

Software requirements specification

Version 3.0

Preface

This document is intended for all the software developers, project managers, testers, and other project stakeholders.

Version	Date	Author	Changes
1.0	2023-11-18	LI Shuhang	Initial version of SRS
2.0	2023-11-21	CHEN Ziyang He Rong	Revised functional requirement
3.0	2023-11-22	YE Chenwei	Modify the system architecture

Introduction

This software system is a command line-based personal information manager. Designed to help users manage their personal information by providing tools. More specifically, the system allows users to create different types of personal information supporting modification, deletion, addition, and search. All personal information data will be stored respectively. The software will also help users manage and organize personal information, optimize time planning, and improve work efficiency.

Glossary

Term	definition
PIM	Command line personal information manager
PIRs	personal information record
MVC	MVC (Model-View-Controller) is a pattern in software design commonly used to implement user interfaces, data, and controlling logic.
Note	A type of PIR (plain text)
Task	A type of PIR (including deadline and task description)
Contact	A type of PIR (including name, address and mobile phone number)
Event	A type of PIR (including start time and alarm and related description)

User requirement definition

Functional Requirements:

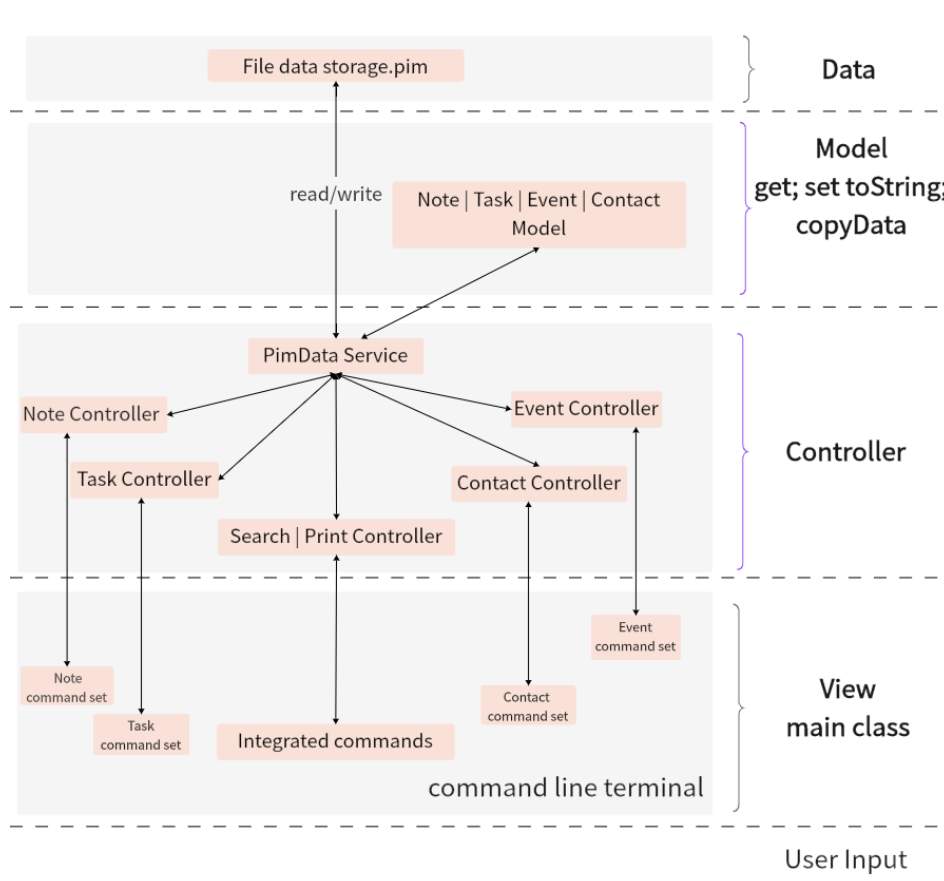
1. The PIM shall allow the user to create different types of PIRs. Including notes, tasks, events, and contacts.
2. The PIM shall allow users to modify any data in PIRs.
3. The system shall allow users to search for PIRs using specific criteria related to their type and the data they contain. These standards can be any attribute, like text or times. In addition, the system allows users to create compound search conditions by using AND (&&), OR (| |), and NOT (!) operators.
4. The PIM shall allow users to print information about a specific PIR(e.g., A unique contact's information or details of an event) or all PIRs.
5. The PIM shall allow users to delete the specified PIR.
6. The PIM shall allow users to modify any data in PIRs.
7. The PIM shall store PIRs in files with a ".pim" extension so that users can use PIM in the future.
8. The PIM shall load PIRs as files with a ".pim" extension so that users can continue to use previously stored PIRs.

Non-Functional Requirements:

1. The system should be user-friendly. Users can master all system functions within 10 minutes through the user manual.
2. The system should be reliable. All changes made by users to PIRs should be appropriately updated and stored.
3. The system should support re-enter operations when the user enters something incorrectly rather than directly exiting the program.

System architecture

1. Overview of the system architecture: The main packages are controller, model, service, and util. The primary function can be seen as a View layer. Each package has its essential functions. The following is a high-level view to facilitate the understanding of the structure and organization of the system.



2. Component description: the four parts of the object controller include adding, modifying, deleting, and listing objects in their respective positions. Search and Print Controller are separate parts that contain specific functions. The model package contains important properties and methods in several classes.
3. Data Flow and Processing Flow: The basic flow is that the command first enters the executeCommand, analyses which operations are: list, add, edit, or delete, and carries out further operations according to the command. Describe the data flow and processing flow in the system. Describe the source, transmission, and destination of data and the algorithms, rules, or logic involved in processing.
 Print controller: Performs different print operations according to the given command. If the command is "all", all Pir objects are printed; if the command is a specific id, the Pir objects corresponding to that id are printed.
 Search controller: Based on the query type, text and time entered by the user, all PIRs that meet the query conditions are returned. Support users to use operators such as "&&" to enter query conditions.

System requirements specification

Functional Requirement

1. The PIM shall allow the user to create different types of PIRs. When creating a PIR, the user first specifies the type of PIR.
 - 1) For a note, users need to input plain text.
 - 2) For a task, users need to provide a due date and description.

- 3) For an event, users need to give a description, start time, and alarm.
- 4) And for a contact, users need to have the name, address, and phone number of contact.

The system then creates the corresponding PIR based on those inputs.

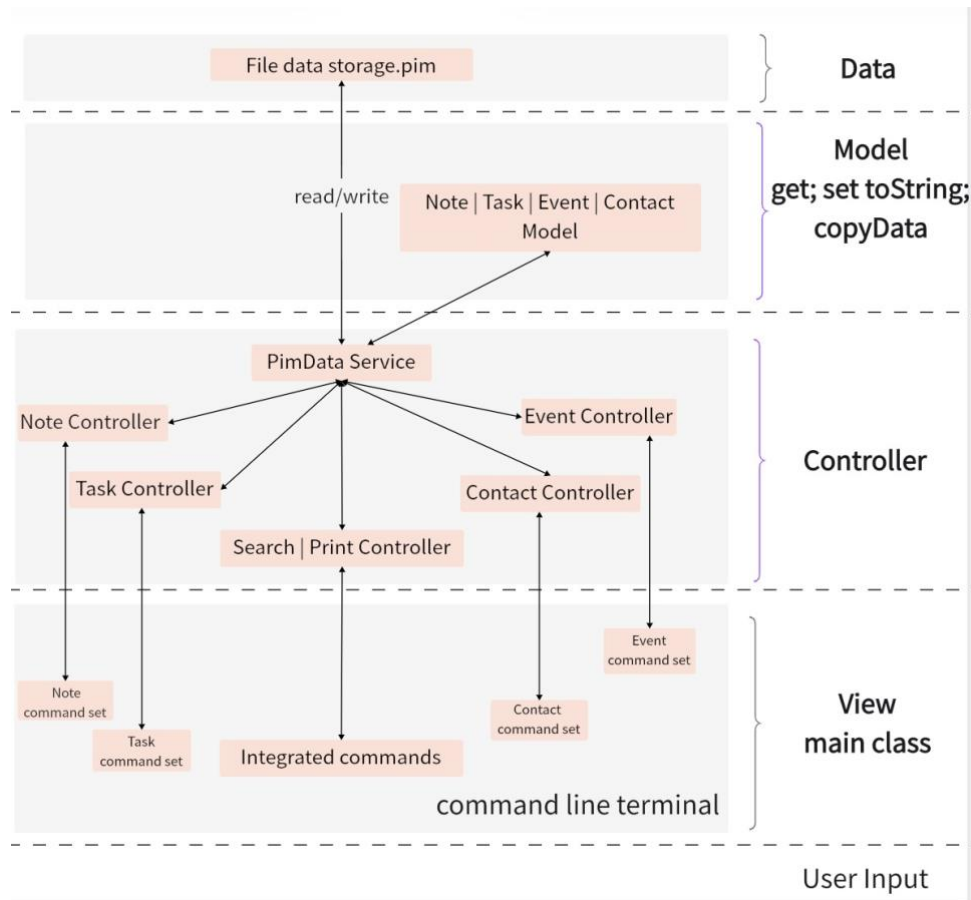
2. The PIM shall allow users to modify any data in PIRs. Users can choose to modify the entire PIR or modify a certain part of the PIR. For example, the system shall allow users to modify both the description and deadline in a specific task or one of them.
3. The system shall allow users to search for PIRs using specific criteria related to their type and the data they contained. Users can search by specific elements such as name, phone number, deadline, or plain text. Users can choose to view records before and after the input time. It is also possible to query all PIRs containing specific content. Additionally, the system allows users to create compound search criteria by logically combining these criteria using AND (&&), OR (||), and NOT (!) operators. The system will return all PIRs that meet the criteria.
4. The PIM shall allow users to print detailed information about a specific PIR or all PIRs. The user enters "print all" and the system will return all PIRs. The user enters "print + ID", and the system will return a specific PIR.
5. The PIM shall allow users to delete the specified PIR. The user enters the ID of the PIR they wish to delete, and the system will delete the PIR specified by the user.
6. PIM shall allow users to delete selected PIRs. The user enters the PIR type and ID that they wish to change, and the system allows the user to modify each key element of the PIR in turn based on the PIR type.
7. The PIM shall store PIRs in files with a ".pim" extension so that users can access them in the future using PIM. Users can click on the data.pim file to view all PIRs data.
8. PIM should load the PIR as a file with a ".pim" extension so that the user can continue to use the previously stored PIR. When the user runs the system and performs operations, the previous files are loaded and all previous PIRs can be operated on.

Non-Functional Requirement

1. The system should be user-friendly. Users can master all functions of the system within 10 minutes through the user manual.
2. The system should be reliable. All changes made by users to Personal Information Records (PIRs) should be updated and stored immediately and accurately without loss of data. Check whether all information in the pim document is saved correctly. Repeat the re-run to see if the changes in between are successfully saved.
3. The system should make the user re-enter (when the user enters something incorrectly) rather than exit the program. Validated through exception handling testing, where various incorrect inputs are fed to the system to ensure it handles them correctly without crashing.

Design document

(1) PIM's architecture



This project adopts **the MVC (Model-View-Controller) architectural pattern**, which is a software design pattern for designing user interfaces. It decomposes the application into three interconnected components, each component is responsible for its own responsibilities. In our project, the components of the MVC pattern are instantiated as follows:

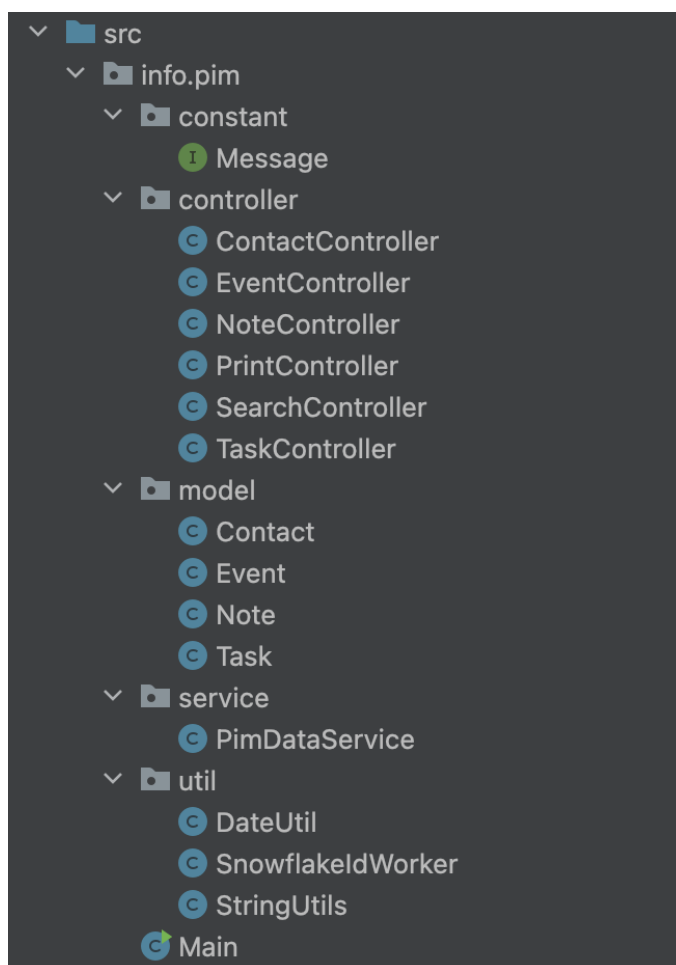
1. The **Model** layer implements entity abstraction of data. The four model classes Note, Task, Event and Contact are defined in the info.pim.model package.
2. The **View** layer interacts with the user through the command line interface, receives system instructions input by the user, and returns the results of the instruction operations to the command line interface. The view is this command line interface. The command line input and output in the main method in the Main class is the implementation of the view.
3. The **Controller** layer includes control classes and service classes. The control class accepts instructions from the view layer, performs some logical processing, and completes data access and update operations through the service class. Four controller classes, NoteController, TaskController, EventController and ContactController, are defined in the info.pim.controller

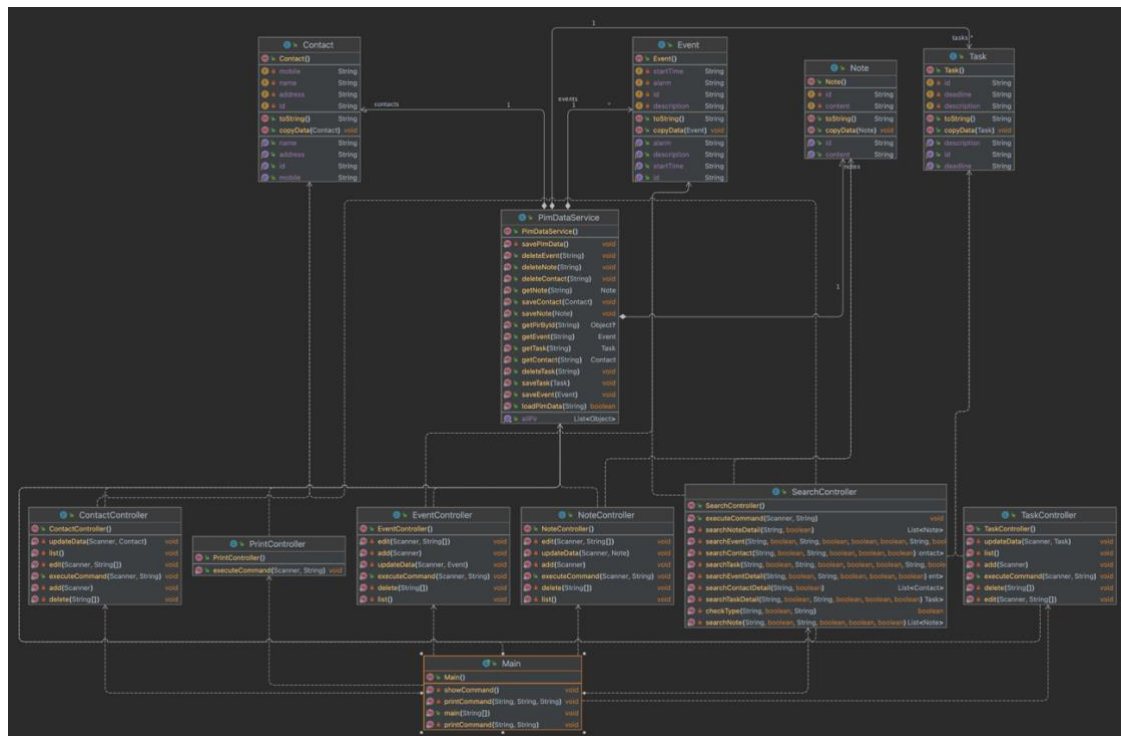
package. They process commands entered from the command line and update the model accordingly.



There are two main reasons why we choose the MVC model:

1. Separation of concerns: The MVC pattern separates data processing (model), user interface (view) and control logic (controller), allowing each part to be developed and modified independently, improving the maintainability and scalability of the code.
2. Code reuse: Since the MVC pattern decomposes the application into three parts that are independent of each other, these parts can be reused in different scenarios. For example, a model can be used by multiple views, or a controller can be used to coordinate multiple models and views.

(2) The structure of the major code components in the PIM



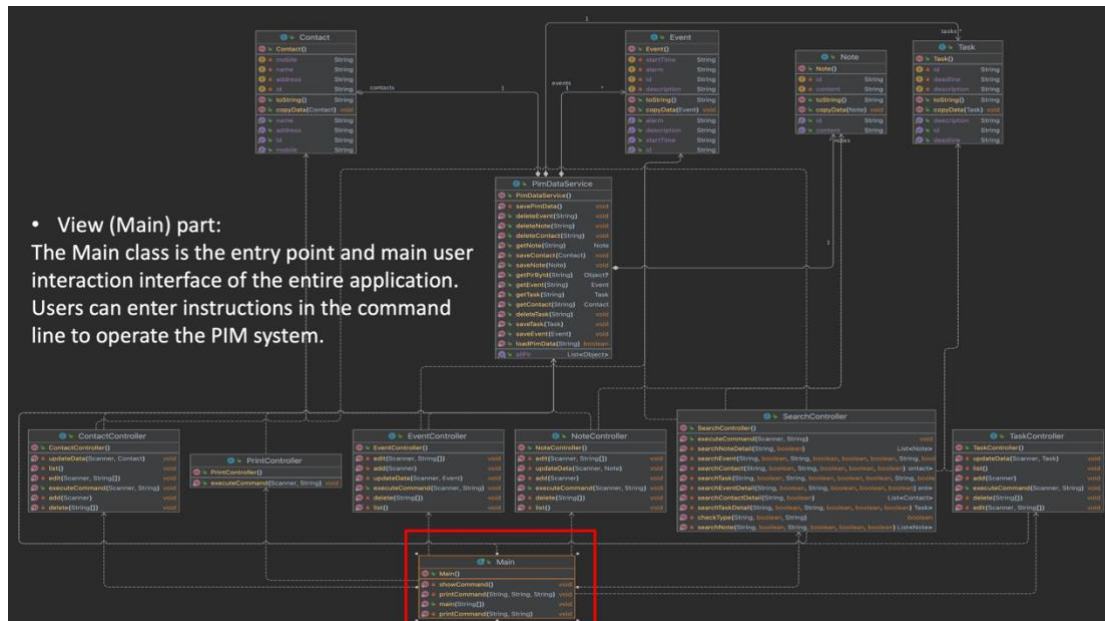


In all the images in this document, the green unlock icon  represents "public". The public modified class/property/method can be accessed from any location. The red lock icon  indicates "private". Private modified properties/methods can only be accessed within the same class.

The main code components in this project mainly includes the following parts.

Main Class

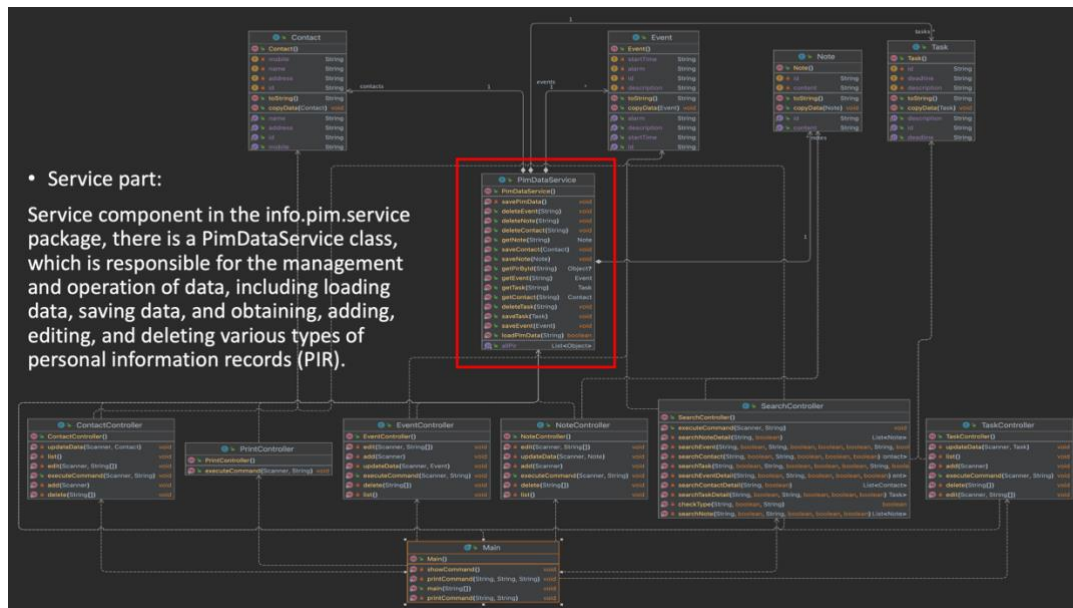
The Main class is the entry point of the entire application and the main user interaction interface. It contains a main method. The main method first attempts to load data from the file, then enters an infinite loop, waits for user input instructions, and calls the corresponding controller method according to the instructions. This is mainly implemented through the command line interface. Users can enter instructions in the command line to operate the PIM system, display information to the user through the System.out.println method and obtain user input through the Scanner object.



- **Fields:** None.
- **Methods:**
 - `main(String[] args)`: This is the entry method of the application. This method is called by the JVM when the application is run. This method first calls the `PimDataService`'s `loadPimData()` method to load the data, then enters a loop that waits for the user to input the instruction and calls the `executeCommand (String command)` method to execute the instruction. Does not return anything.
 - `printWelcome()`: Prints the welcome message to the console. Tell the user how to use this application. Does not return anything.
 - `executeCommand(String command)`: Executes the given command from the user. Does not return anything.
 - `printGoodbye()`: Prints the goodbye message to the console. Does not return anything.

PimDataService Class

Within the Service component of the `info.Pim.Service` package, there is a `PimDataService` class, which is responsible for the management and operation of data, including loading data, saving data, and obtaining, adding, editing, and deleting various types of personal information records (PIR).



- **Fields:**

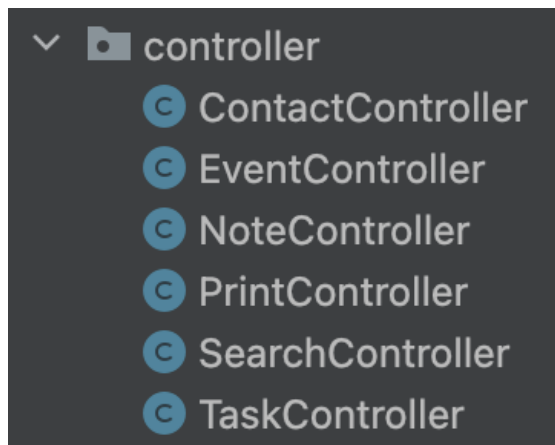
- notes: A List of Note objects.
- tasks: A List of Task objects.
- events: A List of Event objects.
- contacts: A List of Contact objects.

- **Methods:**

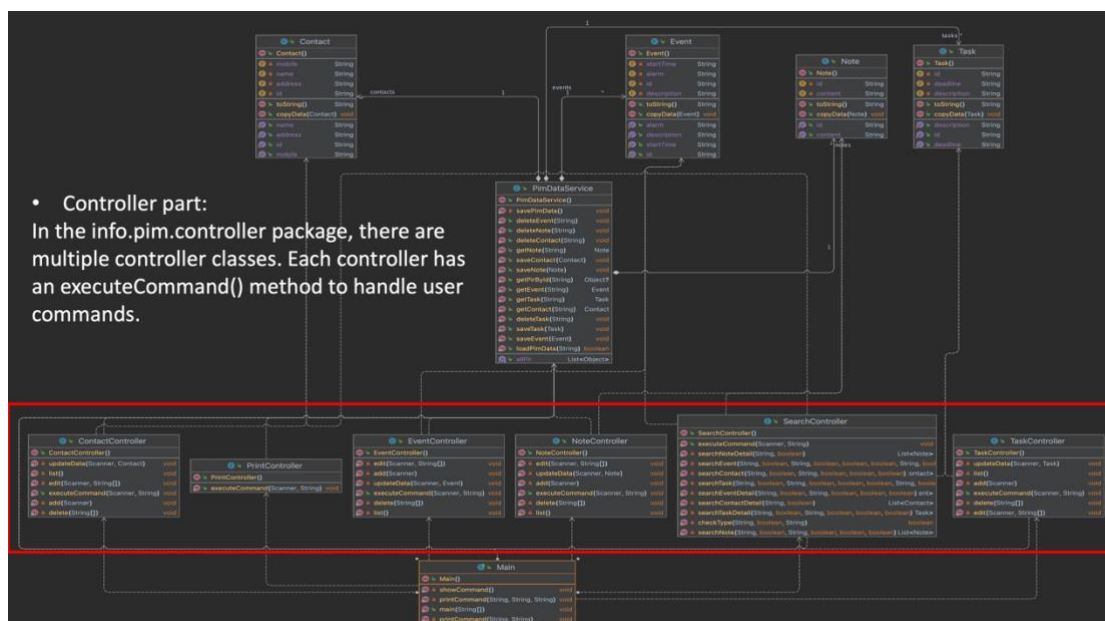
- loadPimData(String path): Loads the PIM data from the given file path. Does not return anything. Throws IOException, ClassNotFoundException.
- savePimData(): Saves the current PIM data to a file. Does not return anything. Throws IOException.
- getNote(String id), getTask(String id), getEvent(String id), getContact(String id): Retrieves the respective PIR with the given id. Returns the respective PIR.
- saveNote(Note note), saveTask(Task task), saveEvent(Event event), saveContact(Contact contact): Saves the respective PIR. Does not return anything.
- deleteNote(String id), deleteTask(String id), deleteEvent(String id), deleteContact(String id): Deletes the respective PIR with the given id. Returns a boolean indicating whether the deletion was successful.
- getAllPir(): Returns a list of all PIRs.
- getPirById(String id): Returns any type of PIR with the given id. If the corresponding PIR cannot be found, it returns null.

Controller Classes

(NoteController, TaskController, EventController, ContactController)



In the info.pim.controller package, there are multiple controller classes including NoteController, TaskController, EventController, ContactController, SearchController, and PrintController. Each controller has an executeCommand() method to handle user commands. For example, the executeCommand() method of the NoteController class will call the corresponding list(), add(), edit() or delete() method according to the user's instructions.



• **Controller part:**
In the info.pim.controller package, there are multiple controller classes. Each controller has an executeCommand() method to handle user commands.

- **Fields:** Each controller has a PimDataService object.
- **Methods:**
 - executeCommand(Scanner sc, String command): Executes the given command from the user. Does not return anything.
 - list(): Prints a list of all PIRs of the respective type. Does not return anything.
 - add(Scanner sc): Adds a new PIR of the respective type. Does not return anything.

- `edit(Scanner sc, String[] commands)`: Edits an existing PIR of the respective type. Does not return anything.
- `delete(String[] commands)`: Deletes an existing PIR of the respective type. Does not return anything.

Here are the main classes of these controller packages:

NoteController class:

- Handles user commands related to notes.
- `executeCommand(Scanner sc, String command)`: This method is used to execute a given user command related to notes. It takes a Scanner object `sc` (for reading user input) and a String command (the user command to be executed). This method does not return anything (return type is void). It does not declare any exceptions. Inside this method, it parses the command, and depending on the command type, it calls the `list()`, `add(Scanner sc)`, `edit(Scanner sc, String[] commands)`, or `delete(String[] commands)` method.
- `list()`: This method is used to list all notes. It does not take any arguments and does not return anything (return type is void). It does not declare any exceptions. The method uses the `PimDataService` to retrieve all notes and then prints them to the console.
- `add(Scanner sc)`: This method is used to add a new note. It takes a Scanner object `sc` (for reading user input). The method does not return anything (return type is void). No exceptions are declared for this method. Inside this method, it prompts the user for the note content, creates a new Note object, and uses the `PimDataService` to save the note.
- `edit(Scanner sc, String[] commands)`: This method is used to edit an existing note. It takes a Scanner object `sc` (for reading user input) and a String array `commands` (the user commands). The method does not return anything (return type is void). No exceptions are declared for this method. Inside this method, it identifies the note to be edited based on the commands, prompts the user for the new content, updates the Note object, and uses the `PimDataService` to save the changes.
- `delete(String[] commands)`: This method is used to delete an existing note. It takes a String array `commands` (the user commands). The method does not return anything (return type is void). No exceptions are declared for this method. Inside this method, it identifies the note to be deleted based on the commands and uses the `PimDataService` to delete the note.

TaskController class:

- Handles user commands related to tasks.
- The `executeCommand(Scanner sc, String command)` method calls the appropriate method based on the user's command.
- The `list()`, `add(Scanner sc)`, `edit(Scanner sc, String[] commands)`, and `delete(String[] commands)` methods are used respectively to list, add, edit, and delete tasks.

EventController class:

- Handles user commands related to events.
- The executeCommand(Scanner sc, String command) method calls the appropriate method based on the user's command.
- The list(), add(Scanner sc), edit(Scanner sc, String[] commands), and delete(String[] commands) methods are used respectively to list, add, edit, and delete events.

ContactController class:

- Handles user commands related to contacts.
- The executeCommand(Scanner sc, String command) method calls the appropriate method based on the user's command.
- The list(), add(Scanner sc), edit(Scanner sc, String[] commands), and delete(String[] commands) methods are used respectively to list, add, edit, and delete contacts.

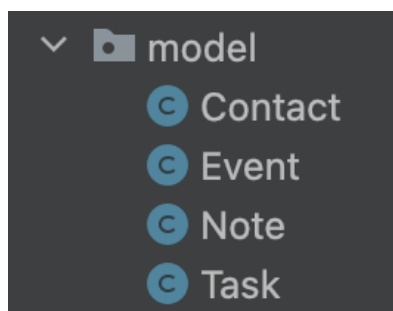
SearchController class:

- Handles user commands related to searching.
- The executeCommand(Scanner sc, String command) method calls the appropriate method based on the user's command.
- search(String[] commands): This method is used to search for PIRs (Personal Information Records) based on the given commands. The commands argument is a String array that contains the search keywords. The method does not return anything (return type is void). No exceptions are declared for this method. This method processes the commands array, extracts the search keywords, and uses the PimDataService to retrieve matching PIRs. The results are then printed to the console.

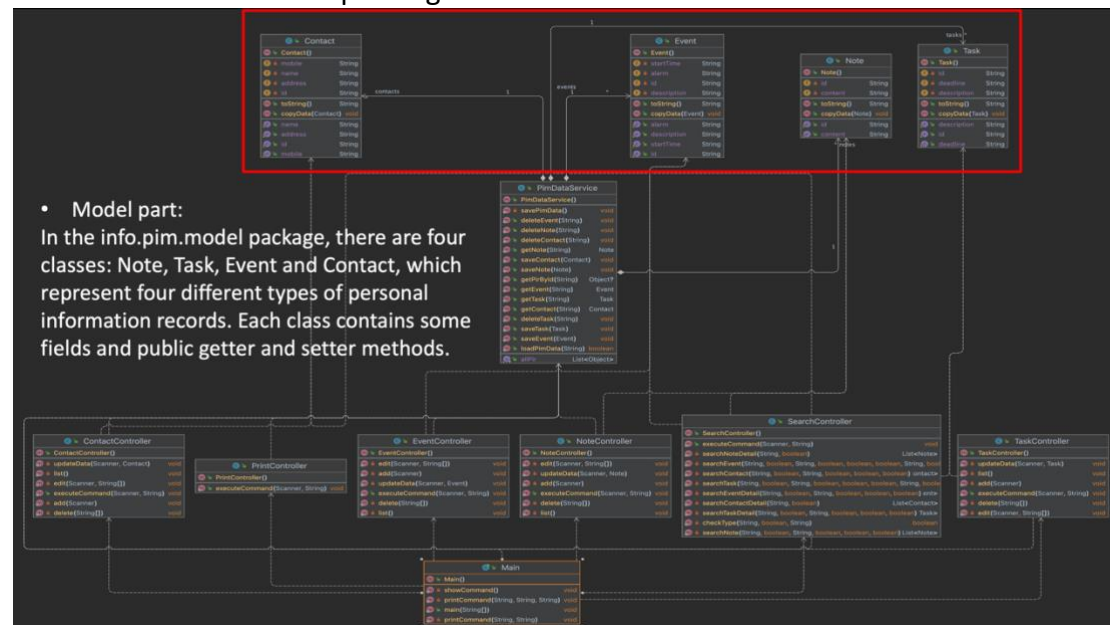
PrintController class:

- Handles user commands related to printing.
- The executeCommand(Scanner sc, String command) method calls the appropriate method based on the user's command.
- printAll(): This method is used to print all PIRs. It does not take any arguments and does not return anything (return type is void). No exceptions are declared for this method. Inside this method, it uses the PimDataService to retrieve all PIRs and then prints them to the console.

Model Classes (Note, Task, Event, Contact)



In the info.pim.model package, there are four classes: Note, Task, Event and Contact, which represent four different types of personal information records. Each class contains some fields and public getter and setter methods.



- **Fields:** Each model has a String id field, and respective fields related to its type.
- **Methods:**
 - getId(): Returns the id of the PIR.
 - setId(String id): Sets the id of the PIR. Does not return anything.
 - copyData(Note/Task/Event/Contact source): Copies data from the source PIR. Does not return anything.
 - toString(): Returns a string representation of the PIR.

The relationship among the classes in the PIM

1. Dependencies:

Main class depends on the following classes: PimDataService, NoteController, TaskController, EventController, ContactController, SearchController, PrintController. Controller classes (NoteController, TaskController, EventController, ContactController, SearchController, PrintController) depend on the following classes: PimDataService, Scanner (Java standard library class)

The PimDataService class depends on the following classes: Note, Task, Event, Contact, File (Java standard library class), FileInputStream (Java standard library class), ObjectInputStream (Java standard library class), FileOutputStream (Java standard library class), ObjectOutputStream (Java Standard library class), StringUtils (a tool class for processing strings).

Model classes (Note, Task, Event, Contact) depend on the Serializable interface (Java standard library interface).

2. Inheritance relationship: In this PIM (Personal Information Manager) project, there

is no explicit inheritance relationship between classes. All classes inherit directly or indirectly from Java's Object class. This is the default behavior of all classes in Java. However, the model classes in this project all implement the `java.io.Serializable` interface, which is a special kind of inheritance. `Serializable` is a tagged interface that does not contain any methods, but classes that implement this interface are considered serializable by the Java Virtual Machine (JVM). Serialization is the process of converting an object's state information into a form that can be stored or transmitted. In this project, the Note, Task, Event, and Contact classes all implement the `Serializable` interface so that their instances can be serialized and deserialized for storage and reading in files.

3. Association: There is an association between the `PimDataService` class and the model classes Note, Task, Event, and Contact. There are four List type fields in the `PimDataService` class: `notes`, `tasks`, `events`, and `contacts`. These fields store Note, Task, Event, and Contact type objects respectively, which indicates that the `PimDataService` class is associated with these four model classes.

The relationship between the main code components in PIM

The Personal Information Manager (PIM) application is composed of several major components: Main, Controller, Model, and Service packages. Each component has a specific role and they interact with each other to form the complete functionality of the PIM.

Main: This is the entry point of the application. It initializes the `PimDataService` and the controller classes. It also handles the user's command-line input, and based on the input, it delegates the request to the appropriate controller.

Controller: The controller package includes several classes like `NoteController`, `TaskController`, `EventController`, `ContactController`, `SearchController`, and `PrintController`. These classes interpret the user's commands and interact with the `PimDataService` to manipulate the data. Each controller corresponds to a specific type of personal information record (PIR) - note, task, event, or contact.

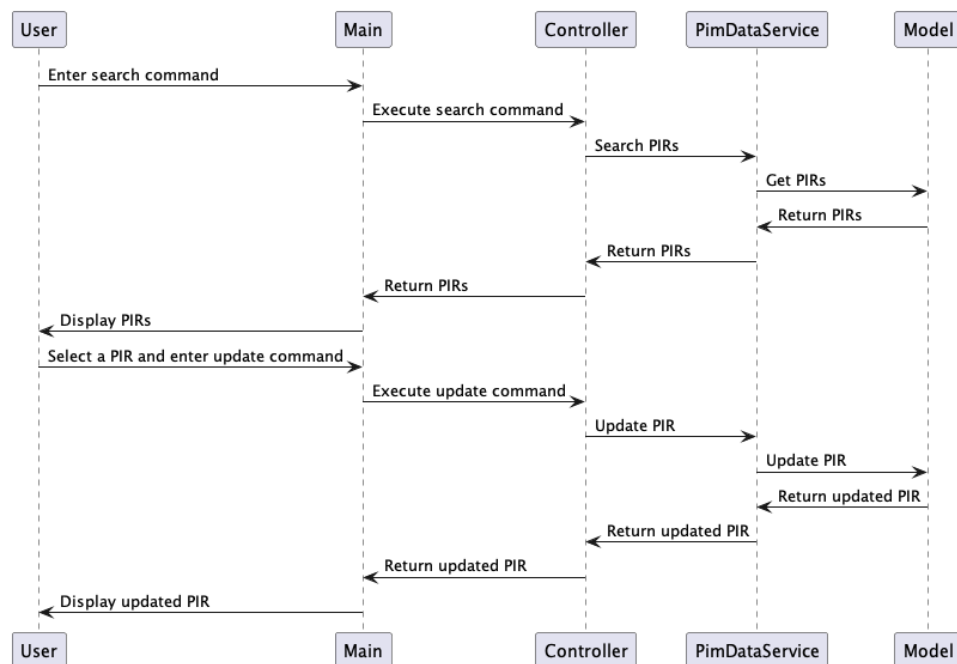
Model: This package includes classes that represent the different types of PIRs: Note, Task, Event, and Contact. These classes mainly encapsulate the data of each record and do not contain business logic. They are used by the `PimDataService` to store data.

Service (`PimDataService`): This class serves as the main service for data handling. It interacts with the model classes to store PIRs and provides methods to add, modify, delete, and retrieve PIRs. It also handles the loading and saving of data from and to the `.pim` file.

The relationships among these components can be summarized as follows:

- Main interacts with the controllers and `PimDataService`.
- Controllers interact with `PimDataService` to manipulate data.
- `PimDataService` interacts with the model classes to store and manage data.

3) A diagram to outline the process of an example use of the PIM, where a user first searches for some personal information records (PIRs) and then updates one of the returned PIRs:



This sequence diagram describes the following process:

1. The user enters a search command on the command line.
2. The main function (Main) receives the user's command and calls the controller (Controller) to execute the command.
3. The controller calls PimDataService to search for PIRs.
4. PimDataService gets the PIR from the model class (Model).
5. The model class returns the PIR to PimDataService.
6. PimDataService Returns the PIR to the controller.
7. The controller returns the PIR to the main function.
8. The main function displays the PIR to the user.
9. The user selects a PIR and enters an update command.
10. The main function receives the user's command and calls the controller to execute the command.
11. The controller calls PimDataService to update the PIR.
12. PimDataService updates the PIR in the model class.
13. The model class returns the updated PIR to PimDataService.
14. PimDataService Returns the updated PIR to the controller.
15. The controller returns the updated PIR to the main function.
16. The main function displays the updated PIR to the user.