Design Document

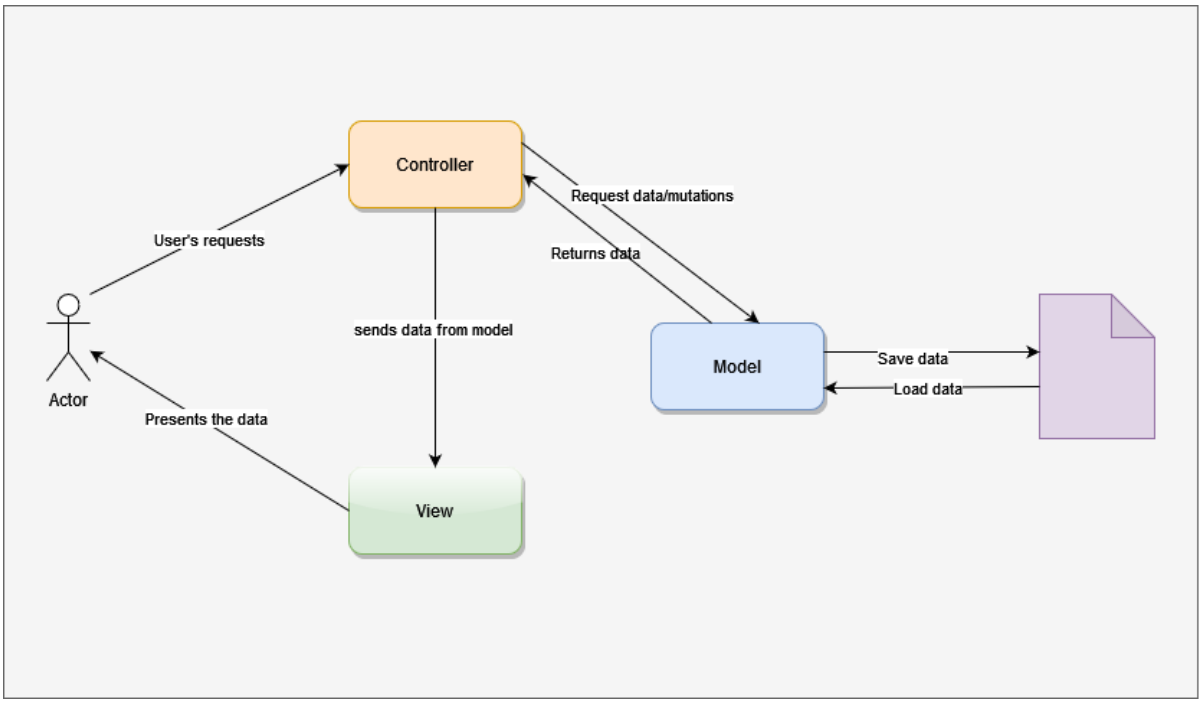
System Architecture

Architecture Pattern

The Model-View-Controller (MVC) pattern is selected for the Personal Information Manager (PIM). It separates presentation and interaction from the system data. The Model component manages all data and accepts actions that could transmit users' commands toward the data. The View component defines the information manager and how the data is presented to users. The Controller component is functioned to manage userzs' interactions, and these interactions are passed to View and Model components.

The reasons of using MVC pattern are as below:

1. The MVC pattern can support different presentations of similar data. The system is separated into three parts, and each part is responsible for different tasks. Hence, it could promote reusability of the functions, and code duplication can be avoided.
2. As the system is separated into three parts, the components are able to be tested independently. Therefore, it is possible to enhance the quality of the codes by identifying and fixing the codebase in the process of development.
3. Similarly, the separation of the system encourages maintainability. Since the components are separated, developers could make amendments to the components without affecting other components.



Overall Structure

The diagram below displays the overall relationship between the user and the PIM system.

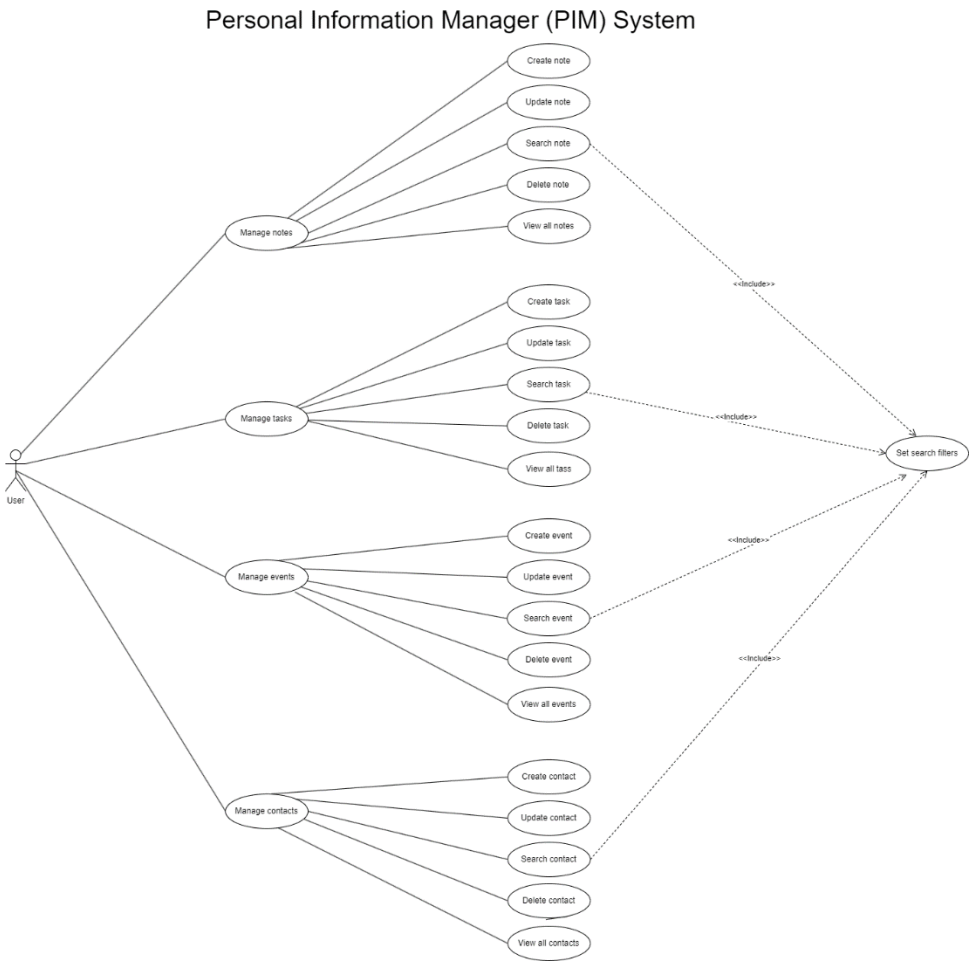
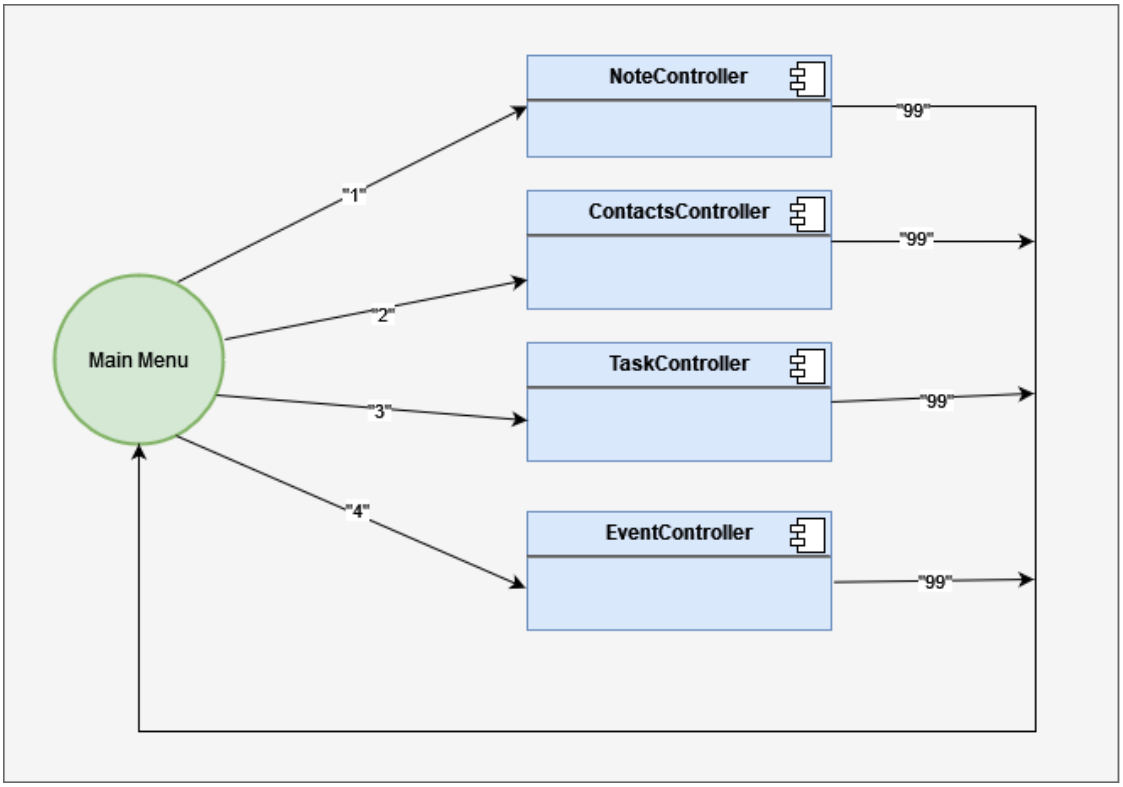


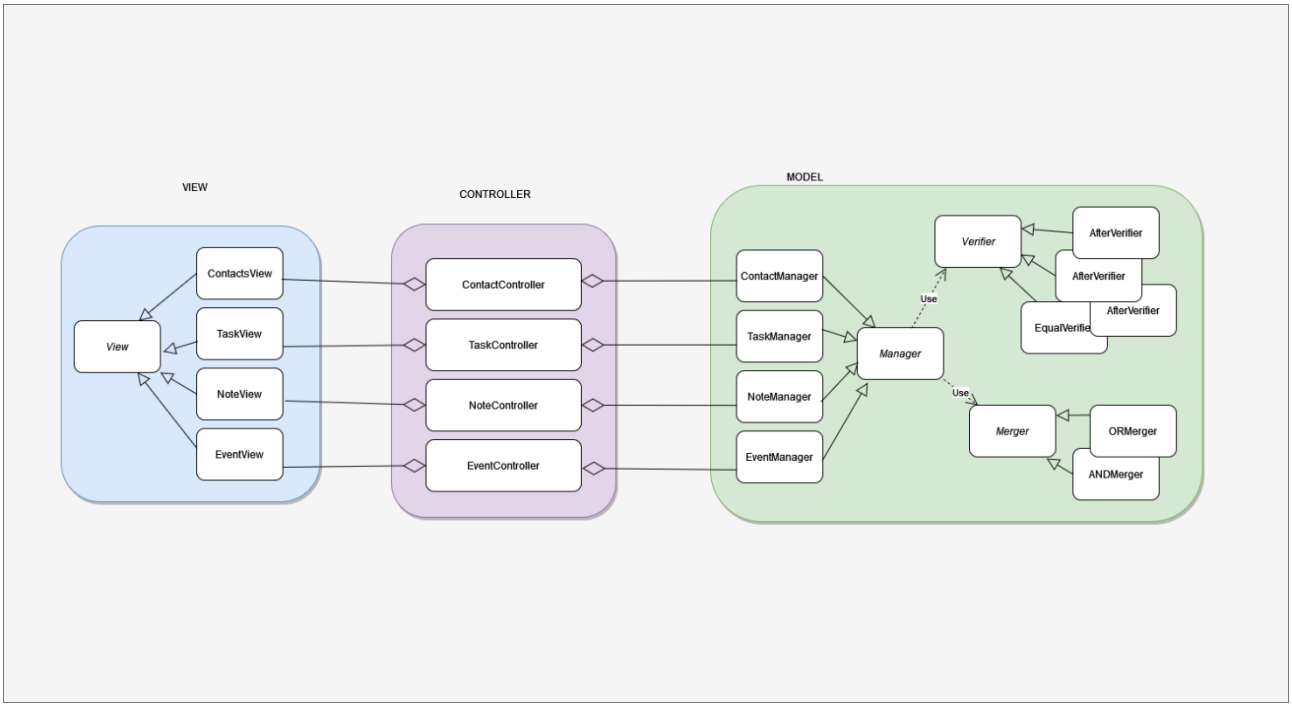
Figure 1: Use case diagram of PIM system

Once the PIM system is initialized, users will be prompted to select the category of the PIR record, including notes, tasks, events, and contacts. Users will then be prompted to perform the provided operations such as create record, update record, search record, delete record, and view all records.

PIM is a CLI-based application that relies on a dialog menu for the user's navigation and sending requests. Below is a diagram of how user requests are delegated to appropriate controllers based on the choice they made on the main menu. Across the application, the user interacts with the system via these choices and each choice changes the state as shown in this state diagram.



Code components



The above is the diagram that shows a summary of the components of the system in models, views, and controllers. They are going to be discussed in detail in later sections.

Controllers

Controllers are the instances that are responsible for handling user interactions and getting the user's request. This involves collecting the user's request details from console input. Upon collecting the user's, they delegate the request to appropriate model functions to handle the request and return or mutate the data as requested by the user. Controllers are also responsible for delegating the data to the appropriate view to handle data representation.

The application now has four controller classes:

1. NoteController
2. TaskController
3. EventController
4. ContactController

A typical controller has the following methods:

Method	**Description**
init()	Initialization of the menu for various function handling. This involves the creation of manager and view instances.
handleCreate()	Function to handle the creation of the PIR.
handleSearch()	Function to handle the search of the PIR.
handleUpdate()	Function to handle the updating of the PIR.
handleDelete()	Function to handle the deletion of the PIR.
handleViewAll()	Function to handle the view of all PIRs.
handleLoad()	Function to handle the loading of existing PIRs.
handleSave()	Function to handle the saving of PIR(s).

Entities

Entities are the domain instances that are independent of the changes of their attributes. They provide functionalities such as getting and setting UID and strings. In our system, the entities

The methods of entity are as below:

Methods	**Description**
setId(String)	Function to support the manager of setting the UID of the PIR.
getId(): String	Function to support the manager of getting the UID of the PIR.
toString(): String	Function to support the manager of allowing the data to be represented in a string.
getCopy(): Entity	Function to support the manager in getting a copy of the PIR.

Each entity concrete class proveides a static method *fromString()* which parses string into the classes object.

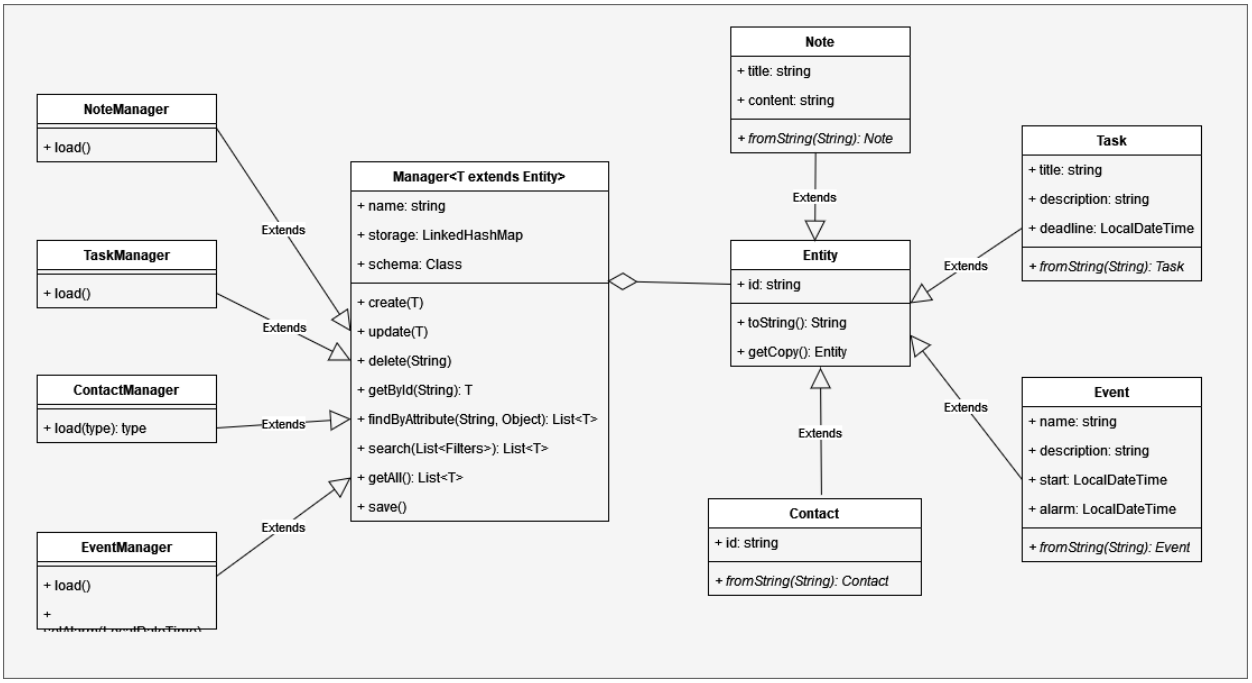
Example:

fromString(String): Note

Returns an object of class Note given a string representation of the instance. It is normally the reverse engineering of the toString() method.

Managers

Managers



Managers are model classes that are responsible for handling the logic involved in information retrieval and mutation of the PIRs. They provide multiple functionalities including creating, updating, editing and retrieving PIRS. They are aslo the component in charge of persistance of the data to the respective files as well as loading them when the system starts up.

The system comprises of 4 main managers:

- Note Manager
- Contact Manager
- Task Manager
- Event Manager

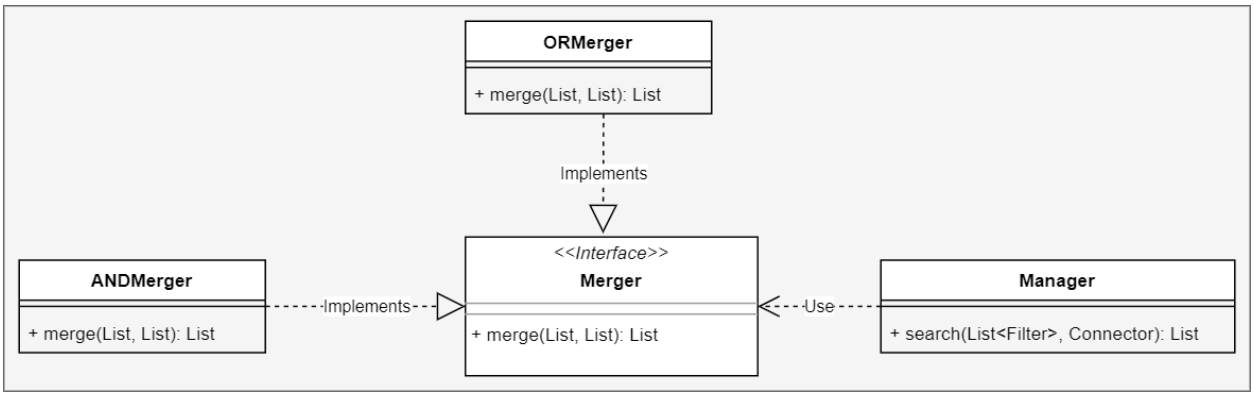
The methods of manager are as below:

Method	Description
create()	Function to create a PIR. This function will first generate an UID before storing the PIR with the UID.
update()	Function to update a PIR. This function will first obtain the UID of the PIR intended to be updated before storing the record with the new data.
delete()	Function to delete a PIR. This function will delete the PIR with the corresponding UID, which is inputted by users.
getAll()	This function will retrieve all the values in the storage before displaying them to the users.

getById()	This function will retrieve the PIR from the storage according to the UID input by users.
findByAttribute()	This function will retrieve the PIR from the storage according to the attribute and the corresponding value input by the users.
search()	This function will return the PIR results after performing logical operations towards the filtered data. This method is supported by findByAttribute().
checkId()	This function is used to check whether a PIR with the associated UID exists in the storage.
load()	Function to load all the PIR and related information from the matching .pim file to a LinkedHashMap.
save()	Function to save all the changes of PIR in the LinkedHashMap to a .pim file.

Mergers

Mergers are the instances responsible for combining the results of multiple filters based on what logical operator the user chooses to use. This is to ensure that the merged list consists of data according to users' search requirements.

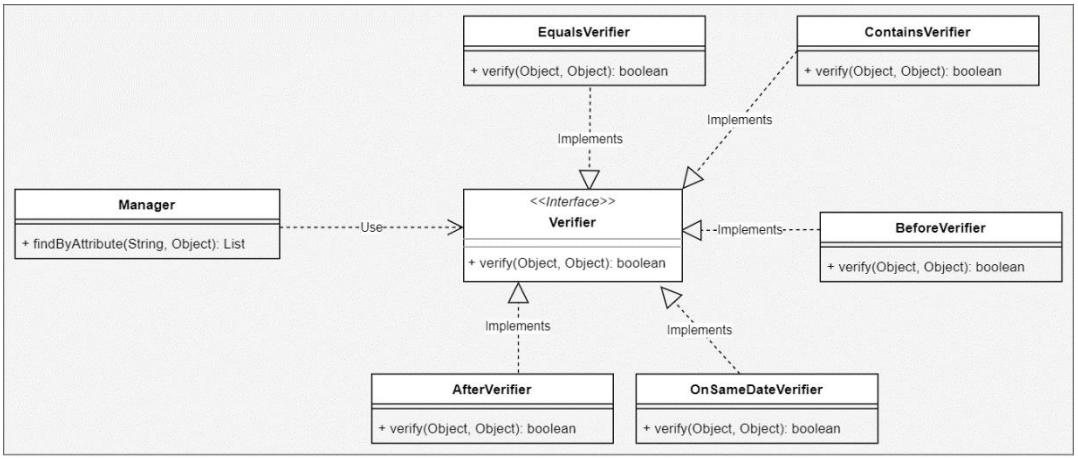


The application has the following methods:

Method	Description
ANDMerger::merge (List l1, List l2)	This merges two lists from filtering into one. This returns only the intersection of the two lists; i.e. the objects that match satisfies both filters.
ORMerger:: merge (List l1, List l2)	Filter and merge the data to a new list according to users' search criteria. This returns a union of the two lists; it means either one of the filters has to be true on one of those objects.

Verifiers

Verifiers are the instances to compare or validate whether an object matches another object according to different operators, such as equalTo, contains, before, and after. This is to ensure that the displayed output matches the users' search requirements.



The application has the following methods:

Method	Description	Applicable for (data type)
AfterVerifier::verify()	Verifies if the former date comes after the latter date.	DateTime
BeforeVerifier::vefrify()	Verifies if the former date comes before the latter date.	DateTime
ContainsVerifier::verify()	Verifies if the former string contains the latter string.	String
EqualsVerifier::verify()	Verifies if both provided strings or timestamps are equal	All (DateTime & String)
OnSameDateVerifier::verify	Verifies if both provided dates are equal.	DateTime

Filter

Filter is a class that contains methods to obtain and modify data based on specific criteria. It encapsulates the attributes, targets, and operators used for performing search operations.

The application has the following methods:

Method	Description
getAttribute()	Obtain the attribute of the data.

setAttribute()	Attach the data with certain attributes.
getTarget()	Obtain the targeted data.
setTarget()	Attach the targeted data.
getOperator()	Obtain the operator for target filtering purposes.
setOperator()	Attach the operator for target filtering purposes.

Operators

Operators are the members of enumerator class of Operator. They are used to indicated which verifier to use when evaluating a filter. These are one-to-one relationship with verifier classes. The system has the following operators: EQUALS, BEFORE, AFTER, CONTAINS, ONSAMEDATE.

- EQUALS indicates that a verify function from EqualVerifier has to be used to evaluate the filter.
- CONTAINS indicates that a verify function from ContainsVerifier has to be used to evaluate the filter.
- BEFORE indicates that a verify function from BeforeVerifier has to be used to evaluate the filter.
- AFTER indicates that a verify function from AfterVerifier has to be used to evaluate the filter.
- ONSAMEDATE indicates that a verify function from AfterVerifier has to be used to evaluate the filter.
-

Connectors

Connectors are used in instances to perform validation procedures.

The application has the following connectors: OR, AND

- OR indicates that a merging function from ORMerger has to be used to evaluate the filter.
- AND indicates that a merging function from ANDMerger has to be userd to evaluate the filter.

Views

Views are the instances that define and manage how data is presented to users. Appropriate methods will be selected to display the data passed by the controller.

The application now has four view classes:

1. NoteView
2. TaskView
3. EventView
4. ContactView

A typical view has the following methods:

Method	Description
displayDetails(T t)	Function to display the details of a specific PIR to users. This is normally used when only one PIR is required to be displayed.
displayList(List<T> t)	Function to display a list of PIRs to users. This is normally used when more than one PIR is required to be displayed.
printError (Exception e)	Function to display the error message to users.

Errors and Exceptions

These are the following exceptions that are supported by the system:

UnsupportedOperation

This exception is normally thrown by a verifier when the caller attempts to perform an operation that is not valid on the type of given instances. For example, trying to run contains method on integer or LocalDateTime. Another example is trying to run BEFORE on non-Chronology Objects.

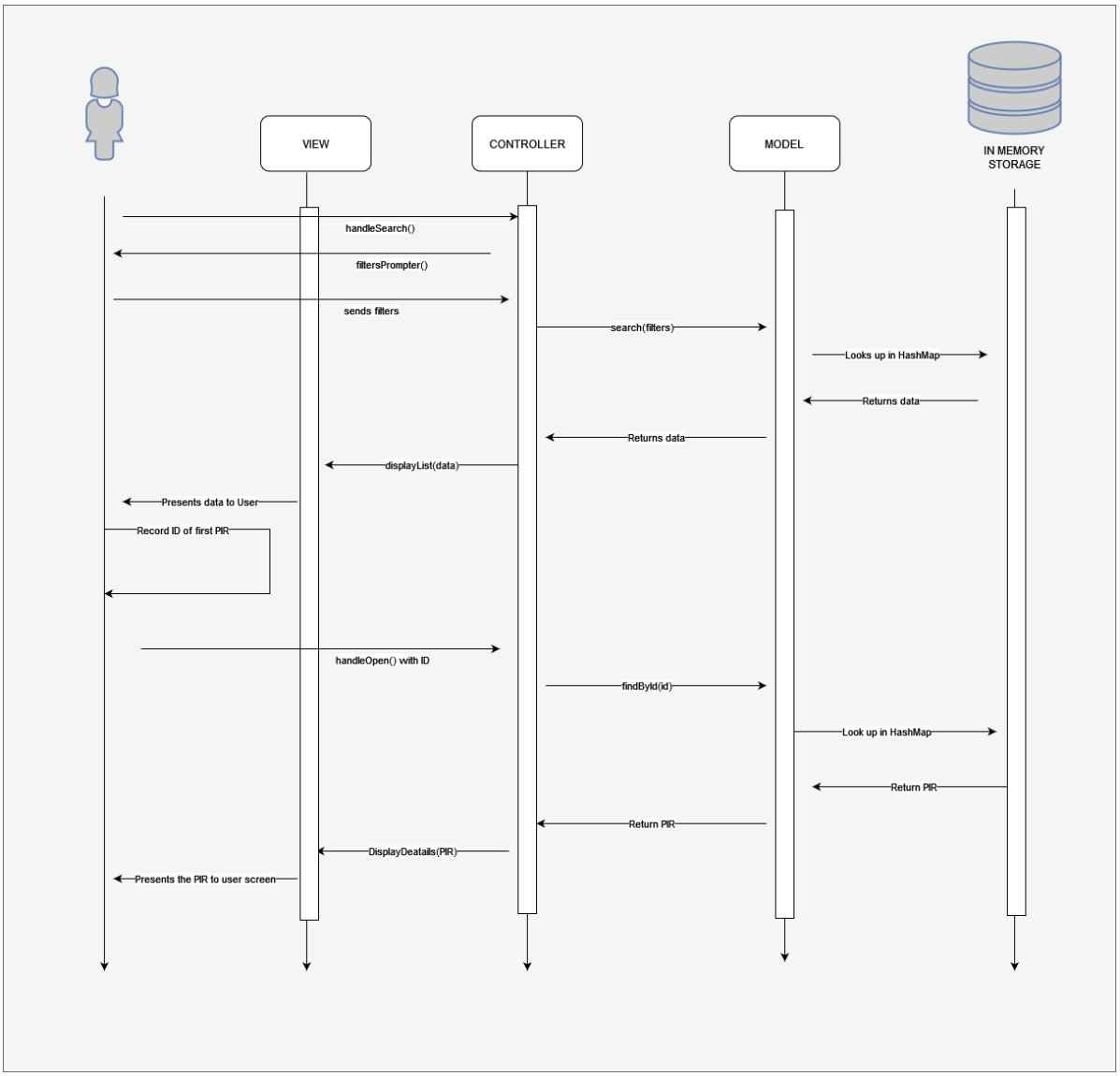
ValidationError

This exception is thrown whenever there is an attempt to violate set constraints on certain behaviors. For instance, this exception is thrown when user attempts to set an alarm for the time which is the past. This action attempts to violate the rules that the alarm should not be set for time in past.

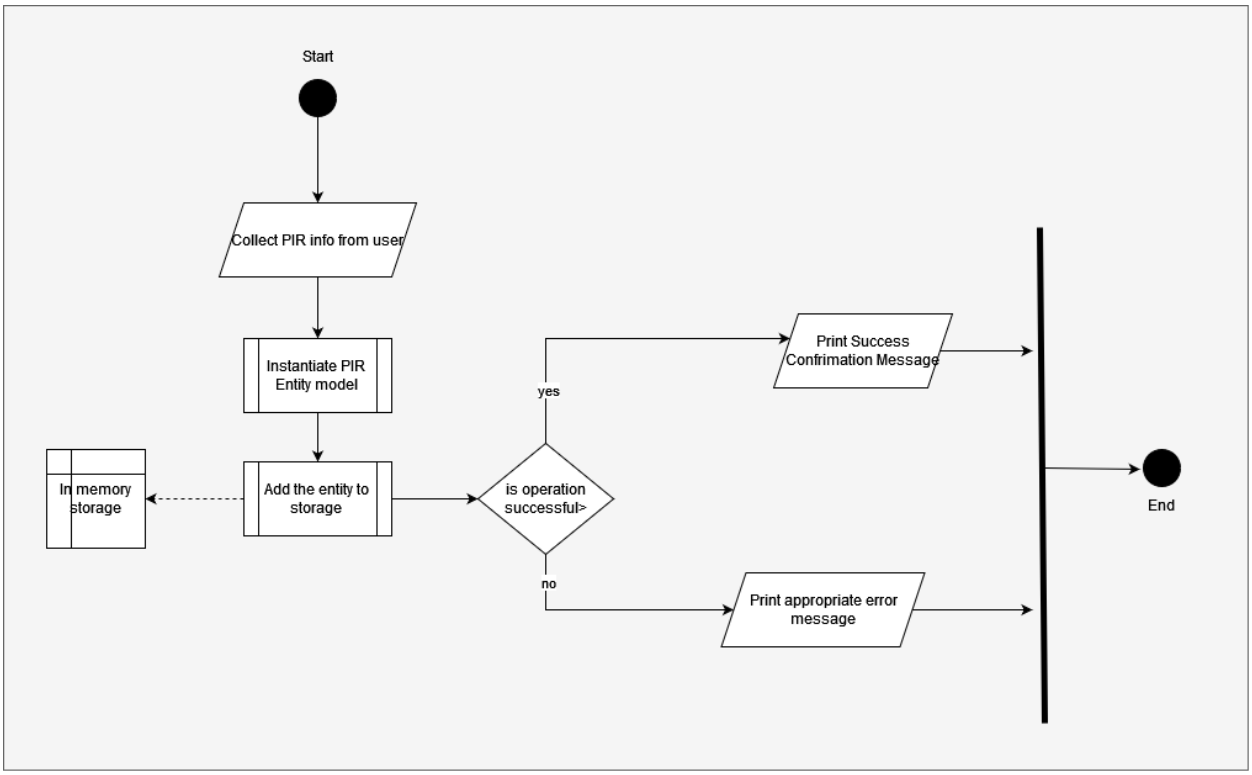
Interaction with the System

PIM is an interactive system that helps users manage their personal information. Below is a sequence diagram of a typical user interaction with the system. The scenario is when the user wants to open a certain PIR. This is done through series of communication between different components of the system.

-

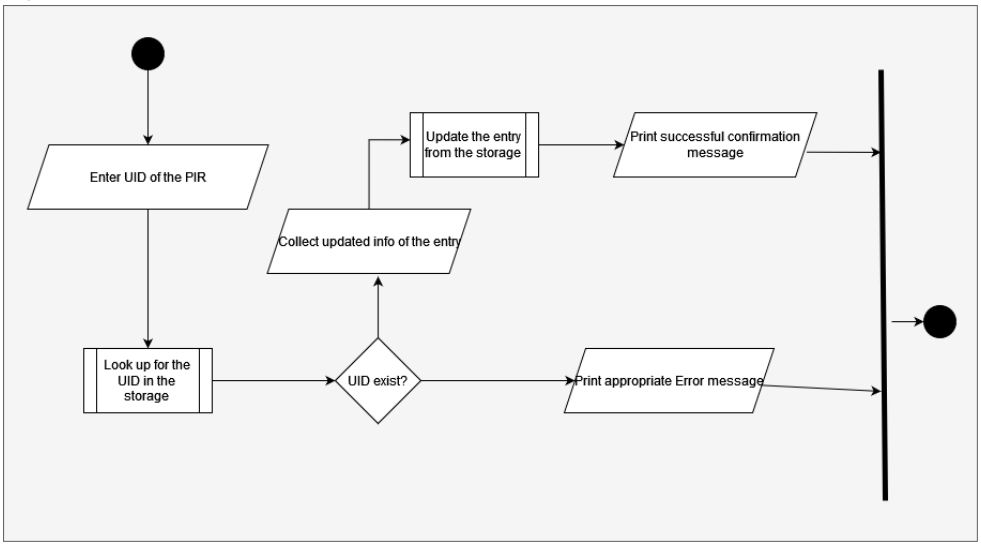


Add record



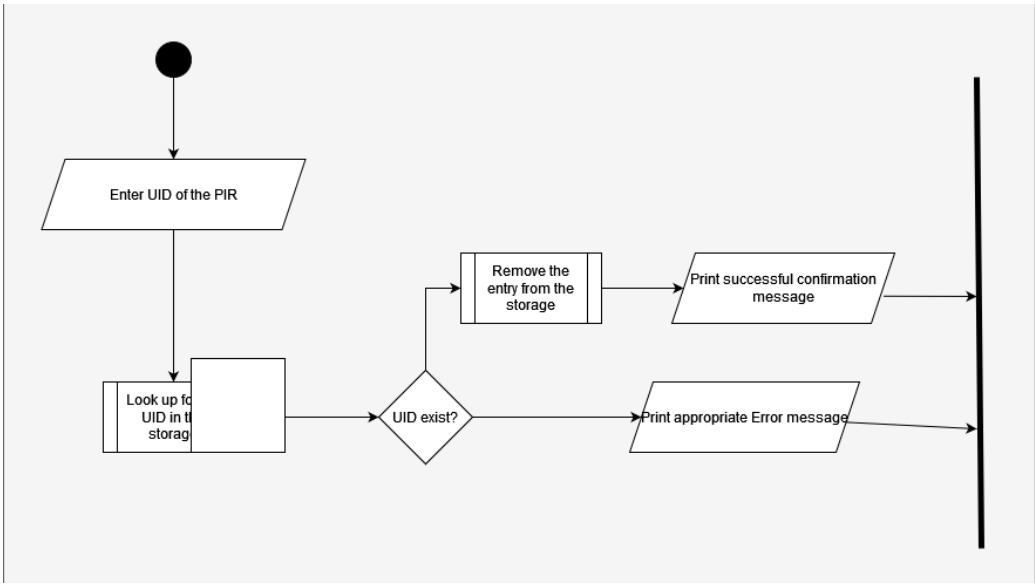
After selecting to add record based on the category, users will be prompted to add two types of data based on the selected category sequentially (except contact, which consists of three fields to be completed). The system will check the existence of empty string from user's inputs, and users will be prompted to input again until empty string is not detected by the system. After inputting all the data, the system will store the data temporarily and display the message to indicate that the record has been created successfully.

Update record with UID



Once users have selected to update the record, users will be prompted to provide the unique identifier of the record (UID). The system will then check the existence of the record with a matched UID. If the system cannot find a record with the provided UID, it will display a message indicating that the record with the inputted UID could not be found. Once the corresponding record has been found, users will be prompted by the system to input the data to override the existing information, and this operation is similar to the process of adding new data. Afterwards, the system will display the message to indicate that the record has been updated successfully.

Delete record with UID

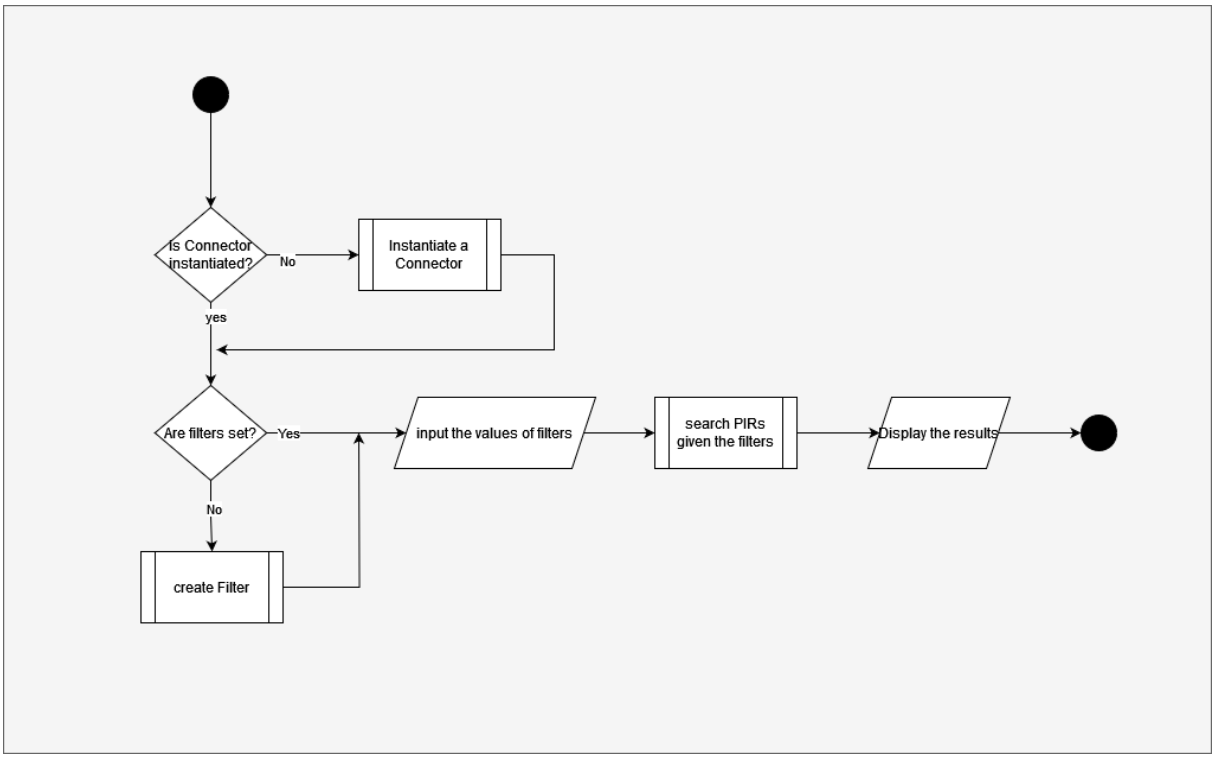


Similar to update record, users will be prompted to enter the unique identifier of the record before proceeding to the deletion process. The system will then first check the existence of the record with matched UID. If the system cannot find a record with the provided UID, it will display a message indicating that the record with the inputted UID could not be found. If a record is found with the UID, the system will proceed to delete the record and display the message to indicate that the record has been deleted.

Search record with UID

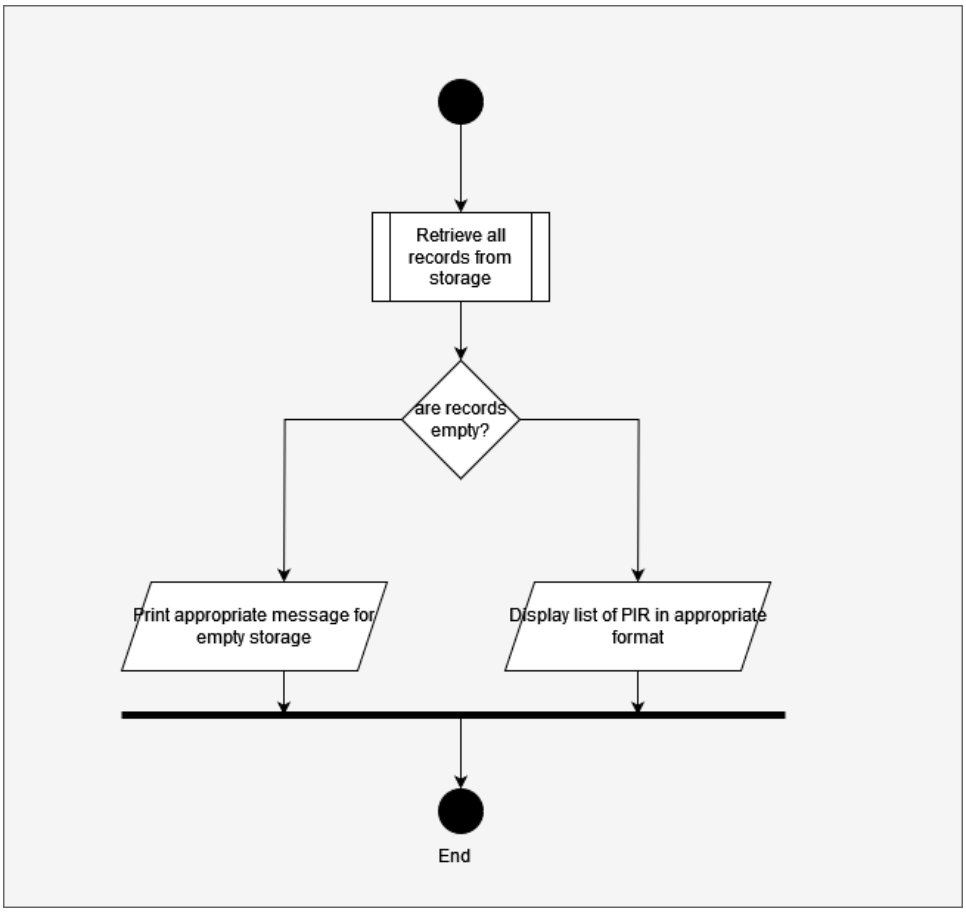
To search for a record with UID, the system will first check whether prior records exist in the system. If there is none, the system will then prompt a message to remind the user to add records before proceeding to use the search feature. If there are existing records, users will be prompted to input the UID to search for a particular record. Afterwards, the system will attempt to search for record with matched UID, and if found, the system will display the record to the user. Otherwise, a message will be displayed to indicate that the record with the UID does not exist.

Search record with attribute



To search for a record with attribute, the system will first check whether prior records exist in the system. If there is none, the system will then prompt a message to remind the user to add records before proceeding to use the search feature. If there are existing records, users will be prompted to input the attribute to search for a particular record. Afterwards, the system will prompt users to input the value that corresponds to the attribute. The system will then filter and gather data based on the criteria and will calculate the size of the resulting data. If the size of the result is zero, a message will be displayed to user to indicate that no record has been found based on the search criteria. Otherwise, if the size of the result is bigger than zero, indicating that matched record based on search criteria has been found. The system will then display the record to users.

View all records



After users have selected to view all records, the system will first check whether prior record(s) exist(s). If there is no presence of previous records, the system will display a message indicating that there is no prior saved record. Otherwise, the system will display all the records to the user.