CSC207 Design Document:
Phase 2

Circus Group

December 8th, 2021

Contents

1 Specification

1.1 Updated Specification

Updates italicized.

Create a program with a user interface that allows users to load a new warehouse layout (made up of Tiles) given width and height specifications in number of Tiles from the user. There are designated tiles for both racks (with a given capacity) and receiving/shipping depots (which are used for receiving new items and shipping items from the warehouse). Once this warehouse layout is loaded and visualized, *users can to continue to edit it within the UI.* Similarly, users will be able to create items *to be stored in the "Part Catalogue"* by providing the item name and description, those of which will eventually be added to their warehouse. Users are able to easily add and remove items in their catalogue as well as access a list of existing items (that will be saved).

The user can create orders to both add new items to the warehouse (increase inventory by adding a given item to a specified rack), change the location of an item in the warehouse, and remove items using the interface. Once the user creates an order for any of the previously mentioned reasons, it will be added to an order queue (possibly behind other orders) so the user can track their order progress as it is processed in the warehouse. The user can make two possible orders: one that receives a new item in the designated depot and places it in the warehouse, and one that ships an item already in the warehouse via the shipping depot. Once an order reaches the front of the queue (*and the order can be processed, i.e. there is a Rack in the Warehouse available for this item*), a robot will be assigned to moving it to it's designated location using a route-finding algorithm to reach the item, pick it up, and move it to its destination via the most efficient path. This robot will only travel in designated paths (will avoid racks, robots, and other obstacles).

2 Design

2.1 Summary

We have attempted to incorporate Clean Architecture, the SOLID principles, and design patterns where relevant throughout the development of our project. We kept our input and output methods in the outer layers of our implementation and when crossing boundaries between inner and outer layers of code, we used interfaces (keeping our code from inconvenient dependency). Our adherence to these principles and guidelines is outlined below:

2.2 Clean Architecture

- Program follows Dependency Rule as displayed in our UML Diagram (zoom in for details).
  - Example frequently used entity `Item` class: Since the `Item` class is an entity it does not rely on any other class/methods (refer to imports). Any class dependant on `Item` is contained in the `warehouse` package (see Packaging Strategies below), similarly any class dependant on `Item` is an entity or use case (application or enterprise business rules).
  - Example in the outer Frameworks and Drivers Layer (the `User Interface)`. This class relies largely on the `WarehouseController` and `WarehouseState` classes, both of which are interface adapters thus adhering to the Dependency Rule. One note is that this class relies on the `PartCatalogue` which is required in the construction of a new `WarehouseState`. This is one aspect of Clean Architecture we would need to work on and could possibly fix by constructing the `PartCatalogue` in the `WarehouseState` constructor itself rather than requiring it as a parameter.

2.2.1 Scenario Walk-Through

Suppose a user wishes to create a new warehouse layout, create an item, and create an order to place this item in their warehouse. The user would first create their new warehouse using the `WarehouseController` via the user interface (user interface has dependency on the controller). During this process the user can input specifications for the phyiscal parameters of the warehouse to load an create a new warehouse layout. They would then be able to establish Rack and Ship/Recieve Depot entities in the warehouse. The `WarehouseController` constructs a `WarehouseState` which constructs a `Warehouse`, a use case, which then contains `Tiles` that are assigned a rack or depot by the user. The user will then go on to create insances of `Item`, an entity, and populate the `Warehouse`'s racks as previously mentioned. They do so by creating `Orders` (i.e. by instantiating one of the concrete implementation of the `Order` entity class, e.g. a `PlaceOrder` object), and then adding it to the `OrderQueue` (use case). This is then processed by the `WarehouseController`, which is responsible for orchestrating actions within the warehouse.

Our program will then employ a pathfinder algorithm in order to calculate the most efficient route to where there is available storage space in the warehouse. This `Rack` will then update to reflect what item it is storing, the number that item it is storing, etc.

2.3 SOLID Design Principles

- **Single Responsibility Principle**: Each module is isolated into functional components those of which have their own responsibilities (similarly, each class has it's own responsibility as well).
  - **Example:** the warehouse module which is responsible for creating and editing the warehouse. Each module within this warehouse module is additionally responsible for their exclusive functions, i.e. the inventory module which is responsible for implementing any functionality related to the warehouse's inventory system; that is, the `Item` and `Part` entities, along with the `PartCatalogue` use case.

- **Open/Closed Principle, Liskov Substitution Principle, & Interface Segregation Principle**: Maintained Open/Close Principle throughout project development as well as when refactoring. This allowed us to easily extend classes without worry of dependency issues. Liskov Substitution Principle is observed wherever an interface is implemented.
  - **Example:** the `Distributable` and the `Receivable` classes allow us to add any number of Distributable/Receivable subtypes, like `Rack` (making it open for extension) without having to re-write affected classes. Using the same example, any subtype of `Distributable` or `Receivable` can be substituted for any other Distributable/Receivable subtypes which is an example of the Liskov Substitution Principle. Similarly, the Distributable and Receivable interfaces both uphold the Interface Segregation Principle. Rather than creating one large interface that would force us to implement irrelevant methods in some of the implementing classes, we separated this interface into two. This ensured interfaces remained small to maintain ease of extending/modifying the design.
- **Dependency Inversion Principle**: Attempted to maintain this throughout development in order to ensure the appropriate flow of control throughout our code
  - **Example:** the `StorageUnitStrategy` interface which introduces interface abstraction for the multi and single type storage strategies. Here, since the dependency inversion principle is followed, any high-level modules can depend on the abstraction, rather than *either* just the multi or single type storage strategies. An example is the `StorageUnit` class where the `canAddItem` method depends on the `StorageUnitStrategy` abstraction which means calls to strategy methods will not be affected by changes in both the `MultiTypeStorageUnitStrategy` or the `SingleTypeStorageUnitStrategy`.

2.4 Design Patterns & Refactoring

Some design patterns to highlight in our code as well as relevant pull requests, much of phase 1 was centred on refactoring for clean design:

- **Builder Design Pattern** in the `Item` class as implemented in PR#37, and then later refined in PR#40.
- **Strategy and Template Design Patterns** in the `StorageUnit` classes (see above for `StorageUnitStrategy`) found in PR#31.
- **Observer Pattern** in the application messaging system and UI events (see `ComponentEventListener`). Implemented in PR#31.
- **Command Pattern** in the shell/command-line interface app framework.
- **(Simple) Factory Design Pattern** for the Tiles.

Additionally, there is further opportunity to implement design patterns for the pathfinder algorithm for example since we have both `complexPathfinding` and `concretePathfinding`.

2.5 Packaging Strategies & Code Organization

We chose to package **by component**. Although packaging by feature was considered, it was evident that packaging by component would best organize our classes since we have so many components (that make up larger ones, like the Tiles in our Warehouse). Similarly, our tests mirror the package structure found in our main folder. See Fig 1 for a diagram of our packaging.

2.6 Code Style, Documentation & GitHub Usage

- We have used Javadoc in each of our classes and have documented each of our methods as well as classes. This ensured team members were made aware of what changes were being made and how to use these changes in their own code.
- Throughout the second phase we made a conscious effort to increase our usage of Pull Requests, which is something we highlighted in Phase 1 to be worked on. This ensured that no incoming branches would inhibit program functionality and additionally enforced code readability.

- We made great use of the Issues feature in GitHub. This ensured team members were aware of what tasks remained incomplete and who was working on what feature. Similarly, this allowed us to connect issues to specific commits/branches to communicate any updates to other team members. Collectively we created a little over forty issues which have, for the most part, all been closed.
- Set up GitHub Actions to collectively monitor the ongoing functionality of our program, additionally aiding pull request reviewers.

3 Functionality

- Program does what is outlined in the specification as required and demo'd in our presentation. Have increased user accessibility in our user interface (more intuitive) as outlined in our accessibility report below.
- We believe the overall function of our program is sufficiently ambitious, given the fact that we have chosen to implement complicated additional components to our code such as pathfinders, warehouse robots, as well as a interactive GUI that were not guaranteed from the beginning.
- Between phase 1 and phase 2 we were able to implement a more complex pathfinding algorithm using the A* pathfinding algorithm[1] as well as implement the Robots and their functionality. Additionally, much of the UI was created in this phase including the addition of interactivity.
- Our program can store and load state as found in the serialization package. This package provides a generic adapter for serializing any object, in any format (provided that an implemented of `FileObjectLoader` `FileObjectSaver` is given). The concrete implementations `JsonFileObjectLoader` and `JsonFileObjectSaver` are leveraged by the UI layer to save and load the `WarehouseState`. This is then used by the editor to display the warehouse and perform order-routing.

3.1 Testing

- Reached 63% test coverage (see Fig 2)

4 Accessibility Report

How does our program adhere to the Principles of Universal Design and how could we further adjust it to increase accessibility?

Principles of Universal Design:

1. **Principle 1: Equitable Use** Every user has equal access to all functionalities of our program and we do not have additional incentives for certain users nor do we segregate/stigmatize users in any way. Additionally, our goal with our UI was to ensure our design is intuitive for all users.
2. **Principle 2: Flexibility in Use** Our UI is designed to be highly flexible with a variety of customize features. The UI is fully themeable, with a WarehouseColourScheme class that allows for change in every style option in the warehouse editor. Similarly, the UI can be resized and editor windows within the app itself are movable and dockable (apart from the warehouse editor), allowing users to customize the layout of their editor to their liking. Similarly, users are offered the ability to zoom in/out of the warehouse layout in the UI: this enhances the user's accuracy and precision if need be (dependent on the user's preferences). Finally, our system supports arbitrary sized warehouses with arbitrary parts which allows for further user flexibility.
3. **Principle 3: Simple and Intuitive Use** Our program is relatively simple and intuitive to use and this is enhanced by some of the features in the UI. For example, the toolbar to draw, remove, add, etc. robots and racks has symbols to represent each tool which accommodates for a wide range of literacy and language skills and eliminates any unnecessary complexity. Similarly our program provides effective prompting and feedback, for example when the user tries to add an item to the warehouse when it is not in their catalogue, they will be met with a warning explaining why this action is impossible.

4. **Principle 4: Perceptible Information** Since our program is relatively intuitive, instructions are not necessary in all cases however there are examples of inputs were inputs are required and every element is labeled and legible. The GUI itself is "dark" so elements stand out against the dark background which might aid those with sensory limitations, similarly important aspects of the warehouse have been designed in bright colours to emphasize their positions.

5. **Principle 5: Tolerance for Error** The goal of our software is to be used with actual robots in a warehouse. Our algorithms are constantly checking for viable paths and valid movements which minimizes overall errors. Similarly, if mistakes are made by a user, (i.e. a user attempts to add an item not in the warehouse parts catalogue) they will be met with a warning straightforward instructions as to why their action failed.

6. **Principle 6: Low Physical Effort** This is one principle we could work on. Drawing in racks is simple, efficient, and not tiring when the warehouse size is relatively small. However, on a larger scale this could prove to be difficult, repetitive, and will require more physical effort than what is necessary. Thus we could implement "rack templates" that automatically organize racks in a certain way given the warehouse height and width.

7. **Principle 7: Size and Space for Approach and Use** This principle is not entirely applicable to our program. The elements of our UI are clear to any seated or standing user and reach to all components is comfortable for, again, any seated or standing user. Since our program is not a mobile app, there is no need to accommodate variations in hand and grip size nor provide adequate space for the user of assistive devices.

Our program would be marketed towards companies/corporations that require large warehouses and software for warehouse management. On a smaller scale our program may not be necessary as we employ robots opposed to people, thus we would target companies with large shipping fulfillment centres that already have the capability and software to use robots in their warehouse. Our program would allow said companies to automate their warehouses, ultimately decreasing the human labour required and minimizing potential for error.
Our program is evidently not likely to be used by individuals nor those who don't require it's services. As mentioned above, it will likely be used by those who already have a relatively large warehouse system in place already as well as the capability to use robots as opposed to people in their warehouse operations. Our program is quite specific in it's target audience, thus there is a large demographic that would likely not find much use of our program.

5 Progress Report

Phase 2 was centered on completing the UI, implementing the Robot functionality, implementing the complex pathfinder using the A* algorithm, updating and completing tests, as well as implementing the ability to load and save the state of our program. There was some additional refactoring to incorporate more design patterns and clean up some code.

5.1 Artem

- Phase 0
  - Worked on the design and the implementation of the core structure of the program.
  - Started working on the implementation and design of Simple Pathfinder.
  - Worked on the initial tests for the basic classes.
- Phase 1
  - Contributed to the initial development of the UI by researching different UI framworks.
  - Implemented and designed the first simple version of the path finder, which was used as the basis for more advanced pathfinder implementation (Simple Pathfinder)
  - Refactored Phase 0 code to adhere to SOLID and Clean Architecture principles.
- Phase 2
  - Worked with Liam, finalized a concrete implementation of the path finder, with a decision to use the Pathfinder controller. It allowed us to avoid all unneeded dependencies. In addition, addition of the other pathfinders to the program and their implementation became easier. (Pathfinding functionality)

  - Introduced additional testing classes for the pathfinder Controller and implemented various Design pattern to keep the code maintainable and extensible.
  - Continued working on bug fixing and restructuring, helped other members of the group with their tasks.

5.2 Audrey

- Phase 0
  - Worked on original project structure + design plan
  - Created CRC cards
  - Wrote phase 0 design document
- Phase 1
  - Research/experimented with JavaFX as a potential for UI
  - Worked on unit tests for Warehouse components (OrderQueue, WarehouseController, and StorageUnits)
  - Implemented OrderQueue system
- Phase 2
  - Wrote phase 2 design document
  - Implemented the order-robot matching system

5.3 Fiony

- Phase 0
  - Contributed to project structure and design
  - Assisted teammates with entity classes of the program
- Phase 1
  - *Didn't contribute anything in code - had issues with IntelliJ recognizing my folders as modules.
  - Researched how to write unit tests for the pathfinding classes my teammates wrote
  - Worked on unit testing WarehouseController
- Phase 2
  - Worked on unit testing - Warehouse inventory classes

5.4 Liam

- Phase 0
  - Contributed to initial project structure
  - Worked on Pathfinder Skeleton
  - Created initial tests
- Phase 1
  - Refactored starter classes with PR
- Phase 2
  - Finalized a concrete implementation of the Pathfinding functionality
  - Introduced additional testing classes
  - Bug fixing, restructuring, and testing
- Pathfinding was most significant contribution

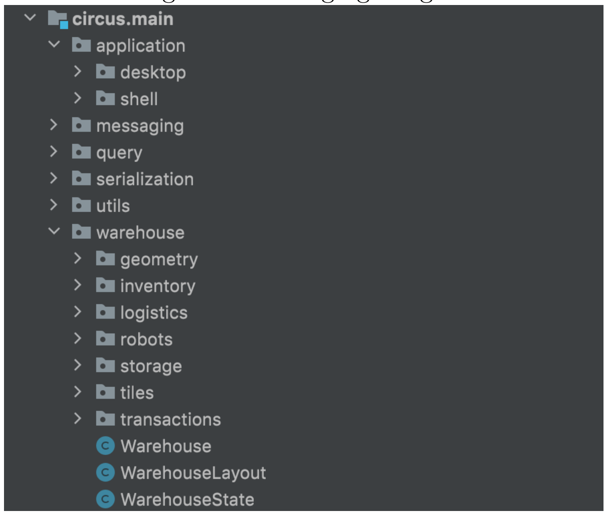
5.5 Shon

- Phase 0
  - I worked on designing the architecture of the program, and implemented the basics of the shell application.
  - Also worked on designing and implementing a query system for the items, which will allow our agents to retrieve items from the warehouse based on generic queries.
- Phase 1
  - Contributed to the initial development of the UI. Researched different UI frameworks/libraries and helped the team come to a decision on which would work best for us—we eventually settled on imgui-java.
  - Wrapped ImGui framework in a component-based framework. Used the Observer pattern to implement a messaging system for UI events.
  - Refactored Phase 0 code to adhere to SOLID and Clean Architecture principles.
  - In PR#31, I applied the *Strategy Pattern* and *Dependency Injection Principle* to refactor the `StorageUnit` class into several smaller components: `StorageUnitStrategy` and `StorageUnitContainer`. This refactor was critical as it provided an elegant way of specifying the behaviour of different storage units (e.g. one storage unit can have an insertion strategy that acts as a filter, while another storage unit can have an insertion strategy that acts as a data validator). Also abstracted the underlying data representation from the storage unit (i.e. the storage unit can be implemented as an array, database, or anything...).
  - In PR#40, I helped Liam refactor the inventory system. Using the Builder pattern, we split the inventory into `Items` and `Parts`, where a part serves as the *blueprint* to an item (i.e. the builder).
- Phase 2
  - Continued working on the logistics backend. In PR#45, built upon Audrey's Order system implementation and integrated it with the rest of the Warehouse system.
  - Continued working on the user interface (desktop application). Implemented the editor panels (warehouse and part catalogue editors), inspector/properties panel, toolbars, and added various UI enhancements.

5.6 Vic

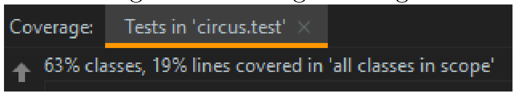
6 Figures

Figure 1: Packaging Diagram



```
circus.main
  application
    desktop
    shell
  messaging
  query
  serialization
  utils
  warehouse
    geometry
    inventory
    logistics
    robots
    storage
    tiles
    transactions
    Warehouse
    WarehouseLayout
    WarehouseState
```

Figure 2: Testing Coverage



63% classes, 19% lines covered in 'all classes in scope'

References

[1] Cox, G. (2021, January 21). *Implementing A* pathfinding in Java*. Baeldung. Retrieved December 5, 2021, from https://www.baeldung.com/java-a-star-pathfinding.