Design document phase 1

Updated Specification:
There are a few things that we decided to change for phase 1. One of the things that we changed is the options menu. We have decided to change the options menu to a number-based rather than word-based. Specifically, rather than having users enter create worker, create department head in the command line, we give them an options menu where 1 means create worker and 2 means another operation which makes it more easier to use for users. Another thing that we decided to change from phase 0 is that when changing attributes about workers like salary, we no longer ask for department head ID as we think that when the HR person changes attributes, they usually already gain permission from corresponding department head so that is redundant. The rest are just additional functionality that is added. We have added a couple new features including deleting workers and department heads by id, search worker by name, search department heads by the number of years of experience and listing all workers and department heads. We also added serialization to our system which allows the system to keep the workers and department heads created in the past.
Major design decisions:
We made some major decisions for phase 1 including the option menu that is mentioned in the specification above which makes our program a bit more user friendly. Another major decision is the direction of where to expand on functionalities. We have decided to include deleting workers and department heads as it is a quite common operation in the HR system. Another reason why we add this feature is that we want to incorporate an undo feature in our final project so this deleting feature is a way to undo creating workers and department heads. Another feature that we decided on adding is the search operations where it would be nice to have when a HR person wants to find or check if a worker or department head is in the system. Another reason for having this search is because we can expand more on the search operations by having different parameters to search but due to time limit, we have only implemented 1 search for workers and 1 for department heads. The other big decision that we make is on what design pattern to use. We have decided to go with the command design pattern. The reason is that first it fits our interface where we have different commands from different user inputs, second it allows us to implement the undo or history features much easier which we are planning to do in phase 2. The last reason is that it gives us a nice layer between controllers and the use case to further decouple them.
How adhere to Clean Architecture:
We will show that our program adheres to Clean Architecture with a CRC model.

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**

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

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** workerManager departmentHeadManager

Class name: Schedule	
Responsibilities • Store day of week, start time and end time for a schedule • Provide getter and setter for each attributes and convert a Schedule to string	**Collaborators** Worker workerManager

Use Case Class:

Class name: WorkerManager	
Responsibilities • Completing all manipulations and operations to workers	**Collaborators** Worker DepartmentHead Schedule SystemController

Class name: DepartmentHeadManager	
Responsibilities ● Completing all manipulations and operations to department heads	**Collaborators** DepartmentHead SystemController

Controller:

Class name: SystemController	
Responsibilities ● Execute the correct command based on user input provided	**Collaborators** WorkerManager DepartmentHeadManager CmdLineUI DepartmentHeadReadWriter WorkerReadWriter

Presenter:

Class name: CmdLineUI	
Responsibilities ● Display the option menu and take in user input	**Collaborators** SystemController

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	**Collaborators** ReadWriter SystemController

a list of workers	

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 SystemController

The CRC model above shows all the classes that we have except the classes created because of the command design pattern which are just a layer between controller and use case class that decouples them further. In general, our program adheres to Clean architecture fairly well except where the readWriter gets passed into the use case class. We have our use case class have an instance of the ReadWriter interface to adhere to Clean Architecture since if we directly have an instance of the implementation of ReadWriter, we will be having the second inner layer to depend on the outermost layer which violates Clean Architecture so we decide to have the use case have instance of the interface which is abstraction and pass in the actual ReadWriter as parameter. This is a way that we demonstrate how we follow the Dependency Rule when working with outer layers by having use cases depend on abstraction and pass in the actual ReadWriter. But then, we have the problem of where we should create this ReadWriter. At the beginning, we thought we should create them in the DemoRun class as if we create them in controller or presenter it would violate the Dependency Rule by having a third layer class depend on the outermost layer. But we asked a TA about this and the TA suggested that we should create it in the controller as if we create it in DemoRun, there will be too many layers to pass through. So we decided to go with the TA's suggestion and have it created in the controller. If you look at import of any file, you will see that except the problem mentioned above about the ReadWriter being created in Controller, all other classes follow clean architecture. In the controller class, we have imported 2 different command classes that we didn't include in the CRC model. This is because those classes are created because we implement the command design pattern so rather than having controllers directly using use case class, we have controllers using this command invokers and having the invokers call different methods in use case class so that import still follows Clean Architecture. In each individual use case class, we have to import the data package which is the outermost layer but this is not a violation as we are only using the ReadWriter interface in our use case class. Lastly, we will have a scenario walk through to further convince you that our program adheres to Clean Architecture.

1. First the CmdLineUI will display the option menu and take in user input and pass it into our Controller.
2. Controller will then create different command objects based on user input
3. Then controller will all the execute command method in the corresponding command invoker class with the command object created and the list of arguments provided
4. The command invoker will then call the execute method that the command object implement with these arguments
5. Then the command object will call corresponding methods in use case class that are able to complete the command with these arguments

6. Then, use case class will do their operations and complete the command and pass the output back to the command object
7. Then the command object execute method returns the same output back to the invoker
8. Then the invoker pass this output back to the controller
9. Then the controller pass this output back to presenter and the presenter will display it on the command line

This is a process that applies to all possible commands in our program and we can see clearly that it is consistent with the Clean Architecture.

How consistent to SOLID:

Adhere to Single Responsibility principle:

In order to make sure that our program adheres to the Single Responsibility principle, we have separated the use case class into 2 classes where one class is only responsible for doing operations on workers and the other is only responsible for department heads. Also, when we implement the command design pattern, we separate the invoker for commands into a worker invoker as well as a department head invoker which separate responsibilities. Another great example that shows we adhere to the Single Responsibility principle is the readWriter class. Rather than having a readWriter implementation for both workers and department heads, we create a workerReadWriter and a DepartmentHeadReadWriter class which separate the responsibility and adhere more to the principle.

Adhere to Open/Close principle:

We believe that our program is free for extensions but not free for modifications. For example, the feature delete worker, we did not need to do any modifications to the original class but rather adding new methods and behaviours to the existing class. Another example will be the command design pattern that we implement. It is so easy to add new features with it as we just need to add an extra command class that implements the command interface and have that command class call some methods (existing or new) from the use case to execute the command.

Adhere to Liskov Substitution principle:

There are lots of places in our program that implement interfaces. For example, the serialization class that we have. In the use case class, we have an instance of the interface readWriter but when we pass in the actual readWriter from the controller, we are passing a readWriter that implements the interface. Another example where we did not violate this principle is where we have employees as the parent class and have workers and department heads extend this parent class. In our workers and department heads class, we keep all the attributes and methods from employees. We are just adding new information specifically about workers or department heads and new methods. We did not remove or modify anything from employees.

Adhere to Interface Segregation principle:

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 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.

Adhere to Dependency Inversion principle:

One of the great examples that our program has that follows this principle is the command design pattern. Rather than having the controller directly use 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 pass in the actual readWriter into the use case. But one of the problems that we have is where we should create the actual readWriter. Should we create the readWriter in the controller or should it be created in the demoRun? Based on what our group thinks, we think that we should create this readWriter in the demoRun as it is in the outermost layer in terms of clean architecture and controller is in the third layer should controller should create an instance of readWriter but one of us asked a TA and he mentioned that it is fine to create it in controller and pass it into use case class so that is what we go with.

Package Strategy and Code Organization:

For packaging Strategy and code organization, I think we organize them in a meaningful way and the name of each package is really descriptive. We have decided to use the package by layer strategy to organize our code because first, it is the most appealing one to our group and second, we spend a lot of our time adhering to Clean Architecture so this way of packaging seems more reasonable to us and it is the way of packaging that we feel the most comfortable with. Based on the Clean architecture, we should divide our project into 4 packages but we have 6 where one of the extra is due to the demo run and the other is because of we apply the command design pattern which gives us lots of class in use case and the way our divide them seems really natural so we decide to have 2 packages for use case package. So we decided to separate the use case layer into 2 packages where one is all commands and methods that can be applied to workers and the other packages are commands and methods that can be applied to department heads. So we have in total 6 packages which are UI, containing the controller and presenter which are codes about the command line interface, Data, containing the implementation of the readWriter for worker and department head as well as the interface which are codes to read and write data to file, Entities which are the 4 entities class that we have, WorkerOperations containing the command class and the worker use case class which are operations that can be applied to worker, DepartmentHeadOperations containing the command class and the department head use case class which are operations can be applied to department head and the last package which is the DemoRun. We think that these names are really descriptive in terms of what is in it and they also follow the package by layer strategy so that is why we decide to use it.

Design Pattern:

We have chosen to implement the command design pattern in our project. We have chosen to implement it because first, it is a great design pattern to implement when we have multiple operations on the user interface, second, it decouples the controller and the use case, and lastly, it encapsulates the detail of how to execute a command. The command design pattern not only gives us these advantages, it also makes our implementation in future phases a bit easier such as the history and the undo feature will be much easier to implement with the command design pattern. We have implemented the command design pattern in the commit called command design pattern and packaging them. We have an additional class that extends the command interface for each operation that our program can perform and we have 2 command invoker classes which executes the command class object. One of the possible design patterns that we can implement if we have more time is the strategy design pattern. Due to the time limit, we have only implemented 1 attribute for workers and 1 attribute for department heads. We are planning on expanding on this feature by allowing users to search by more attributes for workers and department heads. In this case, we thought that it would be good to apply the strategy design pattern where we can have a searcher interface and have each individual search by attributes command implement that interface and worker and department heads will just have an instance of that searcher interface. Then, depending on what the user enters, we can have a specific searcher and execute the search method within it.

Use of Github features:

We have thought of using some github features to develop our code such as pull requests and issues. But when we try to use the pull request, we are not sure for some reason that we are not able to see the pull request on our end so instead of figuring out why it is, we just decide to push the change to the main branch and we look at it there. Then if there are any problems, we just put it in the discord chat and we can discuss them in our next meeting. In terms of issues, we thought of a better and easier way to track todos, feature requests and bugs on a google doc and we can take a look at that google doc regularly to see what others need and what are some bugs that others have.

Testing:

We have tested most parts of our program including all the use case classes as well as methods in the entities class. This covers possibly all functionalities of the program. We did not test the individual commands class that we implement as it just calls the use case method. We also did not test the command invoker class as it just executes the command which are just calling methods in use case class so we think that testing the use case class is enough. We also did not test controller and presenter class as the presenter is just responsible for displaying the interface which has nothing to test and the controller just does the functionality of use case based on user input which we thought is not necessary to test. In terms of difficulties in testing, we are having difficulties in testing the use case class because we implement serialization. As we do any operations, the change will be recorded into the ser file which means that when we run the test without clearing up the file, the test will fail. Also, due to serialization, the order in which we do the operations also matters so we need to see which order IntelliJ tests them in order to make sure the expected output is correct.

Refactoring:

In terms of refactoring, we have made 2 important refactoring from phase 0. One major refactoring that we did is separate the controller and presenter into 2 different classes as our project expanded, it is more organized to separate them. We also changed the user interface from phase 0 to a more user friendly one. Also, we have moved the list of workers and department heads which is where we keep track of the workers and department heads created so far to the use case class rather than having them in entities and controller class. This refactoring makes our code adhere more to the Clean architecture as well as SOLID principles. This refactoring is done in the commit named change based on phase 0 feedback. Another refactoring that we did is using the command design pattern which decouples the controller and use case and encapsulates details of completing operations. This is done in the commit called command design pattern implementation and packaging them. One of the code smells that we observe is the long parameter code smell in our use case method to create workers and department heads. We did not do any refactoring about it as we think that this code smell has its purpose of being there. The use case method just calls the constructor of the worker or department head which needs this information. This code smell is not caused because we combine too many things into one method so we think that this is fine.

Functionality:

For functionality, our specification for phase 0 is really limited and we completed all the functionalities in phase 0 already so we expand and extend functionalities from phase 0. Our program does what the specification says. Given us only a group of 4, we think that adding 5 more features that the user allows to do with a design pattern as well as serialization is sufficiently ambitious. Also, because of serialization, our project is able to store and load state such as workers and department heads created in previous runs.

Code Style and Documentation:

There is a warning in the DemoRun class due to commented out code but it should be there for the TA to test our code so we ignore that warning. The other warning is in the serialization class. In the actual implementation of readWriter class for both worker and department heads, when we convert the object read from the ser file to arraylist, it gives a warning because it is unchecked conversion but we cannot fix this warning so we just suppress the warning. In terms of java doc, all methods are documented with java doc and some comments added where we think is needed. In terms of understandability of our code, we have reviewed each other's commit and we think that the methods name, class name and packages name are all quite descriptive so it should be easy to understand.

Progress Report:

<u>Open Questions:</u>

- We're having difficulties with testing the use case after implementing serialization because it records the previous state and the order of the test will affect the correctness
- We are wondering where should we define the readWriter and pass it into the use case class
- Whether and how to fix the long parameter list code smell when call constructor

<u>What works well:</u>

- We have tested most parts of the code and it works fine.
- We have also doing well with adding extra features and it is really easy to do so as we adhere to the clean architecture and SOLID design principle

<u>Each member's work and future work:</u>

- Shillin Zhang – Expanding the functionality of the program by implementing additional features. Incorporating SOLID and Clean Architecture into the Program. Creating test cases and implementing the command design pattern. Refactoring the program and organizing the program's code based on its features. Developing the design document with each additional change to the code.
- Ruixin Liu – Expanding the functionality of the program by implementing additional features. Incorporating SOLID and Clean Architecture into the Program. Implemented serialization into the program.
- Somtochukwu Oriaku – Expanding the functionality of the program by implementing additional features. Incorporating SOLID and Clean Architecture into the Program. Developing the design document with each additional change to the code.
- Hamza Khan - Expanding the functionality of the program by implementing additional features. Incorporating SOLID and Clean Architecture into the Program.