"
```

This approach creates risks:

- Logic scattered across the system
- Easy to introduce invalid transitions
- Approval/rejection logic duplicated in multiple places

Pattern Implementation

A full **State Pattern** was implemented:

- `ApplicationState` (abstract)
- `PendingState`
- `ApprovedState`
- `RejectedState`

The `Application` class delegates behavior to its active state object:

```
application.approve();  // calls state.approve()
```

Benefits

- Behavior is encapsulated in state classes
- Easy to add new states later
- Prevents invalid transitions
- Cleaner and safer approval workflow

Risk Reduction

This pattern prevents future complexity issues, especially when scaling the application to include more states (e.g., Withdrawn, Under Review).

4. Additional Improvements

4.1 Singleton Repository

`DataRepository` ensures consistent shared memory storage without explicit singleton enforcement.
All controllers reuse the same instance.

4.2 Dependency Injection

Dependencies are passed through constructors, improving modularity and testability.

4.3 Improved Separation of Concerns

Controllers now contain minimal logic and delegate the heavy work to services and domain objects.

4.4 Refined UML and Diagrams

Part 2 includes full updated diagrams showing:

- Class relationships
- Interaction between controllers, services, observers, and states
- Four clean sequence diagrams demonstrating real system behavior

These diagrams strengthen documentation quality.

5. Conclusion

This project demonstrates the practical application of the design principles and patterns covered in SWEN-383:

- **Strategy Pattern** improves flexibility and removes coupling from matching logic
- **Observer Pattern** decouples notifications and scales easily
- **State Pattern** streamlines the lifecycle of applications

The redesigned architecture is cleaner, easier to extend, and resistant to many common anti-patterns such as “God Class”, “Spaghetti Code”, and “Golden Hammer”.

This final submission reflects an evolution from a simple procedural system into a well-structured object-oriented application aligned with real-world software engineering practices.