Academic Honesty Statement

I understand that my learning is dependent on individual effort and struggle, and I acknowledge that this assignment is a 100% original work and that I received no other assistance other than what is listed here.

Acknowledgements and assistance received:

- Discussions with Dr.G and Ari about my project design

I did not use generative AI in any form to create this content and the final content was not adapted from generative AI created content.

I did not view content from any one else's submission including submissions from previous semesters nor am I submitting someone else's previous work in part or in whole.

I am the only creator for this content. All sections are my work and no one else's with the exception being any starter content provided by the instructor. If asked to explain any part of this content, I will be able to.

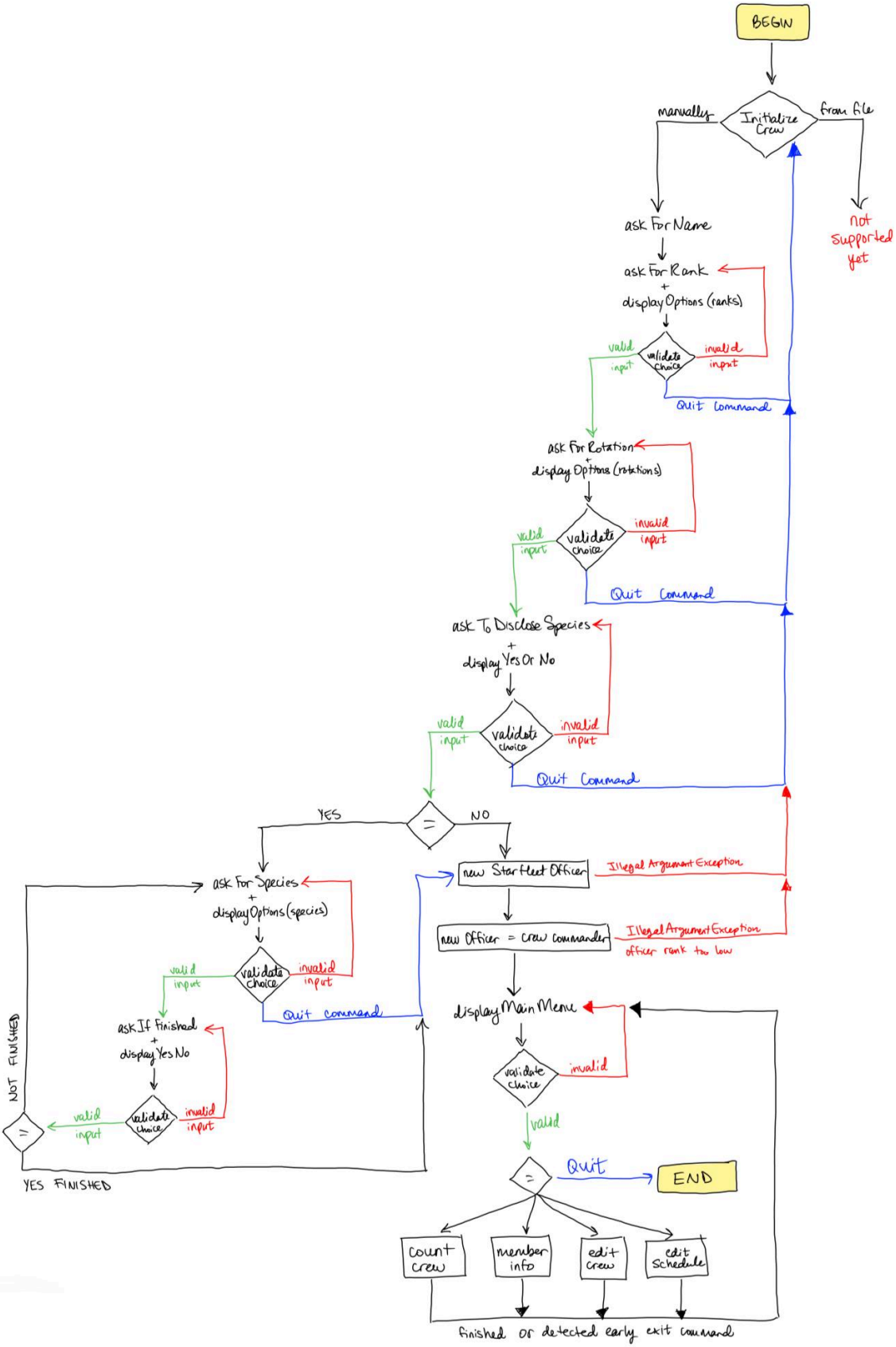
By putting your name and date here you acknowledge that all of the above is true and you acknowledge that lying on this form is a violation of academic integrity and will result in no credit on this assignment and possible further repercussions as determined by the Khoury Academic Integrity Committee.

Name: Estelita Chen	Date: 4/16/2024

Introduction

My main program is a Starfleet Crew Management simulation. The crew members are organized in a tree inside of the crew model, and each crew member has an instance of the shift class. The user interface isn't fully implemented yet, but I marked those methods with //TODO comments and also wrote some pseudocode. I made a smaller application in the starships package to show Interface Segregation that was based off of my original idea to have a Starship class that contains a crew that then contains crew members. (Note: UML diagram is too large to put here so please look in the main folder for the UML)

flow chart for controller menus



GitHub Link for code:

https://github.com/estelita24601/Final_Project_5004

Recursion in Practice - a logical use of recursion that simplifies your code

`BranchNode<T>` class in treeADT package

- line 47 in the `countIf` method
    - the node will count itself if required, and then every single child will use `countIf` on itself as well before returning the total.
    - this will keep on going until every single node in the subtree has been counted. aka until we get to nodes that don't have children to recursively call the `countIf` method
- line 88 in the `findNode` method
    - instead of manually traversing the entire tree I used recursion.
    - if the current node isn't the desired node then each of it's children will use `findNode` on itself until either the node is found or we have finished looking through all the children and can say that the node is nowhere in this subtree.
- many other methods in this class also use recursion to do things to the node and all of its children/descendants.
    - line 110 in `fold` method
    - line 123 in `filterToList` method
    - line 137 in `mapToList` method
    - line 262 in `map` method

Logical Structure and Design using Abstract Classes and Interfaces

Interfaces for the model, controller and view of my program

- `ICrewModel` in the model package
    - This interface then specifically has a compositional relationship with the `ICrewMember` interface so that any given implementation of the crew model can be composed of any object that implements `ICrewMember`
- `ICrewController` in the controller package
- `ICrewView` in the view package

treeADT package

- `ITree<T>` is an interface for my tree data structure that defines what every node in the tree should be able to do and allows them to be interchangeable
- `TreeNode<T>` is an abstract class that should never be used but passes down a lot of code to `BranchNode` and `LeafNode`
    - `BranchNode` uses the code it inherits as a way to apply the methods to itself before going through the entire subtree below it
    - `LeafNode` has no children so uses the inherited code as is and then implements the abstract methods about children nodes so that it doesn't break

starships package

- has interfaces for some of the main capabilities a space ship would have
- then each ship class chooses which of them to implement
- for example `BirdOfPrey` has a cloak and thus implements the `Cloakable` interface while `StarFleetShip` does not. However they can both still be used interchangeably as classes that implement `CombatShip`

Useful and Logical Abstraction using Generics and Lambda Expressions

treeADT package

- All of the classes here use generics so that a tree could be constructed with any data type. This can be seen in the `TreeTest` class in my tests folder where I create one tree that has String data in the nodes, another with integers and the last one with StarFleetOfficers

`Predicate<T>` lambdas

- were used for all the counting methods, filtering methods and also used to find which node we want to perform an operation on.
- this way we can count, filter and find nodes based on any attribute their data has. for the crew members stored in this tree that means we can count, filter and find based on name, rank, department etc.
- locations in `ITree` interface
    - line 23 `moveChildren`
    - line 27 `countIf`
    - line 35 `addChild`
    - line 39 `deleteChild`
    - line 45 `findNode`

    - line 49 `filterToList`

`Function<T, R>` lambdas

- were used for mapping and converting the data in nodes to whatever data type specified.
- locations in `ITree` interface
    - line 51 `map`
    - line 53 `mapToList`

`BiFunction<R,T,R>` lambdas

- Were used mostly for folding. The first parameter of type R is the running total/initial value we start with. The second parameter T is the data in the node of the tree. they return a value of type R that's the result of adding T to the running total.
- location in `ITree` interface
    - line 47 `fold`

`Consumer<T>` lambdas

- These are lambdas that don't return anything out, they would accept the data from the node in the tree as an argument, make modifications to that and then end
- location in `ITree` interface
    - line 15 `editData`

Concrete usages in `StarFleetCrew` class in the model package

- `getMemberInfoList` method at line 141
    - line 142 use the predicate to filter the tree and return a list of crew members that passed the filter
    - line 145 use the function to convert each crew member into a string before adding it to the final list that we return out
    - this way we can filter crew members and then get their information in any way we want
    - example get all the crew members in engineering, but only their name and ranks since we already know they work in engineering
- `editCrewMember` method at line 182
    - line 183 we use the predicate to find the specific crew member we want to edit
    - line 184 we use the consumer to edit
    - this way we can edit crew members any way we want without having to create new methods for every possible edit.

Other Examples

- `FindByName` in the model package
    - functional class that implements `Predicate<ICrewMember>`
    - will compare a crew member's name to the string it received upon initialization
    - used in lines 199 and 212 of `StarFleetCrewTest` in the tests folder
- `ViewDisplayer` in view package
    - a functional interface that only has the method `display()`
    - used in the controller to specify what you want the view to do. This allowed me to make some more generic menus that would have the same options to choose from but would accept a view displayer to determine the initial prompt given to the user.
- in general just all over the `ICrewModel` from the model package and the `StarFleetCommand` class in the controller package

Higher Order Functions Map, Filter, and Fold

Map

- `map` method in line 51 in `ITree` interface of the treeADT package
- implemented on line 66 of `LeafNode` class and line 258 of `BranchNode` class
    - the branch node converts its data and creates a new node, it then goes through each child, maps it to a new node with the new data type and adds it to the children of the new node. the children also call map recursively so the mapped node they return will include all of their children/grandchildren.

Filter

- `filterToList` method at line 49 of `ITree` interface of the treeADT package
- implemented on line 97 of `TreeNode` class and line 120 of `BranchNode` class
    - the node converts it's data into a list of 1, it then recursively calls `filterToList` on all the children and adds the resulting list from all of the children to that initial list and then returns.

Fold

- `fold` method at line 47 of `ITree` interface
    - implemented in line 107 of `BranchNode` and line 92 of `TreeNode`
    - takes in a bifunction that will combine our running "total" with the data in the current node.

- `countIf` and `countAll` methods of `Itree` at lines 27 and 29 respectively
    - also implemented in `BranchNode` and `TreeNode` the only difference is that it folds the tree down to an integer

Hierarchical Data Representation as an ADT or a Linked List ADT

treeADT package

- `ITree` interface for the data structure
- `TreeNode` abstract class that passes code down to `BranchNode` and `TreeNode`
    - `BranchNode` a node in our tree adt that can have child nodes
    - `LeafNode` a node in our tree that cannot have child nodes
- Each tree node is composed of one parent and an ArrayList of child nodes. If a node is the root of the tree then it's parent is null. If a node is one of the leaves of the tree then its ArrayList of child nodes is empty. The `LeafNode` class overrides functions so that it can never have children added to it and it's always a leaf of the tree. The `BranchNode` class *can* have an instance with no children that is the leaf of a tree, but it can also be the root node or one of the branch nodes

Architectures and Design Patterns MVC Design

- I have an interface for my Model (`ICrewModel`) my controller (`ICrewController`) and my View (`ICrewView`). My implementation of the view `TextView` only prints messages to the terminal when the controller asks it to. My model implementation `StarFleetCrew` only makes changes to it's internal state when told to by the controller. The controller implementation `StarFleetCommand` dictates the control flow and menu navigation. It tells the view to display a list of options each with its own number. The controller then takes the user input and makes sure it matches the options it told the view to show them. Then depending on where in the menu it is the controller will call on the model to do something.

SOLID Design Principles

Single Responsibility Principle

- I used this concept in my model package. The `StarFleetCrew` class only has to worry about managing a tree adt with all the crew members. The actual implementation details of that tree is delegated to the tree node objects. The `StarFleetOfficer` class is only responsible for information about each individual crew member, and further delegates responsibility for scheduling and shift management to the `Shift` class. Another example of this in my code is how the `StarFleetCSVReader` class is responsible for reading in a csv file instead of having `StarFleetCrew` be responsible for it.
- Another example is my `createCaptain` method in line 185 of the `StarFleetCommand` class (controller package).
    - This method is trying to create a captain for the crew using the input from the user. It's only responsibility is to run the logic for this sub menu, keep on looping until they successfully create a captain or until they decide to quit. The responsibility for actually getting user input and turning it into an instance of `ICrewMember` is delegated to the `createCrewMember` method on line 216. `createCrewMember` in turn is only responsible for running the menu until the user decides to quit or until the user gives all the necessary information for creating a crew member. It delegates again to methods like `getRank`, `getDepartment` and so forth to deal with the logistics of prompting the user and receiving their input.

Open/Closed Principle

- The `ICrewModel` and `ICrewMember` interfaces are closed for modification, but open to extension. You could easily create a new implementation of `ICrewModel` that used a linked list instead of a tree, and it would still be able to use `StarFleetOfficer` objects since they implement the `ICrewMember` class. And the same vice versa, a new implementation of `ICrewMember` could be used with any implementation of `ICrewModel`.
    - If you wanted to add more capabilities to the crew you could make another interface and have a new or existing crew implementation use the new interface without having to change anything from the original interface or implementation. For example I could make an `ICombatCrew` interface that says my crew has to be able to remove multiple people at once to send them all out to battle together. I could add this to the `StarFleetCrew` class easily without having to modify any of the existing code.
- Another example is with the `Shift`, `Department`, `Rank`, `Species` enums. You can add new values to the enum and rearrange the order they're in without

needing to change any code in the rest of the program. This is possible because our `StarFleetOfficer` class is composed of the shift, department, rank and species enums and will continue to function as long as it receives valid enum values. When the `StarFleetOfficer` wants to create a string representation of itself it can use the `toString` override in all of the enums to represent the data without having to use switch statements/if else statements specific to what values the enum currently has. `StarFleetCommand` also can prompt the user for the enums without needing to know exactly what all the possible enum values are. I already took advantage of this for the `Species` enum and added hologram, ocampan and borg so I could make specific members of the voyager crew have the correct species listed. And I didn't need to make any changes to the model view or controller.

Liskov Substitution Principle

- With the `ICrewMember` interface I am able to switch any subclasses that implement that interface in and out of my crew models. My `StarFleetCrew` class only ever refers to `ICrewMember` and doesn't refer to anything specific to `StarFleetOfficer` even though that is the class I end up using with it in my MVC. The same could be said for `ICrewModel`, my star fleet officers would be able to be used by any crew model.
- My model and controller packages also allow for substitution, if I wanted to make a GUI I could implement `ICrewView` and my `StarFleetCommand` class would still work as expected.

Interface Segregation Principle

- In the starships package I created a small example of interface segregation. If you look at all the different kinds of ships in star trek they have different abilities, so making just one interface called "StarTrekShip" with all the possible methods wouldn't be very good.
- Here I've split it up so each ship class only needs to implement what it needs. `Freighter` has all the functionality of `SpaceFaringVessel` interface, but it doesn't have a cloak or combat abilities so having methods like `activateCloak` (`Cloakable` interface line 4) would be pointless and confusing.
- `BirdOfPrey` and `StarFleetShip` both implement `CombatShip` but starfleet doesn't have cloaks so only `BirdOfPrey` has to implement `Cloakable`

Dependency Inversion Principle

- The way my `ICrewView`, `ICrewModel` and `ICrewController` are set up means that even though the three have dependency on each other none of them

are dependent on a specific implementation of one another. My `ICrewController` could function the same with any lower level class that implements `ICrewModel` without having to change any of it's code. Even if I had a `RomulanEmpireCrew` my controller could still use an instance of that class since it would have all the methods defined in `ICrewModel` that behave more or less as we expect even if there are some small differences.