Venkata Vikranth Jannatha

[Email address]

soen poe part 2

Software Engineering

Table of Contents

[Architecture Pattern 2](#_Toc150758169)

[Description of Each Table 4](#_Toc150758170)

[Software Requirements in Class Diagram 5](#_Toc150758171)

[Appendix 6](#_Toc150758172)

[Part 2 Improvements: 6](#_Toc150758173)

[References 7](#_Toc150758174)

[Figure 1 : UML Diagram 3](#_Toc148687088)

# Architecture Pattern

I’ve chosen the Model-View-Controller (MVC) based on (GeeksforGeeks, 2023). The design pattern for developing the software solution for our non-profit organization. This pattern is widely used for user interfaces, and it separates the program logic into three interconnected elements. This separates an application into three components: Model, View, and Controller. Each component has a specific role and responsibility in the application.

The Model component handles the data and business logic of the application. It defines the structure, format, and operations of the data. It also communicates with the database or other data sources to store and retrieve data. The Model component does not interact with the user directly, but only with the Controller component.

The View component handles the user interface and presentation of the application. It displays the data from the Model component in a user-friendly and interactive way. It also receives the user input and sends it to the Controller component. The View component does not contain any business logic, but only the UI logic.

The Controller component acts as a mediator between the Model and View components. It receives the user input from the View component, processes it using the business logic from the Model component, and updates the View component with the results. The Controller component coordinates the flow of data and actions between the Model and View components.

I believe MVC is a great fit for our case study due to several reasons:

* **Separation of Concerns**: MVC provides a clear separation between domain logic (Model) and user interface (View). This separation allows us to manage complex applications by focusing on one aspect at a time. For instance, we can work on improving our user interface without worrying about the underlying business logic.
* **Easiness of Change**: The separation in MVC ensures that changes made in one component do not affect others. This makes it easy to make changes in our application without affecting other parts of our code.
* **Simultaneous Development**: With MVC, multiple developers can work simultaneously on different parts of the code such as model, view, and controller without waiting for other parts to be finished.
* **High Cohesion**: Each component of MVC has its own responsibilities which means there’s high cohesion within components.
* **Low Coupling**: The components can be tested independently because of low coupling between Model, View, and Controller.

In our case, we could have models for trees, supporters, volunteers, and donations. The views would be responsible for displaying this data in a user-friendly way, while the controllers would handle communication between models and views. This would allow us to effectively track tree-planting efforts, improve communication with supporters and volunteers, manage donations efficiently, and provide adequate information on our organization’s activities and impact.

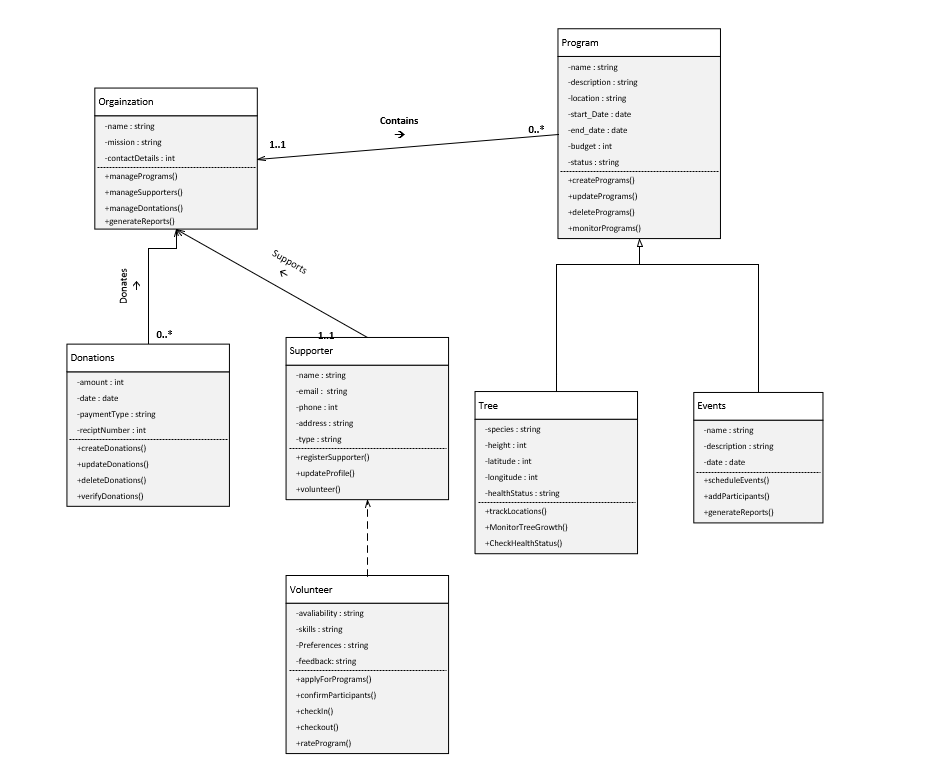


Figure : UML Diagram

# Description of Each Table

The diagram shows the main classes and their relationships, attributes, and operations. The classes are:

* **Organization**: This class represents the non-profit organization and its basic information, such as name, mission, contact details, and website. It has operations to manage its programs, supporters, donations, and reports.
* **Program**: This class represents a tree-planting program that the organization runs. It has attributes such as name, description, location, start date, end date, budget, and status. It has operations to create, update, delete, and monitor a program.
* **Tree**: This class represents a tree that is planted as part of a program. It has attributes such as species, height, diameter, GPS coordinates, and health status. It has operations to update and track its growth and condition.
* **Supporter**: This class represents a person or entity that supports the organization in some way. It has attributes such as name, email, phone number, address, and type (individual or corporate). It has operations to register, update profile, donate, volunteer, and view programs.
* **Event**: It represents an event that the organization organizes to raise awareness and funds for its mission. The attributes are name, description, and date. The methods could include schedule Event(), add Participants(), and generate Report().
* **Donation**: This class represents a monetary contribution made by a supporter to the organization. It has attributes such as amount, date, payment method, and receipt number. It has operations to create, update, delete, and verify a donation.
* **Volunteer**: This class represents a supporter who volunteers to participate in a program. It is a subclass of Supporter and inherits its attributes and operations. It also has additional attributes such as availability, skills, preferences, and feedback. It has additional operations to apply for a program, confirm participation, check-in, and check-out of a program site, and rate a program.

The diagram also shows the associations between the classes and their multiplicities. For example:

* An organization can have zero or more Programs (0…\*), but a Program must belong to one and only one Organization (1).
* A Program can have zero or more Trees (0…\*), but a Tree must belong to one and only one Program (1).
* A Supporter can make zero or more Donations (0…\*), but a Donation must be made by one and only one Supporter (1).
* A Supporter can be a Volunteer for zero or more Programs (0…\*), but a Volunteer must be a Supporter (1).

# Software Requirements in Class Diagram

The diagram also indicates which requirements from the software requirements specification are satisfied by each class or association.

* The Organization class satisfies the requirement [Website Management System]: The website shall provide information about the organization’s mission, vision, values, contact details, and website.
* The Program class satisfies the requirement [Tree-Establishing Management System]: The website shall allow the organization to create, update, delete, and monitor tree-planting programs.
* The Tree class satisfies the requirement [Tree-Establishing Management System]: The website shall allow the organization to track and manage the trees planted in each program.
* The Supporter class satisfies the requirement [Customer Relationship Management (CRM) System:]: The website shall allow supporters to register and update their profiles.
* The Donation class satisfies the requirement [Donation Management System}: The website shall allow supporters to make donations to the organization using various payment methods.
* The Volunteer class satisfies the requirement [Volunteer Management System]: The website shall allow supporters to volunteer for tree-planting programs.
* The association between Organization and Program satisfies the requirement [Website Management System]: The website shall display the current and past programs of the organization.
* The association between Program and Tree satisfies the requirement [Tree-Establishing Management System]: The website shall display the details of each tree planted in a program.
* The association between Supporter and Donation satisfies the requirement [Financial Reporting System]: The website shall display the donation history of each supporter.
* The association between Supporter and Program satisfies the requirement [Customer Relationship Management (CRM) System]: The website shall display the programs that each supporter has volunteered for.

# Appendix

## Part 2 Improvements:

In the second part of my project, I selected the Model-View-Controller (MVC) pattern based on feedback. However, my initial explanation of the MVC pattern was not comprehensive enough, as I only provided a superficial overview without exploring into the core aspects of the MVC pattern. This resulted in a loss of marks.

To rectify this, I have now provided an in-depth explanation of the components that constitute the MVC pattern. I have elaborated on the roles and interactions of the Model, View, and Controller components, and how they contribute to the overall functionality of the application.

In the class diagram, I previously overlooked the inclusion of communication and inheritance. I had failed to specify the verb relationships between tables and did not implement inheritance where it was required. To address these issues, I have now explained the relationships between linked tables, enhancing the clarity of communication in the diagram.

For the implementation of inheritance, I have created a relationship between the ‘Trees’ and ‘Events’ classes, linking them to the ‘Program’ class. This addition demonstrates the hierarchical relationship and shared attributes between these classes, effectively implementing inheritance in the class diagram.



# References

GeeksforGeeks, 2023. *MVC design pattern.* [Online]   
Available at: https://www.geeksforgeeks.org/mvc-design-pattern/  
[Accessed 20 October 2023].