Phase 2 Design Document

Update on Specification:

- We have added an undo feature for worker and department head on undoable features like create and delete worker and department head and change schedule and salary of a worker.
- Even search workers and department heads are not undoable, users still need to undo them before they can undo the one before it.
- A note here is that the list of all employees counts as one operation for the worker and one operation for the department head so if the previous command is to list all employees, you need to undo on worker and department head to fully undo it.
- Users are not able to undo anything if there is nothing that can be undone
- We also check for edge cases when handling user input such as when a user is supposed to enter a number but the user entered a word
- We also extend our search features by allowing users to search on worker and department head ID
- We also updated our option menu page with more options and the format of input of the search feature for both worker and department head has been changed to allow search by 2 different attributes separately
- The search features for department head in Phase 1 where we allowed search by more or less years of experience has been changed to only allow search for more years of experience as we think that HR people usually work for more years of experience and this also makes our input handling a bit easier

Major design decision:

- We have split our presenter and controller into different classes as the presenter and controllers from Phase 1 are too long and as we add more options, the class will get bigger and bigger.
- The presenters are split based on the page that the user should see and the controller is split based on whether it is the controller on worker or department head, then we have an overall presenter which is the CmdLineUI.
- However, we do still think that our controller and presenter are still long but we didn’t have enough time to figure out a way to divide them up but this could be refactored to divide them further.
- The additional features that we added are already planned in phase 1 so there isn’t much decision made there except that we have decide to not implement the search by department on worker and department head as the time that we spend on refactoring is a bit long so our time are fairly limited
- Another major decision is about design pattern where we thought at the beginning that we can use the strategy design pattern for the search operations where the workerManager class have a workerSearcher interface and have search by ID and search by worker name implement the searcher interface
- After we look at the example code on github, we realized that the strategy design pattern requires us to create 2 different workerManager objects in order to perform the 2 search but we can only have one manager since we need to keep track of worker created
- If we have 2 instances of the searcher interface, then there isn’t a reason to use the strategy design pattern and the same reason applies for the department head use case so we just decide to go with just 2 similar methods in both use cases.

- We also have all the messages and the file name into a constant packages so that if we need the program in another language or change the message, they can be easily found and it is easier to modify

How adhere to Clean Architecture:

<u>CRC cards model:</u>

- Since we include this CRC model to convince that we followed Clean Architecture, so in this CRC cards, we only shows classes that is important in terms of Clean Architecture so all the command classes and the command invoker classes are not included as those classes are added just because implementing the command design pattern which should act as a layer between controller and use case class
- The Constant and FileName class which contain constant message and file name, are also not included as they are not in any of the layers of the Clean Architecture and they will not be shown in the collaborators part of this CRC cards.

Entities Class:

Class name: Employees (parent class for Worker and DepartmentHead)	
Responsibilities ● Stores name, ID and department of any employees ● Provide getter and setter for attributes	Collaborators Worker DepartmentHead

Class name: Worker (child class of Employees)	
Responsibilities ● Stores salary and schedule and the 3 attributes that Employees store ● Provide getter and setter for new attributes	Collaborators Schedule workerManager Employees

Class name: DepartmentHead (child class of Employees)	
Responsibilities ● Stores the number of year of experience and the 3 attributes that Employees store ● Provide getter and setter for the new attributes	Collaborators departmentHeadManager Employees

Class name: Schedule	
Responsibilities ● Store day of week, start time and end time for a schedule	Collaborators Worker workerManager

● Provide getter and setter for each attributes and convert a Schedule to string	

Use Case Class:

Class name: WorkerManager	
Responsibilities ● Completing all manipulations and operations to workers	Collaborators Worker Schedule readWriter (interface) WorkerInputHandller

Class name: DepartmentHeadManager	
Responsibilities ● Completing all manipulations and operations to department heads	Collaborators readWriter (interface) DepartmentHead DepartmentHeadInputHandler

Controller:

Class name: WorkerInputHandler	
Responsibilities ● Execute the correct worker commands based on user input provided	Collaborators WorkerManager CmdLineUI WorkerReadWriter

Class name: DepartmentHeadInputHandler	
Responsibilities ● Execute the correct department head commands based on user input provided	Collaborators DepartmentHeadManager CmdLineUI DepartmentHeadReadWriter

Presenter:

Class name: OptionMenuPresenter	
Responsibilities ● Display the option menu	Collaborators CmdLineUI

Class name: WorkerOutputHandler	
Responsibilities ● Display outputs of worker operations that is presented	Collaborators CmdLineUI

Class name: DepartmentHeadOutputHandler	
Responsibilities ● Display outputs of department head operations that is presented	Collaborators CmdLineUI

Class name: CmdLineUI	
Responsibilities ● Use other three presenter to display messages that the user should see and take in user input and pass them to corresponding controller	Collaborators OptionMenuPresenter WorkerOutputHandler DepartmentHeadOutputHandler WorkerInputHandler DepartmentHeadInputHandler

Data Accessing:

Class name: ReadWriter (Interface)	
Responsibilities ● Defines methods that specific readWriter need to implement	Collaborators WorkerManager DepartmentHeadManager WorkerReadWriter DepartmentHeadReadWriter

Class name: WorkerReadWriter	
Responsibilities ● Saving list of workers to ser file and reading from ser file and convert it to a list of workers	Collaborators ReadWriter WorkerInputHandler

Class name: DepartmentHeadReadWriter

Responsibilities ● Saving list of department heads to ser file and reading from ser file and convert it to a list of department head	Collaborators ReadWriter DepartmentHeadInputHandler

- One of the thing that may seems weird is that our third layer controller is creating an instance of readWriter class which is in the outermost layer
- This seems violating the Clean Architecture but in the typical scenario diagram, we can see that the part about use case and readWriter are in adjacent layers
- This means that the readWriter class are actually in the third layer in terms of that diagram and in that diagram, the controller is still in the third layer so they technically belongs to the same layer so having controller create readWriter class doesn't violate Clean Architecture
- Because we don't want our CRC cards to get too long, we omit the command class that we implemented but we will have an UML diagram that shows how the command class interact with the use case class and the controller and how it acts as an layer to further decouple the 2 layers

<u>Simple UML Diagram:</u>

- We have included a simple UML diagram as a pdf on github showing how every class in our program interact but since there are lots of classes and lots of interactions, the diagram is a bit mess
- In terms of the diagram, we can see that most of our arrows are pointing inwards which means that we are having an outer layer or details depending on our inner loop or core logic of our program.
- One reverse direction arrow is the FileName class which is reasonable since our intuition of having the FileName class is to make our program deals with file names easier so the class actually doesn't belong to any layer in terms of Clean Architecture.
- The other reverse direction arrow is about the Message class which is similar to the FileName where it doesn't belong to any layer, it is just there to make some extra features easier such as changing to different language so we can just modify that class rather than looking for strings everywhere in the program so this is not a violation of Clean Architecture
- In terms of the arrow going through layers, we can see that all the arrows going through layers are pointing inwards except arrows about readWriter.
- This is because we need readWriter class in our use case but they are in the fourth layer of Clean Architecture so we created an interface where the 2 concrete readWriter class (should be in the fourth layer) implements and having our 2 use case classes depends on that interface to adhere to Clean Architecture.
- In terms of the concrete readWriter, we decide to have our controller which is the 2 inputHandler classes create the concrete readWriter class and inject that into the Executor class where then the Executor creates use case instance with that readWriter
- This is why there is an arrow between the executor classes to the concrete readWriter class where we did not mean that the executor class creates concrete readWriter, it just that the Executor classes take concrete readWriter as input from controller which to us isn't a violation of Clean Architecture as we didn't create or use methods in the concrete readWriter

- The reason why we decide to create the concrete readWriter object in our controller which is the inputHandler class is mentioned in the CRC cards section so overall, we think that our program adheres to Clean Architecture as much as we can

Typical Scenario Walkthrough:

1. The CmdLineUI (overall presenter) starts off display the welcome message and display option menus
2. Then take user input in and based on the first character of user input, pass the information from user into corresponding inputHandler (worker or department head input handler)
3. Then the inputHandler will create different command object based on first character of user input
4. Then the inputHandler calls execute or undo command methods in the corresponding command invoker class (worker or department head invoker class) to execute or undo the commands based on first character of user input
5. Then the command invoker calls execute or undo methods of the command given
6. Then inside the command class, the execute or undo methods call corresponding use case methods to complete the operations and use case class methods returns output
7. The execute or undo methods return the output back to the invoker class which gets returned back to the inputHandler
8. Then the inputHandler return the output to CmdLineUI and then CmdLineUI calls corresponding outputHandler (worker or department head presenter) with the output to display the output to command line

How consist to SOLID:

Adhere to Single Responsibility principle:

- We have separate our use case class into 2 where one contains all operations on worker and the other contains all operations on department head to adhere to Single Responsibility principles
- Similar examples can be found in our program as we separate the invoker class of our command design pattern into 2 invokers, we have 2 readWriter class, one for worker and one for department head to separate responsibility
- Another refactoring that we did in phase 2 is that we also split the presenter which is the output handler and the controller which is the input handler into 2 where one for worker and one for department head to further separate responsibility
- These examples and refactoring makes our program adhere more to the principle

Adhere to Open/Close principle:

- We believe that our program is free for extensions but not free for modifications.
- An example of this will be the new features that we added which is the undo feature
- But implementing this feature, we did not need to modify any of the existing methods in the use case class, we just need to add more methods and behaviours to the existing use case class
- Our command design pattern and the command class also makes our program adhere to the principle more as when we add features (search by ID features), we just need to create a new command class that implements the command interface which calls methods in use case
- No need to modify any of the existing command classes as well as the command invoker class

Adhere to Liskov Substitution principle:

- We have used interface and inheritance in couple places in our program
- For example, the workerCommand interface where all worker command class must implement

- We are keeping in an instance of workerCommand inside the invoker class but when we pass in the command, we are passing concrete command objects
- Another example where we adhere to the principle is the readWriter interface, the use case class have an instance of the readWriter but when we pass in the readWriter, we pass in concrete readWriter when we actually define use case class
- In terms of adhering to the principle when using inheritance, we have an example where the employees are the parent class of worker and department heads.
- When we implement the worker and department head class, we keep and didn't modify any of the attributes that is defined in the employees class as well as all methods in employees
- But instead, we add new attributes and methods for each of the worker and department head class that is unique to worker and department head

<u>Adhere to Interface Segregation principle:</u>

- It is fairly clear that all of our interfaces are really small and only classes that need to implement the interface implement it.
- For example, the command class that implements the command interface.
- All command classes need to implement the execute methods and the undo methods but no other methods so we are following this principle.
- Another example is the readWriter interface where we know that all readWriter classes will need to implement saveToFile and writeToFile methods and no other methods so the interface readWriter only asks the class to implement these 2 methods.

<u>Adhere to Dependency Inversion principle:</u>

- One of the great examples that our program has that follows this principle is the command design pattern.
- Rather than having the controller directly interact with use case class, we have a command interface and command invokers that separate controllers from use case class and decouples them.
- Another example will be the readWriter that we have. Since we need to use readWriter in our use case but it is a high level class so we cannot use it directly in use case class.
- So we created an interface readWriter and have the different readWriter implements that interface and we have an instance of that interface in the use case and have the high level class which is the controller class inject the actual readWriter into the use case.
- With the UML above, we can see clearly that we only have higher level class using one layer lower class which also shows that our program in general adhere to the principle
- Since we implement the command design pattern, we have our controller depend on the command class and have command class depends on use case class that creates this thin layer between controller and use case which makes our program adhere to Dependency Inversion principle

Packaging Strategy and Code Organization:

- We have used the package by layer strategy to organize our code as we have spend lots of time on adhere to the Clean Architecture model so it is natural and easier for us to package classes by layer
- We have 7 packages but there are only 4 layers in Clean Architecture
- This is because we have a demoRun which doesn't belong to any layer so it belongs to its own layer
- we also break our packages for the second layer (use case) into 2 packages (workerOperations and departmentHeadOperations) because we can group our use case class into 2 classes where

one contains worker use case and the other contains department head use case which is a natural and an understandable way of grouping them and the name are fairly suggestive
- Another reason why we group them is that if we keep them as one package, then the package will just be way too big
- The Entities package, UI package and the Data package are the innermost layer, the third layer and the outermost layer respectively where the name are fairly suggestive
- The last extra packages is the package containing all the constants like messages that the user see and the file name that is used in program

Design Pattern:
- We didn't chose to implement a new design pattern as we have only 4 group members and the time are fairly limited
- We also didn't find a design pattern that suits our new added features
- We originally thought of implementing strategy for search feature but realized that it doesn't really fit for our purpose so we decided to just give up with that design pattern and if we have more time, we can look for strategies to group the 2 similar methods into one and make the program neater
- We extend on the command design pattern that we implemented in phase 1
- In phase 1, the command design pattern is just used to decouple use case class from controller and it is a good pattern to use when we got a option menu
- In phase 2, we use this command design pattern to help us implement the undo features where the command design pattern helps us keep track of command that is executed by storing an arraylist of commands in the invoker class
- When we add the undo feature, we just added an undo methods in each of the command class which undo the command and implement undo methods in use case class
- Another benefit that the command design pattern gives is that each command class can stores the list of arguments needed and modify the arguments after the command execute to correct arguments to perform undo which gives us a great benefit as we don't need to worry about keep track of previous command and storing correct arguments to do the undo operations.
- In phase 2, we also experience the benefit of using command design pattern which makes adding new features like search by ID much easier and clearer
- Another fairly common design pattern is the dependency injection design pattern where our use case class holds an instance of the readWriter interface and have controller inject the concrete readWriter class into the use case class

Use of Github Features:
- We didn't use events and actions as we have a fairly small group so we don't really need to use them to keep track of our working process and we also have a fairly regular meeting so we can just ask questions if they have any
- In terms of todos, we just have them in a google doc and we frequently modify it as we proceed in our implementation
- We have use pull request in our phase 2 implementation which gives us the benefit of having everyone look at the new changes before it is pushed to the main as well as give everyone an opportunity to look at each other's work

Testing:
- We test all the getter and setter methods of entity classes.

- We test the methods in use case classes with some common cases and edge cases.
- We create two test files to store the test data and clear all the data after each test.
- This makes the test easier to access since there is no need to create a file before the test and delete it after the test any more.
- We plan to implement some test cases for the input handler. However, to make the handler interact with the test file makes more changes to codes than we expected, so due to the time limit we give up the test.
- We may fix it by changing the test boolean inside command into a parameter but time is limited so we decide to keep the code that we know functionality works
- The output of inputHandler should be similar to our use case class output and we can see if our individual input cases call the corresponding command class by actually running the code and doing some valid or invalid inputs. Therefore, it is actually fine to leave out tests for this part.
- The other part that we didn't test is the presenter class which are full of void methods
- This means that it will be fairly hard to test and as mentioned before, we can test them by running the program so we decide to omit the tests
- There is one problem that we have encountered for our testing is that when Shilin try to run the test, the 2 test files need to be in HRSystem folder in order to find the file and when Ruixin run the test, the 2 have to be in the outermost folder (outside of HRSystem)
- We have tried to specify like a file path to look for files but we are not successful at that
- Since this doesn't affect running the actual program, so we just decided to keep the 2 test file in the HRSystem and if on someone's computer, the file is not when they run test, then they can try move that file outside of the HRSystem and try again

Refactoring:

- We have done lots of refactoring from phase 1
- One significant one is splitting our presenter based on pages and controller based on worker and department heads
- This refactoring help us reduce the size of our presenter and controller class as well as making our code clearer
- This also helps us to convert our command line UI to an actual Web UI easier as we have clear separation between which class are handling input and which are handling output
- We complete this refactoring in the pull request called splitting controllers and presenters
- Another refactoring that we did is rather than having the constant message directly in presenter, we moved all constants into a constant class as well as storing out file name in the constant packages
- We also handles more edge cases
- This refactoring is done in pull request change output into class and check for edge cases
- We also created 2 separate files for testing which removes the need of creating new files manually before testing the code
- This is done in pull request called create 4 ser files
- We have also added more test cases on the controller (2 inputHandler classes) as well as testing edge cases
- We have also modified our command class to store the list of arguments when they are created rather than having controller passing them as parameters

- This helps us implement the undo feature with command design pattern as we can store arguments to do undo right after we execute the command
- This refactoring is done in the pull request named refactor the command design pattern to suit for future extensions
- We have an long parameter code smell in demoRun where we create our CmdLineUI object but this cannot be fixed as the CmdLineUI is displaying our overall UI page so it needs the 3 presenters and 2 controllers and we cannot have them into a list since they are not same type of object
- Another possible code smell is that we still have a fairly long switch case in our presenter (outputHandler class) and controller (inputHandler) which can be solved if we have more time, by splitting our controller and presenter even further (by splitting them by more specific categories)
- For example, we can probably split them up by operations that they did.
- We can have a presenter and controller for the create operations (one on create worker and create department head), one presenter and controller for delete, one for search and so on.
- This allows us to fix the long switch case code smell but this may make our long parameter list code smell even longer as our overall presenter CmdLineUI will take in more parameters
- But we think that this is a possible refactoring that we can do as it makes our program adhere to Single Responsibility principle even more and get our class shorter
- This problem of having long class also applies to our use case class
- As we expand on more and more features, our use case class gets longer and longer so one of the possible refactoring that we can do in the future is that splitting these 2 use case class into more use case class not only that it makes program adhere more to Single Responsibility principle, it also makes our program neater

Functionality:

- For functionality, our specification for phase 1 is already completed in phase 1 so we decide to expand on new features like undo and search by one more attributes
- After phase 2, our program does what the specification says
- In terms of whether the program is ambitious enough, given us only a group of 4, we think that adding these 2 more features is sufficiently ambitious.
- Our phase 2 seems that it doesn’t have much more features added but we do think that it is ambitious enough as we allow user to undo 6 different operations which gives us lots of work and modification to do and we also added 1 more search features for worker and department head
- This gives a total of 8 new features which we think is ambitious enough for a group of 4 as we need to do refactoring from previous phase as well as adding test cases and java docs
- Also, because of serialization, our project is able to store and load state such as workers and department heads created in previous runs and their information.

Code Style and Documentation:

- In terms of code documentation, we have java docs on every methods that we created except tests and we have make sure that there is no warning in any of the classes
- We also added a bit of comments on where we think we need to add to make programmer understand our code better
- In terms of name of variables and class names, we think that they are fairly suggestive and self-explanatory

- The warning from demoRun class due to the commented out code is fixed as we have 2 separate files for running and testing so we don't need to create file manually
- The other warning mentioned in phase 1 is still there as there isn't a way to fix that warning and we still use the suppress warning to get rid of it.

Progress Report:

What Hamza Works on:

- Splitting the controllers and presenter into two different classes in order to differentiate between input and output flow of data.
- Once again splitting the controllers and presenters based on worker or department head operations
- Creating the search by ID feature for worker objects where existing workers can be found using their id number.

Link of pull request to Hamza works:

- The pull request shows significant contribution through splitting the controller and presenter. This was something that the TA had mentioned as feedback after phase 1.
- This contribution was significant as it was needed for us to adhere to the SOLID principles and Clean Architecture design, before that the input and output of a user for both department head and worker operations were in the same classes.
- After this change, the inputs and outputs could easily be traced whether they were for workers of department heads or controllers or presenters.
- https://github.com/CSC207-UofT/course-project-group-065-1/pull/2

What Ruixin works on:

- Refactor the test to make it easier to use.
- Implement test cases for new methods created in phase 2 and add some edge cases test.
- Implement some undo methods in use case classes.

Link of the pull request to Ruixin works:

- Link: https://github.com/CSC207-UofT/course-project-group-065-1/pull/5
- The pull request contains fixed test cases and implemented undo methods.
- The contribution is significant since the fixed test method and newly added test cases and edge cases ensure the easy accessibility and sophistication of tests and the undo methods are basis to implement the undo feature.

What Somto works on:

- Creating the searchByID feature for the departmentHead objects where they created department heads can be found by id
- Wrote tests for the newly created departmentHead searchByID feature
- Modified the department head search by years of experience feature to search by specifically more years of experience

Link of the pull request to Somto works:

- Link: https://github.com/CSC207-UofT/course-project-group-065-1/pull/8
- The pull request contains the code for the implementation of a new feature "searchByID" in one of our two main classes. searchByID was created for the departmentHead object and is used by the departmentHeadManager to effectively search for a department head using their id. In this pull request, I also modified code for the departmentHeadManager where search by years of experience can now search for departmenHeads with greater than or equal to given years of experience.

- This pull request is an important contribution because it contains the whole functionality for searching for a department head.

What Shilin works on:

- Refactoring the command design pattern to make it easier to implement the undo feature
- Change the constant message and file name into a separate class rather than having them in presenter
- Add checks for edge cases for user input and helper methods to check input
- Implement the undo features in all command classes and command line interface
- Write up some java docs and comments

Link of the pull request to Shilin works:

- The pull request that shows significant contributions is implement the undo functionality
- Link: https://github.com/CSC207-UofT/course-project-group-065-1/commit/96d0350ab9d41e9274cd236c71a3b6049d64cc74
- The reason why this is significant contribution is that our major goal for this phase is to implement the undo feature and he completes all parts about the command class and input and output handling for undo
- Also, undo feature seems like one feature but we actually have 6 possible operations that can be undone so this is why this is a significant contributions
- He also have significant contribution on refactoring as he change the constant message into a class and checks for edge cases in pull request change output into class and check for edge cases
- Link for check edge case: https://github.com/CSC207-UofT/course-project-group-065-1/commit/12ee9eb69b778c2b5891ce7e2b49622f4cf6d20e
- Link for change constant message into class: https://github.com/CSC207-UofT/course-project-group-065-1/commit/722fa0176ce4cd4a5f16af5bf237d42d2e61abb1