Designing and Implementing an Ordered List

Summary: The goal of this assignment is to implement an ADT, and gain experience with layered architecture and choosing appropriate data structures..

1 Background

Learning Objectives:

- LO1: Describe the concept of a layered data structure.
- LO2: Select the appropriate data structure (array or link) to implement a collection ADT.
- LO3: Apply knowledge of data structures to develop an ordered List ADT. (EM@FSE G)
- LO4: Integrate different kinds of knowledge when justifying a data structure choice. (EM@FSE Q)

In this assignment, you will be implementing an Ordered List ADT (idea from Java Software Structures, Lewis and Chase, 2014). An ordered list is a collection where the elements are kept ordered, and where elements may be added or removed. Note that we are building a List ADT - this is slightly different than a linked list data structure. Remember that ADTs are not obligated to be implemented in a specific way. It is possible to implement a list with either an array (an "ArrayList") or a linked approached (a "LinkedList"). To disambiguate terms: we use *list* to refer to operations (ADT API), *linked* to refer to a low level implementation using nodes containing next references, and *linked list* to a ADT with list operations implemented with linked nodes.

This homework introduces and practices two important ideas when working with data structures: the need to develop data structures using an architectural *design*, and *selecting* the appropriate data structure for a problem. Design is a deep topic and involves virtually all aspects of software engineering. In this assignment, we'll apply the idea of a layered system design (see Section 1.1) to structure our program. The structure prescribes how the different elements in our system work together. The decision to use this particular design has already made - we'll just practice working within that context. In more advance classes, you will begin to make these types of high-level decisions that are later reflected in lower level programming. However, it will be up to you to select the data structure you use to implement the ADT. The basic choice is between an array or a list. For a list, you have further choices: it may be singly or doubly linked. The implementation may or may not include a head pointer. It may or may not include a tail pointer. Determining the correct choice of a data structure requires looking at a scenario (potentially: a customer requirement) and deducing what underlying implementation will provide efficient functionality.

The first step in the assignment will be to analyze a scenario and determine an appropriate data structure. You will then implement the ordered list ADT using that structure. The ADT will be part of a "layered" architecture. The goal is to avoid a monolithic design that is hard to change. For example, Lewis provides a series of ADTs for lists that separate the basic list functionality (removing elements, reading elements) from the functionality which adds elements. See the UML diagram below. Notice that there are two interfaces: OrderedListADT builds on ListADT. The ListADT only knows what it means to store and remove data, not how the data is added (which requires additional knowledge how on it should be sorted, or not). The result of this is that one can implement the basic functionality, and then build on top of it to get the desired list ADT: ordered, unordered, indexed (like arrays). In this homework, you'll only be responsible for the OrderedListADT, but, if your code base is properly structured, implementing an UnorderedListADT would take only a fraction of your original development time.

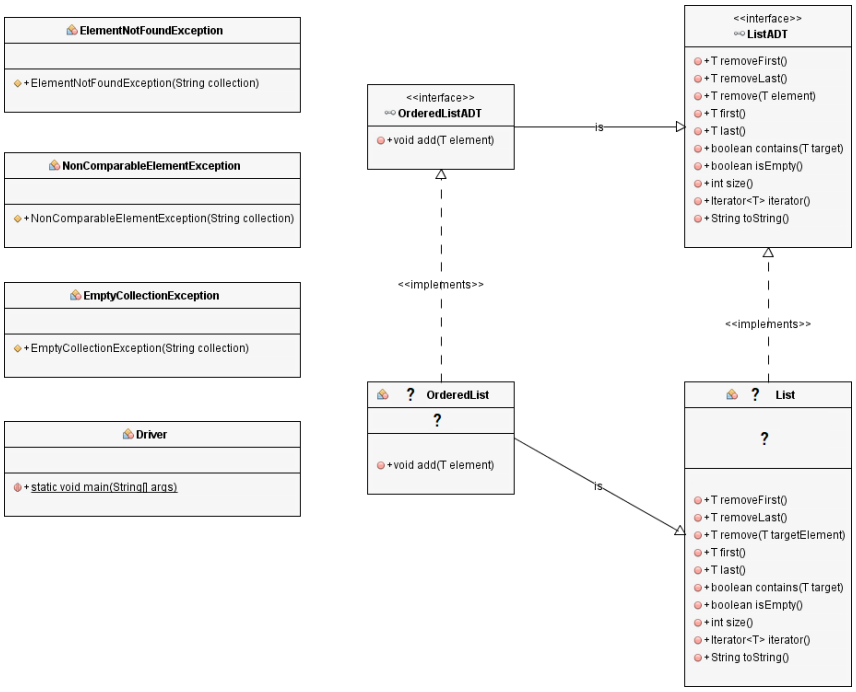

Figure 1: UML Overview

This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we give some basic tests to start to verify your implementation. Lastly, Submission discusses how your source code and writeup should be submitted on Canvas.

1.1 Layered Architecture

Complex software is usually developed with some overall architectural pattern to give the system some initial structure. These include server-client, model-view-controller, layered, and many others. Different architectures have different pros and cons. In this homework, we use a layered architecture. In a layered system, different parts of a system (usually clustered by "responsibilities") are separated into layers. For instance, a game may have three layers: a hardware interaction layer, a gameplay logic layer, and user interface layer. This useful since if a company chooses to port their game to a platform, they only need to change out one layer (e.g., the lowest layer which depends strongly on the current hardware). This helps to ensure a program is maintainable and extensible. See Figure 2 for examples. In this assignment, we'll practice layered architecture in the small by applying it to a data structure. In this case, ListADT is one "layer" and OrderedListADT is another "layer" built on top of it.

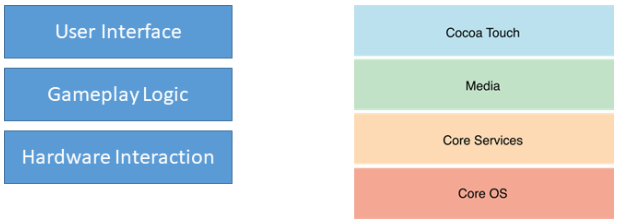

Figure 2: Two examples of layered architecture. Left: a simple game. Right: an operating-system (iOS, diagram from https://developer.apple.com).

A layered system is a way to ensure that different elements of a system are *decoupled*. The amount of

coupling between elements A and B indicates how much a change in A impacts B (meaning it may have to change as well). For example, we might say that A and B are tightly coupled which means that a change to either means the other needs to change as well. Ideally we have systems that are loosely coupled, meaning a change to A does not require a change in B. This is advantageous since upgrades/changes/bugfixes to one part of a system do not cause a cascade of additional work. Companies developing systems that are intended to be extended over time (such as operating-systems) usually aim for a decoupled design.

2 Requirements [32 points total]

The first step in this assignment is to determine if you should use an **array** or **linked** approach. Consider the following scenario: *Suppose that you are on a team that is developing part of a platform for streaming live video such as video games. The main feature is to provide a streaming portal where users can watch live and chat. On this platform, users can follow their favorite content creator to be notified when they are live. To support this functionality, the platform maintains a collection of followers for each content creator. At this point, the site is still small and many content creators have only a few hundred followers who follow or unfollow creators relatively often. You have been tasked with developing an ordered list ADT for storing the followers of each content creator. For example, the main program would have code like: OrderedList<Follower> followers = new XYZOrderedList<>(); The followers would be ordered using the time they followed a creator (so that the platform can support showing the most recent 25 followers). Fortunately, an architect on the team has proposed using a layered interface for the collection so even if there is a problem later with your implementation (or new features need to be called), the system can be easily extended. Your solution needs to efficient for the current size and behavior of the platform.*

Based on this scenario, what would be the most appropriate data structure? Either approach is potentially worth full credit - there is no absolute ideal choice. Rather, we want to see that your reasoning is valid in the write up. (If it is not, it will not impact the grade for the programming portion of your submission.)

2.1 Writeup [6 points]

In a separate document (typically just under one page single spaced), discuss your choice of data structure using the two prompts below. Your discussion **must** use your knowledge of: 1) data structure behavior, 2) the problem scenario, and 3) basics algorithm analysis. For example, you should find how data is accessed (read or written) in the scenario, then discuss how your selected data structure would support that style of access, and then strengthen your argument by describing the data structure's performance using **Big-Oh**. Do not give answers which are independent of the scenario.

- **Justification**: why did you selected the particular data structure for your implementation? Start by indicating the design you selected. [3 points]
- **Downsides:** what would be a potential future usage scenario where your choice of data structure would be problematic? Provide such a scenario. [3 points]

Note that are multiple approaches to these prompts: performance is the most straightforward way, but space usage and maintainability are also important aspects that can be considered.

2.2 Programming [26 points]

As the second step, create a class that implements the OrderedListADT interface using the data structure you selected. In the course GitHub[1], you will find most of the source code for several implementations of stacks and a queues - array and list based - feel free to read over it, and reuse whatever is appropriate. It also contains sample implementations of iterators for both array and linked structures. In particular, notice which exceptions are used. You will need to create a total of three classes: CompletedOrderedList, CompletedList, and ListIterator:

[1] https://github.com/racuna1/ser222-public/tree/master/SER222_01_03_Samples/src

- CompletedList: A class that represents a doubly linked structure, with functionality to remove nodes. Must implement ListADT<T> and Iterable<T>. [15 points]
- ListIterator: A class that provides an iterator for iterating over your array or linked implementation of CompletedList. **This should be implemented as a private inner class within the CompletedList class.** Must implements Iterator<T>. Make sure that your implementation of hasNext() throws ConcurrentModificationException when appropriate, next() throws NoSuchElementException when appropriate, and remove() throws UnsupportedOperationException. Hint: Since the attached source code doesn't have the Iterator interface, look it up by searching "java 9 Iterator interface", it should be the first result! (You don't need to implement forEachRemaining mentioned on the page.) [6 point]
- CompletedOrderedList: A class that extends the functionality from CompletedList by adding a method to add new elements. Must extend CompletedList<T> and implement OrderedListADT<T>. [5 points]

Both CompletedList and CompletedOrderedList must contain a default constructor. Attached to this assignment are five files. The first two are from the Lewis textbook but have been slightly modified.

- ListADT.java: This interface defines the list ADT.
- OrderedListADT.java: This interface defines the ordered list ADT.
- Driver.java: This contains a simple test for DoubleOrderedList, and the output associated with it. Do not submit it, it is purely for your testing.
- CompletedList: a base file.
- CompletedOrderedList: a base file.

2.3 Packages

Do not import any packages other than java.util.ConcurrentModificationException, java.util.Iterator, and java.util.NoSuchElementException. (Do not use any star imports.)

3 Testing

A few simple test operations have already been provided. These are very simple tests - they may not be appropriate for your actual testing. The sample output from these tests is:

```
Test Results:
1 3 7 9 13 14 16 17 23 24
3 9 13 16
```

When you set about writing your own tests, try to focus on testing the methods in terms of the integrity of the overall collection. If you compare the tests that you given, with the parts of your program that they actually use (the "code coverage"), you'll see that these tests only use a fraction of the conditionals that occur in your program. Consider writing tests that use the specific code paths that seem likely to hide a bug. You should also consider testing things that are not readily apparent from the interface specification.

4 Submission

The submission for this assignment has one parts: a writeup and a source code submission. The file should be attached to the homework submission link on Canvas.

Writeup: Submit a PDF that discusses your choice of data structure as decribed in Section 2.1. Include a header that contains your name, the class, and the assignment. Do not submit anything longer than one page.

Source Code: Please zip all of your source code (.java) files together as "LastNameOrderedList.zip" (e.g., "AcunaOrderedList.zip"). The classes must be in the "edu.ser222.m01_03" package, as already done in the provided base files (do not change it!).