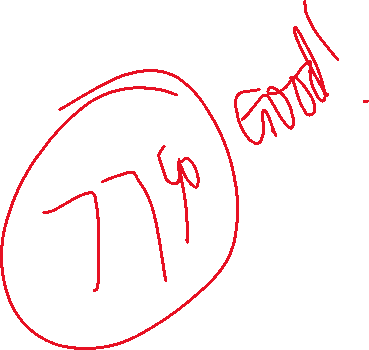
Venkata Vikranth Jannatha

[Email address]

soen poe part 2

Software Engineering



[Architecture Pattern 2](#_Toc148685496)

[Description of Each Table 3](#_Toc148685497)

[Software Requirements in Class Diagram 4](#_Toc148685498)

[References 5](#_Toc148685499)

[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 separation allows us to manage internal representations of information and how this information is presented to and accepted from the user.

The Model component is where all the data-related logic resides. This could be the data being transferred between the View and Controller components or any other data related to our business logic.

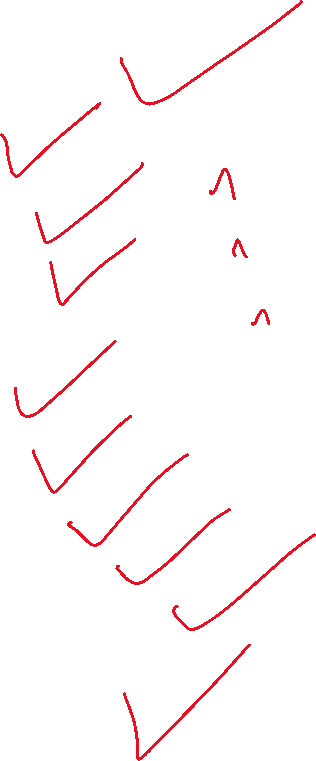


The View component handles all the UI logic of our application. For instance, this is where we create and render views for our supporters.

The Controller acts as an interface between Model and View components. It processes all the business logic and incoming requests, manipulates the data using the Model component, and interacts with the Views to render the final output.

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.

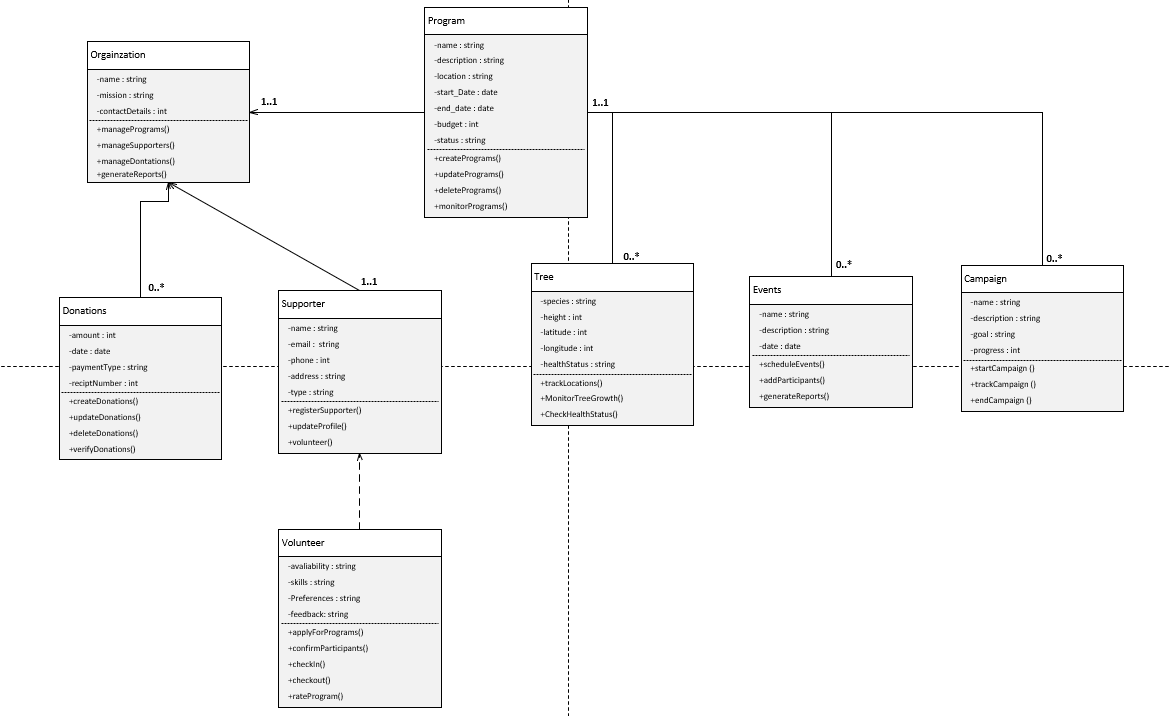




Figure 1 : UML Diagram



# 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.

# References

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