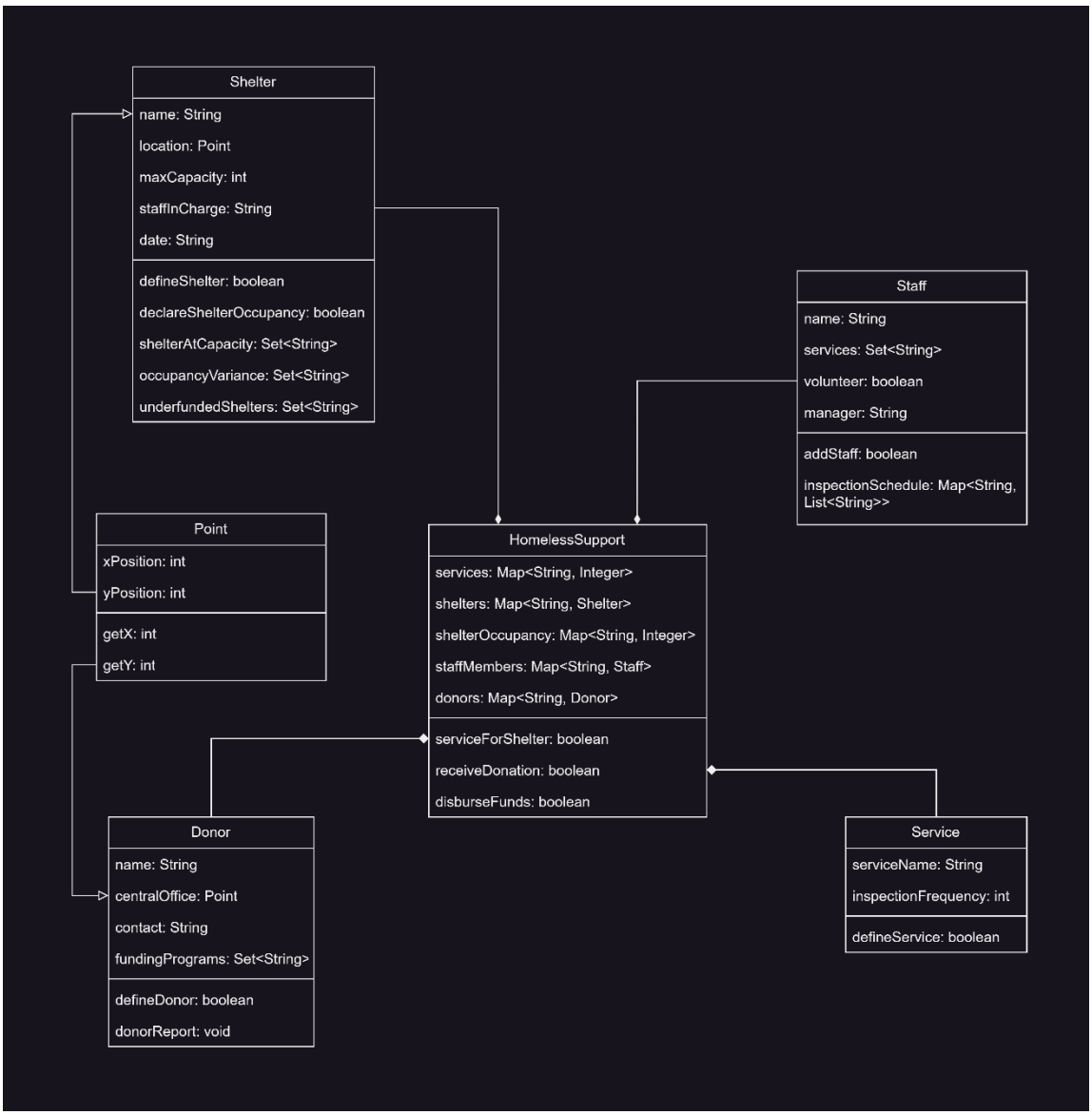
CSCI 3901 PROJECT

Fall 2023

Homeless Support

Class Design and UML

Initial design:
I adopted an iterative development approach for creating my UML class diagram, employing the CRC (Class Responsibilities Collaborators) methodology. The initial design identified six classes: Shelter, Staff, HomelessSupport, Donor, Service, and Point, utilizing a noun-verb approach for class and attribute identification. Here's an overview of the design:



Shelter: Represents a homeless shelter with attributes such as name, location, maximum capacity, staff in charge, and date. Methods include defining shelter and its occupancy, identifying shelters at capacity, and those in need of assistance.

Staff: Represents a staff member with attributes like name, services provided, volunteer status, and manager details. Methods include adding staff and scheduling inspections for the staff.

HomelessSupport: Manages support services for homeless individuals. Includes maps for services and shelters, occupancy details, a list of staff members, and donors. Methods cover service provision for shelters, donor contributions, and fund disbursement.

Donor: Represents a donor with attributes like name, location, contact info, and funding programs. Methods include defining donors and generating reports.

Service: Represents a service provided by the shelter with attributes such as service name and inspection frequency. Includes a method to add a service to the system.

Point: Defines location details with xPosition and yPosition attributes, along with a method to retrieve location coordinates.

Relationships:

1. A Shelter provides Services.
2. A Staff member inspects a Service.
3. A Donor donates to a Shelter.
4. A Donor and a Shelter has a location.

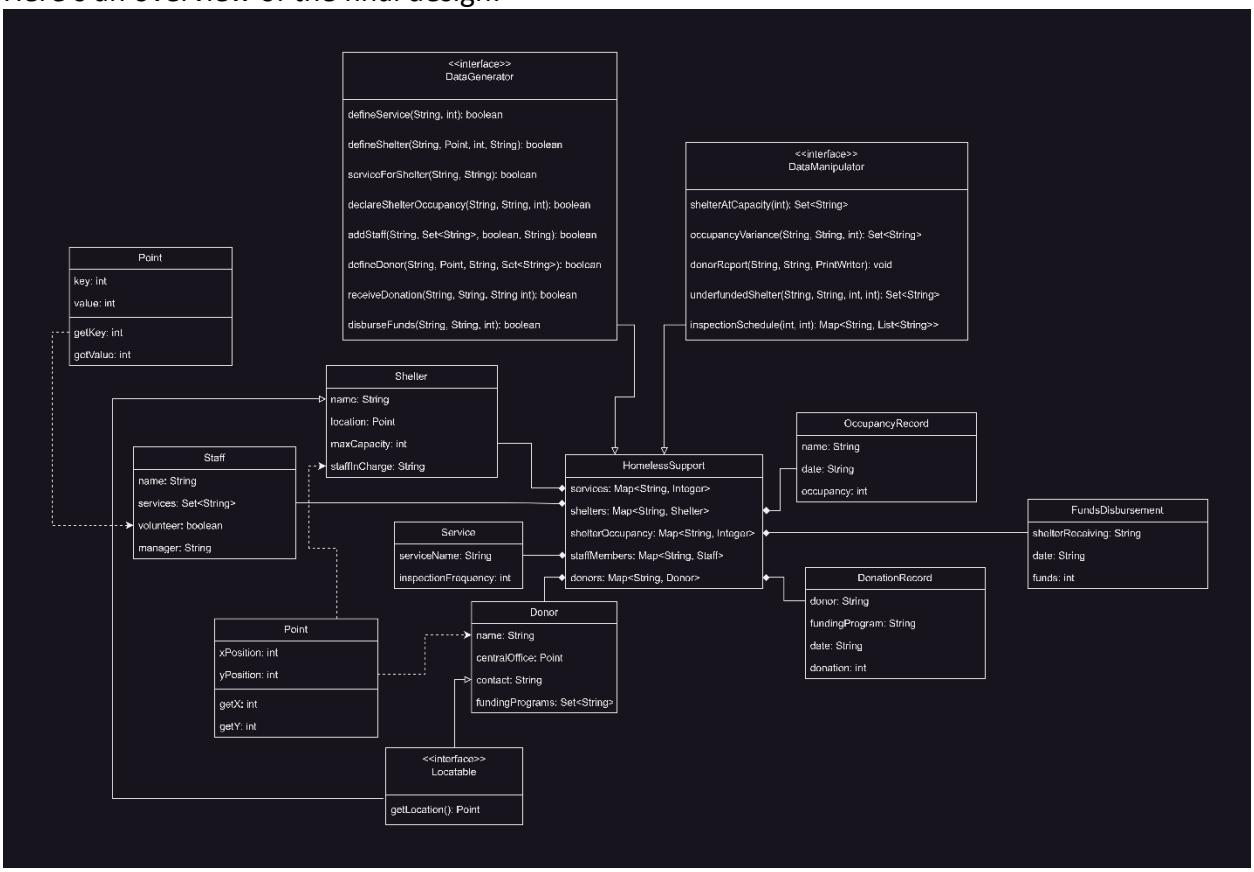
Potential Issues with this design:

1. **Single Responsibility Principle (SRP)**: The HomelessSupport class seems to be responsible for managing services, shelters, staff members, and donors. This could be a violation of the SRP, as this class has more than one responsibility.
2. **Open-Closed Principle (OCP)**: In case of changes, the current design might require modification of the existing classes, which could violate the OCP. It might be better to design the system in a way that it can be extended without modifying the existing code.
3. **Liskov Substitution Principle (LSP)**: There doesn't seem to be any inheritance in the current design, so the LSP doesn't directly apply. However, if inheritance is introduced in the future, care should be taken to ensure that subclasses can substitute their base classes without affecting the correctness of the program.
4. **Interface Segregation Principle (ISP)**: The current design doesn't seem to use interfaces, so the ISP doesn't directly apply. However, if interfaces are introduced in the future, they should be client-specific rather than general to prevent the clients from being forced to depend on interfaces they don't use.
5. **Dependency Inversion Principle (DIP)**: The current design seems to have high-level modules (HomelessShelter) depending on low-level modules (Shelter, Service etc.). This could be a violation of the DIP. It might be better to depend on abstractions (interface)

rather than a concrete class implementation.

Final Design:
Once I had the initial design, I decided to apply SOLID principles wherever needed in the design. Beginning with SRP, I looked at my design and found places where I could use this principle. After making these changes, I checked if they caused any big problems (ripple effects on other classes). If they did, I thought about whether it was worth the trade-off. For example, if it made different parts of the design more connected (cohesion), I asked myself if it was still a good change. If it was, I kept the change and moved on to the next step.
I repeated this process for each of the other SOLID principles. It helped me make sure that each part of my plan did one thing well, and that they all worked together nicely.

Here's an overview of the final design:



Analyzing the final design:

Single Responsibility Principle (SRP):
Donor, Shelter, Service, Staff, ReceiveDonationRecord, DisburseFundRecord, ShelterOccupancyRecord: Each class is responsible for managing its own attributes and behavior related to its specific domain. For example, Donor manages donor-related information, **Shelter** manages shelter-related information, and so on.

Open/Closed Principle (OCP):
Donor, Shelter, Service, Staff: Easily extendable by adding new attributes or methods without modifying existing code. Example: If a new attribute needs to be added to Donor, you can do so without changing existing code.

Liskov Substitution Principle (LSP):
Locatable: For instance, different implementations of Locatable (like Shelter and Donor) can be used interchangeably.

Interface Segregation Principle (ISP):
DataGenerator, DataManipulator, Locatable: Each interface has client specific requirements, and nothing is generic in any of them.

Dependency Inversion Principle (DIP):
DatabaseManager, DataGenerator, DataManipulator: We can have multiple implementations of DatabaseConnection based on the specific database being used (e.g., MySqlDatabaseConnection, PostgresDatabaseConnection). Each of these implementations adheres to the same DatabaseConnection, allowing them to be easily swapped in and out of the DatabaseManager without affecting its core functionality.
The design also follows the Dependency Inversion Principle by depending on abstractions (interfaces) rather than concrete implementations. For instance, HomelessSupport depends on abstractions (interfaces) such as DataGenerator and DataManipulator instead of concrete implementations, promoting flexibility.

Additional Considerations:

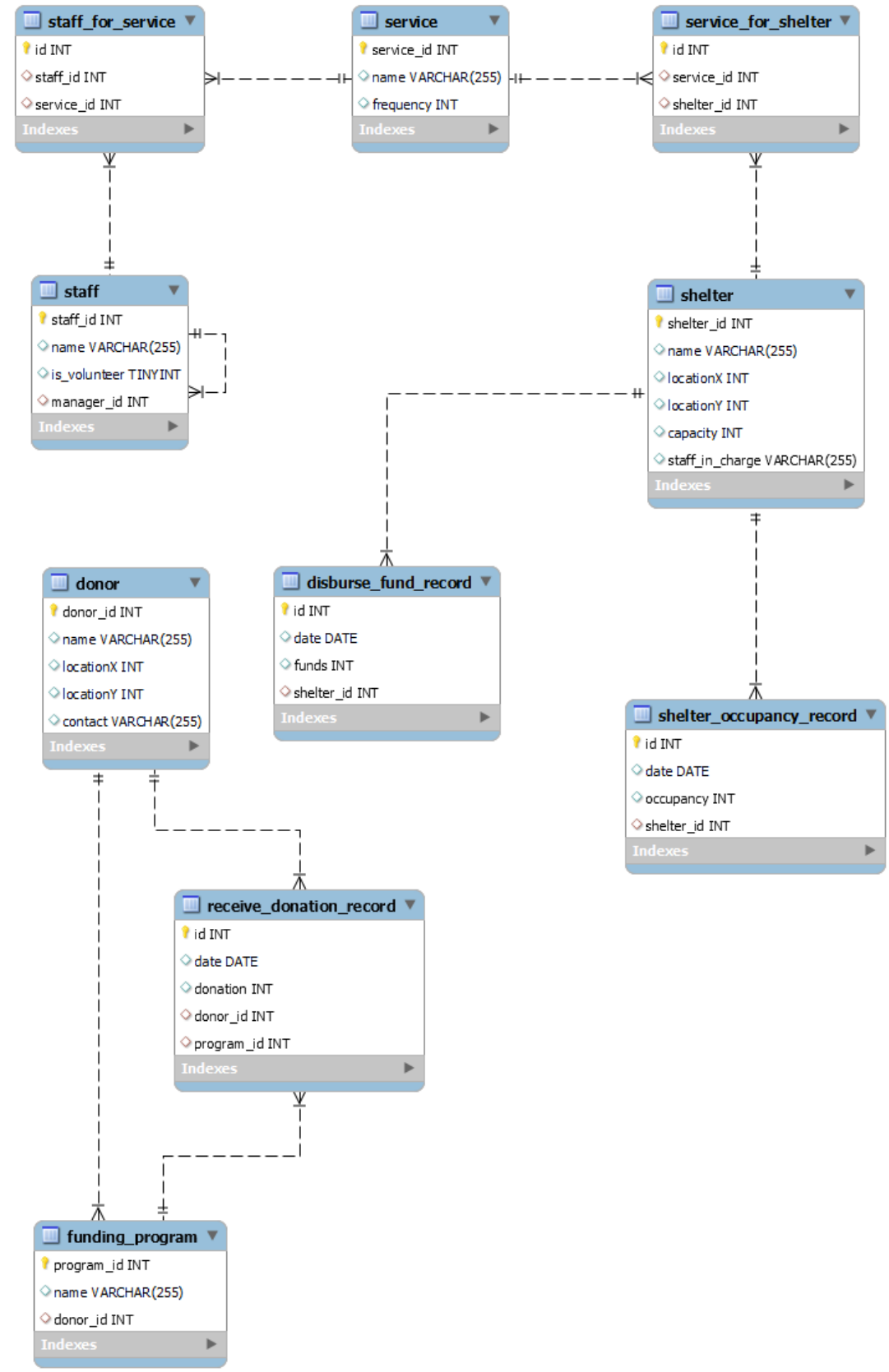
- **Consistency:** Classes follow a consistent naming convention, making the codebase more readable and maintainable.
- **Quality:** Proper encapsulation and information hiding ensure data integrity and reduce the risk of unintended side effects. Relationships between classes are clear, promoting a well-organized design.

Database Design and ERD

You can find the SQL Script for the Database at the following location in directory:

Documentation and Files -> Database Design and ERD -> SQL.sql

ERD:


staff_for_service
id INT
staff_id INT
service_id INT
Indexes
service
service_id INT
name VARCHAR(255)
frequency INT
Indexes
service_for_shelter
id INT
service_id INT
shelter_id INT
Indexes
staff
staff_id INT
name VARCHAR(255)
is_volunteer TINYINT
manager_id INT
Indexes
shelter
shelter_id INT
name VARCHAR(255)
locationX INT
locationY INT
capacity INT
staff_in_charge VARCHAR(255)
Indexes
donor
donor_id INT
name VARCHAR(255)
locationX INT
locationY INT
contact VARCHAR(255)
Indexes
disburse_fund_record
id INT
date DATE
funds INT
shelter_id INT
Indexes
shelter_occupancy_record
id INT
date DATE
occupancy INT
shelter_id INT
Indexes
receive_donation_record
id INT
date DATE
donation INT
donor_id INT
program_id INT
Indexes
funding_program
program_id INT
name VARCHAR(255)
donor_id INT
Indexes

Tables:

1. **Donor Table (donor):**
    - Represents individuals or entities contributing funds.
    - Attributes: **donor_id** (primary key), **name**, **locationX**, **locationY**, **contact**.
    - Ensures uniqueness of donors and their funding programs.
2. **Funding Program Table (funding_program):**
    - Tracks various programs through which donors contribute funds.
    - Attributes: **program_id** (primary key), **name**, **donor_id** (foreign key).
    - Enforces unique combinations of donor and program.
3. **Receive Donation Record Table (receive_donation_record):**
    - Records the donations received from donors.
    - Attributes: **id** (primary key), **date**, **donation**, **donor_id** (foreign key), **program_id** (foreign key).
    - Maintains referential integrity with donor and funding program.
4. **Service Table (service):**
    - Describes services provided at shelters.
    - Attributes: **service_id** (primary key), **name**, **frequency**.
    - Ensures non-negativity of service frequency.
5. **Shelter Table (shelter):**
    - Represents shelters for the homeless.
    - Attributes: **shelter_id** (primary key), **name**, **locationX**, **locationY**, **capacity**, **staff_in_charge**.
    - Enforces positive capacity values.
6. **Service for Shelter Table (service_for_shelter):**
    - Links services with the shelters offering them.
    - Attributes: **id** (primary key), **service_id** (foreign key), **shelter_id** (foreign key).
    - Guarantees the uniqueness of service-shelter associations.
7. **Staff Table (staff):**
    - Captures information about staff members.
    - Attributes: **staff_id** (primary key), **name**, **is_volunteer**, **manager_id** (foreign key).
    - Ensures staff members are associated with a manager.
8. **Staff for Service Table (staff_for_service):**
    - Connects staff members with the services they can handle.
    - Attributes: **id** (primary key), **staff_id** (foreign key), **service_id** (foreign key).
    - Maintains unique staff-service associations.
9. **Shelter Occupancy Record Table (shelter_occupancy_record):**
    - Keeps track of shelter occupancies on specific dates.
    - Attributes: **id** (primary key), **date**, **occupancy**, **shelter_id** (foreign key).
    - Ensures unique records for each shelter on a given date.
10. **Disburse Fund Record Table (disburse_fund_record):**
    - Records the disbursement of funds to shelters.
    - Attributes: **id** (primary key), **date**, **funds**, **shelter_id** (foreign key).
    - Maintains referential integrity with shelters.

Reasons for some of the Data modelling in the design:

1. **Service and Shelter Relationship:** Since shelter and service have many to many relationships, it's better to have an intermediate table service_for_shelter with foreign keys of both the tables.
2. **Staff and Service Relationship:** Since staff and service have many to many relationships, it's better to have an intermediate table staff_for_service with foreign keys of both the tables.
3. **Location Representation:** Instead of creating a separate table for location attribute, it's better to have 2 columns for it (locationX and locationY) as we know that the location will be represented in 2 values only and it's not going to change in future.

Noteworthy Points:

- **Boolean Representation:** The **is_volunteer** attribute in the **staff** table is represented as a boolean (TINYINT(1)) in MySQL.

Test Cases / Plans

Gathering data

defineService

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null Check	serviceName is null	Return false
	Empty Check	serviceName is empty string	Return false
Boundary Case	Min Value	inspectionFrequency is less than 0	Return false
Control Flow	Second call to same method	Calling defineService again with same name but different frequency	Return true and update the frequency.

defineShelter

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null Check	name is null	Return false
	Empty Check	name is empty string	Return false

	Null Check	location is null	Return false
	Null Check	satffInCharge is null	Return false
	Empty Check	staffInCharge is empty	Return false
Boundary case	Min Value	maxCapacity is less than or equals to 0	Return false
	Negative Value	Passing negative UTM values	Return true
Control flow	staffInCharge not found	staffInCharge is not in the system/database	Return false
	Second call to same method	Calling defineShelter again with same name but different location, maxCapacity and staffInCharge (staff in the system)	Return true and update the information.

serviceForShelter

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	shelterName is null	Return false
	Empty check	shelterName is empty string	Return false
	Null Check	serviceName is null	Return false
	Empty Check	serviceName is empty	Return false
Control flow	serviceName is already registered with a shelterName	Passing same serviceName but different shelterName	Return SQLException
	shelterName is already registered with a serviceName	Passing same shelterName but different serviceName	Return SQLException
	Duplication	Calling method with same	Return SQLException

		serviceName and shelterName	
Data flow	serviceName not found	Calling serviceForShelter before defineService (serviceName not in the system/database)	Return false
	shelterName not found	Calling serviceForShelter before defineShelter (shelterName not in the system/database)	Return false

declareShelterOccupancy

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	name is null	Return false
	Empty check	name is empty string	Return false
	Null Check	date is null	Return false
	Empty Check	date is empty	Return false
	Bad Format	date not in “yyyy-MM-dd” format	Return false
Boundary case	Min Value	Occupancy is less than 0	Return false
	Max Value	Occupancy greater than the maxCapacity of that shelter	Return false
Control flow	Date change	Same shelterName and occupancy but different date	Return true
	Duplication	Calling method with same parameters twice (duplicate entry for same shelter on same day)	Return SQLException
	Non-Leap year count exceed	Adding a record for a shelter/camp more than 365 times for a single	Return false

		non leap year	
Data flow	shelterName not found	Calling declareShelterOccupancy Before defineShelter (shelterName is not in system/database.)	Return false

addStaff

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	name is null	Return false
	Empty check	name is empty string	Return false
	Null Check	services is null	Return false
	Empty Check	services is empty	Return false
	Null check	manager is null	Return false
	Empty check	manager is empty	Return false
Control flow	Duplication	Passing same name but different services, volunteer status and manager	Return true and update the information
Data flow	services not found	Calling addStaff before defineService (services are not in the system/database)	Return false
	manager not found	Calling addStaff before adding a manager (Manager is not in the system/database)	Return false

defineDonor

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	name is null	Return false

	Empty check	name is empty string	Return false
	Null Check	centralOffice is null	Return false
	Null Check	contact is null	Return false
	Empty check	contact is empty	Return false
	Null Check	fundingPrograms is null	Return false
	Empty Check	fundingPrograms is empty	Return false
Boundary case	Max Value	centralOffice with maximum UTM coordinates	Return true
	Min Value	centralOffice with -ve (minimum) UTM coordinates	Return true
Control flow	Duplication	Passing same name but different centralLocation, contact and fundingPrograms	Return true and update information

receiveDonation

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	donor is null	Return false
	Empty check	donor is empty string	Return false
	Null Check	fundingProgram is null	Return false
	Empty Check	fundingProgram is empty	Return false
	Null Check	date is null	Return false
	Empty check	date is empty	Return false
	Bad Format	date not in “yyyy-MM-dd” format	Return false
Boundary case	Min Value	donation is less than or equal to 0	Return false

Control flow	No association	The donor and programs are not associated with each other	Return false
	Duplication	Calling function with same values again	Return true (A donor can contribute to the same program twice)
Data flow	Donor Not Found	Calling receiveDonation before defineDonor (donor is not in the system/database)	Return false

disburseFunds

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	shelterReceiving is null	Return false
	Empty check	shelterReceiving is empty string	Return false
	Null Check	date is null	Return false
	Empty check	date is empty	Return false
	Bad Format	date not in “yyyy-MM-dd” format	Return false
Boundary case	Min Value	funds is less than or equal to 0	Return false
Control flow	No association	The donor and programs are not associated with each other	Return false
	Duplication	Calling function with same values again	Return true (a shelter can receive through same program twice)
Data flow	Shelter Not Found	Calling disburseFunds before defineShelter (shelter is not in	Return false

		the system/database)	

Doing something with the data

shelterAtCapacity

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Boundary case	Min Value	Threshold less than 0	Return null
	Max Value	Threshold is more than 100	Return null
Control flow	Exact Value	Shelters operating exactly at the threshold	Return Set of those shelters and shelters with threshold more than that
	Above	shelters operating above the threshold	Return Set of shelters working at threshold above the given value
	Empty	shelters operating below the threshold	Return Empty Set
Data flow	No shelters	Calling shelterAtCapacity before calling defineShelter (before adding any shelters to the system/database)	Return Empty Set

occupancyVariance

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	startDate is null	Return null
	Empty Check	startDate is empty	Return null
	Bad Format	startDate not in “yyyy-MM-dd” format	Return null
	Null check	endDate is null	Return null
	Empty Check	endDate is empty	Return null
	Bad Format	endDate not in	Return null

		“yyyy-MM-dd” format	
Boundary case	Min Value	Threshold less than 0	Return null
	Max Value	Threshold is more than 100	Return null
Control flow	Exact Value	Shelters having a variance equal to the threshold	Return Set of those shelters and shelters with threshold more than that
	Above	shelters having a variance above the threshold	Return Set of the shelter which have a variance more than threshold
	Empty	shelters with no variance (occupancy constant throughout the date range)	Return Empty Set (unless passing threshold = 0)
	Same start and end date	Passing same start and end date in date range and threshold >= 1	Return Empty Set (the variance on same day will always be 0)
Data flow	No shelters	Calling occupancyVariance before calling defineShelter (before adding any shelters to the system/database)	Return Empty Set

donorReport

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	startDate is null	Return void
	Empty Check	startDate is empty	Return void
	Bad Format	startDate not in “yyyy-MM-dd” format	Return void
	Null check	endDate is null	Return void
	Empty Check	endDate is empty	Return void

	Bad Format	endDate not in “yyyy-MM-dd” format	Return void
	Null check	outstream is null	Return void
Control flow	Multiple donations to same program	donors who donated multiple times to same program in the given time range.	Print the donation amount as the summation of all donations towards that program by that donor
Data flow	No Donors	Calling donorReport before calling defineDonor (before adding any donors or programs to the system/database)	Print empty

underfundedShelter

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Input validation	Null check	startDate is null	Return null
	Empty check	startDate is empty string	Return null
	Null Check	endDate is null	Return null
	Empty Check	endDate is empty	Return null
Boundary case	Min Value	Distance is less than 0	Return null
	Min Value	Threshold is less than 0	Return null
Control flow	No Data	No shelters received fundings between the given date range	Return null
	Within the distance	Shelters that have funding needs and are within the specified distance.	Return Set of the shelters who have the lowest per-occupant funding
	Proportional Funding Distribution	Three shelters within the specified distance, having	Return Set of top two shelters which have lowest per-occupant funding

		capacities (100, 50, 75) and all having funding needs and threshold = 2.	
	Nearest Shelters	Donor with no shelters within the specified distance range	Return Set of the nearest shelters with lowest per-occupant funding
	Equal Funding needs	Multiple shelters within the specified distance, all having equal funding needs.	Return Set of shelters which received the lowest disburse-funds during the same period (date-range)
Data flow	No Shelters	Calling underfundedShelters before calling defineShelter (No shelters are in the system/db)	Return null
	No Donors	Calling underfundedShelters before calling defineDonors (No donors are in the system/db)	Return null

inspectionSchedule

Type of the test case	**Property of the test case**	**Property of the input data**	**Expected result**
Boundary case	Min Value	scheduleDays is less than 0	Return null
	Min Value	inspectLimit is less than 0	Return null
Control flow	Single Entities	Single staff inspects single service at a single shelter	Return Map of that staff name along with that shelter-service pair
	Multiple Staff Members, Multiple Services, Multiple Shelters	Multiple staff members, each inspecting multiple services each day for the specified scheduleDays and at shelters which	Return Map of Staff and the respective shelter-service pair ensuring that the scheduleDays and inspectLimit constraints are met.

		offer those services.	
	Inspect limit exceed	Services which exceed the inspectLimit. One staff member who can inspect more services in one day than the inspectLimit allows.	Return Map of Staff which can inspect those services for mentioned scheduleDays and scheduling the services of later days to earlier days if they exceed the inspectLimit of the day. The excess inspections are shifted to earlier days in the schedule.
	Services with Different Inspection Frequencies	Services that have different inspection frequencies (some daily, every 2 days, etc.)	Return Map of staff which inspects those services and ensuring the scheduling accommodates these different frequencies
	Services Requiring Periodic Inspection	Services that require periodic inspection	Return Map of staff which Ensures that there is at most one staff member who can inspect each service
Data flow	No Shelters	Calling inspectionSchedule before calling defineShelter (No shelters are in the system/dB)	Return null
	No Services	Calling inspectionSchedule before calling defineService (No shelters are in the system/dB)	Return null
	No Services are associated with any shelters in the system	Calling inspectionSchedule before calling serviceForShelter	Return null
	No Staff	Calling inspectionSchedule before calling addStaff (No staff in are in the system/dB)	Return null

External Documentation

Overview

The **Homeless Support System** is a comprehensive and compassionate initiative designed to streamline and enhance the support mechanisms for the homeless population. The primary objective of this project is to create an efficient, technology-driven system that facilitates collaboration among donors, shelters, and staff members to provide effective aid to those in need.

The Homeless Support System is designed to tackle homelessness by looking at shelter occupancy, funding, and inspections. We aim to understand which shelters have the most variable usage and which ones are almost full.

We also want to identify shelters that need more funding based on their capacity. Additionally, we're checking how much each donor contributes over time and creating inspection schedules for staff members. The goal is to build a responsive and data-driven system to support those facing homelessness.

Files and external data

All in all, there are 14 classes and 3 Interfaces in the project (class diagram above):

1. Constants.java: The "Constants" class defines integer constants for "NOT_FOUND" values and a DateTimeFormatter for date formatting.
2. DatabaseManager.java: This class provides methods for managing database connections. The use of static in the DatabaseManager class aligns with the utility class pattern, providing global accessibility for managing database connections.
3. DonationRecord.java: DonationRecord class represents a record of a donation, capturing details such as donor information, funding program, donation date, and amount. It provides a method to receive and record donations in the system.
4. Donor.java: Donor class represents a donor entity with information such as name, location, contact details, funding programs, and donation amounts. It implements the Locatable interface to provide location-related functionality.
5. FundsDisbursement.java: FundsDisbursement class represents a record of funds being disbursed to a shelter, capturing details such as the shelter receiving the funds, disbursement date, and the amount of funds. It provides a method to disburse funds and record the transaction in the system.
6. HelperMethod.java: HelperMethod class provides various helper methods for checking the existence of entities in the system, date validation, identifying non-duplicate elements, and leap year determination.
7. HomelessSupport.java: The class HomelessSupport implements DataGenerator and DataManipulator interfaces to manage various aspects of shelters, camps, staff, donors, donations, and reports.
8. Main.java: Contains the main method and basically holds the driver code. Handles the

running of each method in the system.

9. OccupancyRecord.java: The OccupancyRecord class represents the occupancy record of a shelter or camp on a specific date. It provides methods to declare and validate shelter occupancy.
10. Pair.java: The Pair class represents a simple key-value pair.
11. Point.java: Capture a two-dimensional (x, y) point, with integer coordinates.
12. Service.java: Represents a service that can be available at a shelter or a camp.
13. Shelter.java: Represents a shelter or camp, providing methods to define, retrieve information, and analyze data related to shelters.
14. Staff.java: Represents staff members in the system with information such as name, services, volunteer status, and manager.
15. DataGenerator.java: Methods to add data in the system.
16. DataManipulator.java: Methods to do something with that data in the system.
17. Locatable.java: An interface representing entities with a location.

Data structures and their relations to each other

In the design of this system, data is predominantly stored in the database, with a primary focus on leveraging database capabilities. However, certain data structures are strategically employed for temporary storage and data transformation within the system.

Shelter.java:

1. **Shelter at Capacity (Set<String>):**
    - Used to identify shelters at capacity and manage the corresponding data subset.
2. **Occupancy Variance (Set<String>):**
    - Facilitates the identification and handling of shelters with occupancy variations.
3. **Underfunded Shelters (List<Shelter>):**
    - Donor-Shelter Mapping (Map<Donor, List<Shelter>>): Establishes a mapping between donors and the list of shelters they have contributed to within a specified date range.
    - Shelter Calculated Donation Map (Map<Shelter, Integer>): Stores calculated donations for each shelter based on donor contributions, aiding in the determination of underfunded shelters.
    - Nearest Shelters (List<Shelter>): Represents shelters in proximity, assisting in calculations related to donor contributions.
    - Sorted Underfunded Shelters (List<Shelter>): Utilized for sorting and managing shelters based on their calculated donation amounts.

Staff.java (Schedule Inspection):

1. **Staff-Service-Shelter Map (Map<Pair<String, String>, Service>):**
    - Plays a central role in generating inspection schedules, capturing relationships between staff, services, and shelters.
2. **Schedule List (List<String>):**

- Temporarily stores inspection schedules for individual staff members during the scheduling process.

3. **Earlier Pairs (List<String>):**
    - Used in the dynamic process of shifting inspection pairs to earlier days within the inspection schedule.
4. **Elements to Append (List<String>):**
    - Represents pairs designated for appending during the shifting of inspection pairs in the schedule.
5. **Current Pairs (String[]):**
    - Holds pairs scheduled for the current day, crucial for various operations related to the inspection schedule.

Assumptions

1. Suppose we're creating shelters or camps; we assume their capacity is more than zero because it doesn't make sense to establish spaces that can't accommodate homeless individuals.
2. We assume that the occupancy of a shelter or camp on a specific day can be zero. There might be days when no one stays there for various reasons.
3. We assume that a donor can make multiple contributions to the same program on a given day. The system should not restrict donors from making several donations.
4. We assume that a donor's donation amount on a given day is greater than zero. Recording a donation amount of zero doesn't provide any meaningful information.
5. We assume that a shelter or camp can receive multiple donations on a given day.
6. We assume that a shelter or camp would receive a donation greater than zero on a given day. Recording a donation amount of zero doesn't provide any meaningful information.

Choices

1. If something goes wrong in the “doing something with the data methods”, return null and not empty set. This indicates that an error has happened whereas an empty data structure would be counted as a result and not error.
2. A donor must have a non-null and non-empty contact. From a business perspective, ensuring a donor has valid contact information is essential for potential future communication.
3. When multiple underfunded shelters share the same per-occupant funding, the system will analyze their disbursement records and return the shelters with the lowest disbursement amounts. This approach ensures an equitable distribution of funds, directing financial support to the shelters that have historically received fewer funds for the same per-occupant funding, thereby promoting a fair allocation of resources.

Key algorithms and design elements

Algorithm for underfundedShelter Method:

Inputs:

- **startDate**: Start date for donation records.
- **endDate**: End date for donation records.
- **distance**: Maximum distance for considering a shelter within range.
- **threshold**: Threshold for selecting underfunded shelters.

Outputs:

- A set of shelter names meeting the specified criteria.

Steps:

1. **Fetch Donors and Shelters:** Query the database to get a list of donors and shelters within the specified date range (**getAllDonors** and **getAllShelters** methods).
2. **Create Data Structures:** Create data structures to store donor-shelter mappings and calculated funds for each shelter.
3. **Check for Empty Data:** If there are no donors or shelters in the system, return **null**.
4. **Map Donors to Shelters:** For each donor, find shelters within the specified distance and add them to the **donorShelterMap**.
5. **Calculate Funds for Each Shelter:** Iterate through the donor-shelter mappings and calculate the funds for each shelter based on its capacity.
6. **Sort Underfunded Shelters:** Sort the shelters based on the ratio of calculated donation to max capacity.
7. **Select Shelters based on Threshold:** Add shelter names to the result set up to the specified threshold.
8. **Return Result:** Return the set of shelter names meeting the specified criteria.

Algorithm for inspectionSchedule Method:

Inputs:

- **scheduleDays: The number of days to schedule inspections.**
- **inspectLimit: The maximum number of inspections per day**

Outputs:

- A map containing staff names and their corresponding inspection schedules represented as lists of strings.

Steps:

1. Initialization:

- Initialize a map (**result**) to store staff names and their inspection schedules.
- Get all staff-service-shelter pairs and corresponding services from the database.

2. **Iterate Through Staff-Service-Shelter Pairs:**

- For each staff-service-shelter pair:
    - Get the name of the staff and create an empty list of strings for that staff.
    - Get the service that needs to be inspected by this staff.
    - Schedule the services according to the their inspection frequency.

3. **Shift Pairs to Meet Inspect Limit:**

- After scheduling, check if the inspection matches the **inspectLimit**.
- If not, shift the inspection pairs to earlier days while considering constraints.

4. **Return Result:**
   - Return the generated inspection schedule map.

Design Pattern:

Singleton Pattern

The **DatabaseManager** class employs the Singleton Pattern to ensure that only one instance of the database manager is created throughout the application's lifecycle. This design choice optimizes resource utilization and centralizes control over database connections.

Facade Pattern

The **HelperMethod** class may exhibit characteristics of the Facade Pattern by providing a simplified interface to a set of more complex subsystem functionalities. It abstracts and centralizes interactions with the database and other components, offering a higher-level interface for the rest of the application. This simplification enhances maintainability and encapsulates implementation details.

The Singleton Pattern ensures controlled access to the database manager, while the Facade Pattern enhances code organization and maintainability.

Limitations

The staffInCharge attribute is of type String, which limits the information that can be stored about the staff member in charge. It would be more appropriate to have this attribute be of type Staff.