Project 1: Sea Port Program

Justin VanWinkle[1,2]

[1] University of Maryland University College

[2] Wycliffe Associates, Inc.

Author Note

Correspondence concerning this article should be addressed to Justin VanWinkle, 281 4th St, Geneva, FL 32732. E-mail: justin@swissarmydev.com

Abstract

This paper is dedicated to a project developed in a course on object-oriented programming and concurrency. It explores the decisions made while designing the program, the way a user should go about making use of the program, standard practices for testing the program, lessons learned while

Keywords: java,object-oriented

Project 1: 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

- It was determined that all classes that are representative of objects in the input file would not instantiate a scanner so as to prevent repetitive code across classes and also to prevent each class from needing to know how its definition is formed in the input file. This loosens the coupling in such a way that eases changes as a result of changes in the structure of the input file.
- 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,

User's Guide

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.

Special Features

Nothing notable.

Test Plan

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. 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

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.

Future Iterations

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.

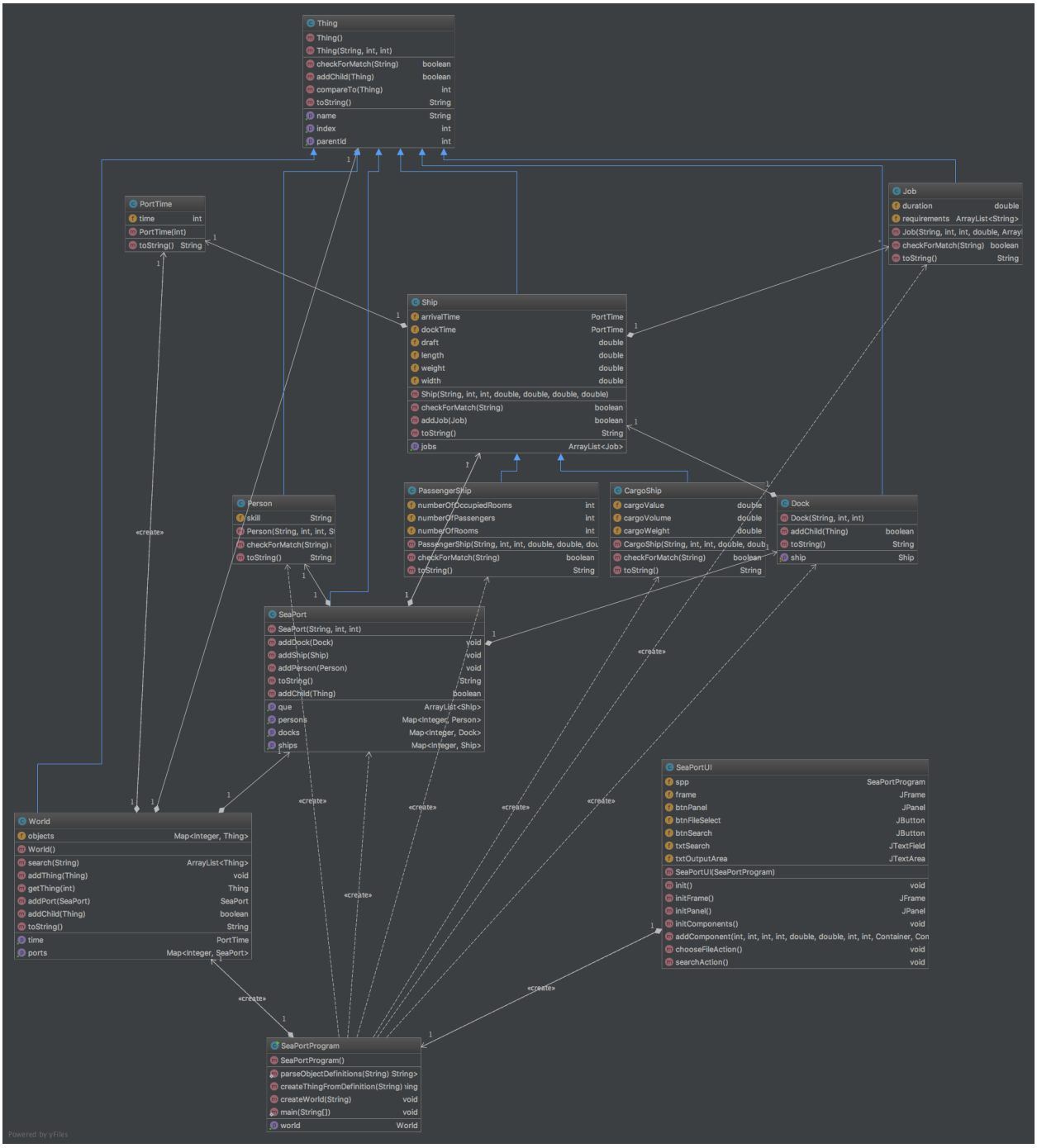

Figure 1. UML Diagram