Project 2: Sea Port Program

Justin VanWinkle

9 April 2017

Author Note

Project 2: Sea Port Program

Design

From a high level, this program has been built with a class that contains the UI separate from any workflow logic. This paves the way for several simplistic approaches when concurrency is included into the program in a future iteration. One such example would be creating the world using a thread that is separate from that of the Event Dispatch Thread that would be used by default when calling a function from an event listener that is attached to a swing component. Such a separation becomes easily manageable when business logic and view logic remain isolated.

Decisions

- I chose not to use the comparable interface on the Thing class, because I don't believe it is needed and I don't want dead code.
- TDD was used for some areas of the program to ensure that changes in future iterations do not alter the functionality of those methods.
- A map was used in the World class to track all objects for two main reasons. First, the complexity of creating the model is greatly reduced since parent objects can be found without having to search for them. Second, this allows for $O(n)$ lookup time for any object that exists in the World. So for the purposes of this class, the time needed to create the World is drastically reduced – especially for very large input files.

Meanings

The various classes, packages, variables, and methods in this program are organized in a logical manner that show good practices in object-oriented design. Each of the items has been placed in such a way that it has some sort of direct relationship with its containing parent or its contained child.

Classes. All classes assume no knowledge of their calling class whether by passing context or interface. This is likely to change in a future iteration, however.

The Thing Package. Each class in the *thing* package as well as classes in the sub packages of *thing* represent real world objects that reflect their name.

Other Classes. The PortTime class, in a similar manner represents the time of a port. Due to ambiguous requirements, however, the developer was unable to determine the intended use of this class. Therefore, it is assumed that this class will be better integrated in a future iteration.

Variables. In each class, there are sets of instance variables that represent a *has-a* relationship between the class and the item that variable holds. For instance, a SeaPort has a (or many) Dock(s).

Methods. The SeaPortProgram class contains the heart of creating the world including all of the logic for parsing out the input file, creating the objects, and modeling a world that represents what is defined in the input file. While the requirement documentation *suggests* that a scanner should be used, a string tokenizer was used instead. While this adds some additional coding overhead,

<div align="center">

User's Guide

</div>

This serves as a guide for all you need to know to get up and running with this program

How to start the program

To compile, cd into the src directory and run the command `javac *.java */*.java */*/*.java`. Once that command has completed, run `java -classpath <path-to-src-folder> main.SeaPortProgram`.

How to create a world

1. Click "Select File".
2. Navigate to and select a valid input file.

3. Select "Open".

4. World is automatically modeled. Notice the output that matches the data in the chosen input file.

Search

Searches using regex provide the user with more control over the matching pattern. Matches look at three criteria: name, index, and skill.

How to search.

1. Input a valid java regex engine pattern in the search box. (Valid regex patterns for the java regex engine can be obtained here.

2. Click "Search". Searching returns a list of all *Things* that have a Name or Index that match the criteria. Some additional cases exist such as searching the skills of a Person or the duration of a Job.

Examples.

- Search Pattern: "ar"

    – Matches "Sara"
    – Does not Match "Archie"

- Search Pattern: "[A|a]r"

    – Matches "Sara" and "Archie" For more extended documentation on regex patterns, go here.

Sorting

Sorting can be done on all lists and sub lists in the program. For convenience, the sort dialog will remain open until the user closes it.

How to sort.

1. Once a file has loaded, the `sort` button can be clicked which will open a sort picking dialog.

2. On the sort picking dialog, select the desired sort options and click `sort`.

3. Repeat until the desired sorts are achieved.

Special Features

Nothing notable.

<div align="center">

Test Plan

</div>

1. Base Case

   - **Input:** (1.txt) Using the input file outlined in the requirements documentation
   - **Expected:** Ouput that matches the output outlined in the documentation.
   - **Result:** An exact match between the expected output and the actual output.

2. Very large input file

   - **Input:** (2.txt) Using an input file that is extremely large to ensure no allocation issues
   - **Expected:** No errors and output that displays an appropriate data structure to match the input.
   - **Result:** No errors thrown and output matches an appropriate construction of input.

3. Searching "ar"

   - **Input:** (1.txt) Containing names with *ar* and Ar*.
   - **Expected:** Matches with names only containing *ar*.
   - **Result:** Search returns only names that contain *ar*. (See figure 1)

4. Sorting All Ascending by name

- **Input:** (2.txt) An input file containing multiple ports.

- **Expected:** All items in the tree will be sorted in ascending order by name.

- **Result:** All items are sorted in ascending order in the tree. (See figure 2)

5. Unit tests

- Unit tests were written to ensure that units of code perform the same basic function even after changes are made.

- **Expected:** All unit tests should pass.

- **Result:** All unit tests pass.

Lessons Learned

In the coming sections, consideration will be given to particular pain points, gleanings, and takeaways.

About This Documentation

Since discovering Pandoc, I have written all of my professional documentation and papers in markdown and used Pandoc to convert them to whatever end format I require. Markdown allows me to focus on content rather than formatting and thus saves great amounts of time. One particular hurdle in writing this documentation was finding a way to use LaTeX to handle the formatting in a way that was consistent with the APA style guide. As a result, I came across a set of tools that allows me to continue using Pandoc to pipe everything through a template that LaTeX uses to ultimately format my paper perfectly according to the APA style – and all I ever did was write this in markdown with some yaml front matter. A glorious combination when paired with Vim.

New Classes Used

JScrollPane. While I purposefully avoid Java's built in UI libraries, it came as no surprise when I found myself using the JScrollPane class for the first time in this program. I did find JScrollPane to be comparatively easy to work with for the sake of just getting a scroll view set up.

JTree. I found JTree to be easy to work with and rather intuitive. It did force me to refactor all of my `toString()` methods – but for the better.

JSplitPane. Once again, easy and intuitive. I don't quite like JSplitPane, however, because I feel that it paves the way for a bad UX. That's purely opinion at this point, though.

Future Iterations

Project 1: In a future iteration, I would like to focus on pushing all UI driven events to the EDT and all other tasks onto other threads. In the words of Vern Gosdin, "This ain't my first rodeo." Additionally, I would likely opt for some sort of utility class if too much begins to be repeated among the various classes. I already see where this might be possible as the search expands and I could offload the matching to a static utility method that just grabs the specified fields from each of the classes for comparison. Thing implements the comparable interface, but makes no actual use of the interface. Currently, the compareTo() method is overridden in Thing, but in the future, Thing need to be an abstract class and all subclasses will need to override this comparable method. Lastly, there is plenty of room for additional unit tests. New unit tests should be written on any logic code before any alterations are made for concurrency to ensure proper results.

Project 2: The main UI class is getting cumbersome. I would like to see it be split out into a second class as appropriate. The two UI classes have some repetitive code that could go into a static method of a utility class. The search could allow for more robust options such as searching only on Persons.

Figures
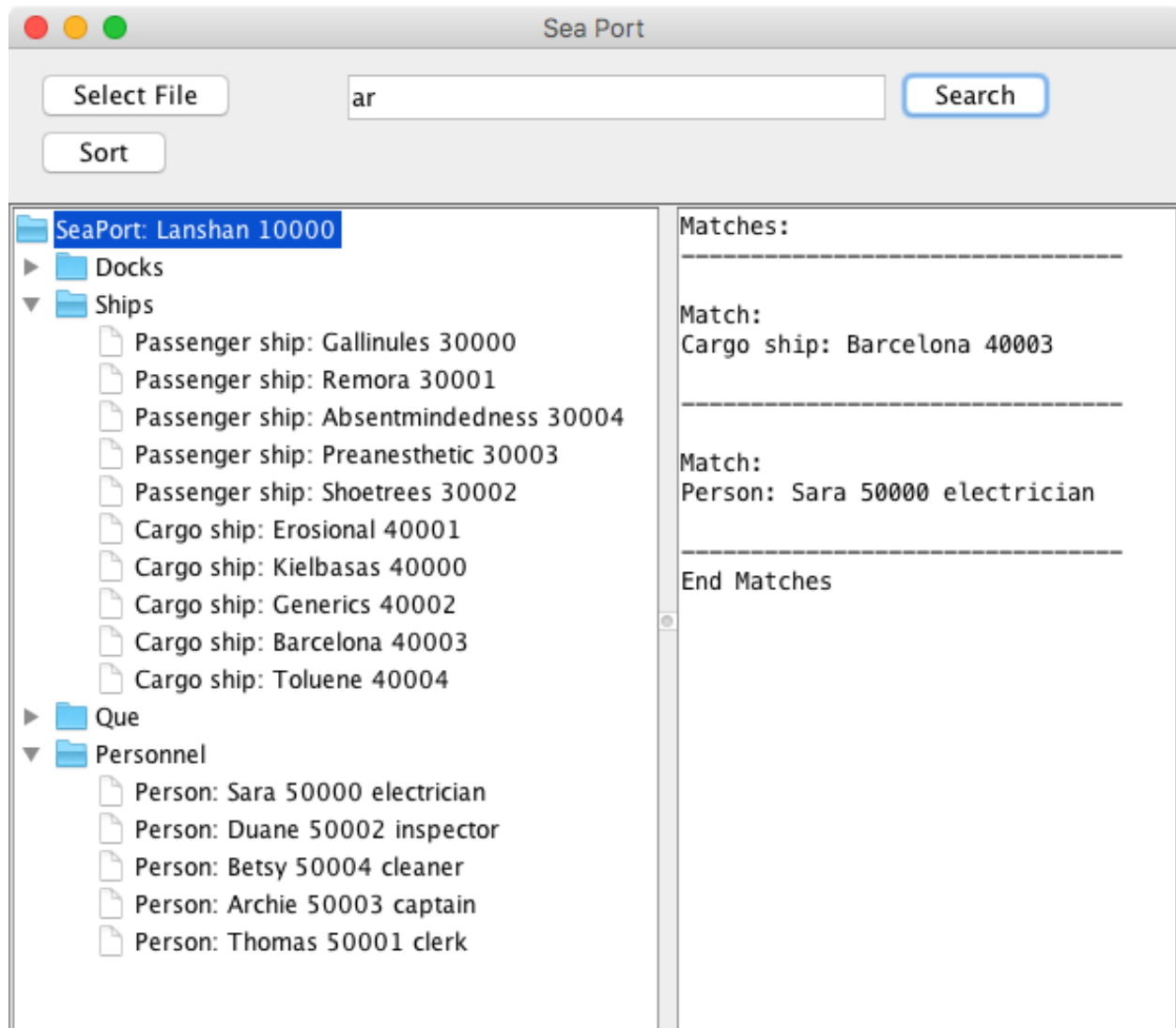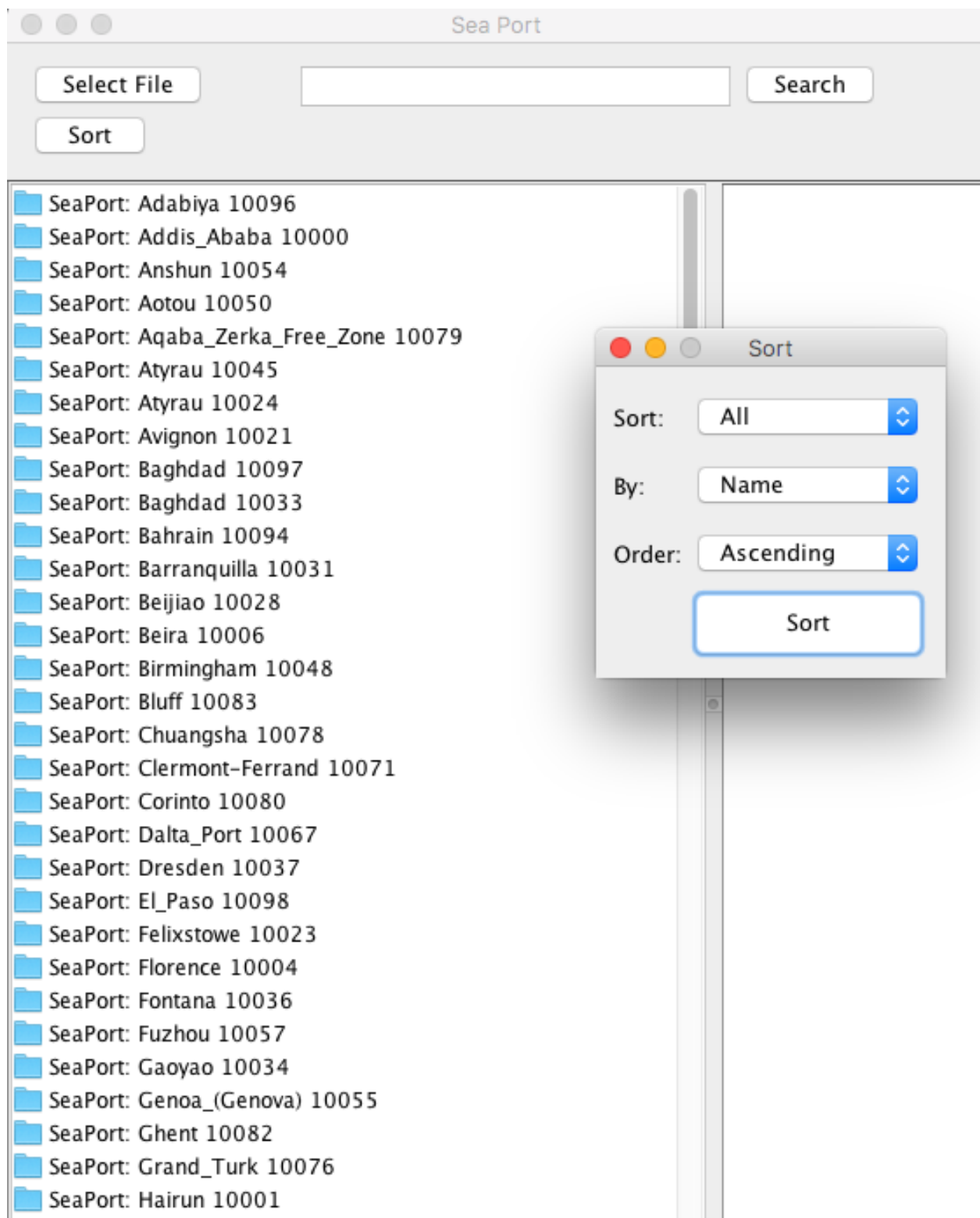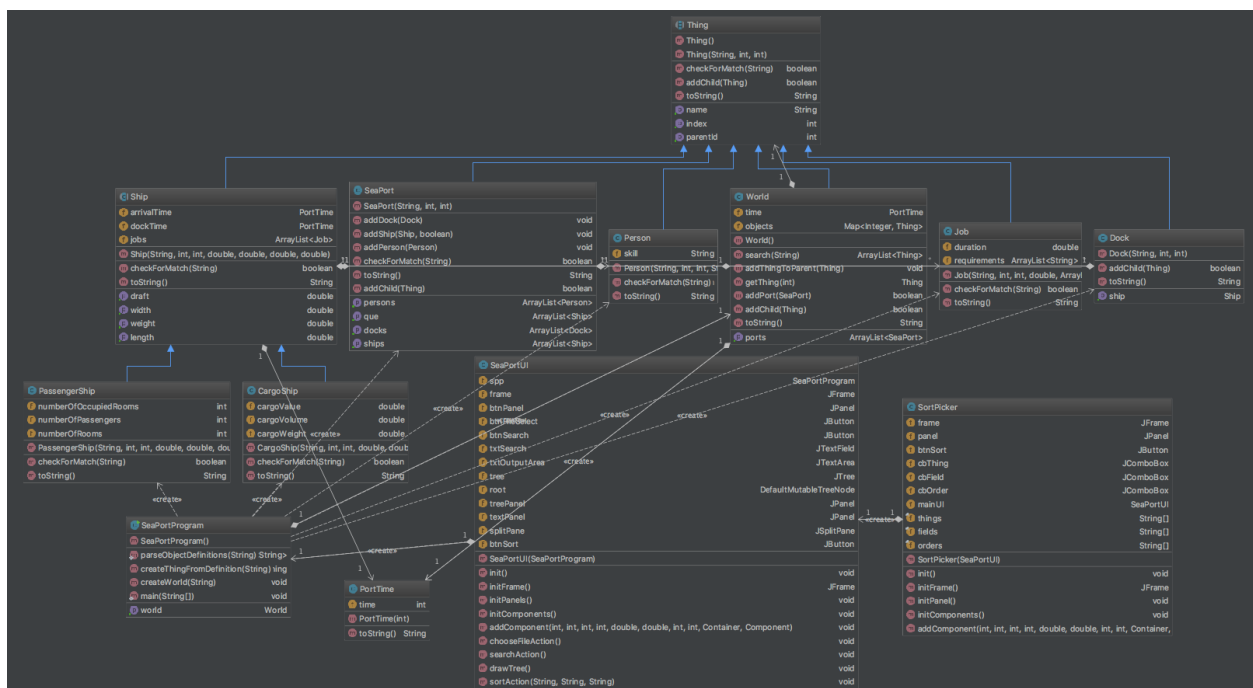
Figure 1. Search Test

Figure 2. Sort Test

Figure 3. UML Diagram