Design Document for

Personal Information Manager (PIM)

COMP3211 Software Engineering Project

Group 26

Han Wenyu	21097519D
Li Yangxiaoxuan	20102514D
Tam Chung Man	21073064D
Zhou Siyu	21094655D

Content

Diagram describe the PIM's architecture

Architecture Patterns

The Model-View-Controller (MVC) separates the presentation from interaction from the system data. The entire system is mutually divided into three logic blocks. The Model component block manages all data, and accepts actions that transmit user commands. The View component block defines the user interface. The Controller component block only performs user instructions, and these operations are transmitted to View and Model.

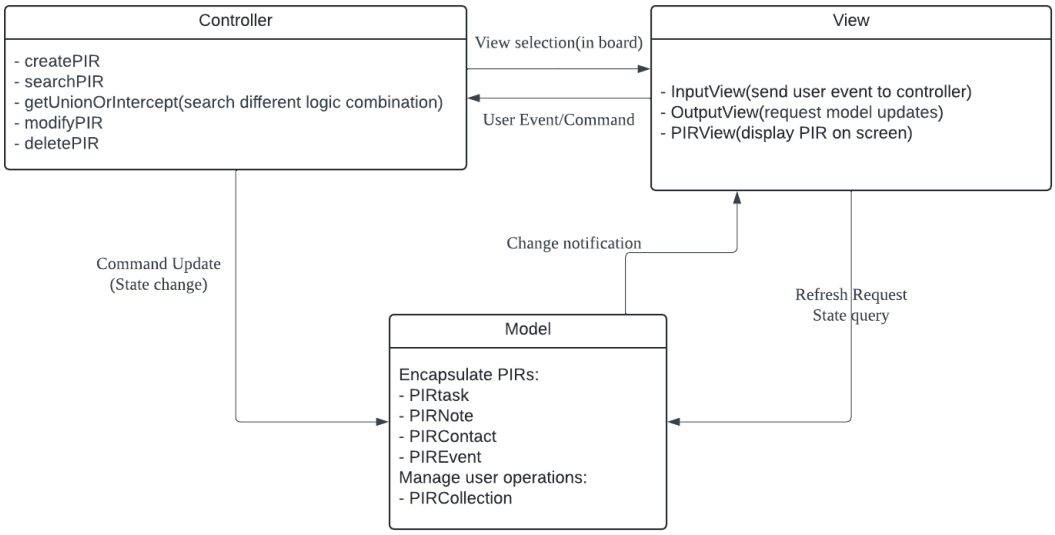
Reasons

The generic architecture pattern we picked for our PIM system is the Model-View-Controller (MVC). It helps in clear structuring in different components of code, enhancing modularity and maintainability.

The system is structured into three logical components that interact with each other, the model component manages PIRs and business logic, the view component defines how the information is presented to the user, and the controller component manages user commands and passes these interactions to the view and the model.

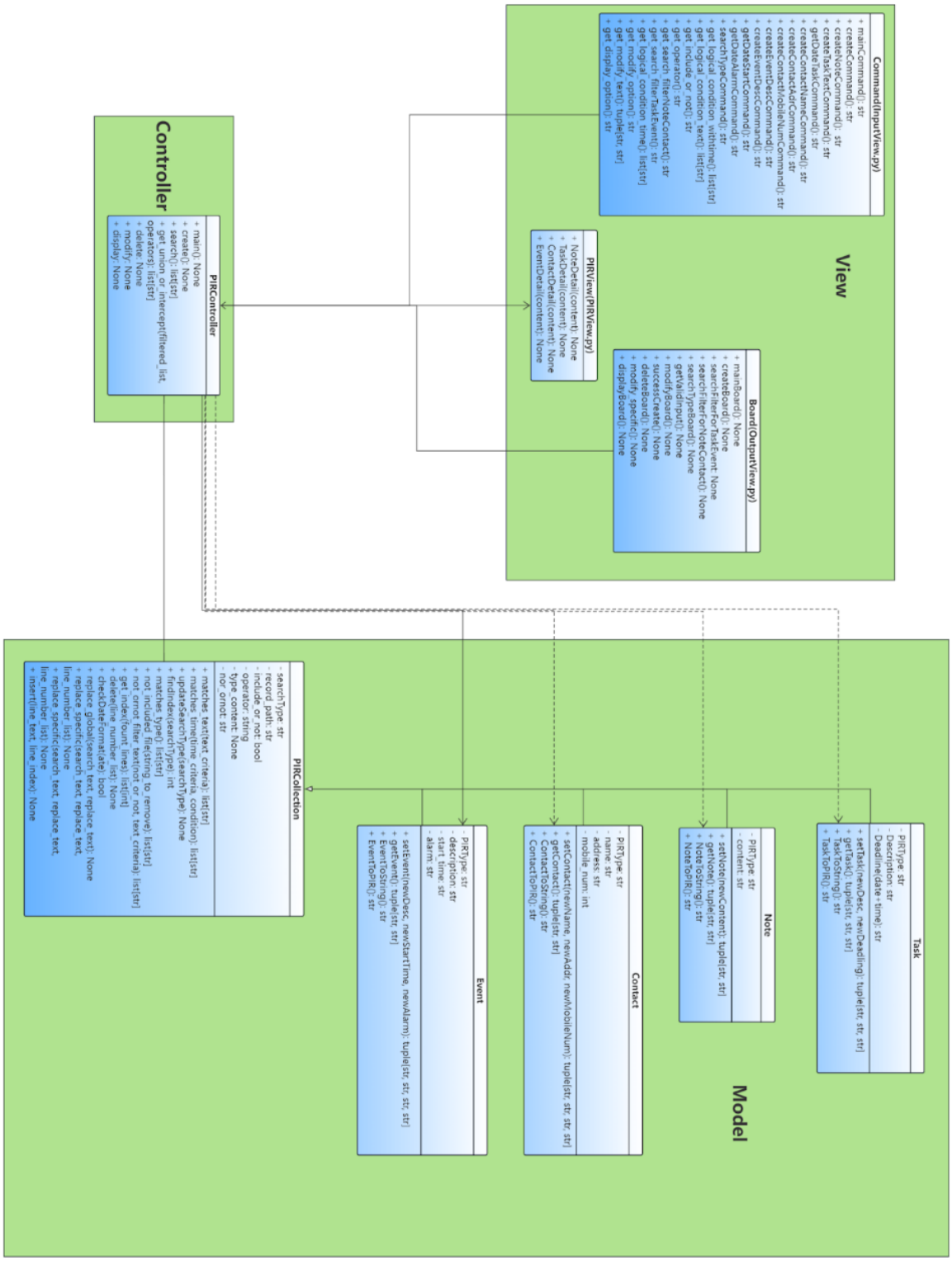
The separation of system allows each component to be developed and maintained independently, which makes it easier to code resumes for adapting or extending functions, the minimal impact on other existing models can reduce the likelihood of introducing bugs. Also, the pattern also becomes easier to write unit tests for each component to ensure the correctness. Overall, the MVC patterns as a widely-used architecture, making developers better responsible for different components to have clearer organization for designing the PIM.

Overall Structure: Model-View-Controller Diagram



The overall architecture is divided into three parts including model, view, and controller(See the Figure above). For the Model part, it contains encapsulated PIRs (PIRTask, PIRNote, PIRContact, PIREvent) and user operations of all types of PIR (PIRCollection). For the View part, it mainly contains InputView(receiving users' operation), OutputView(request model update, hints), PIRView(display PIR on screen) and other visualized contents with interface for action in PIR creation, search, modification, deletion. For the Controller part, it mainly receives users' operation with interface in View parts, and turns command into logic and then does the operation to PIR using the model.

Structure of and relationships among major code components



Model

PIRNote

Provide a *Note* class for storing PIR in *Note* type.

__init__(content): None - This method is to initialize different types of variables used for PIR *Note* type, such as PIRType, description, and deadline

```
def __init__(self, content):
    self.PIRType = 'Note'
    self.content = content
```

setNote(newContent): tuple[str, str] - This method is to set the *Note* with input *newContent*.

```
# create (return info)
def setNote(self, newContent):
    self.content = newContent
    return self.PIRType, self.content
```

getNote(): tuple[str, str] - This method is to get *Note* information by returning *PIRType*, and *content*.

```
# read(return info)
def getPIMNote(self):
    return self.PIRType, self.content
```

NoteToString(): str - This method is to get *Note* information in the string information in displaying formatting information on UI.

```
# tostring(return string to printed PIR record on interface)
def NoteToString(self):
    string = self.PIRType + ":\nContent: " + self.content
    return string
```

NoteToPIR(): str - This method is to get *Task* string information as PIR form into the "*record.pim*" file.

```
# note to PIR record form(return in PIR form)
def NoteToPIR(self):
    return self.content
```

PIRTask

Provide a *Task* class for storing PIR in *Task* type.

__init__(description, deadline): None - This method is to initialize different types of variables used for PIR *Note* type, such as *PIRType*, *description*, and *deadline.*

```
def __init__(self, description, deadline):
    self.PIRType = 'Task'
    self.description = description
    self.deadline = deadline
```

setTask(newDesc, newDeadline): tuple[str, str, str] - This method is to set the *Task* with input *newDesc*, and *newDeadline*.

```
# create(return info)
def setTask(self, newDesc, newDeadline):
    self.description = newDesc
    self.deadline = newDeadline
    return self.PIRType, self.description, self.deadline
```

getTask(): tuple[str, str, str] - This method is to get *Task* information by returning *PIRType*, *description*, and *deadline*.

```
# read(return info)
def getPIMTask(self):
    return  self.PIRType, self.description, self.deadline
```

TaskToString(): str - This method is to get *Task* information in the string information in displaying formatting information on UI.

```
# tostring(return string to printed PIR record on interface)
def TaskToString(self):
    string = self.PIRType + ":\nDescription: " + self.description + "\nDeadline: " + self.deadline
    return string
```

TaskToPIR(): str - This method is to get *Task* string information as PIR form into the "*record.pim*" file.

```
# task to PIR record form(return in PIR form)
def TaskToPIR(self):
    return self.description + "," + self.deadline
```

PIRContact

Provide *Contact* class for storing PIR in *Contact* type.

__init__(name, address, mobile_num): None - This method is to initialize different types of variables used for PIR *Contact* type, such as *PIRType*, *name*, *address*, and *mobile_num.*

```
def __init__(self, name, address, mobile_num):
    self.PIRType = 'Contact'
    self.name = name
    self.address = address
    self.mobile_num = mobile_num
```

setContact(newName, newAddr, newMobileNum): tuple[str, str, str, str] - This method is to set the *Contact* with input *newName*, *newAddr*, and *newMobileNum*.

```
# create (return info)
def setContact(self, newName, newAddr, newMobileNum):
    self.name = newName
    self.address = newAddr
    self.mobile_num = newMobileNum
    return self.PIRType, self.name, self.address, self.mobile_num
```

getContact(): tuple[str, str] - This method is to get *Contact* information by returning *PIRType*, *name*, *address*, and *mobile_num*.

```
# read(return info)
def getContact(self):
    return self.PIRType, self.name, self.address, self.mobile_num
```

ContactToString(): str - This method is to get *Contact* information in the string information in displaying formatting information on UI.

```
# tostring(return string to printed PIR record on interface)
def ContactToString(self):
    string = self.PIRType + ":\nName: " + self.name + "\nAddress:" + self.address + "\nMobile Number: " + self.mobile_num
    return string
```

ContactToPIR(): str - This method is to get *Contact* string information as PIR form into the "*record.pim*" file.

```
# contact to PIR record form(return in PIR form)
def ContactToPIR(self):
    return self.name + "," + self.address + "," + self.mobile_num
```

PIREvent

Provide *Event* class for storing PIR in *Event* type.

__init__(description, start_time, alarm): None - This method is to initialize different types of variables used for PIR *Event* type, such as *PIRType*, *description*, *start_time*, and *alarm*.

```
def __init__(self, description, start_time, alarm):
    self.PIRType = 'Event'
    self.description = description
    self.start_time = start_time
    self.alarm = alarm
```

setEvent(newDesc, newStartTime, newAlarm): tuple[str, str, str, str] - This method is to set the *Event* with input *newDesc*, *newStartTime*, and *newAlarm*.

```
# create event (return info)
def setEvent(self, newDesc, newStartTime, newAlarm):
    self.description = newDesc
    self.start_time = newStartTime
    self.alarm = newAlarm
    return self.PIRType, self.description, self.start_time, self.alarm
```

getEvent(): tuple[str, str] - This method is to get *Event* information by returning *PIRType*, *name*, *address*, and *mobile_num*.

```
# read (return info)
def getPIMEvent(self):
    return self.PIRType, self.description, self.start_time, self.alarm
```

EventToString(): str - This method is to get *Event* information in the string information in displaying formatting information on UI.

```
# tostring(return string to printed PIR record on interface)
def EventToString(self):
    string = self.PIRType + ":\nDescription: " + self.description + "\nStart Time:" + self.start_time + "\nAlarm Time: " + self.alarm
    return string
```

EventToPIR(): str - This method is to get *Event* string information as PIR form into the "*record.pim*" file.

```
# event to PIR record form(return in PIR form)
def EventToPIR(self):
    return self.description + "," + self.start_time + "," + self.alarm
```

PIRCollection

Provide a *PIRCollection* class for logical judgment of correctness

__init__(): None - This method initialized different types of variables used for operations for the four PIR types collections, such as *searchType*, *record_path*, *include_or_not*, *operator*, *type_content*, *not_ornot*.

```
def __init__(self):
    self.searchType = "All"
    self.record_path = os.path.join(os.path.dirname(os.path.dirname(os.path.abspath(__file__))),'records.pim')
    self.include_or_not = False
    self.operator = "&&"
    self.type_content = None
    self.not_ornot = ""
```

match_text(text_criteria): list[str] - This method is for search text logic. It first defines *found_lines* as *list[str]* form, then splits content received by users, then checks if record contains it, then saves found lines into *found_lines* list.

```
def matches_text(self,text_criteria):
    found_lines = []
    for line in self.type_content:
        line = line.strip()
        parts = line.split(",")
        parts1 = line.split(" ")
        if text_criteria in parts or text_criteria in parts1:
            found_lines.append(line)
    return found_lines
```

matches_times(time_criteria, condition): list[str] - This method is for search time logic. This method uses the *checkDateFormat* method to check date format, and check different kinds of condition, then save into *found_lines* list.

```
def matches_time(self,time_criteria,condition):
    found_lines = []
    for line in self.type_content:
        parts = line.split(",")
        for part in parts:
            if not self.checkDateFormat(part.strip()):
                continue
            time = datetime.strptime(time_criteria.strip(),"%Y/%m/%d %H:%M")
            value = datetime.strptime(part.strip(),"%Y/%m/%d %H:%M")
            condition = condition
            if condition == "<" and value < time:
                found_lines.append(line.strip())
            elif condition == ">" and value > time:
                found_lines.append(line.strip())
            elif condition == "=" and value == time:
                found_lines.append(line.strip())
            else:
                pass
    return found_lines
```

updateSearchType(searchType): None - This method is for updating search type in initialized variable *seachType*.

```
def updateSearchType(self, searchType):
    self.searchType = searchType
```

findIndex(searchType): int - This method is for finding the index in a saved "*record.pim*" file, and then return line index range of searched PIR type.

```
def findIndex(self, searchType):
    with open(self.record_path, "r") as file:
        lines = file.readlines()
    if searchType == 1:
        word_to_find = "Note"
    elif searchType == 2:
        word_to_find = "Task"
    elif searchType == 3:
        word_to_find = "Contact"
    elif searchType == 4:
        word_to_find = "Event"
    else:
        word_to_find = "End"
    for i, line in enumerate(lines):
        if word_to_find in line:
            index = i
            return index
```

matches_type(): list[str] - This method is for matching type of PIR records, checking a line index range of a specific PIR type in "*records.pim*" file using *findIndex(searchType)* method

```
# match type
def matches_type(self):
    with open(self.record_path, "r") as file:
        lines = file.readlines()
        if self.searchType == 1 or self.searchType == 2 or self.searchType == 3 or self.searchType == 4:
            self.type_content = lines[self.findIndex(self.searchType)+1:self.findIndex(self.searchType+1)]
        else:
            self.type_content = lines
    return self.type_content
```

not_included_file(string_to_remove): list[str] - This method is for checking if a line is included in the "*record.pim*" file during deletion operation.

```
# check if included in file
def not_included_file(self,strings_to_remove):
    found_lines = []
    for line in self.type_content:
        if not any(string in line for string in strings_to_remove):
            found_lines.append(line.rstrip())
    return found_lines
```

not_ornot_filter_text(not_ornot, text_criteria): list[str] - This method is to filter the text using function *not_included_file(strings_to_remove)* and function *matches_type()*, then return a *list[str]*.

```
def not_ornot_filter_text(self,not_ornot,text_criteria):
    if not_ornot == "-":
        return self.not_included_file(self.matches_text(text_criteria))
    else:
        return self.matches_text(text_criteria)
```

not_ornot_filter_time(not_ornot,time_criteria, condition): list[str] - This method is to filter time using function not_included_file(strings_to_remove) and function matches_time(), then return a list[str] of record containing time.

```
def not_ornot_filter_time(self,not_ornot,time_criteria,condition):
    if not_ornot == "-":
        return self.not_included_file(self.matches_time(time_criteria,condition))
    else:
        return self.matches_time(time_criteria,condition)
```

get_index(found_lines): list[int] - This method is to get an *index_list* of *found_lines* in "*record.pim*" file.

```
def get_index(self,found_lines):
    with open(self.record_path, 'r') as file:
        lines = file.readlines()
    index_list = []
    for index, line in enumerate(lines):
        line = line.strip()
        for found_line in found_lines:
            if found_line == line:
                index_list.append(index)
    return index_list
```

delete(line_number_list): None - This method is to directly delete searched lines in got *line_number_list* from *get_index(found_lines) method* then delete those records in "*records.pim*" file.

```
#  delete a line from lines list, then copy left lines into file
def delete(self,line_number_list):
    with open(self.record_path, 'r') as file:
        lines = file.readlines()
    for line_number in sorted(line_number_list,reverse=True):
        if line_number > 0 and line_number <= len(lines):
            del lines[line_number]
    with open(self.record_path, 'w') as file:
        file.writelines(lines)
```

checkDateFormat(date): bool - This method is to check if the date format of the user's command is correct or not by importing from the *datetime* module in class *datetime*.

```
def checkDateFormat(self,date):
    if date is None:
        return False
    date_format = "%Y/%m/%d %H:%M"
    try:
        datetime.strptime(date.strip(), date_format)
    except ValueError:
        return False
    return True
```

replace_global(search_text, replace_text): None - This method is to replace whole parts of the "*records.pim*" file by replacing *searched_text* by *replace_text*.

```
def replace_global(self,search_text, replace_text):
    with open(self.record_path,'r') as file:
        data = file.read()
        data = data.replace(search_text, replace_text)
    with open(self.record_path,'w') as file:
        file.write(data)
```

replace_specific(search_text, replace_text,line_number_list): None - This method is to replace in a specific line of the "*records.pim*" file by replacing *searched_text* by *replace_text*.

```
def replace_specific(self,search_text, replace_text,line_number_list):
    with open(self.record_path,'r') as file:
        data = file.readlines()
    for line_number in line_number_list:
        data[line_number] = data[line_number].replace(search_text,replace_text)
    with open(self.record_path,'w') as file:
        file.writelines(data)
```

insert(line_text, line_index): None - This method is to insert a specific line into the "*records.pim*" file.

```
def insert(self,line_text, line_index):
    # Read the existing contents of the file
    with open(self.record_path, 'r') as file:
        lines = file.readlines()

    # Insert the new line at the specified position
    lines.insert(line_index , line_text + '\n')

    # Write the modified contents back to the file
    with open(self.record_path, 'w') as file:
        file.writelines(lines)
```

View

Command（InputView.py):

mainCommand(): str - This method is for getting command from users coming after the main board

- "Getting Command" methods from users in creation operation are as follows. These methods are all using *input("some string")*, and the return type is string
    - *createCommand(): str*
    - *createNoteCommand(): str*
    - *createTaskTextCommand(): str*
    - *getDateTaskCommand(): str*
    - *createContactNameCommand(): str*
    - *createContactAdrCommand(): str*
    - *createContactMobileNumCommand(): str*
    - *createEventDescCommand(): str*
    - *createEventDescCommand(): str*
    - *getDateStartCommand(): str*
    - *getDateAlarmCommand(): str*
    - *searchTypeCommand(): str*

- "Getting Command" methods from users in search operation are as follows. These methods are all using *input("some string")*, and the return type is string
    - *get_logical_condition_withtime(): list[str]*
    - *get_logical_condition_text(): list[str]*
    - *get_include_or_not(): str*
    - *get_operator(): str*
    - *get_search_filterNoteContact(): str*
    - *get_search_filterTaskEvent(): str*
    - *get_logical_condition_time(): list[str]*

- "Getting Command" methods from users in modification operation are as follows. These methods are all using *input("some string")*, and the return type is string
    - *get_modify_option(): str*
    - *get_modify_text(): tuple[str, str]*
- "Getting Command" methods from users in display operation are as follows. These methods are all using *input("some string")*, and the return type is string
    - get_display_option(): str

Board (OutputView.py)

“Giving Out Boards / Instructions / Hints” methods that system give are as follows. These methods are all using *print(“some string”).*

mainBoard(): None

createBoard(): None

searchFilterForTaskEvent: None

searchFilterForNoteContact(): None

searchTypeBoard(): None

getValidInput(): None

modifyBoard(): None

successCreate(): None

deleteBoard(): None

modify_specific(): None

displayBoard(): None

PIRView: These methods make up formats for different types of PIR form and are used by display methods in the Controller.

NoteDetail(content): None

TaskDetail(content): None

ContactDetail(content): None

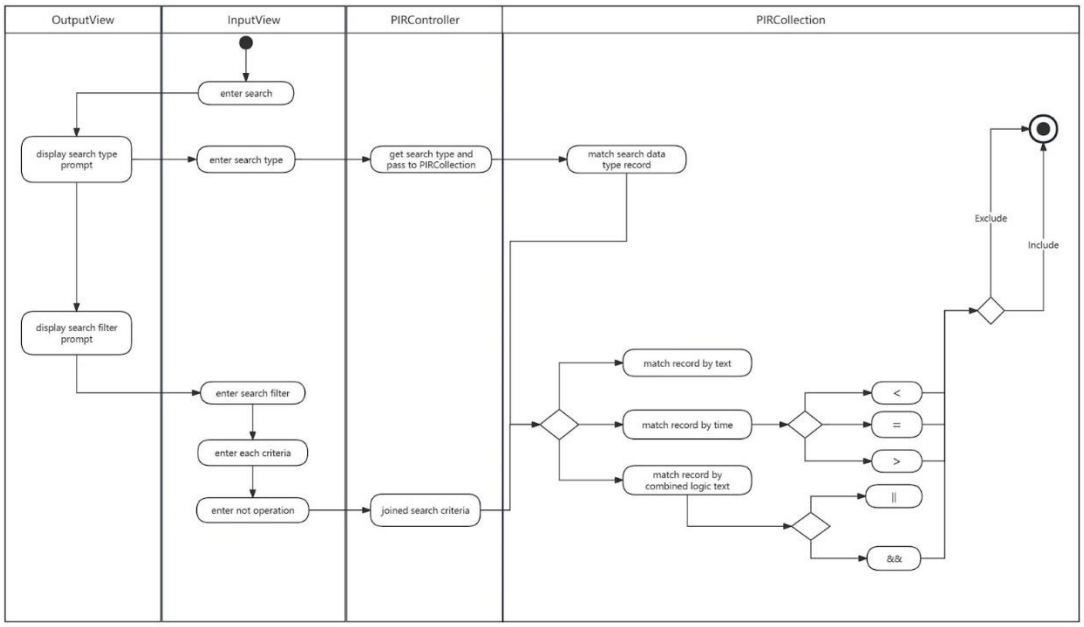
EventDetail(content): None

Controller

PIRController:

- *main(): None* - This method functionally provides a home page directed into different operations: creation of PIR, search of PIR, deletion of PIR, display of PIR.
- *create(): None* - This method functionally provides the creation function of *Notes*, *Task*, *Contact* and *Event.*
- *search(): list[str]* - This method functionally provides the searching functions of *Notes*, *Task*, *Contact* and *Event.*
- get_union_or_intercept(filtered_list, operators): list[str] - This method functionally provides operations of the way for combining different kinds of searching conditions/criterias together, and then returning the final list of PIR records in "*record.pim*" files.
- *delete(): None* - This method functionally provides the deletion function of *Notes*, *Task*, *Contact* and *Event* PIR.
- *modify(): None* - This method functionally provides the modification function of *Notes*, *Task*, *Contact* and *Event* PIR.
- *display(): None* - This method functionally provides the modification function of *Notes*, *Task*, *Contact* and *Event* PIR.

Example use (Activity Diagram)



InputView prompts the user to enter the search type.

The PIRController class receives the search type input from InputView and passes it to the PIRCollection class.

PIRCollection uses the updateSearchType() method to update the search type and narrow down the search result scope to the specific data type.

PIRCollection uses the matches_type() method to filter the data based on the search type.

OutputView displays a search filter prompt and asks the user to input the search filter type.

For Note and Contact search types, the user has two filter choices: single text filter or combined logic filter.

For Task and Event types, which have a time attribute, there is an additional filter choice: single time search filter. This filter can be used to filter out time records before, after, or equal to the specified time criteria.

InputView asks the user to input the search criteria and the logical operation (include or exclude) for the criteria.

If the user chooses the combined logic filter, PIRController receives both the search criteria and the logical operator from the user.

PIRCollection applies the logical operation to each criteria one by one, filtering the data accordingly.