Design Document

Members: Huang Tianji (21099573d)[1], Zhang Zhiyuan (21096414d)[1],
Zheng Shouwei (21097982d)[1], Zhou Taiqi (21106717d)[1]

Team: 20 *1. Hong Kong Polytechnic University*

Table of Contents

1 Introduction

The Personal Information Manager (PIM) is a command-line application designed for managing various types of personal information, such as contacts, events, tasks, and quick notes. This design document outlines the system architecture, module design, and interface design. It provides a detailed roadmap for the development and implementation of the PIM.

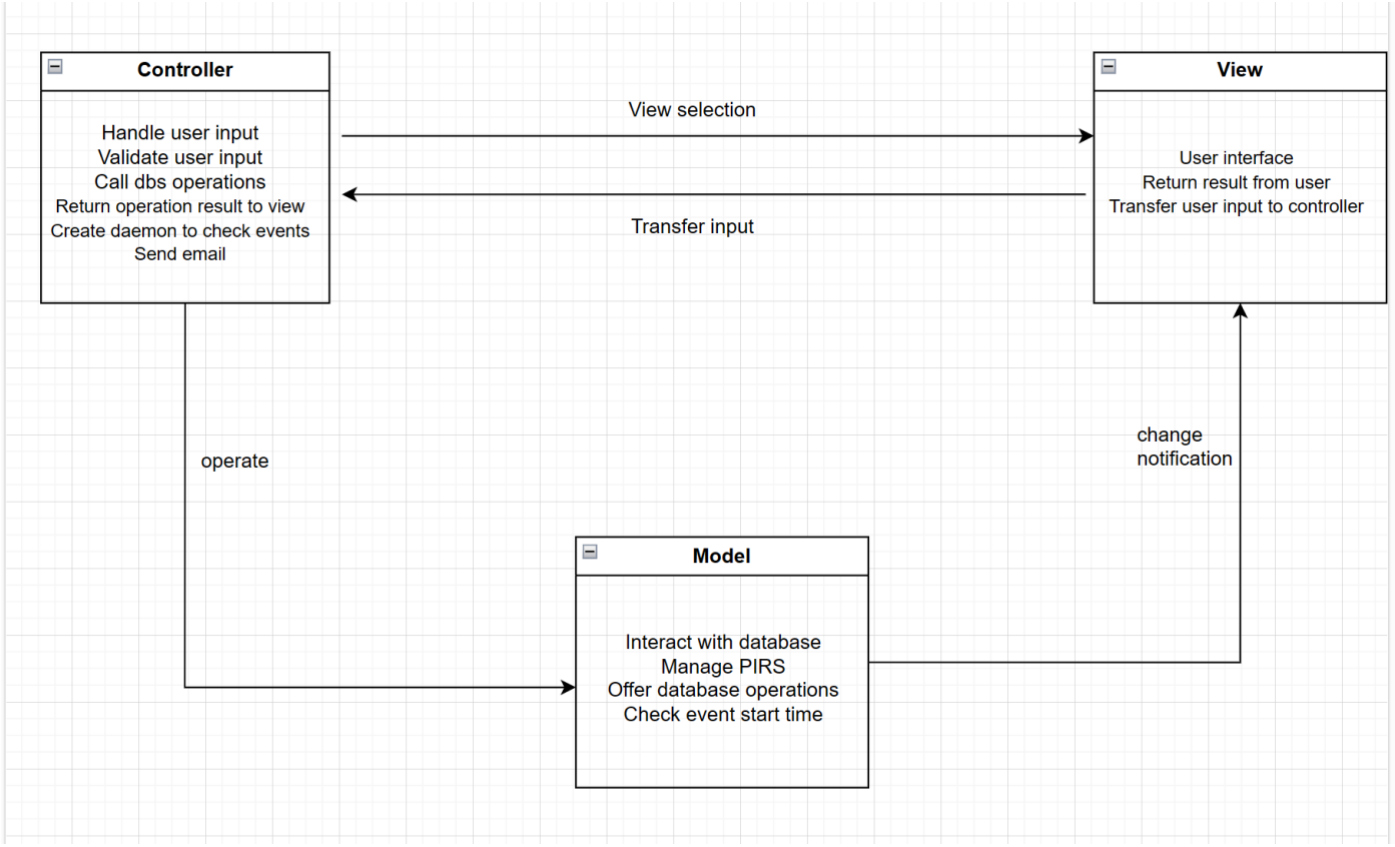
2 Architectural Design

1. MVC Pattern: The Model-View-Controller (MVC) architectural pattern is at the heart of the PIM, a cornerstone in modern software design. This pattern is celebrated for its ability to neatly separate concerns within the application, enhancing maintainability and scalability.

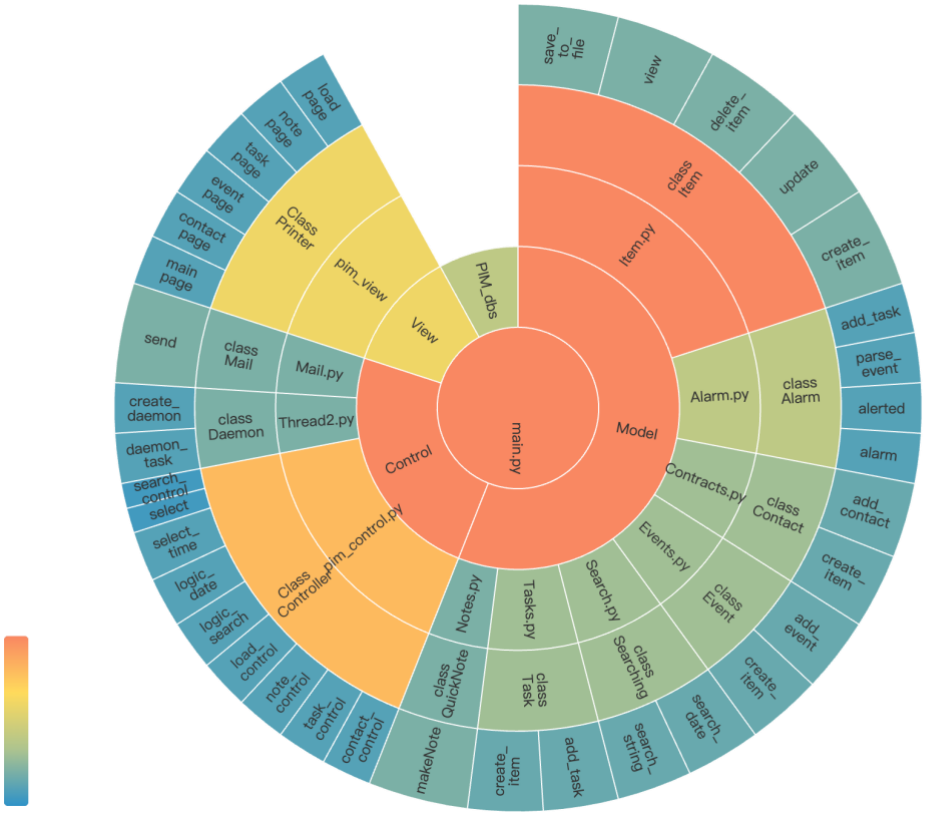
2. Model: The Model is the bedrock of data management in PIM. It encompasses a suite of classes - Contacts, Events, Mails, Tasks, Notes, and Alarm. These classes, stemming from the foundational Item class, are pivotal in managing specific data operations like addition, deletion, and updates. When the user engages in actions such as initiating a new contact or scheduling an event, these classes spring into action. The Item class lays down the basic framework of essential PIM attributes and behaviors. In contrast, its subclasses are tailored to manage data pertinent to their unique contexts.

3. View: Interaction with the user is managed through the View component, primarily via the command-line interface. Central to this module is the Printer class, which is instantiated whenever the system interacts with the user. It upholds a consistent and user-friendly interface, responsible for presenting information and menus.

4. Controller: The Controller class is the conductor of the PIM's symphony, initiating its operations at the program's commencement. It serves as an intermediary between the Model and View, adeptly processing user commands and coordinating the system's responses. It's supported by auxiliary classes like $Mail$. For email notifications, and $Thread2$, which handle background processes, each instantiated as required for specific tasks.



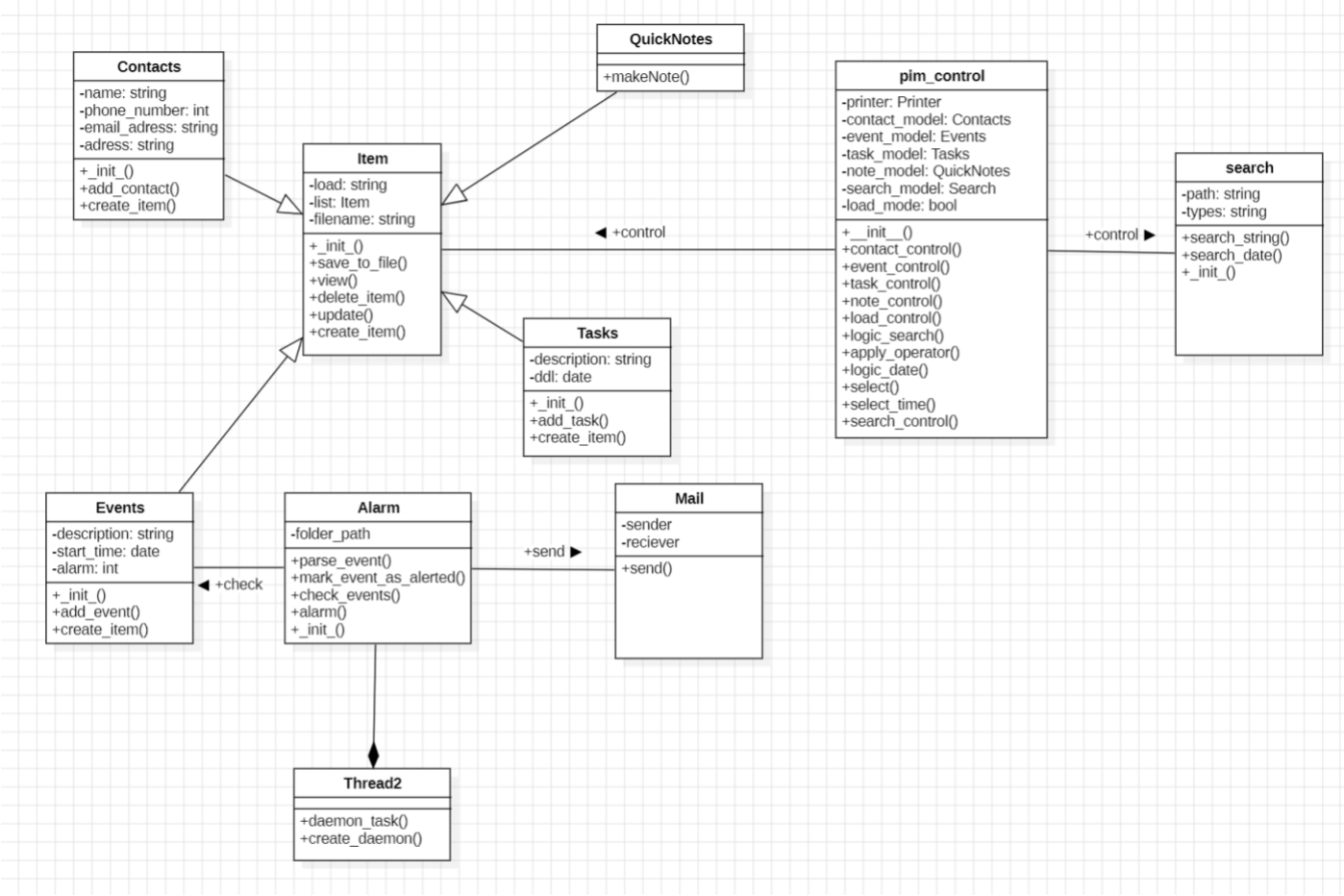
[This MVC diagram describes the PIM's architecture]



[This diagram describes the PIM's detailed architecture]

3 Structure of and relationship among major code components

Since Python does not have explicit divisions and declarations for private, protected, and public access modifiers, our program development is based on the principle that 'everyone is an adult.' During the design and development process, we write our code following the norms of private, protected, and public access (but this does not guarantee restrictions as strict as those in Java).



[UML diagram]

3.1 Model

Item Class

The item class is the superclass of all the pir events (Contacts, QuickNotes, Events, and Tasks). It contains functions such as write, update, view, delete, and save_to_file. And abstract functions such as create_item. Subclassed only needs to inherit from it, use its function, and rewrite some of the abstract methods. Therefore, it reduces the complexity of the code, making it easier to organize and maintain.

Fields include:

- **load:** it specifies the name of the pir events. When the user wants to name a PIR file, the input is treated as transfer as load. So that the system can create, update, and delete in the database via this load. We also offer debug mode (in this condition, the user does not need to specify the name of the file, but with a default value, the current date), which sets load to be '0'
- **list:** it creates an empty list; the list is used to store various pirs. (it means, in a single file, we can input more than one pir)
- **filename**: The path of the file, the relative path. (In PIM_dbs)

Methods include:

init: initialize the fields

```
class Item:
    def __init__(self, pir, load):
        self.load = load
        if load == 0:
            filename = f"{pir}_{datetime.now().strftime('%Y-%m-%d')}.pim"
            self.filename = os.path.join(os.path.dirname(__file__), '..', 'PIM_dbs', filename)
            with open(self.filename, "a") as f:
                pass
        else:
            self.filename = os.path.join(os.path.dirname(__file__), '..', 'PIM_dbs', load)
            with open(self.filename, "a") as f:
                pass

        self.list = []
```

save_to_file: a simple save function; save the file to target directory

```
    def save_to_file(self, item_data):
        with open(self.filename, "a") as f:
            for key, value in item_data.items():
                f.write(f"{key}: {value}\n")
            f.write("------------------\n")
        print(f"{type(self).__name__} added successfully.")
```

view: print out the information of the pir, report FileNotFound when fail to find the file

```
    def view(self):
        try:
            file1 = open(self.filename, 'r')
            Lines = file1.readlines()
            for line in Lines:
                print(line)
        except FileNotFoundError:
            print(FileNotFoundError)
```

delete_item: delete a certain pir by traversing through the file and matching. If the match is successful, print out a notice of success, else inform the user the keywords not found.

```
    def delete_item(self, title, identifier):
        with open(self.filename, 'r') as file:
            items_data = file.read()

        # split different blocks
        items_blocks = items_data.split("------------------\n")
        items_to_delete = f"{title}: {identifier}\n"
        items_to_keep = []

        for item in items_blocks:
            if not item.startswith(items_to_delete):
                items_to_keep.append(item)

        # if the deleted file remain same as before indicates that no deletion
        if len(items_to_keep) == len(items_blocks):
            print(f"{type(self).__name__} not found.")
            return

        # update new
        with open(self.filename, 'w') as file:
            for item in items_to_keep:
                if item.strip() != "":
                    file.write(item)
                    file.write("------------------\n")

        print(f"{type(self).__name__} deleted successfully.")
        file.close()
```

update: this method shows the logic of the working of the system. Every time users apply operations on the system, the update function is called to load the information from the database by assigning the list.

```
def update(self):
    self.list = []  # Clear the current items list to avoid duplicates
    try:
        with open(self.filename, 'r') as file:
            item_data = {}
            for line in file:
                if '------------------' in line:  # '------------------' make a clear boundary between every pir
                    if item_data:
                        self.list.append(self.create_item(item_data))
                        item_data = {}
                else:
                    key, value = line.strip().split(': ')
                    item_data[key] = value
    except FileNotFoundError:
        print(f"The file {self.filename} was not found.")

# an abstract method that need to be overriden by subclass
黄添骥
def create_item(self, item_data):
    raise NotImplementedError("must be overriden")
```

create_item: this method is an abstract method, and its subclass needs to override it. This function will be used in the update function.

Contacts Class

It is used to manage contact type data in PIM. This class enables users to add, update, view, and delete contact information (such as name, phone number, and email address).

Fields include:

- **name**: Contactor name
- **phone**: Contactor phone number
- **email**: Contactor email address
- **address**: Contactor physical address

These fields do not have specific usage currently; it is designed for future maintenance and make the debug procedure easier.

Methods include:

init: initialize the fields.

```
class Contact(Item):
    #  Override init function by specifying more information
    黄添骥
    def __init__(self, name, phone, email, address, load):
        super().__init__("Contacts", load)
        self.name = name
        self.phone = phone
        self.email = email
        self.address = address
```

create_item: override the method of Item by specifying more information

```
    def create_item(self, item_data):
        return Contact(item_data['Name'], item_data['Phone'], item_data['Email'], item_data['Address'], self.load)
```

add_contact: add specific information of pir to file

```
def add_contact(self, name, phone, email, address):
    event_data = {
        "Name": name,
        "Phone": phone,
        "Email": email,
        "Address": address
    }
    self.save_to_file(event_data)
    self.update()
```

Events Class

It is used to manage contact type data in PIM. This class enables users to add, update, view, and delete event information Properties with description, start time, and alert settings. Typically used with the Alarm class and interacts with the PIM database to store related data.

Fields include:

- **description**: Event description
- **start time**: Event start time
- **alert**: Event alert, time to inform users in advance

These fields do not have specific usage currently, it is designed for future maintenance and make the debug procedure more easily.

Methods include:
init: initialize the fields.

```
class Event(Item):
    #  Override init function by specifying more information
    黄添骥
    def __init__(self, description, start_time, alarm, load):
        super().__init__("Events", load)
        self.description = description
        self.start_time = start_time
        self.alarm = alarm
```

create_item: override the method of Item by specifying more information.

```
def create_item(self, item_data):
    return Event(item_data['Description'], item_data['Start Time'], item_data['Alarm'], self.load)
```

add_event: add specific information to the file.

```
def add_event(self, description, start_time, alarm):
    event_data = {
        "Description": description,
        "Start Time": start_time,
        "Alarm": alarm
    }
    self.save_to_file(event_data)
    self.update()
```

QuickNotes Class

Provides a structure for creating and managing quick text annotations. Supports adding and retrieving short text entries. The note class primarily interacts with the local file system to store and load note entries.

No Fields included.

Methods include:

init: call the superclass

```
class QuickNote(Item):
    # Only call superclass' init function, no information added
    黄添骥
    def __init__(self, load):
        super().__init__("QuickNotes", load)
```

makeNote: use while true to repeatedly receive user input. Detect 'END' as the termination of the program. Then offer four types of choice to handle the completion of writing.

```
def makeNote(self):
    QNote = []
    flag = True
    mode = 'r'
    while True:
        line = input()
        if line.strip() == 'END':
            # enter w to rewrite the previous file,
            # enter a to append information
            # enter c to continue writing taking END as an input
            # enter q to quit without save
            confirm = input("Rewrite(w) | Append(a) | Continue(c) | Quit without save(q) ")
            if confirm.lower() in ['a', 'w']:
                mode = confirm
                break
            elif confirm.lower() == 'q':
                flag = False
                print("Quit successfully")
                break
            elif confirm.lower() == 'c':
                print("=====continue=======")
            else:
                print("Wrong choice,try to enter 'END' again")

        else:
            QNote.append(line)

    # indicate user want to end taking note and want to save
    if flag:
        note_text = '\n'.join(QNote)
        try:
            with open(self.filename, mode) as file:
                file.write(note_text)
            print("Note saved successfully")
        except FileNotFoundError:
            print("File save fail")
```

Tasks Class

For task management, tracking to-dos and deadlines. It enables users to manage task type information (including descriptions and due dates). It also interacts with database and search classes.

Fields include:

- **description**: Task description
- **ddl**: deadline of the task

These fields do not have specific usage currently; it is designed for future maintenance and make the debug procedure easier.

Methods include:

init: initialize the fields.

```
class Task(Item):
    #  Override init function by specifying more information
    黄添骥
    def __init__(self, description, ddl, load):
        super().__init__("Tasks", load)
        self.description = description
        self.ddl = ddl
```

create_item: override the method of Item by specifying more information.

```
    def create_item(self, item_data):
        return Task(item_data['Description'], item_data['DDL'], self.load)
```

add_contact: add specific information of pir to file.

```
    def add_task(self, description, ddl):
        event_data = {
            "Description": description,
            "DDL": ddl
        }
        self.save_to_file(event_data)
        self.update()
```

Search Class

Provides methods to search for PIRs using keywords, dates, and logical operators throughout the PIM system. The principle is to scan the PIM database for matches and return the search results to the user interface.

Fields include:

- **type**: indicate what kind of file the query want such as Contacts, Events, Tasks, QuickNotes.
- **path**: the path of PIM_dbs is the place to store files.

Methods include:

init: initialize the fields.

```
def __init__(self, types):
    self.path = os.path.join(os.path.dirname(__file__), '..', 'PIM_dbs')
    self.types = types
```

Search_string: handle with the query(the keywords), and search for the matching file contain this keywords in it.

```
def search_string(self, query):
    match = []
    for filename in os.listdir(self.path):
        if filename.startswith(self.types):
            file_path = os.path.join(self.path, filename)
            with open(file_path, 'r') as file:
                contents = file.read()
                if query in contents:
                    match.append(filename)
    return match
```

Search_date: handle with the query (the date), and search for the matching file contain this keywords in it.

```
def search_date(self, query):
    match = []

    operator, time = query.split(' ', maxsplit=1)

    query_time = datetime.strptime(time, '%Y-%m-%d %H:%M')

    for filename in os.listdir(self.path):
        if filename.startswith(self.types):
            file_path = os.path.join(self.path, filename)
            with open(file_path, 'r') as file:
                contents = file.read()
                events = contents.split('--------------------')
                # check by matching with Start time or DDL
                for event in events:
                    if 'Start Time:' in event:
                        start_time_str = event.split('Start Time: ')[1].split('\n')[0].strip()

                        start_time = datetime.strptime(start_time_str, '%Y-%m-%d %H:%M:%S')
                        # Check if the start time matches the query
                        if ((operator == '<' and start_time < query_time) or
                                (operator == '>' and start_time > query_time) or
                                (operator == '=' and start_time == query_time)):
                            match.append(filename)
```

3.2 Control

pim_control Class

Acts as the command center for PIM. Coordinate the user input and operations on database offered by the Model module. It provides various control model to carry out the related operations.

```
class Controller:
    def __init__(self):
        self.printer = Printer()
        self.contact_model = None
        self.event_model = None
        self.task_model = None
        self.note_model = None
        self.load = 0
        self.load_mode = False  # to identify the current mode, if it is load from dbs, or it is firs time being created
        self.search_model = None
```

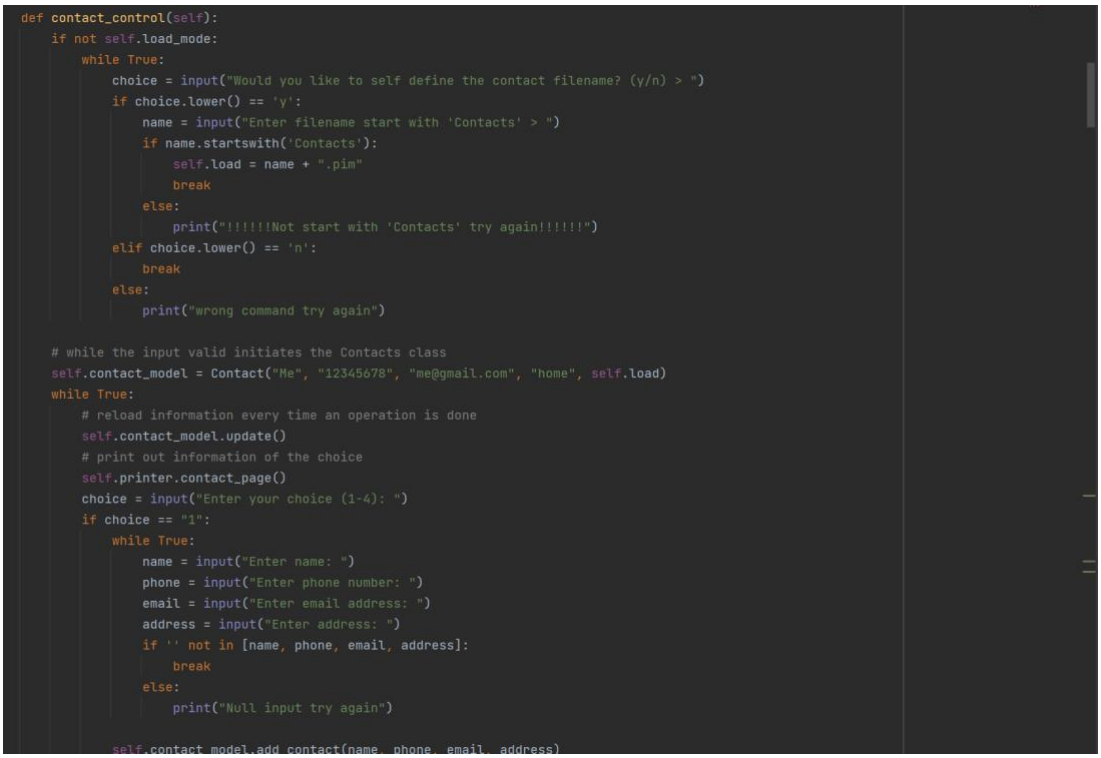
Fields include:

- **printer**: initialize the Printer class, mainly to offer user the interface and notification.
- **contact_model**: the control center of contact type operation.
- **event_model**: the control center of event type operation.
- **note_model**: the control center of note type operation.
- **search_model**: the control center of search type operation.
- **load**: indicate whether the current mode is call by load function or others
- **load_mode**: the filename specifiaction

Methods include (only present major methods with picture):

init: initialize the fields.

contact_control: control the operation with contact model. It mainly include receiving user input and call different function in response to that input. It also include input detecting and notifier to ensure user make the right input and not damage the system.



```
def contact_control(self):
    if not self.load_mode:
        while True:
            choice = input("Would you like to self define the contact filename? (y/n) > ")
            if choice.lower() == 'y':
                name = input("Enter filename start with 'Contacts' > ")
                if name.startswith('Contacts'):
                    self.load = name + ".pim"
                    break
                else:
                    print("!!!!!!Not start with 'Contacts' try again!!!!!!")
            elif choice.lower() == 'n':
                break
            else:
                print("wrong command try again")

    # while the input valid initiates the Contacts class
    self.contact_model = Contact("Me", "12345678", "me@gmail.com", "home", self.load)
    while True:
        # reload information every time an operation is done
        self.contact_model.update()
        # print out information of the choice
        self.printer.contact_page()
        choice = input("Enter your choice (1-4): ")
        if choice == "1":
            while True:
                name = input("Enter name: ")
                phone = input("Enter phone number: ")
                email = input("Enter email address: ")
                address = input("Enter address: ")
                if '' not in [name, phone, email, address]:
                    break
                else:
                    print("Null input try again")

            self.contact_model.add_contact(name, phone, email, address)
```

event_control: it uses the same logic as contact_control. Details are omitted.

task_control: it uses the same logic as contact_control. Details are omitted.

note_control: it uses the same logic as contact_control. Details are omitted.

load_contol: This function does not interact with the user database but calls all the functions in control. It will specify the load_mode to be true to inform that the current call is from load_control.

```python
def load_control(self):
    self.load_mode = True
    while True:
        filename = input("enter the file name or 'q' to exit > ")
        if filename == 'q':
            return
        flag = True
        if not filename.endswith('.pim'):
            print("format error : not end with pim")
            flag = False
        file_path = os.path.join(os.path.dirname(__file__), '..', 'PIM_dbs', filename)
        if not os.path.exists(file_path):
            print(f" {filename} not exit in PIM_dbs ")
            flag = False
        if flag:
            break
    self.load = filename
    if filename.startswith("Contacts"):
        print("========load successfully=========")
        self.contact_control()
    elif filename.startswith("Events"):
        print("========load successfully=========")
        self.event_control()
    elif filename.startswith("QuickNotes"):
        print("========load successfully=========")
        self.note_control()
    elif filename.startswith("Tasks"):
        print("========load successfully=========")
        self.task_control()
    else:
        raise SystemError("unknown error occurs")
    self.load_mode = False
```

search_control: this is a complicated function it coordinates with **logic_search, logic_date, apply_operator, select** and **select_time** function. Depending on different types of search, offer different types of functionality.

```python
def search_control(self):
    while True:
        choice1 = input("input type of pir you want to search quit(q): ")
        if choice1.lower() in ['contacts', 'quicknotes']:
            self.select(choice1)
        elif choice1.lower() in ['tasks', 'events']:
            while True:
                if choice1.lower() == 'events':
                    print("1)search by keywords 2)search by start time : ")
                else:
                    print("1)search by keywords  2)search by DDL : ")
                choice2 = input()
                if choice2 == '1':
                    self.select(choice1)
                    break
                elif choice2 == '2':
                    self.select_time(choice1)
                    break
                else:
                    print("wrong command, try again")
        elif choice1.lower() == 'q':
            return
        else:
            print("wrong command, try again")
```

logic_search: Not important component. Details are omitted. It supports the logic to search the keywords.

logic_date: Not important component. Details are omitted. It supports the logic to search the dates.

apply_operator: Not an important component. Details are omitted. It supports the logic of handling the operators(‘||’, ‘&&’, ‘!’)

select: Not important component. Details are omitted. Identify notes or contacts.

select_time: Not important component. Details are omitted. Identify contacts or events.

3.3 View

Printer class

Since the project is a command-line-based system. There is no specific GUI function. However, to complete the structure of MVC more thoroughly, we employed the Printer class, utilizing the print function to display the user interaction interface.

```
def __init__(self):
    pass

黄添骥
def main_page(self):
    print("Personal Information Manager (PIM)")
    print("1. Contacts")
    print("2. Events")
    print("3. Take Quick Notes")
    print("4. Tasks")
    print("5. Load File")
    print("6. Search")
    print("9. Exit")
```

4 Example Use Case and Diagrams

Users need to start the program from *main.py*. The command line will then display an interactive option. Enter '6' to search.

```
Enter your choice (1-4, 9): 6
```

Enter a type (like 'Contacts).

```
input type of pir you want to search quit(q): Contacts
```

Enter keywords.

```
input keywords (support !, ||, &&): sdc || Vivi || laura
```

System returns matching file name.

```
file find as follows:
['Contacts_zhengshouwen.pim', 'Contacts4.pim', 'Contacts_2023-11-11.pim', 'Contacts_2023-11-12.pim', 'Contacts1.pim', 'Contacts_2023-11-23.pim']
```

Enter '7' to load and enter the file name.

```
Enter your choice (1-4, 9): 5
enter the file name or 'q' to exit > Contacts_zhengshouwen.pim
========load successfully=========
1. Add Contact
2. View Contacts
3. Delete Contact
4. Back to Main Menu
```

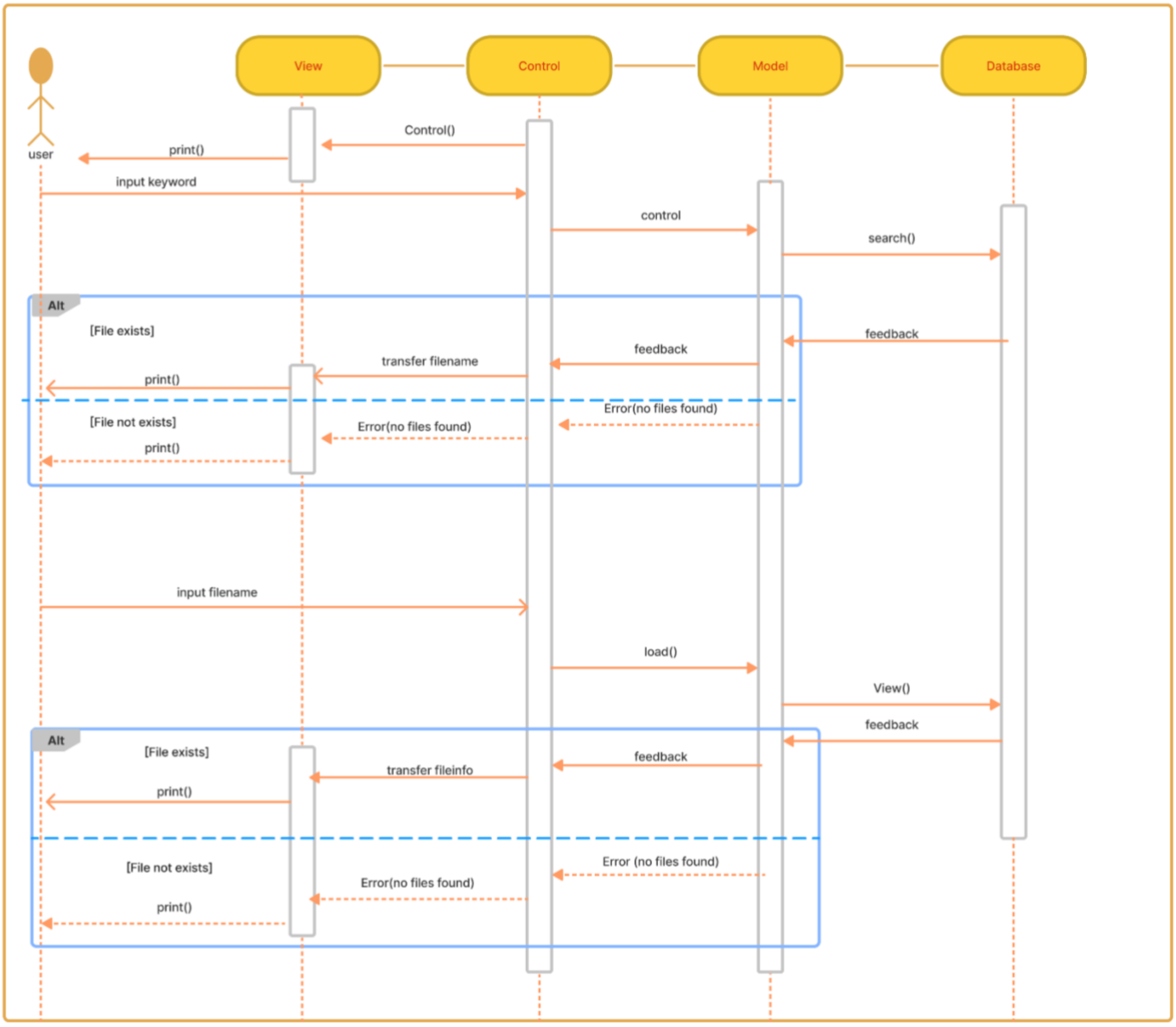
Enter '2' to view the contents of the file.

```
Enter your choice (1-4): 2
```

System outputs:

```
Name: sdcj

Phone: dsacdasc

Email: sdcasdc

Address: sadc

-----------------
```

According to the above process, we can further draw the corresponding simple sequence diagram：



[Sequence diagram]

5 Conclusion

The design of the PIM application is aimed at providing a robust and user-friendly tool for personal information management. The MVC architecture facilitates a clean separation of concerns, enhancing maintainability and scalability. The design ensures a comprehensive coverage of functionalities as required by the system's objectives.