# Design Document

---

## 1. Major design decisions

- Serialization is a slow process, so we did not choose to save data after every user action. Instead, the user has to type 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.

## 2. 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 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 does not depend on a class from an outer 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. Brief scenario walkthrough:
    - User action: The user types “newTask;project phase 1;2021-11-15” 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: As long as one entity stores a collection of another entity, this dilemma is unavoidable. We have a *Project* class which stores a collection of *Task*, but

*Project* is itself an entity instead of a use case or controller. We have two choices: either let *Project* implement *addTask*, *delTask*, which are almost use-case-like methods that mutate its collection of *Task*, or return the whole collection to use case classes and let them do the mutation. Both seem problematic, but the second approach is much worse than the first one, because we do not want use case classes to know how *Project* stores *Task*; we are using a *HashMap* now, but we might change it to a *HashTable* or *ArrayList* later. Therefore, we decided to let *Project* implement *addTask* and *delTask*. We are aware that this is a potential drawback to our program structure and that there are probably better solutions, but we have not seen any example that properly avoids letting one entity store a collection of another entity (when applicable), so we ran out of ideas.

## 3. SOLID

- Single Responsibility Principle: The majority of our classes obviously stick to the SRP because they only have one or two methods that are clearly inseparable, but there are indeed a few classes that seem a bit colossal, for example, the *TaskUseCases* class and the *User* class. We argue that it is not necessary to split up the responsibilities of *TaskUseCases* classes because they are quite closely related (e.g. addTask, completeTask, rename...). The *User* class also has to store information of a user, which includes their tasks, projects, and teams.
- 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* or let *User* store more information.
- Liskov Substitution Principle: So far 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 entities outwards to the driver, without jumping layers.

## 4. 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 *Project* and *Team* because *Project* stores a collection of *Task* and *Team* stores a collection of *User*, they don't want to reveal how they store them, and we need to use enhanced for loops on both of 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.

## 5. 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 except in test classes.
- Refactoring: We have used refactoring to a degree we wouldn't believe ourselves. Almost all code from phase 0 has been changed and we were continuing to refactor as we made progress through phase 1. Just several pull requests involving refactoring are # 57, 59, 60, 61, 70, 74, 75.
- Testing and functionality: We have tested a few commands and use cases with 36% overall coverage so far. All tests pass and our program functions well according to our specification. More testing progress will be made in phase 2.