POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION

Institute of Computing Science

Bachelor's thesis

VOLUMAN-BACKEND: A SERVER FOR THE VOLUMAN VOLUNTEER MANAGEMENT SYSTEM

Jędrzej Mikołajczyk, 151550

Jacek Młynarczyk, 151747

Klaudiusz Szklarkowski, 151801

Łukasz Walicki, 151061

Supervisor

prof. dr hab. inż. Jerzy Nawrocki

POZNAŃ 2025

Abstract

Volunteers play an essential role in addressing global challenges such as pandemics, natural disasters, and migration crises. However, managing volunteer activities efficiently becomes increasingly challenging as the scale of operations grows. Existing IT tools, such as spreadsheets and basic communication platforms, fail to meet the demands of dynamic, large-scale volunteer coordination.

This project addresses the problem by redesigning the server-side architecture of VoluMan, a volunteer management system initially developed by another student team, which lacked flexibility, making it difficult to maintain and extend.

The scope of work included defining the system's interface and ensuring functional completeness by verifying the coverage of defined use cases.

The results demonstrate that the new server implementation covers 90% of the identified use cases, ensuring that it meets the functional requirements of volunteer status management, action scheduling, and coordination. Additionally, the server is fully compatible with future integration of frontend and mobile application components.

In conclusion, the redesigned VoluMan system provides a foundation for volunteer management, achieving level 4 on the 9-level scale of the Technology Readiness Levels, which means it is verified in laboratory environment. It can be further developed to include advanced features.

Contents

Chapter 1

Introduction

1.1 Problem Statement

Natural disasters, global pandemics, and large-scale migrations caused by conflicts pose significant challenges that require rapid and well-coordinated responses. The outbreak of COVID-19 and the migration of millions of Ukrainians following the Russian invasion are two striking examples of situations in which humanitarian efforts must be mobilized swiftly. In such crises, volunteers play an indispensable role by providing essential services, offering logistical support, and ensuring that aid reaches those in need as efficiently as possible.

According to global reports, volunteerism is a crucial pillar of crisis response, bridging the gap between institutional support and the immediate needs of affected communities [7]. Volunteers contribute in various capacities, from assisting in emergency shelters to distributing supplies and facilitating integration for displaced individuals. However, despite their importance, the effective coordination of volunteer efforts remains a significant challenge, particularly in large-scale operations where the number of volunteers grows rapidly.

As the scale of crises increases, so does the complexity of managing volunteer work. Ensuring that volunteers are assigned to appropriate tasks, preventing overstaffing or shortages, and maintaining clear communication channels between organizers and participants are all essential for maximizing the impact of humanitarian initiatives. In the European Union alone, volunteer activities have been systematically analyzed, highlighting both their societal benefits and the difficulties organizations face in structuring them effectively [1].

A volunteer is not an employee, so the organization must adjust its schedules to the volunteers' capabilities. In addition, assigning too many volunteers to a given campaign can be demotivating, and with too few volunteers, they will be overworked and the quality of services will suffer. Therefore, it is important to support volunteer leaders in managing campaigns. Unfortunately, there is a lack of appropriate IT tools that could support these processes. In many organizations, the only tools that leaders have are telephones, text messages, and spreadsheets. With a small number of volunteers, this may be enough, but when the number of volunteers is large, such tools become ineffective.

The scale of volunteer involvement in recent global crises further highlights the need for advanced management systems. For instance, during the COVID-19 pandemic, thousands of volunteers were mobilized to deliver essential services, such as food distribution and vaccination support. Similarly, the Russian invasion of Ukraine in 2022 resulted in over 9 million refugees entering Poland[6] and around 1 million still lives in Poland[5], requiring large-scale coordination of volunteers to provide aid in reception points and shelters. In such cases, the lack of a centralized management system led to inefficiencies, including misallocated resources, uneven workload distribution, and unnecessary administrative burdens on volunteer leaders.

Historically, volunteer efforts have played a crucial role in disaster response and crisis management. However, the tools available to support these efforts have not evolved significantly. While organizations have transitioned from paper-based methods to digital tools like spreadsheets, these solutions remain inadequate for managing the scale and complexity of modern volunteer campaigns. For example, during large-scale emergencies, manual coordination often results in duplication of effort in some areas and insufficient support in others.

A notable example is the reception points for Ukrainian refugees established in Poznań. These required volunteers to work in shifts, providing food, medical assistance, and logistical support. Without a centralized system, leaders relied on manual scheduling and tracking, which was both time-consuming and error-prone. This resulted in uneven task allocation, overburdened volunteers, and delays in responding to changing circumstances.

These challenges underline the critical need for a reliable, scalable, and efficient volunteer management system. Such a system should support dynamic scheduling, adaptability to real-time changes, and effective coordination, allowing volunteer leaders to focus on delivering aid rather than administrative tasks.

1.2 Market Analysis

The organization and management of volunteer activities require robust and flexible IT systems. Although there are various solutions available on the market, most fail to fully address the complexities of volunteer management, particularly in dynamic and large-scale operations. This section reviews some existing tools and identifies their limitations to highlight the motivation for the VoluMan system.

Existing Solutions

Several software solutions for volunteer management are currently available. Examples include:

- **Bloomerang Volunteer Management**: This tool focuses on tracking volunteer hours and automating communication. While comprehensive in terms of donor and volunteer engagement, it offers limited scalability for complex scheduling or real-time adjustments in volunteer assignments.
- **Volgistics**: A popular solution providing tools for scheduling and volunteer tracking. However, its rigid structure makes it less suitable for scenarios requiring flexible or dynamic allocation of volunteers.
- **Galaxy Digital's Get Connected**: A platform offering features like volunteer opportunity matching and reporting. Despite its versatility, it is primarily targeted at specific markets and often lacks customization options for unique organizational needs.

Limitations of Current Tools

While these tools provide valuable functionalities, they share several limitations that restrict their applicability in demanding scenarios:

- **Limited Scheduling Flexibility**: Most systems rely on static scheduling, which is unsuitable for large-scale or rapidly changing volunteer campaigns.
- **Language and Localization Constraints**: Many solutions lack multi-language support, making them less accessible to diverse volunteer groups or international organizations.
- **Cost-Prohibitive Models**: Subscription-based pricing can be expensive for non-profit organizations with limited budgets, reducing the adoption of these tools.
- **Inefficiency in Handling Large Volunteer Numbers**: The architecture of many systems struggles to handle the complexities of coordinating hundreds or thousands of volunteers effectively.

Opportunities for Improvement

The VoluMan system aims to address these shortcomings by focusing on:

- Providing dynamic scheduling capabilities to adapt to real-time changes in volunteer availability and campaign demands.
- Ensuring scalability and flexibility to accommodate both small and large volunteer teams.
- Supporting multilingual environments to improve accessibility and inclusivity.
- Leveraging open-source technologies to make the system more affordable for non-profit organizations.

This analysis of existing tools and their limitations highlights the need for an advanced, reliable, and scalable volunteer management system. The VoluMan project seeks to fill this gap by providing an efficient back-end architecture as a foundation for further development.

1.3 Scope and Objective of the Project

To address these challenges, a student team at Poznań University of Technology initiated a volunteer management system aimed at improving the planning of duty hours, handling schedule changes, and streamlining essential processes such as candidate recruitment, promotion, demotion, and removal. Despite their efforts, the initial implementation did not have the required reliability and scalability, making the system unsuitable for deployment.

This work focuses on redesigning the server-side architecture of the VoluMan system. Our contribution includes a new back-end implementation designed to meet the requirements of scalability, reliability, and usability, as outlined by our project supervisor.

The current report documents the new approach to specifying and implementing the server. This work establishes a foundation for further development, addressing broader aspects of volunteer and campaign management in subsequent phases.

License and Open Access

Chapter 2

System Logic and Use Cases

This section provides an overview of the whole system. It will also be used to check the functional completeness of the proposed server model. The presented notation and use cases are based on the bachelor thesis by Agnieszka Gruszczyńska, Łukasz Jankowski, Maksim Likhaivanenka and Helena Masłowska[3].

2.1 Notation

The following notation will be used to describe the use cases:

either α **or** β : depending on the user's choice, either α or β action will be executed.

oneOf *Set* : an element of *Set* chosen by the user. All the elements of *Set* are presented to the user (in a way depending on the user interface), and then the user makes the decision.

thisActor : this operator identifies the user executing a given use case. For instance, expression **thisActor** $\in a.Leaders$ is true if and only if the user belongs to the leaders of action a.

input *Atrib* is an operator responsible for reading attribute values *Atrib* or their modification, and it involves displaying their current values beforehand. For example, **input** $a.Description$ causes the current description of action a to be displayed and allows for its modification.

present *Atrib* shows the indicated attributes to the user. The attributes can pertain to a single object or an entire set of objects. If S is a set, then **present** $S.Atr$ describes the system's response involving the display of the attributes Atr of all elements in the set S.

... *text* indicates an action described in natural language (in some cases a formal description would be overly complex and therefore less readable).

We will also use the following symbols:

$\mathcal{A}$ = a set of actions run by a given organization.

$\mathcal{C}$ = a set of candidates, i.e., people who applied for volunteering.

$\mathcal{L}$ = a set of leaders of all actions run by the organization.

$\mathcal{V}$ = a set of volunteers.

In this work, the specification of functional requirements is procedural in nature: the functions performed by the system will affect the state of the aforementioned sets. This means that at each step, a new version of the set will be created based on the old version. To differentiate between these versions, it has been established that the new version of a set will be named with a prime symbol. For example, $\mathcal{C}'$ denotes the new version of the set $\mathcal{C}$.

The identified use cases have been grouped into the following functional modules:

- Management of the volunteer status (MVS)
- Management of actions (MA)
- Management of schedules (MS)

2.2 Management of Actions

The main attributes of an action include:

- *Heading* – a short title describing the action.
- *Description* – a detailed description of the action.
- *Leaders* – a set of volunteers responsible for managing the action.

An action can be created, modified, closed, or viewed by different actors in the system. Table 2.1 provides a summary of the key use cases concerning the management of actions.

MA-1: Actor = Admin
Creating a new action:
a = **newAction**()
$\mathcal{A}' = \mathcal{A} \cup \{a\}$
a.Heading = “”
a.Description = “”
a.Leaders’ = a.Leaders $\cup$ **oneOf** $\mathcal{V}$

MA-2: Actor = Admin
Closing an action:
a = **oneOf** $\mathcal{A}$
$\mathcal{A}' = \mathcal{A} \setminus \{a\}$

MA-3: Actor = Leader
Modifying an action description:
present $\mathcal{A}$.Heading
a = **oneOf** $\{x \in \mathcal{A} : \textbf{thisActor} \in x.Leaders\}$
input a.Description

MA-4: Actor = Guest
Viewing action descriptions:
present $\mathcal{A}$.Description

Table 2.1: The use cases concerning the management of actions (MA).

2.3 Management of the Volunteer Status

The main attributes of a volunteer include:

- *Personal data* that include, among others, given name, family name, and contact data.
- *Status* which can assume one of the following values: *Candidate*, *Volunteer*, or *Leader*.
- *Lim* indicating the maximum number of slots the volunteer is willing to spend for the organization in a given week.
- *Avail* representing availability of the candidate in a given week (it shows for each day the slots the volunteer could work for the organization).
- *Timetable* which shows for each day the slots and the action the volunteer is assigned to.

An external person (who is not volunteering for the organization yet) has the **Guest** status. After applying for volunteering and being accepted by a person in the Recruiter role he or she becomes a **Volunteer**. **Admin** can promote a volunteer to the **Leader** position. A **Leader** is also a **Volunteer**.

In Table 2.2 the reader will find a brief description of the use cases concerning the management of volunteer status. According to the naming convention proposed by Ivar Jacobson (and adopted in UML), the user is called *actor*. In the table, we follow the rule.

MVS-1:	Actor = Guest **Application for volunteering**: c= newUser() **input** c.Personal_data *... Guest accepts the organization's bylaws* $\mathcal{C}'= \mathcal{C} \cup \{c\}$
MVS-2:	Actor = Recruiter **Accepting a candidate**: c= **oneOf** $\mathcal{C}$ **either** $\mathcal{V}' = \mathcal{V} \cup \{c\}$ # acceptance **or** $\mathcal{V}' = \mathcal{V}$ # rejection $\mathcal{C}' = \mathcal{C} \setminus \{c\}$
MVS-3:	Actor = Admin **Promoting to leader**: v= **oneOf** $\mathcal{V}$ $\mathcal{L}' = \mathcal{L} \cup \{v\}$
MVS-4:	Actor = Admin **Revoking a leader**: l= **oneOf** $\mathcal{L}$ $\mathcal{L}' = \mathcal{L} \setminus \{l\}$
MVS-5:	Actor = Admin **Dismissal of a volunteer**: v= **oneOf** $\mathcal{V}$ $\mathcal{L}' = \mathcal{L} \setminus \{v\}$ $\mathcal{V}' = \mathcal{V} \setminus \{v\}$

Table 2.2: The use cases concerning the management of volunteer status (MVS).

2.4 Management of Schedules

The Schedule management module ensures proper coordination between volunteers, actions, and schedules. It supports operations such as updating participation preferences, declaring availability, managing volunteer needs, and reviewing or modifying schedules. The main attributes related to schedules include:

- *Schedule* – the specific time slots and actions assigned to a volunteer or leader.
- *Availability* – the time slots during which a volunteer is available.
- *Participation Preferences* – the volunteer's interest or priority regarding participation in specific actions.
- *Volunteer Needs* – the number of volunteers required for a specific action and time slot.

Table 2.3 provides a summary of the key use cases concerning the management of schedules.

MS-1: Actor = Volunteer
Updating action participation:
present $\mathcal{A}$.Description
a = **oneOf** $\mathcal{A}$
w = **thisActor**
input $P(w, a)$ # Participation preferences for a
a.Volunteers' = a.Volunteers $\cup\{w\}$ if $P(w, a) \in \{T, R\}$
a.Decided' = a.Decided $\cup\{w\}$ if $P(w, a) = T$

MS-2: Actor = Volunteer
Declaring availability:
w = **thisActor**
input w.Lim
input w.Avail

MS-3: Actor = Leader
Declaring need for volunteers:
a = **oneOf** $\{x \in \mathcal{A} : \textbf{thisActor} \in x.Leaders\}$
input a.Beg, a.End
input a.Need

MS-4: Actor = Volunteer
Modifying schedule:
w = **thisActor**
input w.Schedule

MS-5: Actor = Leader
Viewing schedule:
a = **oneOf** $\{x \in \mathcal{A} : \textbf{thisActor} \in x.Leaders\}$
present a.Schedule

Table 2.3: The use cases concerning the management of schedules (MS).

Chapter 3

The server interface

3.1 Overview and General Remarks

The description of the server interface has been split into three parts:

- Management of action descriptions,
- Management of volunteers data, and
- Scheduling.

They are described in the following sections. Each part of the interface is further split into descriptions of the data structures and operations that manipulate them.

It is assumed the reader knows at least the basics of Java.

3.2 Action Management

3.2.1 Description of an Action

A volunteer organization can run actions such as a gift stockroom or a refugee reception point. Each action has a description, which is presented to prospective volunteers to help them decide if they want to join. There are two kinds of descriptions: simple and compound. Their examples are given in Fig. 3.1. A compound description is more detailed as it lists roles along with their duty descriptions. For instance, the Refugee Reception Point (see Fig. 3.1) needs volunteers for four positions, including the Bar and the Information desk. In Listing 3.1, a role is represented by the `Role` class.

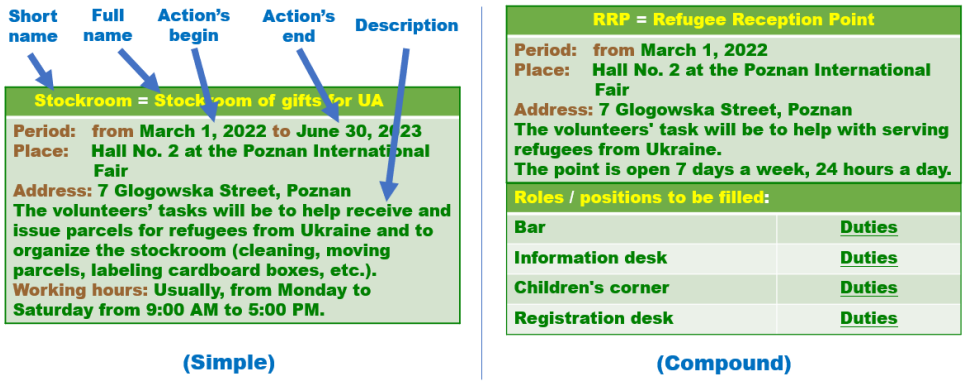

FIGURE 3.1: Two examples of action descriptions: simple and compound.

Each action, simple or compound, is described by an object of the `Action` class. It can have its begin and end date that are stored as the `begin` and `end` attributes — see Listing 3.1. They are optional, and `noDate` denotes no date has been specified.

The main part of an `Action` class is a set of action descriptions given in each of the prospective volunteers' languages. The `descr` attribute is used to store descriptions of the action in various languages. In Listing 3.1, the set of languages is implemented as an enumeration type, `Lang`. There are only three languages there: English, Polish, and Ukrainian. Their 2-letter symbols (EN, PL, UK) comply with the ISO 639 standard [...ISO639...]. Action descriptions, `descr`, is a mapping from the set of languages, `Lang`, to the set of descriptions, each given by an object of class `Version`:

```
import java.util.HashMap;
       HashMap<Lang, Version> descr;
```

The diagram 3.2 illustrates the relationships between an action, its descriptions in various languages, and roles assigned to the action. It visually complements the data structure described above.

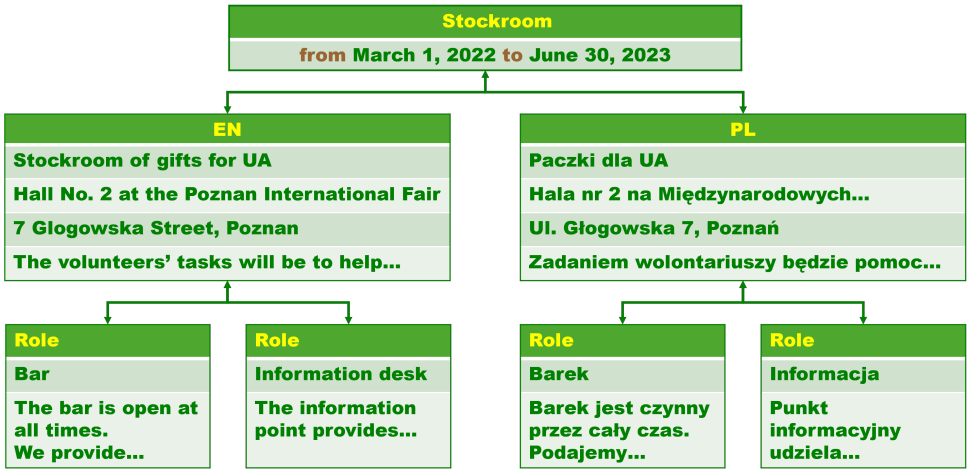

FIGURE 3.2: Relationships between an action, language-specific descriptions, and roles.

An object of the `Version` class can contain data that can be *valid* or *invalid*. Just after creating an object of the `Action` class, all `Version` objects are *invalid* as their attributes have not been set yet. When a `Version` object is ready to use, it is said *valid*. There are two methods of a `Version` class object for managing the validity status:

```
boolean isValid();           // returns the validity status (true == valid)
void    setValid(boolean v); // sets the validity status
```

The information provided by a valid object of the `Version` class is illustrated in Fig. 3.1. If an action description contains role descriptions (i.e., it is compound), the attribute `roles` of `Version` provides a list of descriptions of all the roles, each described by an object of class `Role`.

```
import java.util.ArrayList;
       ArrayList<Role> roles; // List of roles
```

What a user interface needs when presenting an action description to the user is a complete description of an action in a given language, i.e., all the attributes of the `Version` class, along with information about the beginning and end dates of the action. This information is provided by an object of the `Description` class.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashMap;

class Role{
      String name;    // e.g. "Information desk"
      String duties;  // e.g. "Providing info in Ukrainian or Russian."
      }
class Version{
      // Description of the action in a given language:
      boolean valid;
      String  fullName;       // Full name of the action
      String  shortName;      // and its abbreviation or acronym
      String  place;          // Name of the place
      String  address;        // and its address
      String  description;    // Description of the action
      String  hours;          // Working hours
      ArrayList<Role> roles;  // List of roles
      boolean isValid(){
              return valid; }
      void    setValid(boolean v){
              valid= v;  }
      public  Version() {
              valid= false;
              roles= new ArrayList<Role>(); }
      }
enum  Lang {
      EN, // English
      PL, // Polish
      UA  // Ukrainian
      // When extending, use the symbols from
      // https://en.wikipedia.org/wiki/List_of_ISO_639_language_codes
      }
class Action{
      LocalDate begin;
      LocalDate end;
      // noDate = "1970-01-01"
      HashMap<Lang, Version> descr;

      // Constructor initializes default values :
      public Action() {
             descr= new HashMap<Lang, Version>();
             for (Lang l: Lang.values()){
                 descr.put(l, new Version()); }
             begin= noDate;
             end=   noDate; }
      }
class Description extends Version{
      // A complete action description in a given language.
      LocalDate begin;
      LocalDate end;
      }
```

LISTING 3.1: Action descriptions. Each action is described by an object of class `Action`.

3.2.2 Operations

The operations associated with action descriptions are defined by the `Actions` interface presented in Listing 3.2. They can be divided into two sets:

- **Existential**: Using them, one can create, delete, or update the action descriptions.
- **Personal**: They allow to personalize the set of action descriptions, i.e., to track which actions a volunteer has joined and which ones they are not interested in.

Existential operations

To explain the existential operations, the following terms will be used:

- **Action identifier**: It identifies a given action, i.e., there is a one-to-one mapping between an identifier and the corresponding action description. The type associated with action identifiers will be denoted as `ID` and implemented as `int`.
- **Empty date**: It marks that no date has been specified (it's insignificant in a given context). It will be denoted as `noDate` and implemented as `1970-01-01`.
- **Empty action description**: It is an object of the `Action` class with an empty set of descriptions `descr`, and the `begin` and `end` dates set to **noDate**.

The `Actions` module can be perceived as a pool of action descriptions. The two most important operations are `create` and `remove`. The former creates an empty action description (i.e., invalid), adds it to the pool, and returns its identifier. The latter removes an action description of a given identifier from the pool. The following operations allow us to set and get the values of `begin` and `end` dates of an action:

```
// a = action, b = begin date, e = end date
Errors    setBeg(ID a, LocalDate b);
Errors    setEnd(ID a, LocalDate e);
LocalDate getBeg(ID a);
LocalDate getEnd(ID a);
```

The next set of operations is intended to manage the content of action descriptions.
Here they are:

```
// a = action, l = language, v = version of the description in language l
Errors      setVersion( ID a, Lang l, Version v );
Errors      remVersion( ID a, Lang l );
Description getDescript(ID a, Lang l );
```

The `setVersion` operation 'assigns' description `v` in language `l` to action represented by an identifier `a` (if `a` has had a description in language `l`, that description is overwritten). After successful completion of this operation, the description for language `l` becomes valid.

After executing `remVersion`, the description of action `a` in language `l` becomes invalid (it's equivalent to removing that description). The `getDescription` method returns a complete description of action `a` in language `l`, i.e., including the beginning and end dates. Using the `isValid` method of the `Description` class (which inherits it from the `Version` class), one can check if the obtained description is valid or not.

The last set of existential operations is given below:

```
ArrayList<ID>          getAllIds();
ArrayList<Description> getAllDesc(Lang l);
```

The `getAllIds` method returns a list of identifiers of all action descriptions in the pool of actions (i.e. the number of `Action` objects). The second method, `getAllDesc`, returns a list of all the descriptions in the pool. It can be perceived as redundant, as that list can also be obtained using the `getAllIds` and `getDesc` operations. It was introduced to the interface for the sake of efficiency: Instead of several calls to the `getDesc` method, one can obtain all the descriptions with one call to the `getAllDesc` operation (as this is a client-server interface, with each call of the server method, there is some time overhead associated with it).

Personal operations

As an organization runs several actions, a volunteer can subscribe to a subset of them. To be more precise, from a volunteer's perspective, the set of all action descriptions can be split into the following subsets:

- **Strongly Mine (S)**: They are the actions a volunteer definitely wants to participate in.
- **Weakly Mine (W)**: Here are those actions that a volunteer might join if it's necessary but would prefer not.
- **Rejected (R)**: This set contains actions a volunteer doesn't want to participate in (the reason could be they don't feel to possess the required skills).
- **Undecided (U)**: In this set, there are actions a volunteer has not decided yet if they want to join.

The partition of actions into those subsets is personal: For one volunteer, an action can belong to **Strongly Mine**, while for another, that action could be rejected. The following method allow us to get each of the sets mentioned above:

```
List<Description> getPref(String pref , Long volId);
```

Knowing those sets for each of the volunteers allows us to take volunteer's preferences into account at the time of preparation of timetable for each of the actions which encompass assignment of volunteers to actions.

To move, on behalf of a volunteer, an action from one set to another one can use the following operation:

```
Errors setPref(Long aId, String pref, Long volId);
```

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashMap;

class ID {int id}
enum  Lang {EN, // English
            PL, // Polish
            UA  // Ukrainian
            // when extending, use the symbols from
            // https://en.wikipedia.org/wiki/List_of_ISO_639_language_codes
           }
interface Actions {
      final     noDate= "1970-01-01";
      // a = action, b = begin date, e = end date
      ID        create();
      Errors    remove(ID a);
      Errors    setBeg(ID a, LocalDate b);
      Errors    setEnd(ID a, LocalDate e);
      LocalDate getBeg(ID a);
      LocalDate getEnd(ID a);

      // l = language, v = version of the description in language l
      Errors      setDesc(ID a, Lang l, Version v);
      Errors      remDesc(ID a, Lang l);
      Description getDesc(ID a, Lang l);

      ArrayList<ID>          getAllIds();
      ArrayList<Description> getAllDesc(Lang l);

      Errors setPref(Long aId, String pref, Long volId);
      List<Description> getPref(String pref , Long volId);
      Errors isError();
      }
```

Listing 3.2: Operations of the Actions module.

Listing 3.2 presents all the operations of the `Actions` module. Together with Listing 3.1, they describe the whole interface of the module responsible for the management of action descriptions.

3.3 Volunteer Management

3.3.1 Volunteer Data

The volunteer management system is responsible for handling all aspects of volunteer information, including their personal data, assigned roles, preferences, availabilities. Each volunteer is represented by an object of the `Volunteer` class.

The `PersonalData` class provides comprehensive information about the volunteer, including their `firstname`, `lastname`, `email`, `phone`, and address details such as `street`, `city`, and `postalNumber`. This ensures that each volunteer's profile allows for proper identification and communication.

Each volunteer is assigned a specific rank, defined by the `Position` enumeration, which governs their access level and responsibilities in the organization:

```
// Default positions available in system:
CANDIDATE,
VOLUNTEER,
LEADER,
RECRUITER,
ADMIN
```

The system also tracks additional details, such as `limitOfWeeklyHours` attribute, which defines the maximum number of hours a volunteer is willing to work per week, while `actualWeeklyHours` tracks the hours they have already worked.

The `preferences` attribute categorizes actions into four sets described in 3.2.2: `S, W, U, R`, illustrated in Fig. 3.3. By default, all actions are placed in the `U` (Undecided) category when the `Volunteer` object is created.

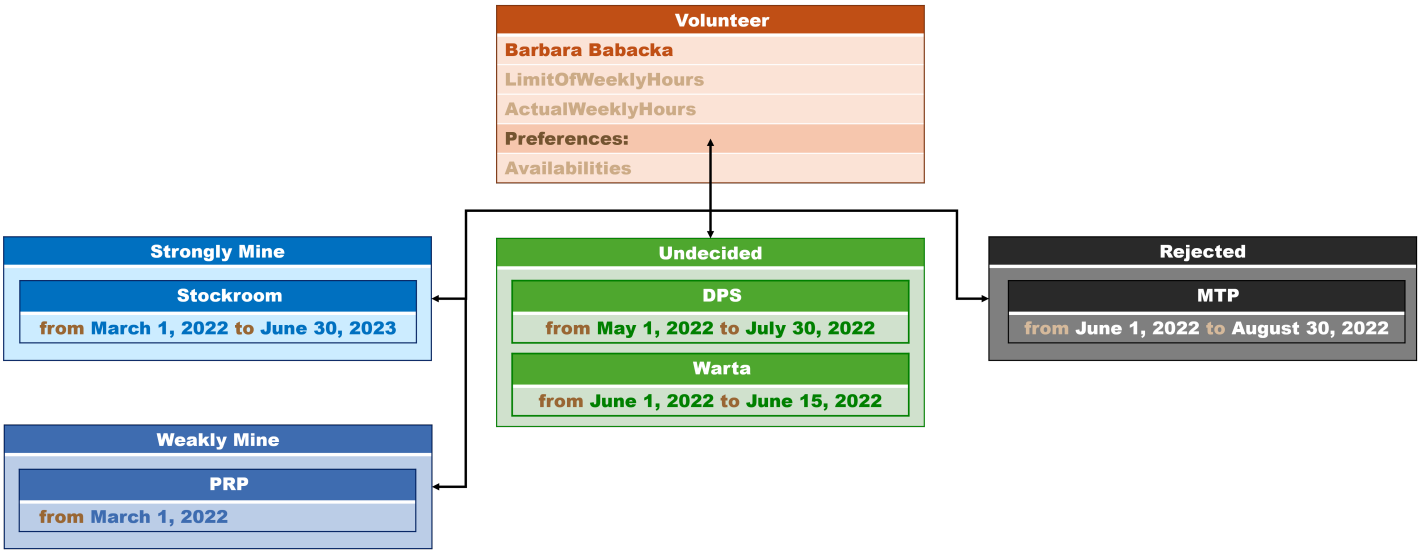

FIGURE 3.3: Volunteer preferences categorized into Strongly Mine, Weakly Mine, Undecided, and Rejected.

The `Availability` class is responsible for managing a volunteer's availability for scheduling and task assignment. Each volunteer maintains a list of their `availabilities`, which specify the days they are available to work. Availability is further divided into `AvailabilityIntervals`, which define the exact periods during a day when a volunteer is free to participate in an action.

Each `AvailabilityInterval` consists of a start time and an end time, defining the exact duration during which a volunteer is available. These intervals are stored within the corresponding Availability object, allowing the system to dynamically adjust scheduling assignments based on specific time frames.

Availability is declared in advance, allowing the system to generate schedules on a weekly basis. Volunteers specify their working hours for the upcoming period, and after the schedule is created, they have the opportunity to review and adjust their availability before confirming their participation.

Figure 3.4 illustrates an example of a volunteer's availability, showing declared availability intervals across multiple days.

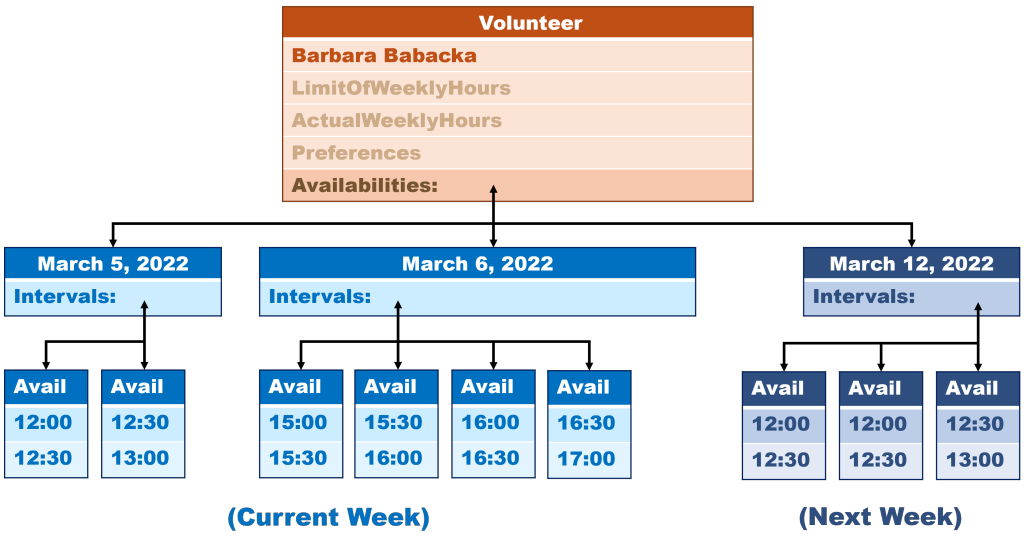

FIGURE 3.4: Diagram illustrating the connection between a Volunteer and their Availabilities.

The overall structure of the `Volunteer` class, including its attributes and relationships, is provided in Listing 3.3

```
import java.util.List;
import java.util.Set;

public enum Position {
    // Default ranks available in system:
    CANDIDATE,
    VOLUNTEER,
    LEADER,
    RECRUITER,
    ADMIN
}

public class Preferences {
    Set<Action> S;
    Set<Action> W;
    Set<Action> R;
    Set<Action> U;
}

public class AvailabilityInterval {
    LocalTime startTime;            // Start time of the interval
    LocalTime endTime;              // End time of the interval
    Availability availability;      // Associated availability
}

public class Availability {
    LocalDate date;                     // Date of availability
    Volunteer volunteer;                // Associated volunteer
    Set<AvailabilityInterval> slots;    // Time slots of availability
}

public class PersonalData {
    // Personal details of the volunteer:
    String firstName;           // First name of the volunteer
    String lastName;            // Last name of the volunteer
    String email;               // Email address for communication
    String phone;               // Phone number
    LocalDate dateOfBirth;      // Date of birth
    String address;             // Full address
}

public class Volunteer extends PersonalData {
    // Volunteer-specific attributes:
    Position position;               // Assigned position, e.g. CANDIDATE, LEADER
    float limitOfWeeklyHours;              // Weekly hour limit
    float actualWeeklyHours;               // Hours already worked in the actual
    week
    ArrayList<Availability> availabilities;// List of availability time slots
    Preferences preferences;               // Volunteer preferences for actions

    // Constructor initializes default values:
    public Volunteer() {
        limitOfWeeklyHours = 0.0;
        actualWeeklyHours = 0.0;
        availabilities = new ArrayList<>();
        preferences = new Preferences();
    }
}
```

LISTING 3.3: Volunteer data structure. Each volunteer is described by an object of class `Volunteer`.

3.3.2 Operations

The operations associated with managing volunteer data are defined by the `Volunteer` interface, as shown in Listing 3.4. They can be categorized into the following sets:

- **Inner:** Operations that manage the lifecycle of a volunteer, such as creating, editing, and deleting volunteer data.
- **Position Management:** Assigning and changing Positions of a volunteer.
- **Preference Management:** Managing the initialization and modification of a volunteer's preferences in the context of actions.
- **Scheduling and Task Management:** Handling availability and task assignments for a volunteer.

Inner Operations

These operations manage the core lifecycle of a volunteer:

- `createVolunteer`: Creates a new volunteer with default settings, including initial preferences and position.
- `editVolunteer`: Updates personal details or other attributes of a volunteer.
- `deleteVolunteer`: Removes a volunteer from the system, along with their preferences, availability, and duties.
- `getVolunteerById`: Retrieves the complete data of a specific volunteer.
- `assignPosition`: Assigns a new position to a volunteer, ensuring compliance with position transition rules.
- `getPosition`: Retrieves the current position of a volunteer.

Scheduling and Task Management

These operations handle volunteer availability and assigned duties:

- `setAvailability`: Defines the availability of a volunteer, including specific dates and time slots.
- `getAvailability`: Retrieves a volunteer's availability for scheduling purposes.

```
interface Volunteer {
      ID         createVolunteer();
      Errors     editVolunteer(ID volunteerId, PersonalData details);
      Errors     deleteVolunteer(ID volunteerId);
      Volunteer  getVolunteerById(ID volunteerId);

      Errors     assignPosition(ID volunteerId, Position newPosition);
      Position   getPosition(ID volunteerId);

      Errors     initializePreferences(ID volunteerId);
      Errors     updatePreferences(ID volunteerId, Action action,
                                   PreferenceType type);
      Errors     applyPreferencesToSchedule(ID volunteerId);

      Errors     setAvailability(ID volunteerId, Availability availability);
      Availability getAvailability(ID volunteerId);

}
```

Listing 3.4: Operations of the `Volunteer` module.

Listing 3.4 presents all the operations of the `Volunteer` module. Together with Listing 3.3, they describe the whole interface of the module responsible for the management of volunteers.

3.4 Schedule Management

3.4.1 Schedule Data Model

The `Schedule` module is responsible for managing and generating schedules for actions in the system. It integrates data such as volunteer preferences, availability, and action demands to create create structured weekly schedules.

To generate a schedule, the system requires predefined action demands, which specify the number of volunteers needed at specific time intervals for a given action. An `ActionDemand` object is linked to an `Action` and provides a structured way to define staffing requirements over time. Each `ActionDemand` consists of multiple `ActionDemandInterval`, representing distinct time slots with defined minimum (`needMin`) and maximum (`needMax`) volunteer requirements.

FIGURE 3.5: Diagram of Volunteer Assignment to Demand

A `Schedule` represents a structured weekly plan, where each `ActionDemand` specifies volunteer requirements for a given day within that week. Each schedule contains:

- A defined weekly range, represented by `startDate` and `endDate`.
- A set of actions occurring within that period.
- The corresponding `ActionDemands` for each action.

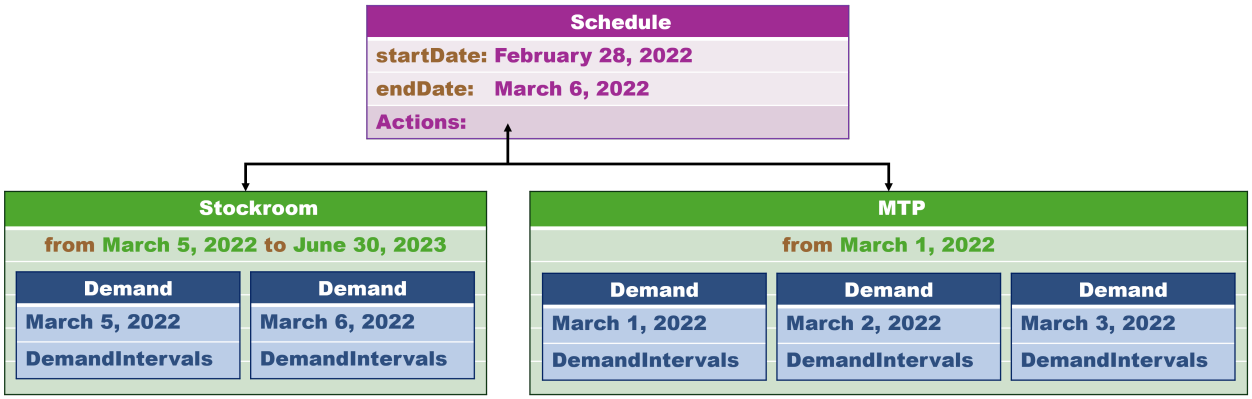

FIGURE 3.6: A weekly schedule grouping multiple actions and their corresponding action demands.

The planning process involves a heuristic approach that prioritizes:

- **Matching preferences:** Volunteers marked as `Strongly Mine (S)` are given the highest priority, followed by `Weakly Mine (W)`.
- **Availability:** Volunteers must be available during the time slots required by the action demands.
- **Demand fulfillment:** Ensuring that the minimum number of volunteers (`needMin`) is met for each demand interval.
- **Weekly hour limits:** Volunteers cannot exceed their defined weekly hour limits (`limitOfWeeklyHours`).

The heuristic function balances volunteer workloads while maximizing demand fulfillment. The structure of the `Schedule` class is presented in Listing 3.5.

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;
import java.util.Set;
import java.util.ArrayList;
import java.util.HashMap;

public class ActionDemandInterval {
    Long intervalId;    // Unique identifier for the action demand interval
    LocalTime startTime;// Start time of the interval
    LocalTime endTime;  // End time of the interval
    Long needMin;       // Minimum required volunteers
    Long needMax;       // Maximum allowed volunteers
    Set<Volunteer> assignedVolunteers; // Assigned volunteers for the interval
}

public class ActionDemand {
    Long demandId;      // Unique identifier for the demand
    LocalDate date;     // Date for which the demand applies
    Action action;      // Associated action
    Set<ActionDemandInterval> actionDemandIntervals; // Demand intervals for the
    action

}

public class Schedule {
    final noDate= "1970-01-01";
    Long scheduleId;            // Unique identifier for the schedule
    LocalDate startDate         // First day of the scheduled week
    LocalDate endDate           // Last day of the scheduled week
    ArrayList<Action> actions   // List of actions taking place in a given week
    List<ActionDemand> actionDemands; // All demands for the scheduled week


    // Constructor initializes default values :
    public Schedule() {
        startDate = noDate;
        endDate =   noDate;
        actions = new ArrayList<>();
        actionDemands = new ArrayList<>()}
    }
}
```

LISTING 3.5: Schedule data structure. Each schedule is described by an object of class `Schedule`.

3.4.2 Operations

The operations associated with managing schedules are defined by the `Schedule` interface, as shown in Listing 3.6. These operations are categorized into the following sets:

- **Existential:** Operations that manage the lifecycle of schedules, such as creation, retrieval, and deletion.
- **Schedule Generation:** Creating optimized schedules based on volunteer preferences, availability, and action demands.
- **Schedule Modification:** Adjusting or modifying schedules dynamically to reflect changes in preferences, availability, or action demands.
- **Volunteer Management:** Managing volunteer assignments and duties within schedules.

Existential Operations

These operations manage the core lifecycle of schedules:

- `createSchedule`: Creates a new schedule for a specific action and time frame.
- `getScheduleById`: Retrieves the complete details of a specific schedule using its unique identifier.
- `deleteSchedule`: Deletes a schedule and all associated assignments and duties.

Schedule Generation

These operations are responsible for generating optimized schedules:

- `generateSchedule`: Generates a schedule for a specified week and action by balancing volunteer preferences, availability, and action demands.

Schedule Modification

These operations allow schedules to be adjusted dynamically:

- `modifySchedule`: Updates a schedule to reflect changes in volunteer availability, preferences, or action demands.
- `updateDemand`: Modifies the demand intervals for a specific action within a schedule.
- `adjustAssignments`: Reassigns volunteers to duties based on updated availability or preferences.

Volunteer Management

These operations handle volunteer assignments and duties within schedules:

- `assignVolunteerToDemand`: Assigns a volunteer to a specific duty interval within the schedule.
- `removeVolunteerFromDemand`: Removes a volunteer from a specific duty interval, ensuring the schedule remains balanced.
- `getVolunteerSchedule`: Retrieves the schedule details for a specific volunteer, including their assigned duties and time slots.

```
interface Schedule {
      // Existential operations:
      ID        createSchedule(Action action, LocalDate startDate, LocalDate
    endDate);
      Schedule  getScheduleById(ID scheduleId);
      Errors    deleteSchedule(ID scheduleId);

      // Schedule generation:
      Errors    generateSchedule(LocalDate weekStart, Action action);

      // Schedule modification:
      Errors    modifySchedule(ID scheduleId, ModifyScheduleRequest modifications);
      Errors    updateDemand(ID actionId, Demand demand);
      Errors    adjustAssignments(ID scheduleId);

      // Volunteer management:
      Errors assignVolunteerToDemand(Long volunteerId, ActionDemand actionDemand);
      Errors removeVolunteerFromDemand(Long volunteerId, ActionDemand actionDemand)
    ;
      Schedule  getVolunteerSchedule(ID volunteerId);
}
```

Listing 3.6: Operations of the `Schedule` module.

Listing 3.6 presents all the operations of the `Schedule` module. Together with Listing 3.5, they describe the whole interface of the module responsible for the management of schedules.

Chapter 4

Technical Details

4.1 Overview

This chapter provides an in-depth look into the technical aspects of the VoluMan system. It is divided into three sections, each focusing on a key area of the server-side implementation:

- **Architecture**: A detailed explanation of the architectural decisions made during the development of the server, including the rationale behind these choices.
- **Server Endpoints**: A comprehensive description of the RESTful API endpoints exposed by the server, including their functionality and usage.
- **JSON Representation**: An overview of the data structures used for communication between the server and the client, with examples of JSON payloads.

These sections collectively provide a clear understanding of how the server is structured, how it interacts with the client, and how data is exchanged within the system.

4.2 Architecture

The server-side architecture of the VoluMan system was designed to ensure maintainability and clarity. The choice of technologies and architectural patterns was guided by several key factors to balance efficiency, reliability, and ease of development.

When selecting the technology stack, the team prioritized the following criteria:

- **Scalability:** The system was designed to handle a growing number of users and increasing data loads. The chosen technologies support horizontal and vertical scaling, ensuring long-term usability.
- **Maintainability:** The architecture follows well-established design patterns and coding principles to facilitate future modifications and feature extensions.
- **Industry Adoption and Community Support:** Using widely accepted frameworks and technologies ensures long-term support, security updates, and availability of learning resources.
- **Code Readability and Efficiency:** Preference was given to technologies that allow concise, readable, and modular code, which simplifies debugging and future maintenance.

- **Security Considerations:** The system was designed with built-in authentication and role-based access control to ensure data integrity and user security.

This section describes the key architectural decisions made during the development of the system, covering the backend framework, database, authentication mechanisms, and API design. Each decision was made to ensure that the system meets the functional requirements while remaining adaptable for future improvements.

Architectural Pattern: MVC The system is built using the Model-View-Controller (MVC) architectural pattern. This pattern provides a clear separation of concerns:

- **Model**: Handles data-related logic and interactions with the database. Implemented using the Repository layer in Spring Framework, it ensures efficient and structured data access.
- **View**: In the context of the server-side, the view is represented by RESTful API responses, allowing the client-side application to consume the data.
- **Controller**: Manages the logic for handling HTTP requests and routing them to appropriate services. It acts as an intermediary between the client and the backend logic.

The use of MVC enhances modularity and simplifies debugging, as each component is responsible for a specific aspect of the system.

API Design The server exposes a RESTful API for communication with the client-side application [2]. REST was chosen for its simplicity, scalability, and wide adoption in modern web services. JSON is used as the data exchange format due to its readability and ease of use.

Security and Authentication Authentication and role-based access control are implemented using JSON Web Tokens (JWT). This ensures secure communication between clients and the server, while also providing an efficient way to manage user sessions without storing state on the server side.

Database and Data Access The system uses a MySQL relational database, managed via Spring JPA and Hibernate. These technologies were selected for their stability, flexibility in query management, and ease of integration with the Java ecosystem. The database schema was designed to support scalability and avoid redundancy.

Programming Principles and Patterns The system adheres to several design principles and patterns to ensure clean and maintainable code:

- **Clean Code and SOLID Principles**: Emphasis was placed on readability, reusability, and adherence to the five SOLID principles to create a robust and extensible codebase.
- **Singleton Pattern**: Applied where a single instance of a class is required throughout the application, such as configuration management.

Coding Conventions To maintain consistency throughout the codebase, camelCase naming conventions were used for variables, methods, and class names. This ensures uniformity and readability.

4.2.1 Technologies Used

This section provides an overview of the specific frameworks, libraries, and tools used in the implementation.

Backend Development The backend of the system is implemented in Java, utilizing the Spring Boot framework for building RESTful web services. Key dependencies include:

- **Java 17** – the primary programming language.
- **Spring Boot 3.1.3** – provides essential components for REST API development [4].
  - spring-boot-starter-web 3.1.3 – handles HTTP request processing.
  - spring-boot-starter-data-jpa 3.1.3 – integrates Hibernate ORM for database operations.
  - spring-boot-starter-security 3.1.3 – provides authentication and authorization mechanisms.
- **Hibernate 6.2.9.Final** – ORM framework used for managing database entities.
- **Jakarta Validation 3.0.2** – facilitates input validation in models.
- **Lombok** – simplifies Java class definitions by reducing boilerplate code.

Database and Persistence The system relies on MySQL as the relational database, with Flyway used to manage schema migrations.

- **MySQL** – stores volunteer, action, and schedule-related data.

Security and Authentication User authentication and authorization are managed using industry-standard security practices:

- **Spring Security 3.1.3** – provides role-based access control (RBAC).
- **JWT (java-jwt 4.3.0)** – enables token-based authentication.

API Documentation and Testing To facilitate API usage and validation, the following tools were employed:

- **Spring Security Test** – provides tools for testing security configurations.
- **Spring Boot Starter Test** – framework for writing and executing unit tests.

Development and Deployment Tools To streamline the development and deployment workflow, the following technologies were incorporated:

- **Maven** – used for dependency management and build automation.
- **Spring Boot Maven Plugin** – enables easy packaging and execution of the application.

4.3 LangSet Module

4.3.1 Architecture

Overview

LangSet is a language resource management system that enables centralized control over available languages. Its core principle involves maintaining a list of languages, each represented by a code compliant with the ISO 639-1 standard, that specifies two-letter codes to uniquely identify languages, such as "en" for English or "fr" for French. This standard includes more than 180 language codes that cover the most widely used languages worldwide. The system allows for adding, removing, and verifying the existence of languages to ensure data consistency and integrity. The entire system is based on consistent validation rules and API.

Functional Description

The module's core functionality revolves around the management of language resources:

- **Retrieve All Languages:** Retrieve a complete list of all languages managed by the system.
- **Add Language:** Add a new language if it is in accordance with the ISO 639-1 standard and is not already present.
- **Remove Language:** Remove a language if it exists within the managed list.
- **Clear All Languages:** Remove all currently stored languages.
- **Check Language Existence:** Verify whether a specific language exists in the system.

Module Relationships

- **LangSetController:** Exposes the REST API endpoints to interact with the module.
- **LangSetService:** Contains the core business logic for language management, including validation and operations on the language list.
- **LangSet Interface:** Defines the contract for managing languages, including required methods for retrieval, addition, deletion, and validation.
- **Lang Class:** Represents a single language with its associated ISO 639-1 code.

LangSet Interface

The `LangSet` interface serves as the abstraction layer for language management. It ensures that any implementation follows the defined contract.

```
public interface LangSet {
    Set<Lang> getAllLang();              // Retrieve all languages

    void addLang(Lang language);     // Add a language if it does not exist

    void removeLang(LangISO639 code);    // Remove a language if it exists

    void emptyLang();                    // Remove all languages

    boolean isLang(LangISO639 code);     // Check if a language exists
}
```

4.3.2 Implementation

Functionality Realization

The LangSet module implements the `LangSet` interface and connects the API controller (`LangSetController`) with the core business logic (`LangSetService`). This implementation ensures separation of concerns:

- **Controller Layer:** Handles user requests and responses.
- **Service Layer:** Implements the business logic, including validation and persistence operations.

How Functions Are Realized

- **Retrieve All Languages:**
  - The controller method `getAllLanguages()` retrieves the list of languages via the service layer.
  - The service method `getAllLanguages()` fetches the list of `Lang` objects from the internal storage.
- **Add Language:**
  - The controller method `addLanguage(Lang language)` forwards the request to the service layer.
  - The service validates the language code for uniqueness and ISO 639-1 compliance and then adds it to the internal storage.
- **Remove Language:**
  - The controller method `removeLanguage(String code)` calls the service to delete the specified language.
  - The service validates the existence of the language and removes it if present.

- **Clear All Languages:**
  - The controller method `emptyLanguages()` triggers a service call to clear the list.
  - The service removes all entries from the language list.
- **Check Language Existence:**
  - The controller method `checkIfLanguageExists(String code)` queries the service for the existence of a language.
  - The service checks if the specified code exists in the language list.

Endpoints

The `LangSetController` exposes the following REST API endpoints:

- **Retrieve All Languages**
  - **Method:** `GET`
  - **URL:** `/api/languages`
  - **Response:**

```
[
  { "code": "en" },
  { "code": "pl" }
]
```

- **Add a New Language**
  - **Method:** `POST`
  - **URL:** `/api/languages`
  - **Request:**

```
{ "code": "es" }
```

  - **Response (Success):**

```
Language has been added!
```

  - **Response (Error):**

```
Invalid ISO 639-1 code: xx
```

- **Remove a Language**
  - **Method:** `DELETE`
  - **URL:** `/api/languages/{code}`
  - **Response (Success):**

```
Language has been deleted!
```

- **Response (Error):**

```
Language with code xx does not exist.
```

- **Clear All Languages**
  - **Method:** `DELETE`
  - **URL:** `/api/languages/empty`
  - **Response:**

```
All languages have been deleted.
```

- **Check Language Existence**
  - **Method:** `GET`
  - **URL:** `/api/languages/exists/{code}`
  - **Response (Success):**

```
HTTP 200 OK
```

  - **Response (Error):**

```
HTTP 404 NOT FOUND
```

Lang Class

The `Lang` class represents a single language in the system.

```
public class Lang {
    private String code; // ISO 639-1 language code

    // Getters and setters (Lombok)
    @Getter @Setter
    public Lang() {}

    public Lang(String code) {
        this.code = code;
    }
}
```

Workflow

1. The user sends a request to the API (e.g., `POST /api/languages`).
2. The request is processed by `LangSetController`.
3. The controller invokes the appropriate method in `LangSetService`:
   - Validations are performed (e.g., ISO 639-1 compliance, uniqueness).
   - Business logic is executed.
4. The response is returned to the user, indicating success or failure.

4.4 Document Management

4.4.1 Architecture

Introduction

Within document management, the `Document` interface stores files associated with a language, a type, and a numeric version (e.g. 1.0.1). Adding documents requires validation of their language and version, ensuring that a specific type-version combination does not already exist. Documents are stored in Azure Blob Storage and can be retrieved efficiently. The system detects outdated documents and implements version increment functionalities for patch, minor, and major changes, enabling controlled evolution of document resources.

Functional Description

The Document Management System supports the following functionalities:

- **Retrieve All Documents:** Allows retrieval of all documents currently stored in the system.
- **Retrieve Documents by Type:** Filters and retrieves documents based on their specified type (e.g. manual, policy).
- **Add Document:** Validates the language, type and version of a document, ensuring that the combination of type and version does not already exist before saving it to Azure Blob Storage.
- **Detect Outdated Documents:** Identifies and retrieves outdated documents based on their type and versioning.
- **Retrieve Document by File Name:** Allows direct retrieval of a document using its file name.
- **Version Validation:** Ensures no duplicate type-version combinations exist and identifies the maximum version for a document type.

Module Relationships

- **DocumentController:** Exposes REST API endpoints to manage documents.
- **DocumentService:** Implements the business logic for document operations, including validation, versioning, and interactions with Azure Blob Storage.
- **Document Interface:** Defines the contract for managing documents in the system.
- **Document Class:** Represents an individual document with associated metadata such as language, type, and version.

Document Interface

The `Document` interface defines the contract for managing documents in the system.

```
public interface Document {
    Set<Document> getAllDocuments();     // Retrieve all documents

    void addDocument(LangISO639 languageCode, String type,
            VersionChangeRequest versionChange, String fileName,
            InputStream fileStream);
            // Add a new document after validating language, type, and version

    Set<Document> getOutdatedDocuments(Set<Document> allDocuments);
        // Retrieve a list of outdated documents

    void downloadDocument(String documentName);
        // Download a document
}
```

4.4.2 Implementation

Document Class

Description: Represents a single document stored in the system.

Fields:

- **languageCode (String):** The ISO 639-1 language code representing the document's language (e.g. `"en"` for English).
- **type (String):** The type of document (e.g. `"manual"`, `"policy"`).
- **version (`List<Integer>`):** A list of integers representing the document version in the format `x.y.z` (e.g. `[1, 0, 2]`).
- **filePath (String):** The URL pointing to the document's location in Azure Blob Storage.

DocumentController Class

Description: Handles HTTP requests related to documents and provides a REST API for managing documents.

Endpoints:

- **Retrieve All Documents**
  - **Method:** `GET /api/documents`
  - **Description:** Returns all documents stored in the system.
  - **Response:**

```
[
  {
    "languageCode": "en",
    "type": "manual",
    "version": [1, 0, 2],
    "filePath": "https://storageaccount.blob.core.windows.net/
container/document_1.0.2.pdf"
  },
  {
    "languageCode": "pl",
    "type": "policy",
    "version": [2, 1, 0],
    "filePath": "https://storageaccount.blob.core.windows.net/
container/document_2.1.0.pdf"
  }
]
```

- **Add a New Document**
  - **Method:** `POST /api/documents`
  - **Description:** Adds a new document to the system and saves it in Azure Blob Storage.
  - **Request:**

```
{
    "file": "document.pdf",
    "languageCode": "en",
    "type": "manual",
    "version": "INCREASE_MINOR"
}
```

  - **Response (Success):**

```
{
    "message": "Document has been added to Azure Blob Storage."
}
```

  - **Response (Error):**

```
{
    "message": "Invalid ISO 639-1 code: xx"
}
```

- **Download a Document by Name**
  - **Method:** `GET /api/documents/file/{filename}`
  - **Description:** Retrieves a document from Azure Blob Storage by its filename.
  - **Response:** The file is downloaded with a response header:

```
Content-Disposition: attachment; filename="document_1.0.2.pdf"
```

- **Check Outdated Documents**
  - **Method:** `GET /api/documents/check`
  - **Description:** Returns a list of outdated documents, optionally filtered by type.
  - **Response:**

```
[
  {
    "languageCode": "pl",
    "type": "policy",
    "version": [1, 0, 0],
    "filePath": "https://storageaccount.blob.core.windows.net/
container/document_1.0.0.pdf"
  }
]
```

Workflow

1. A user sends an API request, such as adding or retrieving a document.
2. The `DocumentController` handles the request and delegates it to the appropriate method in `DocumentService`.
3. The `DocumentService` performs validation, interacts with Azure Blob Storage, and executes business logic.
4. The system responds to the user, providing the outcome of the requested operation.

4.5 Position Management

4.5.1 Architecture

Overview

The Position Management module is responsible for managing volunteer positions within the platform. It defines the possible positions for users and provides mechanisms for position transitions. The module enforces strict transition rules to ensure consistent and valid position changes.

Functional Description

The Position Management System provides the following functionalities:

- **Retrieve Position Transition Rules:** Retrieves the defined position transition rules stored in the system.
- **Assign Position:** Allows administrators to assign a specific position to a volunteer, validating the transition.
- **Validate Position Transition:** Checks whether a position transition is permitted between two positions.
- **Initialize Position Transitions:** Ensures all position transitions are properly defined during system initialization.

Module Relationships

- **PositionController:** Provides REST API endpoints for position management, such as assigning positions and validating transitions.
- **PositionService:** Implements the business logic for assigning positions and validating transitions based on the `PositionTransitionTable`.
- **PositionTransitionTable:** Defines the allowed position transitions and validates the transition rules.
- **VolunteerPosition Enum:** Represents the predefined positions in the system, such as `CANDIDATE`, `VOLUNTEER`, and `LEADER`.

Position Interface

The `Position` interface defines the abstraction for managing volunteer positions. It ensures that any implementation adheres to the required operations for position management.

```
public interface Position {
    void getTransitionRules();
        // Retrieve the transition rules for all positions

    void assignPosition(Volunteer volunteer, VolunteerPosition newPosition);
        // Assign a new position to a volunteer after validating the transition

    boolean canTransition(VolunteerPosition fromPosition, VolunteerPosition
    toPosition);
        // Check if a transition is allowed between two positions
```

```
    void validateTransitions();
        // Validate the transitions to ensure all transitions are properly defined
}
```

4.5.2 Implementation

Introduction

The implementation of the Position Management System focuses on defining strict transition rules between `positions` presented in 3.3.1 and ensuring that all operations adhere to these rules. The transition logic is encapsulated in the `PositionTransitionTable` and validated during system initialization.
The allowed transitions are represented in tabular form, shown in Table 4.1.

	CANDIDATE	VOLUNTEER	LEADER	RECRUITER	ADMIN
CANDIDATE	false	true	false	false	false
VOLUNTEER	false	false	true	true	true
LEADER	false	true	false	false	false
RECRUITER	false	true	false	false	false
ADMIN	false	true	true	true	false

Table 4.1: Table of allowed transitions between positions

PositionTransitionTable Class

Description: Stores the position transition table and defines the rules for allowed transitions between positions. Ensures consistency and prevents unauthorized transitions.

Fields:

- **transitions (Map¡VolunteerPosition, Map¡VolunteerPosition, Boolean»):** A nested map defining the allowed transitions between positions.

Methods:

Method Name	Return Type	Description
`setTransition(VolunteerPosition fromPosition, VolunteerPosition toPosition, boolean canTransition)`	`void`	Defines whether a transition between two positions is allowed.
`canTransition(VolunteerPosition fromPosition, VolunteerPosition toPosition)`	`Boolean`	Checks if a transition between two positions is possible. Returns `true`, `false`, or `null` if undefined.
`validateTable()`	`void`	Validates that all transitions in the table are explicitly defined. Throws an exception if any transition is undefined.

Table 4.2: Methods of the `PositionTransitionTable` Class

PositionService Class

Description: Implements the logic for position assignments and transition validations based on the `PositionTransitionTable`.

Fields:

- **transitionTable (PositionTransitionTable):** Stores and validates position transition rules.

Methods:

Method Name	Return Type	Description
`assignPosition(Volunteer volunteer, VolunteerPosition newPosition)`	`void`	Assigns a new position to a volunteer. Validates the transition using the `PositionTransitionTable`. Logs the operation.
`canTransition(VolunteerPosition fromPosition, VolunteerPosition toPosition)`	`Boolean`	Checks whether a position transition is allowed between two positions.
`validateTransitions()`	`void`	Ensures that all transitions in the `PositionTransitionTable` are explicitly defined.

Table 4.3: Methods of the `PositionService` Class

4.6 Server Endpoints

This section provides a detailed overview of the endpoints exposed by the server across all key modules of the system, including `Action`, `Demand`, `Volunteer`, and `Schedule`. Each endpoint is described with its HTTP method, URL, purpose, and expected behavior. The endpoints collectively enable interaction with the system by handling core functionalities such as managing actions, volunteers, schedules, preferences, and demands.

The server's REST API adheres to the principles of RESTful design, ensuring scalability, simplicity, and ease of integration. It provides a clear interface for managing various components of the system while enforcing appropriate access controls. The endpoints are categorized into the following modules:

- `Action` **Module:** This module is responsible for handling operations related to the management of actions. Endpoints allow users to retrieve, create, update, and delete actions, as well as manage descriptions and headings associated with actions.
- `Demand` **Module:** Focused on managing the demands associated with actions, this module provides endpoints to retrieve and define demands, including the specification of time intervals and the number of required volunteers.
- `Volunteer` **Module:** This module provides endpoints to manage volunteers, including operations for retrieving volunteer details, managing roles, setting weekly hour limits, and handling availability and preferences. It also includes features for candidate management, such as accepting or rejecting candidates.

- `Schedule` **Module:** The `Schedule` module handles the generation and management of schedules. It provides endpoints to generate schedules based on preferences, availability, and demands, modify existing schedules, and retrieve schedules by action or volunteer. This module also supports setting volunteer preferences for actions and defining action demands.

The endpoints within each module are designed to enable communication between the system's components and provide a foundation for both administrative and volunteer interactions. Detailed descriptions of each endpoint, including their expected input and output, can be found in subsections dedicated to each module.

4.6.1 Action Endpoints

The `ActionController` provides the following REST API endpoints for managing actions:

- **Retrieve All Actions**
  - **Method:** GET
  - **URL:** `/actions`
  - **Description:** Retrieves a list of all actions stored in the system.
  - **Response:** A list of `Action` objects.
- **Retrieve a Specific Action**
  - **Method:** GET
  - **URL:** `/actions/{idAction}`
  - **Description:** Retrieves the details of a specific action identified by its unique ID.
  - **Response:** An `Action` object if the ID exists, or a `404 Not Found` error if it does not.
- **Retrieve Action Description**
  - **Method:** GET
  - **URL:** `/actions/{idAction}/description`
  - **Description:** Retrieves the description of a specific action.
  - **Response:** A `DescriptionResponse` object containing the action description.
- **Retrieve Action Heading**
  - **Method:** GET
  - **URL:** `/actions/{idAction}/heading`
  - **Description:** Retrieves the heading of a specific action.
  - **Response:** A `HeadingResponse` object containing the action heading.

- **Create a New Action**
  - **Method:** POST
  - **URL:** `/actions`
  - **Description:** Adds a new action to the system.
  - **Request Body:** An `AddActionRequest` object containing details about the new action.
  - **Response:** A `201 Created` status with the created `Action` object, or an appropriate error code if validation fails.
- **Delete an Action**
  - **Method:** DELETE
  - **URL:** `/actions/{actionId}`
  - **Description:** Deletes a specific action identified by its ID.
  - **Response:** A `200 OK` status on success, or a `404 Not Found` error if the action does not exist.
- **Update Action Description**
  - **Method:** PUT
  - **URL:** `/actions/{idAction}/description`
  - **Description:** Updates the description of a specific action.
  - **Request Body:** A `ChangeDescriptionRequest` object containing the updated description.
  - **Response:** A `200 OK` status on success, or an appropriate error code if validation fails.
- **Close an Action**
  - **Method:** PUT
  - **URL:** `/actions/{idAction}/close`
  - **Description:** Marks a specific action as closed.
  - **Request Body:** A `CloseActionRequest` object.
  - **Response:** A `200 OK` status on success, or an appropriate error code if validation fails.

4.6.2 Volunteer Endpoints

The `VolunteerController` and related controllers expose the following REST API endpoints for managing volunteers, their roles, availability, preferences, duties, and user-related actions:

- **Retrieve All Candidates**
  - **Method:** GET
  - **URL:** `/candidates`
  - **Description:** Retrieves a list of all candidates.
  - **Request Parameter:** `recruiterId`, used to validate access.
  - **Response:** A list of `Candidate` objects.

- **Accept a Candidate**
  - **Method:** POST
  - **URL:** `/candidates/{idCandidate}/accept`
  - **Description:** Accepts a candidate and upgrades their status.
  - **Request Parameter:** `recruiterId`, used to validate the action.
  - **Response:** The accepted `Candidate` object.
- **Refuse a Candidate**
  - **Method:** DELETE
  - **URL:** `/candidates/{idCandidate}/refuse`
  - **Description:** Refuses a candidate, effectively removing them from consideration.
  - **Request Parameter:** `recruiterId`, used to validate the action.
  - **Response:** A `200 OK` status on success.
- **Retrieve All Volunteers**
  - **Method:** GET
  - **URL:** `/volunteers`
  - **Description:** Retrieves a list of all volunteers stored in the system.
  - **Response:** A list of `Volunteer` objects.
- **Retrieve a Specific Volunteer**
  - **Method:** GET
  - **URL:** `/volunteers/{idVolunteer}`
  - **Description:** Retrieves details of a specific volunteer identified by their unique ID.
  - **Response:** A `Volunteer` object if the ID exists, or a `404 Not Found` error if it does not.
- **Add a New Volunteer**
  - **Method:** POST
  - **URL:** `/volunteers`
  - **Description:** Adds a new volunteer to the system.
  - **Request Body:** A `Volunteer` object containing the details of the new volunteer.
  - **Response:** The newly created `Volunteer` object.
- **Delete a Volunteer**
  - **Method:** DELETE
  - **URL:** `/volunteers/{idVolunteer}`
  - **Description:** Deletes a specific volunteer identified by their ID.
  - **Request Body:** An `AdminRequest` object containing the admin's credentials.
  - **Response:** A `200 OK` status on success, or an error code if the deletion fails.

- **Change a Volunteer's Position**
  - **Method:** PUT
  - **URL:** `/volunteers/{idVolunteer}/roles`
  - **Description:** Changes the position of a volunteer.
  - **Request Parameters:** `position` parameter specifying the new position.
  - **Request Body:** An `AdminRequest` object.
  - **Response:** A `200 OK` status on success, or a relevant error code.
- **Retrieve a Volunteer's Weekly Hour Limit**
  - **Method:** GET
  - **URL:** `/volunteers/{idVolunteer}/limit-weekly-hours`
  - **Description:** Retrieves the weekly hour limit of a specific volunteer.
  - **Response:** A `WeeklyHoursResponse` object containing the limit value.
- **Set a Volunteer's Weekly Hour Limit**
  - **Method:** PUT
  - **URL:** `/volunteers/{idVolunteer}/limit-weekly-hours`
  - **Description:** Sets the weekly hour limit for a specific volunteer.
  - **Request Body:** A `WeeklyHoursRequest` object specifying the new limit.
  - **Response:** A `200 OK` status on success.
- **Retrieve All Volunteer Availabilities**
  - **Method:** GET
  - **URL:** `/volunteers/availabilities`
  - **Description:** Retrieves all availability records of volunteers.
  - **Response:** A list of `AvailResponse` objects.
- **Retrieve Availability of a Specific Volunteer**
  - **Method:** GET
  - **URL:** `/volunteers/{volunteerId}/availabilities`
  - **Description:** Retrieves availability details for a specific volunteer.
  - **Response:** A `VolunteerAvailResponse` object.
- **Set Volunteer Availability**
  - **Method:** POST
  - **URL:** `/volunteers/{volunteerId}/availabilities`
  - **Description:** Allows a volunteer to define their availability.
  - **Request Body:** A `VolunteerAvailRequest` object specifying the availability slots.
  - **Response:** A `200 OK` status on success, or an error code if fails.

4.6.3 Schedule Endpoints

The `ScheduleController` exposes the following REST API endpoints for managing schedules, preferences, demands, and generating scheduling plans:

- **Set Volunteer Preferences for an Action**
  - **Method:** POST
  - **URL:** `/actions/{actionId}/preferences`
  - **Description:** Allows a volunteer to set their preferences (e.g., preferred, undecided, or rejected) for a specific action.
  - **Request Body:** `ActionPrefRequest` object containing the volunteer ID and decision (`S`, `W`, `R`, or `U`).
  - **Response:** `200 OK` status on success, or an error code if the action or volunteer ID is invalid.
- **Set Demands for an Action**
  - **Method:** POST
  - **URL:** `/actions/{actionId}/demands`
  - **Description:** Allows a leader to define the demands (e.g., required volunteers) for a specific action on specific days and times.
  - **Request Parameters:** `year` and `week`, specifying the time frame.
  - **Request Body:** `ActionNeedRequest` object containing the leader ID and detailed demand slots.
  - **Response:** `200 OK` status on success, or an error code if the leader or action validation fails.
- **Retrieve Demands for an Action**
  - **Method:** GET
  - **URL:** `/actions/{actionId}/demands`
  - **Description:** Retrieves demands for a specific action.
  - **Response:** A list of `ActionDemand` objects.
- **Generate Weekly Schedule**
  - **Method:** POST
  - **URL:** `/schedules/generate`
  - **Description:** Generates a weekly schedule based on volunteers' preferences, availabilities, and the demands of actions.
  - **Request Body:** `GenerateScheduleRequest` object containing the admin ID and target date.
  - **Response:** `200 OK` status on success, or an error code if validation fails.

- **Modify a Volunteer's Schedule**
  - **Method:** PUT
  - **URL:** `/schedules/{scheduleId}/modify`
  - **Description:** Allows a volunteer to modify their schedule for a specific week.
  - **Request Parameters:** `year` and `week`, specifying the time frame.
  - **Request Body:** `ModifyScheduleRequest` object containing the volunteer ID, duty intervals and type of modification.
  - **Response:** `200 OK` status on success, or an error code if validation fails.
- **Retrieve Schedule by Action**
  - **Method:** GET
  - **URL:** `/schedules/actions/{actionId}`
  - **Description:** Retrieves the schedule for a specific action, including assigned volunteers and their duties.
  - **Request Parameter:** `leaderId`, validating access to this information.
  - **Response:** An `ActionScheduleDto` object containing the schedule details, or an error code if validation fails.
- **Retrieve Schedule by Volunteer**
  - **Method:** GET
  - **URL:** `/schedules/volunteers/{volunteerId}`
  - **Description:** Retrieves the schedule for a specific volunteer for a given week.
  - **Request Parameters:** `year` and `week`, specifying the time frame.
  - **Response:** A `VolunteerScheduleDto` object containing the volunteer's schedule, or an error code if validation fails.

Chapter 5

Functional completeness of the interface

To show that the proposed server interface is functionally complete, we will outline the implementation of all the use cases presented in Section 2 using the methods introduced in Chapter 3. When describing an implementation of the use cases, the following notation will be used:

- To make a clear distinction between the server logic and the user interface, the methods belonging to the proposed **server interface** will be presented in **blue** font and those corresponding to the **user interface** will be given in **violet**.
- For the sake of simplicity, the Python-like notation with significant indentation will be used.
- Also, for the sake of simplicity, it is assumed that possible errors will be communicated and handled via exception handling. That allows us to skip code dealing with error handling.

5.1 Analysis of Functional Completeness

Management of Actions

MA-1: Actor = Admin

Creating a new action:

a = **newAction**()

$\mathcal{A}' = \mathcal{A} \cup \{a\}$

a.Heading = “”

a.Description = “”

a.Leaders’ = a.Leaders $\cup$ **oneOf** $\mathcal{V}$

MA-1: Actor = Admin

Creating a new action:

Template = **getNewActionTemplate**()

present Template

event Fill details:

input a.Heading, a.Description

Leaders = **getPersData**(LEADERS)

present Leaders

event Assign leader $l \in$ Leaders:

a.Leaders = $\{l\}$

create(a, ACTION)

log(thisActor, `Action-Created`, $[a]$)

Table 5.1: Outline of the implementation of the MA-1 use cases.

MA-2: Actor = Admin

Closing an action:

a = **oneOf** $\mathcal{A}$

$\mathcal{A}' = \mathcal{A} \setminus \{a\}$

MA-2: Actor = Admin

Closing an action:

Headings = **getAllHeadings**(ACTIONS)

present Headings

event Select action $a \in$ ACTIONS:

Description = **getDesc**(a)

present Description

remove(a, ACTION)

log(thisActor, `Action-Closed`, [a])

MA-3: Actor = Leader

Modifying an action description:

present $\mathcal{A}$.Heading

a = **oneOf** $\{x \in \mathcal{A} :$ **thisActor** $\in x.Leaders\}$

input a.Description

MA-3: Actor = Leader

Modifying action description:

Headings = **getAllHeadings**(ACTIONS)

present Headings

event Select action $a \in$ ACTIONS:

Description = **getDesc**(a)

present Description

input a.Description

setDesc(a, a.Description)

log(thisActor, `Desc-Modified`, [a])

MA-4: Actor = Guest

Viewing action descriptions:

present $\mathcal{A}$.Description

MA-4: Actor = Guest

Viewing action descriptions:

Descriptions = **getAllDesc**(ACTIONS)

present Descriptions

TABLE 5.2: Outline of the implementation of the MA-2, MA-3, MA-4 use cases.

Management of the Volunteer Status

MVS-1: Actor = Guest

Application for volunteering:

c= newUser()

input c.Personal_data

... *Guest accepts the organization's bylaws*

$\mathcal{C}' = \mathcal{C} \cup \{c\}$

MVS-1: Actor = Guest

Application for volunteering:

input Lang

Actions= **getAllDesc**(Lang)

present Actions

event Submit application:

Bylaws= **getBylaws**(Lang)

present Bylaws

event Accept the bylaws:

input Personal_Data, Lang

u= **newUser** (Personal_Data)

assignPosition(u, CANDIDATE)

log(u, `Applic`, ...)

MVS-2: Actor = Recruiter

Accepting a candidate:

c= **oneOf** C

either $V' = V \cup \{c\}$ *# acceptance*

or $V' = V$ *# rejection*

$C' = C \setminus \{c\}$

MVS-2: Actor = Recruiter

Accepting a candidate:

Lang = **getLang**(thisActor)

C = **getAll**(CANDIDATES)

present C

event Accept candidate $c \in$ C:

assignPosition(c, VOLUNTEER)

log(thisActor, `Candid-Accept`, [c])

event Reject of $c \in$ C:

deleteVolunteer(c, CANDIDATE)

log(thisActor, `Candid-Reject`, [c])

TABLE 5.3: Outline of the implementation of the MVS-1 and MVS-2 use cases.

MVS-3: Actor = Admin

Promoting to leader:

v= oneOf V

$L' = L \cup \{v\}$

MVS-3: Actor = Admin

Promoting to leader:

Lang = **getLang**(thisActor)

V.PersonalData = **getPersData**(VOLUNTEERS)

Log_V = **getLog**(VOLUNTEERS)

present V.PersonalData, Log_V

event Selection of $v \in V$:

assignPosition(v, LEADER)

log(thisActor, `Leader-Promote`, [v])

MVS-4: Actor = Admin

Removing a leader:

l= oneOf L

$L' = L \setminus \{l\}$

MVS-4: Actor = Admin

Removing a leader:

Lang = **getLang**(thisActor)

L.PersonalData = **getPersData**(L)

Log_L = **getLog**(L)

present L.PersonalData, Log_L

event Selection of $l \in L$:

assignPosition(l, VOLUNTEER)

log(thisActor, `Leader-Degrade`, [l])

MVS-5: Actor = Admin

Removing a volunteer:

v= oneOf V

$L' = L \setminus \{v\}$ (if applicable)

$V' = V \setminus \{v\}$

MVS-5: Actor = Admin

Removing a volunteer:

Lang = **getLang**(thisActor)

V.PersonalData = **getPersData**(VOLUNTEERS)

Log_V = **getLog**(VOLUNTEERS)

present V.PersonalData, Log_V

event Selection of $v \in V$:

deleteVolunteer(v, VOLUNTEERS)

log(thisActor, `Vol-Remove`, [v])

TABLE 5.4: Outline of the implementation of the MVS-3, MVS-4 and MVS-5 use cases.

Management of Schedules

MS-1: Actor = Volunteer

Updating action participation:

present $\mathcal{A}$.Description

a = **oneOf** $\mathcal{A}$

input a.ParticipationPreferences

MS-1: Actor = Volunteer

Updating action participation:

Descriptions = **getAllDesc**(ACTIONS)

present Descriptions

event Select action $a \in$ ACTIONS:

Preferences = **getPrefs**(a)

present Preferences

input a.ParticipationPreferences:

updatePreferences(a, a.ParticipationPreferences)

log(thisActor, `PartPrefs-Updated`, [a])

MS-2: Actor = Volunteer

Declaring availability:

input Volunteer.Lim

input Volunteer.Avail

MS-2: Actor = Volunteer

Declaring availability:

Lim = **getLimitOfWeeklyHours** (thisActor)

present Lim

input Volunteer.Lim:

updateLimitOfWeeklyHours(thisActor, Volunteer.Lim)

log(thisActor, `Limit-Declared`)

Avail = **getAvailability**(thisActor)

present Avail

input Volunteer.Avail:

updateAvailability(thisActor, Volunteer.Avail)

log(thisActor, `Avail-Declared`)

MS-3: Actor = Leader

Declaring need for volunteers:

a = **oneOf** $\mathcal{A}$

input a.Beg, a.End, a.Need

MS-3: Actor = Leader

Declaring need for volunteers:

Headings = **getAllHeadings**(ACTIONS)

present Headings

event Select action $a \in$ ACTIONS:

Details = **getDetails**(a)

present Details

input a.Beg, a.End, a.Need:

updateNeed(a, a.Beg, a.End, a.Need)

log(thisActor, `Need-Declared`, [a])

TABLE 5.5: Outline of the implementation of the MS-1, MS-2, MS-3 use cases.

MS-4: Actor = Volunteer

Modifying schedule:

present Volunteer.Schedule

ActionInterval = **oneOf** Schedule

input ActionInterval

MS-4: Actor = Volunteer

Modifying schedule:

Schedule = **getVolunteerSchedule**(thisActor)

present Schedule

event Select ActionInterval from Schedule:

Details = **getDetails**(ActionInterval)

present Details

input ActionInterval:

updateSchedule(thisActor, ActionInterval)

log(thisActor, `Schedule-Modified`, [ActionInterval])

MS-5: Actor = Leader

Viewing schedule:

present $\mathcal{A}$.Schedule

MS-5: Actor = Leader

Viewing schedule:

Headings = **getAllHeadings**(ACTIONS)

present Headings

event Select action $a \in$ ACTIONS:

Schedules = **getActionsSchedules**(a)

present Schedule

TABLE 5.6: Outline of the implementation of the MS-4, MS-5 use cases.

5.2 Quality Evaluation

The functional completeness of the interface is a fundamental indicator of the degree to which the system implementation meets the project requirements. It determines whether the designed API provides sufficient capabilities to support all required operations, covering user interactions and system automation.

To assess the functional completeness of the `VoluMan` system, its implementation was evaluated by performing tests for each use case outlined in Section 2. These tests were designed to verify whether the API effectively supports all expected operations and interactions within the system.

Coverage of Use Cases

The results of the use case testing confirm that the system supports the following key functionalities:

- **Volunteer Management:** The system facilitates the entire lifecycle of volunteers, from application as candidates, through acceptance by recruiters, position assignment, and updates to personal preferences and availability.
- **Position Assignment and Transitions:** The position management module enforces strict rules for position transitions, ensuring that only valid changes occur based on predefined

constraints.

- **Action Management:** Administrators and leaders have the ability to create, modify, and close actions while ensuring that appropriate volunteers are assigned based on their declared preferences and availability.
- **Scheduling Mechanisms:** The scheduling module matches volunteers to actions based on availability and demand requirements, ensuring appropriate task distribution.
- **Security and Authentication:** The system employs authentication and role-based access control to prevent unauthorized modifications, ensuring data integrity and user-specific access restrictions.
- **Logging and Auditability:** All key operations, including role assignments, action modifications, and scheduling updates, are logged to maintain traceability and accountability within the system.

API Completeness and Consistency

The API was designed with the goal of providing a complete and structured interface for managing volunteer activities. The evaluation of functional completeness confirmed that:

- CRUD (Create, Read, Update, Delete) operations are implemented for all major entities, including volunteers, positions, actions, and schedules.
- The API ensures a clear distinction between user positions, restricting certain operations to authorized personnel (e.g., only administrators can assign positions or generate schedules).
- The responses returned by the API maintain a consistent JSON structure, ensuring that clients receive well-formed and predictable data.
- Scheduling mechanisms provide a working foundation for matching volunteers to actions.

Limitations and Future Enhancements

Despite achieving a high level of functional completeness, several areas for improvement have been identified:

- **Integration of the Document and Language Management Modules:** Although the document and language management modules were developed, they were not fully integrated into the system. Future iterations should focus on completing this integration to allow for multilingual support and document handling.
- **Optimization of Scheduling Algorithms:** The current scheduling mechanism assigns volunteers based on basic criteria. The implementation of an optimized scheduling algorithm that balances workloads, minimizes conflicts, and improves efficiency is a key area for future development.
- **Scalability Considerations:** Future iterations may include optimizations for handling a larger number of concurrent users and actions, ensuring smooth operation as the system scales.

Conclusion

The functional completeness of the `VoluMan` system has been evaluated through extensive testing of use cases, verifying that all core functionalities operate as expected. The system provides a structured API supporting volunteer management, position assignments, action handling, and scheduling. While the integration of the document and language management modules remains incomplete, the core system is functional and meets the primary project objectives. Future enhancements should focus on improving scheduling algorithms, and expanding system scalability to ensure long-term maintainability and efficiency.

Chapter 6

Conclusions

This chapter summarizes the work conducted during the development of the VoluMan system, highlights the key lessons learned, and outlines opportunities for future development. It also provides a detailed breakdown of responsibilities among the project team members.

The VoluMan system addresses the core requirements for volunteer management, including handling schedules, preferences, and actions. Through the application of modern technologies and programming principles, the system establishes a scalable and reliable backend that can serve as a foundation for further enhancements, such as the implementation of a dedicated frontend and advanced communication tools.

6.1 Lessons Learned

The development of the VoluMan system provided numerous opportunities to gain valuable technical, organizational, and collaborative experience. This section outlines the most significant lessons learned during the project.

Team Collaboration and Responsibility Distribution Working in a team of four students required clear communication, task division, and coordination. Each team member was assigned specific responsibilities, which enabled parallel development of different system components. Regular meetings and the use of project management tools facilitated effective collaboration and helped to prevent conflicts in code integration. One of the key lessons was the importance of maintaining a shared understanding of the project's goals and design principles to ensure alignment among all contributors.

Testing and Validation The project emphasized the importance of thorough testing and validation at every stage of development. Unit tests were written for critical components of the backend to ensure their correctness and reliability. Integration testing played a vital role in verifying the interaction between modules, particularly the database, API endpoints, and authentication mechanisms. Additionally, we learned that automating tests, while initially time-consuming, significantly improved efficiency in the long term and helped identify issues early in the development cycle.

Backend Development and Architecture Design The project provided a deep understanding of backend development, particularly using the Spring Boot framework. Designing a modular architecture based on the Model-View-Controller (MVC) pattern simplified debugging and ensured a clear separation of concerns. The use of Spring JPA and Hibernate demonstrated the advantages of object-relational mapping for managing database interactions. Furthermore, we gained practical experience in handling RESTful API design and implementation, which improved our skills in creating scalable and maintainable server-side systems.

Adherence to Programming Principles Throughout the project, we adhered to programming principles such as Clean Code and SOLID. This approach not only improved the readability and maintainability of the codebase but also encouraged reusability and scalability. By following these principles, we minimized technical debt and established a foundation for future development. Additionally, implementing design patterns, such as Singleton, provided practical experience in applying theoretical concepts to solve real-world problems.

Documentation and Communication of Technical Details One of the most valuable lessons was the importance of clear and comprehensive technical documentation. Writing detailed documentation not only aided in internal collaboration but also provided a resource for potential future developers.

Project Challenges and Evolving Requirements The project highlighted the challenges of managing evolving requirements and addressing unforeseen obstacles. Initial assumptions about system functionality and scope often required adjustments as new constraints or needs were identified. For instance, the integration of dynamic scheduling with real-time volunteer preferences proved more complex than anticipated. This experience underscored the importance of flexibility in both technical and organizational planning. Furthermore, time constraints necessitated prioritization of core features, teaching us how to balance technical ambition with practical deadlines.

6.2 Future Development Opportunities

The VoluMan system provides the foundation for volunteer management, but there are several opportunities for future development to expand its functionality and usability. This section outlines the most important directions for further enhancement.

1. Implementation of a Dedicated Frontend One of the key areas for future development is the creation of a dedicated frontend for the VoluMan system. Currently, the system provides only backend functionality, and a separate user interface would significantly enhance its usability. The frontend could include:

- **User Interfaces for Different Roles:** Separate interfaces for volunteers, leaders, and administrators, tailored to their specific needs. For example:
  - Volunteers could access their schedules, update availability, and apply for actions.
  - Leaders could manage schedules, assign roles, and monitor volunteer performance.
  - Administrators could oversee the system, manage user roles, and generate reports.
- **Responsive Design:** The frontend should be designed to work across devices, including desktops, tablets, and smartphones, ensuring accessibility for all users.

- **Localization Support:** A multilingual interface would allow the system to be used by organizations operating in different regions and languages.
- **Integration with Backend:** The frontend would interact with the backend API to fetch data, submit forms, and handle user actions securely and efficiently.

The implementation of a frontend is a crucial step in making the system more accessible to end-users and enabling its adoption by a broader audience.

2. Development of a Mobile Application Another important direction for future development is the creation of a mobile application. A mobile app would provide users with convenient access to their schedules, notifications, and tasks on the go. Key features of the mobile app could include:

- Real-time push notifications for updates and changes in schedules or actions.
- Offline access to schedules and other critical information.
- Quick interaction options, such as one-click confirmation of availability or participation.

The mobile app could be built using cross-platform development frameworks like Flutter or React Native, enabling deployment on both Android and iOS platforms.

3. Advanced Reporting and Analytics To provide better insights into the system's performance, an advanced reporting module could be developed. This module would allow leaders and administrators to generate detailed reports on:

- Volunteer participation rates and activity levels.
- Success metrics for individual actions or campaigns.
- Overall system performance, including areas for improvement.

Integration with external tools like Power BI or Tableau could enable more advanced data visualization and analytics.

4. Enhanced Communication and Notification System A robust communication system is essential for effective volunteer coordination. Future enhancements could include:

- **In-App Notifications:** Alerts for schedule changes, new action announcements, or role updates.
- **Email and SMS Integration:** Notifications sent directly to users' emails or phones to ensure they are informed even when not using the system.
- **Message Threads:** A built-in messaging system to facilitate communication between volunteers, leaders, and administrators.

These improvements would streamline communication and reduce the chances of miscommunication or missed updates.

6.3 Scope of Work and Distribution of Responsibility

The aim of this thesis was to achieve following tasks:

1. Implementation of the backend system to handle core functionalities, including the management of volunteers, actions, preferences, and scheduling. This involved defining interfaces, creating RESTful APIs, and implementing the necessary business logic.
2. Development of a scheduling module capable of generating timetables based on volunteer availabilities, preferences, and action demands. The module also supports modifications to schedules and resolves potential conflicts.
3. Design and implementation of a relational database schema for storing and managing data related to volunteers, actions, preferences, and schedules. The schema ensures data consistency and efficient querying.
4. Creation of administrative tools to facilitate the management of actions and volunteers, including functionalities for updating roles, assigning duties, and monitoring schedules.
5. Verification and validation of the system through testing of individual modules.
6. Compilation of documentation for system architecture, requirements, and functionalities with technical descriptions of key components.

The responsibility for completing these tasks was distributed among the authors of this thesis as follows:

- **Klaudiusz Szklarkowski** was responsible for implementing the backend logic for volunteer and action management, as well as defining the communication interfaces between system modules.
- **Łukasz Walicki** worked on the design and development of the scheduling module, focusing on algorithms for timetable generation and schedule modification.
- **Jacek Młynarczyk** contributed to the creation and optimization of the relational database schema, ensuring proper integration with the backend system.
- **Jędrzej Mikołajczyk** conducted testing and validation of system modules and contributed to the documentation of system requirements and architecture.

The work was carried out collaboratively to ensure that all components of the system were aligned with the overall objectives and implemented according to best practices.

Bibliography

[1] Eurostat. Volunteering statistics in the eu. [on-line], 2025. Accessed: 2025.

[2] GeeksforGeeks. Best practices while making rest apis in spring boot application. [on-line], 2025. Accessed: 2024.

[3] Agnieszka Gruszczyńska, Łukasz Jankowski, Maksim Likhaivanenka, and Helena Masłowska. *Voluman - System Wspomagający Zarządzanie Wolontariuszami*. 2024.

[4] Spring.io. Building rest services with spring. [on-line], 2025. Accessed: 2024.

[5] UNHCR. Ukraine refugee situation. [on-line], 2025. Accessed: 2025.

[6] UNICEF. UNICEF Poland Overview. [on-line], 2025. Accessed: 2025.

[7] United Nations Volunteers (UNV). State of the world's volunteerism report 2022. [on-line], 2022. Accessed: 2025.

Poznan University of Technology
Faculty of Computing and Telecommunication
Institute of Computing Science

Typeset using LaTeX.