Design Document
CSC207 Project (Group-059)

Specification

Our chosen domain was a productivity application that would combine the features of a typical scheduling app with a Pomodoro timer. The program now contains the following features:

1. Tasks (events and to-do) with descriptions and due dates/times
2. A to-do list
3. A timeline with time blocks
4. Adding tasks to the to-do list and timeline
5. Printing the to-do list and timeline
6. Editing attributes of individual tasks
7. A Pomodoro timer
8. A list of suggestions for the day, sorted by due date
9. Saving and loading lists in .csv format

The program currently runs in the command line. At the start of the program, a saved TaskList is loaded if it exists. If it doesn't, then an empty TaskList is created. At any point, the user can read out the Timeline or TaskList, get suggestions on what Task to do, and start a Pomodoro timer with text notifications at the end of work/break periods. The user can add or delete Tasks to and from the TaskList and edit their name, description, and start/end date and time. If it is an EventTask, the program also adds these Tasks to the Timeline. When the user exits the program, the TaskList is saved as a .csv file which is loaded once the program is run again.

SOLID

- **SRP**: Starting in Phase 1, more emphasis was placed on fulfilling the Single Responsibility Principle. While it is difficult to ensure that all classes follow the principle, we would occasionally either split up or relocate a piece of code to a new class. For example, the cleanup of UserFunctions resulted in the creation of new classes, such as TaskCreator and TaskEditInUI, which ensures that the class only deals with processing inputs and returning outputs.
- **OCP**: We made more frequent use of abstract and interface classes to satisfy the Open/Closed Principle. Turning the original Task class into an abstract class allows us to add new types of Tasks without needing to repeatedly modify the Task class. Another instance of this principle, noted in the Refactoring section, is updating EditStrategy with an overwrite method, so creating a new strategy does not require modifying the original strategy interface.

- **LSP**: The Liskov Substitution Principle is also difficult to fully ensure, but we have made sure to adhere to it in more important instances, such as different types of Tasks. Methods which use Task objects are able to deal with EventTasks and TodoTasks in the same way.
- **ISP**: Every use of interfaces (such as in design patterns) was related to functionality, making sure each method was used in classes that implemented them. For example, the design pattern interfaces and Storable all contain one or two methods which are specifically meant to be used in all implementing classes.
- **DIP**: The Dependency Inversion Principle is also related to our adjustments to follow Clean Architecture, which can be seen below. For example, the responsibility of the CSVManager class (a gateway) is shifted to an interface to avoid having Storable objects (use cases) depend on it when it loads in data. Storable objects now depend on abstractions.

Clean Architecture

Phase 1: We revised our CRC cards in accordance with the advice of our TA. Most of our revisions involved rearranging and splitting classes to fit the layers of Clean Architecture. One instance of this, as mentioned above, is using an interface to deal with storage, which avoids letting use cases depend on the CSVManager when loading data from CSV files.
Phase 2: When necessary, we would revise our class structure similar to how we did in Phase 1. In our classes, we made sure that dependencies would not skip two layers, which used to be the case in UserFunctions. Using the advice of our TA, we implemented more controllers to more precisely handle inputs between UserFunctions and use cases. Our newly revised CRC cards (though somewhat dense and cluttered) can be viewed here.

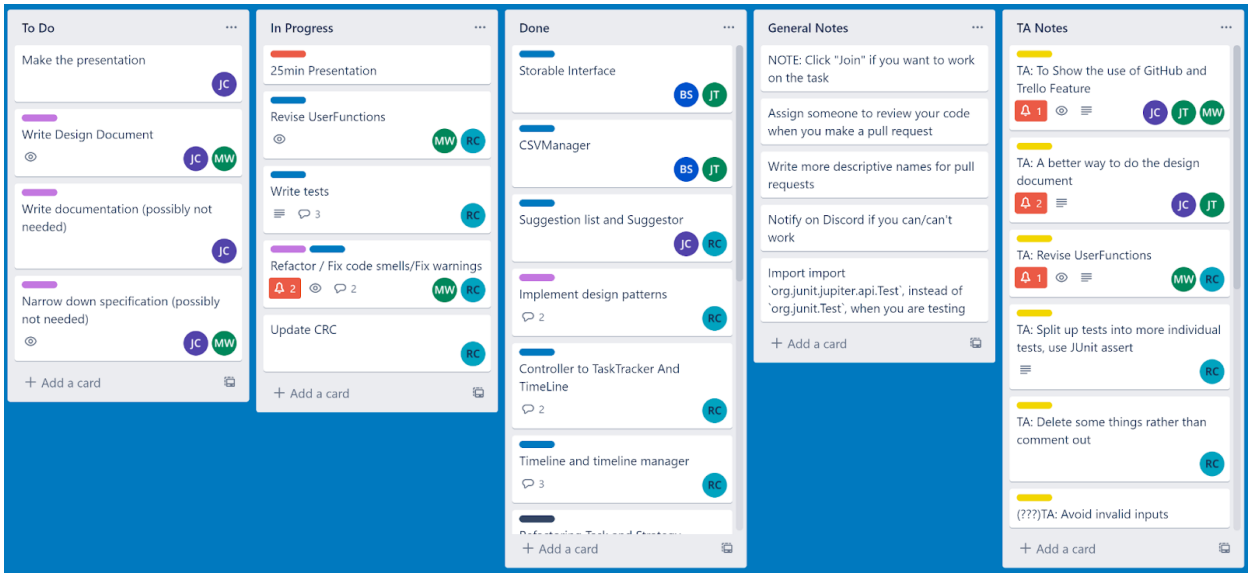
Design Patterns

For our project, we made use of the Strategy and Observer Design Patterns. The interfaces for these design patterns can be found as “EditStrategy” and “TaskObserver.” The Strategy Design Pattern is used to account for the different ways a user would edit a task. The Observer Design Pattern is used to notify other classes about changes made in the list of Tasks in TaskTracker.

Use of GitHub Features and Trello

Github: The process of adding an implementation or change to the repository typically involved members working on their individual branches and submitting pull requests. Each branch has a descriptive name so we could easily tell what changes have been made. When a pull request was made, one or two members were requested to review the code before merging. In Phase 1, we would occasionally just commit to the main branch if other members were inactive, though improved communication meant that this was not necessary in Phase 2.

Trello: We used a Trello board to split up and assign tasks. The tasks were sorted by status (To-do, In Progress, Done), and it also included more general notes for work. Each task would have one or two members assigned to it. A screenshot of the board can be seen below.



Code Style and Documentation

Our code now contains more documentation than it did in Phase 1, though the documentation still remains fairly sparse. As the number of files increase, it becomes more difficult to check if each class has sufficient documentation written by its respective member, though the most important classes likely have sufficient comments. This time, we made sure to delete more code rather than commenting it out once we were confident it would remain unused. Some warnings still remain with code that we planned to use if given more time.

Testing

Unlike Phase 1 where we were unable to write sufficient tests, we were able to include a wider range of testing in Phase 2 by having members write tests for their respective features. Although we have not tested all parts/uses of the program, our tests account for the most important components. Some classes, such as the UserFunctions/UI, are more difficult to test since there isn't a feasible method of testing user input/output. Overall, each feature listed in our specification has been tested. It's important to note, however, that each test in the test classes must be run individually to pass.

Refactoring

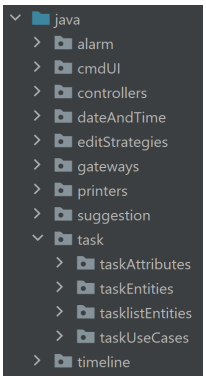
During Phase 2, there were more instances of refactoring during implementation rather than the original plan of refactoring at the end. Here are some examples of pull requests which focused on refactoring:

- We started Phase 2 by adding Gradle to our project and updating the packaging so it had a more consistent "by-feature" format (see pull request)
- With the advice of our TA, the EditStrategy was updated to follow the OCP by removing if-statements and instead using a method to overwrite (see pull request)
- UserFunctions (a UI-related object) was updated to follow Clean Architecture by removing uses of entities such as Task, TaskList, and Timeline (see pull request)

We have noted some code smells that still remain, such as long classes/methods/parameter lists (TaskEditInUI, for example, includes a very long if-statement that is difficult to reduce) which we were unable to fix before the deadline.

Code Organization

In Phase 1, we chose to roughly sort our classes by feature. We chose this format so that classes related to each other are in the same exact place, removing unnecessary time spent looking for files. Based on our TA's advice, we avoided putting all interfaces and abstract classes in the same folder, though we still have controllers and gateways grouped together. Our current packaging format can be seen in the image on the right.



Functionality

By the end of Phase 2, we were able to accomplish most of the functionality listed in our initial specification. The features in our initial specification which weren't included were reminders, sub-tasks, and evaluations. The Specification section above is the list of all features currently implemented and accessible in our program. After Phase 1, the following features have been added to our program:

1. Editing attributes of individual tasks
2. A Pomodoro timer
3. A list of suggestions for the day, sorted by due date
4. Saving and loading lists in .csv format

In the future, some features that could be added to improve user accessibility are:

1. Better input error handling
2. Customizable GUI
3. Sound cues added to notifications
4. Narration
5. Other methods of input: voice commands, on-screen buttons, keyboard shortcuts

Accessibility

Principles of Universal Design:

1. Equitable Use
    - Accessible to people who are not blind and can type on the keyboard
    - Only has one design that may not be appealing to all users
    - Could implement a more user-friendly GUI with more features
2. Flexibility in Use
    - Program can be used with any type of computer
    - Currently only used by typing on the keyboard
    - No UI customization, could be improved with better GUI
3. Simple and Intuitive Use
    - Users are given instructions for valid input
    - Information is communicated in an easy to understand way
    - UI is simple and not unnecessarily complex
    - Could have better formatting for Task printing
4. Perceptible Information
    - Information is communicated through text only
    - Program could use sound cues, more importantly with Pomodoro/Alarm
    - Could implement a GUI with a more engaging and readable design (bigger fonts, different pages/tabs based on features)
5. Tolerance for Error
    - Currently does not handle input errors
    - No (majorly) hazardous elements when using this program

    - Designed to be used actively for a very short interval of time, so there is no risk of getting eye strain from this program
    - Could provide confirmation prompt when deleting tasks
    - Could save backups of TaskLists/Timelines
6. Low Physical Effort
    - Program can be used with only the keyboard
    - No mouse usage
    - Input commands are short
7. Size and Space for Approach and Use
    - Mainly meant for use on a computer, not designed to be used comfortably without a physical keyboard
    - If feasible, it could make use of a voice recognition service

This program is mainly designed for use by people who complete large pieces of work. That is a demographic which would primarily consist of students and employees/workers, though it can also branch out to anyone who desires to use an enriched to-do list. It can be more easily marketed to those who need help with procrastination. Since this program is designed for use on a typical computer, it may also be additionally restricted to those who use technology more frequently.

Since this program deals with general work/tasks similar to a to-do list, its intended purpose of scheduling/management could appeal to almost anyone. However, as mentioned above, this is mainly catered to students/workers who use technology, so it may not appeal to those who do not use more modern technology such as computers or phones. Due to the monotonous nature of the UI and its lack of customizability, it is also not easy to use by those who are physically or visually impaired.

Progress Report

Open Questions

- Which code smells we still have
- What design patterns may be useful
- How to fix difficult code smells
- How to increase accessibility
- How to design/implement a more organized UI

Successes

Phase 0+1:

- Code consistently adheres to Clean Architecture
- Successfully implemented design patterns “Strategy” and “Observer”
- Few conflicts in planning and implementation through the use of Trello

Phase 2:

- Program architecture was further improved with more classes (such as controllers) to make dependency straightforward
- Reviewing and accepting pull requests happened more quickly
- Improved communication (related to pull requests)
- Each member was involved in implementation

Phase 2 Progress

Bamdad

- CSVManager save
- CSVManager load
- Pull request (Implemented CSVManager)

Max

- UserFunctions and Main
- Printing
- Reviewing pull requests
- Design document
- Pull request 1 or Pull request 2 (Extended UserFunctions)
    - Provides the bulk of the user’s ability to interact with the program’s features from the UI

Jacky

- Suggestor/SuggestorList
- Design document
- Presentation
- Pull request (Implemented Suggestor)
-

Jennifer

- Storable interface and CSVManager class
- Updating Pomodoro to fix code smell
- Design document
- Pull request (Implemented Storable+CSVManager)
    - Implements an important feature of saving and loading .csv files.

Robby

- Refactoring the Packaging and implement Gradle to the project
- Fix the Task strategy to satisfy OCP
- Implement Timeline and TimeLineManager
- Implement controllers
- Revise or Refactor other member's codes to make them follow the clean architecture.
- Write Tests for the features mentioned above, and for others' features.
- Pull request (Implemented Task adding feature)
    - Entities and use cases dealing with timeline-related features are added.
    - And a controller dealing with both tasklist and timeline features.
    - They are tested and revised in my next pull request.