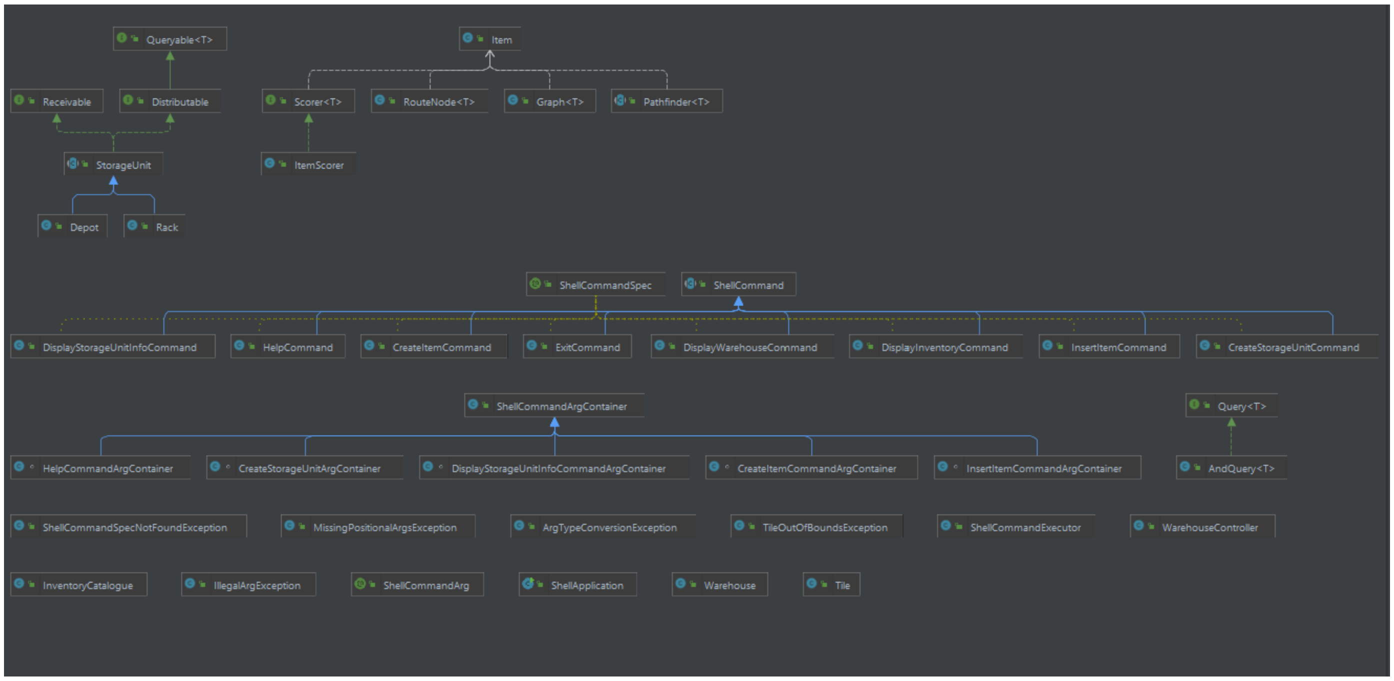
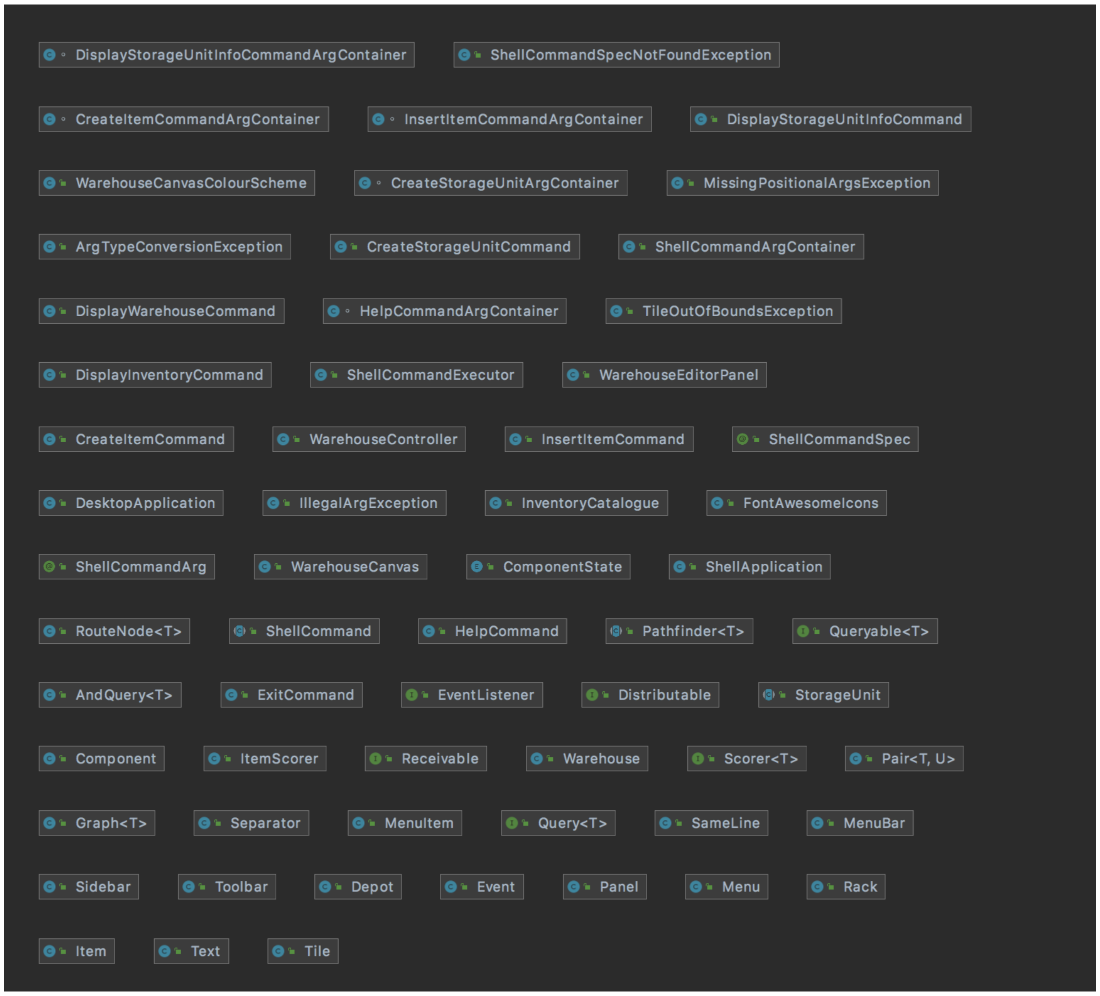
CSC207 DESIGN DOCUMENT PHASE 1

Group Name: Circus

Clean Architecture, Solid Principles and Organization

We made sure to follow the Single Responsibility Principle by isolating each module of our development into functional components. This was helpful in creating individual tasks for group-members to complete, because methods did not rely on other external models/ functionalities. We maintained the Open/ Closed Principle throughout the project, particularly during the refactoring process. Adhering to this principle made it easy to extend classes without having to worry about dependency problems. Similarly to the OCP, the Liskov Substitution Principle is maintained throughout our code, and is observed wherever an interface is implemented. Furthermore, we segregated our interfaces, (following ISP), in many places in our code, such as our implementation of Receivable and Distributable. Finally, the Dependency Inversion Principle can be found across our code. For example in the ShellApplication where our commands are declared but not implemented. We did our best to have this rule consistently implemented across our code in order to maintain the appropriate flow of control throughout our code. The code for our program as well as the testing files are organized in a logical manner, based on interface and functionality. Navigating our packages is intuitive, and they are organized in a few major folders.The hierarchical structure of our code is both functional and inline with Clean Architecture. As we worked we kept our options open to maximize the flexibility of our code. Furthermore we kept our input and output methods in the outer layers of our implementation. When crossing the boundaries of inner and outer layers of code, we use interfaces. This kept our code from inconvenient dependency. All of these instances are in line with clean architecture and benefitted the development of our program by keeping us organized.


Queryable<T>
Item
Receivable
Distributable
Scorer<T>
RouteNode<T>
Graph<T>
Pathfinder<T>
StorageUnit
ItemScorer
Depot
Rack
ShellCommandSpec
ShellCommand
DisplayStorageUnitInfoCommand
HelpCommand
CreateItemCommand
ExitCommand
DisplayWarehouseCommand
DisplayInventoryCommand
InsertItemCommand
CreateStorageUnitCommand
ShellCommandArgContainer
Query<T>
HelpCommandArgContainer
CreateStorageUnitArgContainer
DisplayStorageUnitInfoCommandArgContainer
CreateItemCommandArgContainer
InsertItemCommandArgContainer
AndQuery<T>
ShellCommandSpecNotFoundException
MissingPositionalArgsException
ArgTypeConversionException
TileOutOfBoundsException
ShellCommandExecutor
WarehouseController
InventoryCatalogue
IllegalArgException
ShellCommandArg
ShellApplication
Warehouse
Tile


DisplayStorageUnitInfoCommandArgContainer
ShellCommandSpecNotFoundException
CreateItemCommandArgContainer
InsertItemCommandArgContainer
DisplayStorageUnitInfoCommand
WarehouseCanvasColourScheme
CreateStorageUnitArgContainer
MissingPositionalArgsException
ArgTypeConversionException
CreateStorageUnitCommand
ShellCommandArgContainer
DisplayWarehouseCommand
HelpCommandArgContainer
TileOutOfBoundsException
DisplayInventoryCommand
ShellCommandExecutor
WarehouseEditorPanel
CreateItemCommand
WarehouseController
InsertItemCommand
ShellCommandSpec
DesktopApplication
IllegalArgException
InventoryCatalogue
FontAwesomeIcons
ShellCommandArg
WarehouseCanvas
ComponentState
ShellApplication
RouteNode<T>
ShellCommand
HelpCommand
Pathfinder<T>
Queryable<T>
AndQuery<T>
ExitCommand
EventListener
Distributable
StorageUnit
Component
ItemScorer
Receivable
Warehouse
Scorer<T>
Pair<T, U>
Graph<T>
Separator
MenuItem
Query<T>
SameLine
MenuBar
Sidebar
Toolbar
Depot
Event
Panel
Menu
Rack
Item
Text
Tile

Scenario Walkthrough and UI Example

A User who wishes to create a new warehouse layout would do so using the WarehouseController. They would then be able to establish Racks and StorageUnit. During this process the user can input specifications for the physical parameters of the warehouse to load and create a new warehouse layout. The command shell visualizes the warehouse in grid form with which the user can interface, zooming in and out and scrolling to navigate the screen. The User would then go on to great instances of Item and populate the warehouse. The details necessary for this include the item name, description, and id. Once this item is created, the user can then create an order to add this item to the warehouse. The program will then employ our pathfinder algorithm in order to calculate the most efficient route to where there is available storage space. The algorithm will also endeavor to navigate to the nearest available rack for maximum efficiency. The rack will then update to reflect what item it is storing.

Design Patterns and Refactoring

Throughout phase 1 we focus on refactoring to ensure our code adhered to Clean Architecture and SOLID design principles as much as possible. For example, the storage units were refactored, implementing the strategy pattern and the template pattern for cleaner navigation and easier to understand functionality. We also refactored the pathfinder in order to have it adhere more clearly to clean architecture. We have had the opportunity to implement many design patterns throughout our code. In our design that creates and tracks UI events, we implemented the observer pattern. This pattern can be found in the events folder, specifically in Event and EventListener. Furthermore, our code implements the command pattern for the shell. This can be found in the commands folder, where the pattern is employed in creating commands for Item, StorageUnit, Inventory, and Warehouse. We also refactored the Item class to implement the Builder design pattern, this can be seen in our pull requests. Finally, StorageUnit was refactored in this phase as well.This is substantial progress since phase 0, when we weren't collectively sure on which design patterns would work best for our implementation.

Functionality and Testing

For all the code that we have finished so far, we have included detailed specifications. While there remains some testing to be done, the functionality of the code is in line with the descriptions included. Based on the names of the packages, the functionality is clear and hence easy to navigate and understand. Even if someone unfamiliar were to select a random file, the JavaDoc we have for each interface and method is sufficiently descriptive to communicate the context in which the file is situated in the greater code. We believe that the overall function of our program is sufficiently ambitious, given the fact that we have chosen to implement complicated extra components to our code such as a pathfinder, warehouse robots as well as a visual display representation of our warehouse. As our code gets increasingly complex, we must implement more tests to keep up with the enhanced functionality. There remains some testing to be done for our code, especially in our UI and pathfinder, because we have yet to fully implement the functional details of the code. Therefore we are expecting to have progressively more tests as we get closer to finishing our project. Due to dependency inversion in our shell file, fully comprehensive testing for all of the commands will be best done when most of the implementation is complete.

Future Considerations

Our group was very active in our use of Github functions, especially in creating Issues that we could comment on to claim. We also made use of Actions to collectively monitor the ongoing functionality of our program. In the future we can make better use of pull requests, because they are more formal than communication via chat platforms that replaced this process. We still have some code smells present, particularly in the UI code. We hope to abstract and refactor DesktopApplication so that it will be handled by separate classes. This is to improve the functionality of the UI. In terms of load state, we still have some implementation to do. Thus far we have a basic json saving and loading system, but we hope to improve the functionality of this as well.