Design Document & Progress Report

1. Changes from phase 1

- We refactored the User class by creating three manager classes to take care of the tasks, projects, and teams that belong to each user. This eliminates some of the “bloaters” code smell and improves the overall structure by removing use-case-like methods from the User entity.
- We wrote quite a lot more test cases, which will be further discussed in section 5.
- We added gradle to our project and set up Github Action to perform autotesting.
- We used Javafx to build a simple GUI for our app.
- We fixed some bugs related to the functionality of the program (changing username, adding team members, etc.).

2. Major design decisions

- We decided to let our GUI repeat what CLI does to a certain extent, which is to let the user type commands and execute them, just so that we can directly reuse our *Command* classes without having to write extra code to change the page layout by too much after every user action. This decision was motivated by combined factors of time limitation, difficulty, and our group dynamics. We need the implementation of the GUI to be simple enough that we can actually pull it off and make it run.
- Serialization is a slow process, so we did not choose to save data after every user action. Instead, the user has to execute one of the following three commands in order to properly save data: the “save” command, the “logout” command, or the “exit” command.
- Unlike Git-hub Classroom, we did not allow the user to search up teams and choose a team to join; intead, a member from the team has to add the user to their team, like on Markus, Gradescope, and Crowdmark. We chose this design because we believed that our to-do-list system resembles Crowdmark more than it resembles Git-hub Classroom. There can be many many teams in our system existing for many different reasons, and a team project is not necessary. For example, even a family can constitute a team where parents can assign tasks to their children.
- We could have used Strategy Design Pattern for sorting tasks, but we think that there is really no reason to view all upcoming tasks in a different way other than in chronological order with stars in front of starred tasks. If one strategy proves to be by far the most reasonable, then allowing other strategies to exist is simply unnecessary and contrived.

3. Clean Architecture

Please see our class diagram for a visualization of our program’s structure.

- We took vast inspiration from Week 6’s “CleanArchLoginSample”, so the structure of our program is quite similar to the structure of that sample program, except that we create our entities by reading data from a local .ser file using serialization. After that, we inject entities into use cases, and we inject use cases into controllers as input boundaries.
- The dependency rule is consistently followed. For example, the *DataMemoryUseCases* class is responsible for saving data, so it needs to use our *DataManager* class, which is a gateway class that directly interacts with our database. To adhere to the dependency

rule, we are injecting our *DataManager* class as the *DataSaver* interface into the *DataMemoryUseCases* class, so that the use case class does not depend on a class from the controller/presenter/gateway layer.

- At run time, a use case method will return the result of an action as a boolean or as an Enum; a controller will be responsible for dealing with the result (e.g. raise an exception, or return a String...); classes from the outer layer (*Command*, *CommandExecutor*, *CLI*) will display whatever is returned back from the controller. Here is a brief scenario walkthrough with CLI (GUI will be extremely similar):
    - User action: The user types “newTask;project phase 2;2021-12-08” into the console after logging in.
    - Driver to controller: The *CLI* class passes the input to *CommandExecutor*, which finds the *NewTask* command and calls *execute* on it through the *Command* interface. *NewTask* checks argument length and delegates the task to *TaskController*.
    - Controller to use case: *TaskController* hands off the work to *TaskUseCases* through the *TaskInputBoundary* interface.
    - Use case to entity: *TaskUseCases* runs its *newTask* method, in which it creates a new *Task* entity, adds it to the *Project* entity specified by the user, and adds it to the *User* entity.
- Potential violation and solution:
    - As discussed in phase 1, entities storing collections of other entities bring out a dilemma. Our *User* class, which stores collections of *Project*, *Task*, and *Team*, used to have implementations of methods such as *addTask* and *delTask*, which are almost use-case-like. We chose to let this happen because otherwise we would have to let *User* reveal how it stores *Task* to use case classes.
    - However, we did find a way to improve the structure. For each of those three entities, *User* now stores an instance of a manager interface (such as *TaskList* implemented by *TaskManager*) which actually stores that entity and does the mutation. Dependency inversion is used and the dilemma is perfectly avoided.

4. SOLID

- Single Responsibility Principle: All of our classes should stick to the SRP now because their methods perform strongly related tasks and are clearly inseparable.
- Open/Closed Principle: All of our classes can be extended without having to modify existing code. For example, we can freely create new use case methods in *QueryUseCases*, let *User* store more information, or create a subclass of *Task* which has specific steps attached to it.
- Liskov Substitution Principle: We have only been using interfaces and no hierarchies are present in our program, so the LSP is vacuously satisfied.
- Interface Segregation Principle: Public interfaces of our classes all appear to be reasonable and straightforward, and there does not exist a single irrelevant part (e.g. attribute or method) that a programmer has to implement.
- Dependency Inversion Principle: As explained before in the previous section, our active use of interfaces such as *DataSaver* and input boundaries removed a lot of dependency in our code. We ensured that dependency only flowed in one way: from driver inwards to the entities, without jumping layers.

5. Design patterns

Implemented ones (with latest pull request #):

- Singleton design pattern (47): We used this design pattern on each of our controller classes because they should have one instance only and they should be easily accessible by interface classes.
- Iterator design pattern (61): We used this design pattern on *ProjectList*, *Project*, and *Team* because they store collections of other entities, they don't want to reveal how they store them, and we need to use enhanced for loops on them.
- Dependency Injection Design Pattern (55): We used this design pattern on *User*, *Project*, *Team*, and *UserList*, which all store collections of other entities. Instead of passing entities into their constructors, we inject those entities through methods like *addTask* or *addUser*, so that polymorphism will be supported when we have different types of *Task* or *User* in the future.
- Command design pattern (53): We used this on *CommandExecutor* and all our *Command* classes because our goal is to delegate each user action to a use case method with a command line interface. *CommandExecutor* calls the *execute* method of a specific command based on user input through the *Command* interface, which further delegates tasks to controllers.
- Memento design pattern (73): We used this design pattern because we want to allow "undo" and "redo" multiple times in a row. *UserList* and *DataMemoryUseCases* are where this design pattern is specifically implemented, the former as the "Originator" and the latter as the "Caretaker".

For future extension:

- Strategy design pattern: This could be implemented if we want to allow different sorting strategies for displaying entities (e.g. tasks, projects, etc.).
- Builder design pattern: This could be implemented if the way an entity like *Task* or *Project* is created gets complicated and we want to allow multiple ways to create them.
- Factory design pattern: If we have different types of Task or User, we can use this design pattern when we want to construct without specifying the exact class.
- Observer design pattern: This could work on our GUI where a pressed button needs to inform a Javafx *Pane* or *Scene* to change its layout.

6. Code

- Code organization and packaging strategy: Our packages directly correspond to layers of clean architecture because this reflects our program structure most clearly.
- Code style and documentation: We have eliminated all warnings except ones that complain about some controller methods only returning one possible String (indicating success) because the other cases are all Exceptions. We included Javadocs and comments everywhere.
- Refactoring: We were constantly refactoring as we made progress through phase 1 and 2. Several pull requests involving refactoring are # 57, 59, 60, 61, 70, 74, 75. The biggest refactoring that happened in phase 2 was the one addressed in section 0 with pull request # 91 ("refactoring User class").
- Testing and functionality: We have tested every single command including the general commands. For each command, all possible cases which the user may end up

encountering have been tested. For example, when a user creates a new task, the task name may already exist, the due date may be in the past, and so on. The test coverage is 69% because classes related to the GUI (setting up the layout of the screen) take up many lines of code (it was 83% before we added GUI). All tests pass and our program functions well according to our specification.

7. Group member tasks in phase 2

- Jiayang: wrote comments.
  - Test comments: https://github.com/CSC207-UofT/course-project-howtodoit/pull/116. Comments are important.
- Jingyang: wrote comments, made an alternative Android version of our app (not included in the main branch of our repo).
  - Added some comments, fixed some typos: https://github.com/CSC207-UofT/course-project-howtodoit/commit/22c2419c1d6002bbc7c44c0f1913454e5b7da139. Comments are important.
- Krystal: made GUI.
  - Complete UserActivityScene: https://github.com/CSC207-UofT/course-project-howtodoit/commit/c932f237d1658987f6daa5ce4819e94753559a0b. GUI is a big part of our phase 2 work.
- Richard: refactored User class, wrote test classes, made GUI, debugged, modified design document.
  - Refactoring user class: https://github.com/CSC207-UofT/course-project-howtodoit/pull/91. Incorporated feedback from phase 1 and improved overall code structure.
- Yixin: wrote test classes, set up and completed auto-grading with Action, made GUI, debugged, modified design document.
  - Testing and completing auto-grading: https://github.com/CSC207-UofT/course-project-howtodoit/pull/106. Allowing Github to run all our JUnit tests every time changes are pushed.
  - Finished GUI, fixed "undo" feature, debug and refactor code https://github.com/CSC207-UofT/course-project-howtodoit/pull/117 Finishing half of the GUI, fixing a glitch within the "undo" feature, fixing several other bugs that would break the project.
- Zixiu: made GUI, debugged, wrote test classes.
  - Add login and register button: https://github.com/CSC207-UofT/course-project-howtodoit/pull/115. GUI is a big part of our phase 2 work.

8. To-do-list for "phase 3"

- Improve program structure based on feedback from phase 2.
- Do password encryption.
- Create more constants:

    - Use more Enums for use case results.
    - Create specific Exceptions, but also remove controller warnings and let them return more Strings instead of throwing Exceptions (maybe).
- Improve GUI:
    - Multiple panes (projects, tasks, teams, tasks in project, members in team).
    - When clicking on query buttons, switch pane.
    - When clicking on mutation buttons, do mutation and execute the last query button.
- Improve user experience and functionality:
    - Display how many days are left until the due date.
    - Deal with overdue tasks (auto deletion or display differently).