Project 4: Sea Port Program

Justin VanWinkle

7 May 2017

Author Note

Project 4: 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. Being designed with a standard object-oriented approach has separated concerns of the role and responsibility of each class – as OOD is intended to do.

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.

- Each class that uses a thread is left as responsible for its own thread. This wasn't as much of a decision as it was a convention.

- One of the goals stated that the GUI should show the current resources that have been acquired, resource requests that are outstanding, and the availability of specific skills. My approach for this was a sort of minimum viable product approach so as to avoid going too far outside the scope of this course. It didn't seem to me that show these items had much to do with OOP or concurrency – and I felt that I had adequately demonstrated my capabilities with the Swing library in such a way that showing these items in a different way wouldn't bring any additional value at this time. My specific approach to these items was as follows:

- Available persons/skills: Next to each person, I added a substring that states whether each person is available or unavailable. This can be used to see which specific skills are available as well as how many of each.
- Resources acquired: If a job is progressing, then the skills required for that ship are acquired by that job until the job is either cancelled or completed. The specific skills that are acquired can be seen on the string that represents the job in the JTree.
- Outstanding resource requests: If a job is not progressing, then the skills required for that job all stand as unfulfilled requests. The reason that all resources stand as unfulfilled is because the ship does not take and hold those resources until all are available. I considered showing only the skills that aren't available, but felt that this would be an inaccurate representation of unfulfilled skills.
- When a job requires no skills, I determined that this was an automated job on the ship that required no personnel to complete. This could be something like a self-diagnostic test that the ship needs to perform while in port or just time that it takes to receive its full payload of fuel for the next trip.
-

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.

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.

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.

How to cancel or suspend a job. *Note*: Cancelled jobs are handled similar to a completed job, but have no additional effort applied. This allows the ship to undock. 1. While jobs are running, click "Suspend" or "Cancel". 2. Enter the index of the job you wish to interact with. 3. Click "Ok" and note that the job has stopped progressing.

Special Features

The JTree automatically refreshes its Ship and Job nodes to show progress in a near-realtime manner. If a new file is loaded, the JTree abandoned and reloaded with the new incoming data model.

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. And All ships will be immediately undocked.
   - **Result:** An exact match between the expected output and the actual output. And all ships are immediately undocked.

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:** An out of memory error is thrown and no further actions are completed by the program. This is caused by the OS limitation on the number of threads it will allow to be thrown on the allocated heap.

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:** (4.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.

6. Small Data Set

   - **Input** (4.txt) An input file containing a small set of data
   - **Expected:** All jobs will progress to completion and docked ships will be replaced with ships in que.
   - **Result:** All ships have all jobs completed while docked.

7. Large Data Set

- **Input** (3.txt) An input file containing a relatively large set of data
- **Expected:** All jobs will progress to completion and docked ships will be replaced with ships in que.
- **Result:** All ships have all jobs completed while docked.

Lessons Learned

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

Concurrency

I haven't worked with concurrency in Java for several years. Having said that, I have been using GoLang at work for the past year and have come to love the implementations of the libs surrounding concurrency – so much so that it was painful to not have go routines and channels. Java has, however, become slightly nicer in this area with the introduction of lambda functions. I haven't looked into it, but it almost makes me think that functions can be handled as first-class citizens in Java now. I'd be interested to know!

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. JTree has proven to be quite painful when working with concurrency. Since it is modeling a wide array of data, it is easy to miss a section that should be synchronized.

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

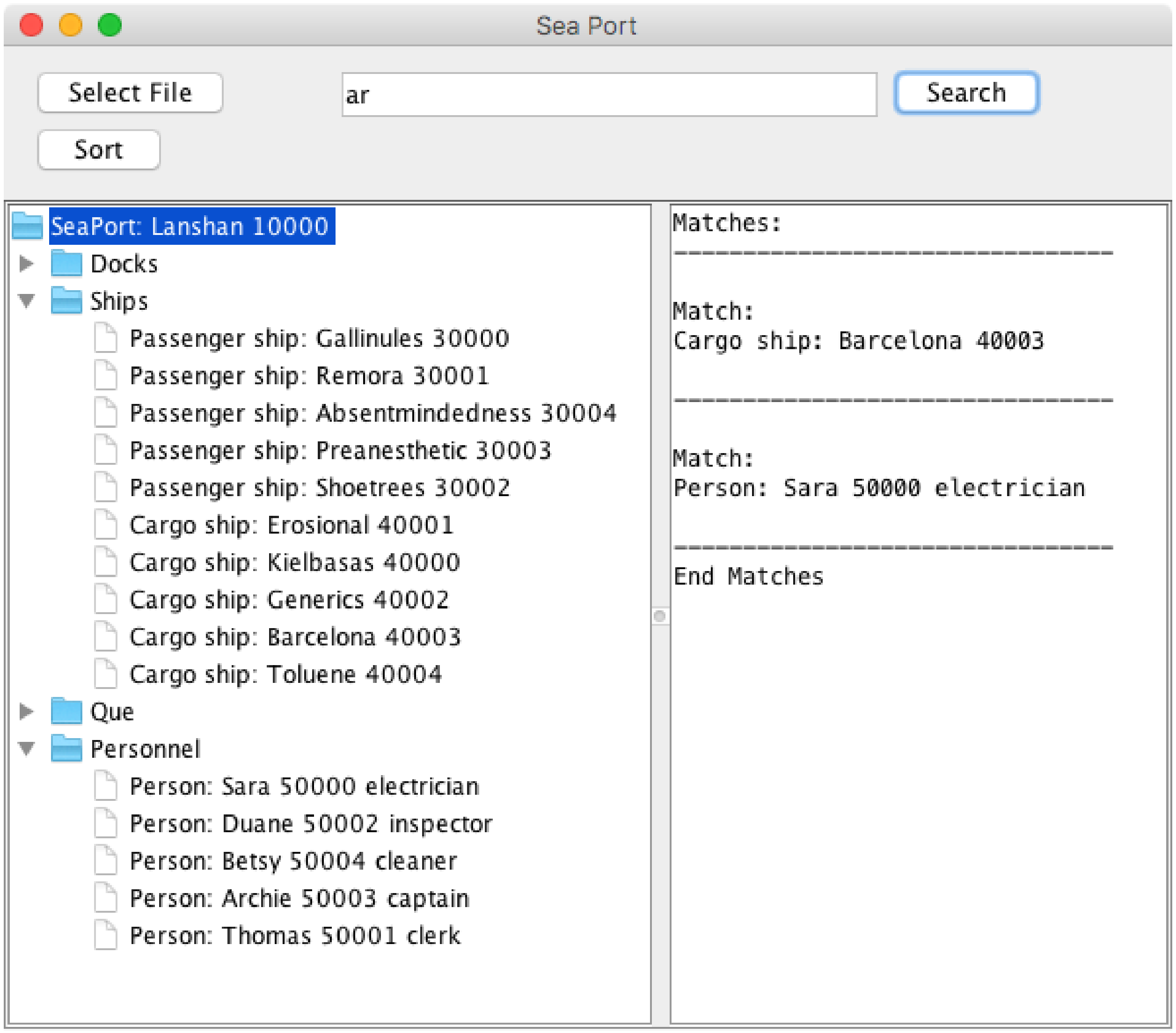
Project 1: In a future iteration, I would like to focus on pushing all UI driven events to the EDT and all other tasks onto other threads. In the words of Vern Gosdin, "This ain't my first rodeo." Additionally, I would likely opt for some sort of utility class if too much begins to be repeated among 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.

Project 3: Similar to the previous project, I feel that the primary UI file is getting very cumbersome. While it is fairly well split into functional areas, it is calling desperately for a refactor to place actions and utils into a util class. Some additional work should also be done to clean up some small bugs with the JTree not rendering the nodes properly. I think this could be fixed by using an invokeLater() or using a graphics buffer.

Project 4: If I were to continue with an additional iteration outside of this course, I would refactor the way that the classes from the SeaPort down interact. I don't think it makes much sense for the seaport to assign personnel to jobs – instead, I would probably consider something more like a model where the Job can reach out to the resource pool and request the resources it needs. In addition, I would consider extending the functionality of the user interface to provide more options for interaction (assuming this would be a real-world application).

Figures

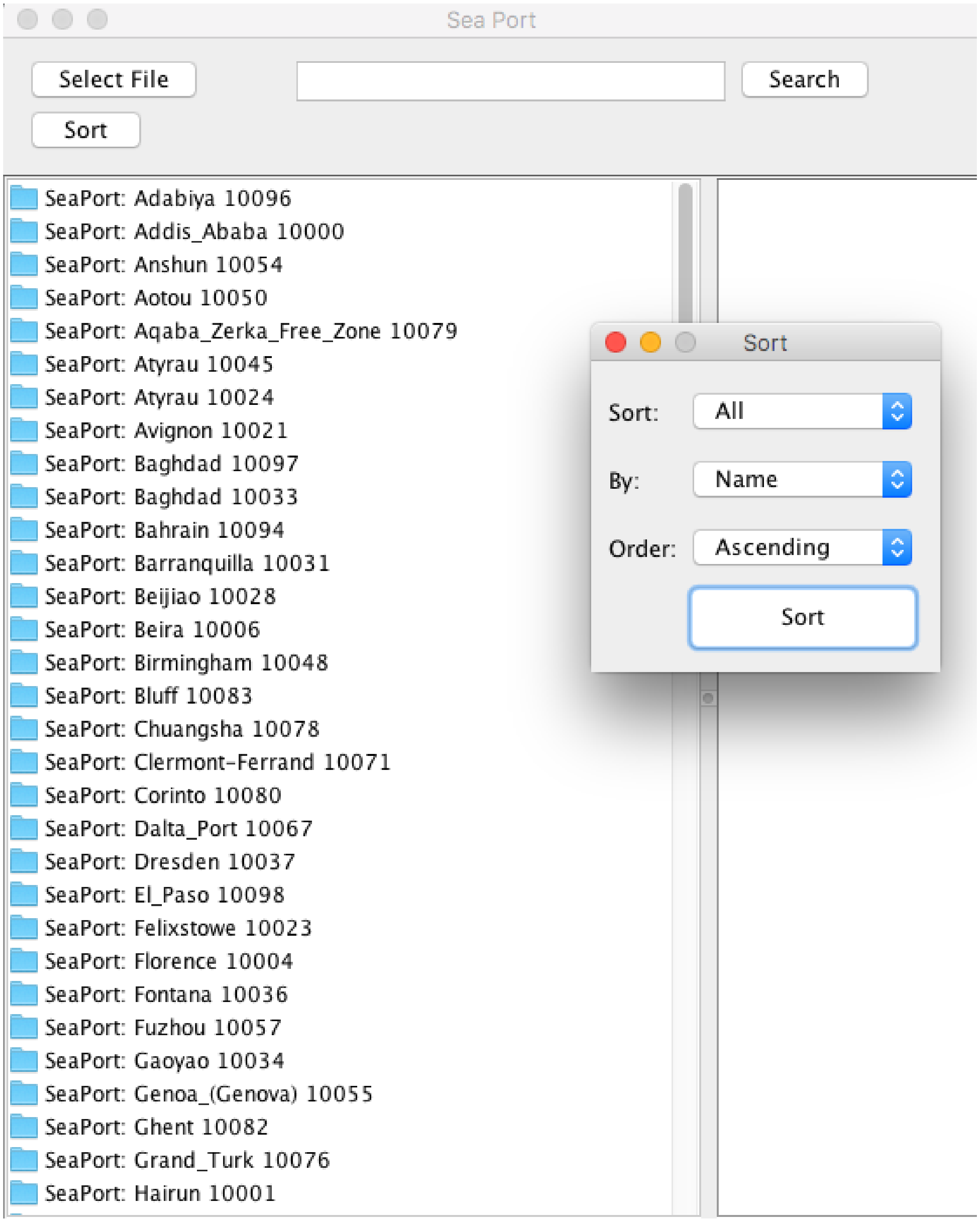


```
ar
```

```
SeaPort: Lanshan 10000
  Docks
  Ships
      Passenger ship: Gallinules 30000
      Passenger ship: Remora 30001
      Passenger ship: Absentmindedness 30004
      Passenger ship: Preanesthetic 30003
      Passenger ship: Shoetrees 30002
      Cargo ship: Erosional 40001
      Cargo ship: Kielbasas 40000
      Cargo ship: Generics 40002
      Cargo ship: Barcelona 40003
      Cargo ship: Toluene 40004
  Que
  Personnel
      Person: Sara 50000 electrician
      Person: Duane 50002 inspector
      Person: Betsy 50004 cleaner
      Person: Archie 50003 captain
      Person: Thomas 50001 clerk
```

```
Matches:
--------------------------------

Match:
Cargo ship: Barcelona 40003

--------------------------------

Match:
Person: Sara 50000 electrician

--------------------------------
End Matches
```

Figure 1. Search Test

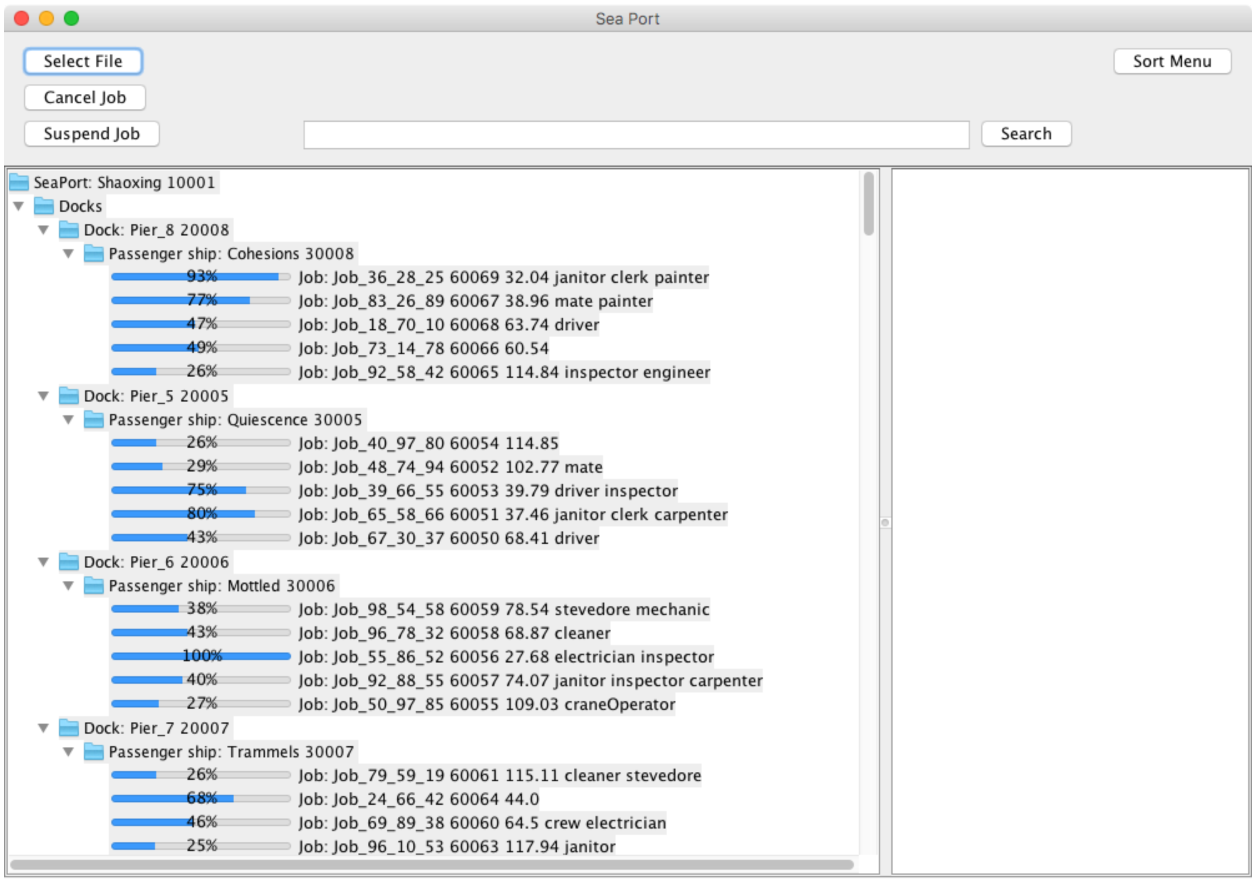


SeaPort: Adabiya 10096
SeaPort: Addis_Ababa 10000
SeaPort: Anshun 10054
SeaPort: Aotou 10050
SeaPort: Aqaba_Zerka_Free_Zone 10079
SeaPort: Atyrau 10045
SeaPort: Atyrau 10024
SeaPort: Avignon 10021
SeaPort: Baghdad 10097
SeaPort: Baghdad 10033
SeaPort: Bahrain 10094
SeaPort: Barranquilla 10031
SeaPort: Beijiao 10028
SeaPort: Beira 10006
SeaPort: Birmingham 10048
SeaPort: Bluff 10083
SeaPort: Chuangsha 10078
SeaPort: Clermont-Ferrand 10071
SeaPort: Corinto 10080
SeaPort: Dalta_Port 10067
SeaPort: Dresden 10037
SeaPort: El_Paso 10098
SeaPort: Felixstowe 10023
SeaPort: Florence 10004
SeaPort: Fontana 10036
SeaPort: Fuzhou 10057
SeaPort: Gaoyao 10034
SeaPort: Genoa_(Genova) 10055
SeaPort: Ghent 10082
SeaPort: Grand_Turk 10076
SeaPort: Hairun 10001

Sort

Sort: All
By: Name
Order: Ascending

Figure 2. Sort Test



```
SeaPort: Shaoxing 10001
▼ Docks
  ▼ Dock: Pier_8 20008
    ▼ Passenger ship: Cohesions 30008
        93%   Job: Job_36_28_25 60069 32.04 janitor clerk painter
        77%   Job: Job_83_26_89 60067 38.96 mate painter
        47%   Job: Job_18_70_10 60068 63.74 driver
        49%   Job: Job_73_14_78 60066 60.54
        26%   Job: Job_92_58_42 60065 114.84 inspector engineer
  ▼ Dock: Pier_5 20005
    ▼ Passenger ship: Quiescence 30005
        26%   Job: Job_40_97_80 60054 114.85
        29%   Job: Job_48_74_94 60052 102.77 mate
        75%   Job: Job_39_66_55 60053 39.79 driver inspector
        80%   Job: Job_65_58_66 60051 37.46 janitor clerk carpenter
        43%   Job: Job_67_30_37 60050 68.41 driver
  ▼ Dock: Pier_6 20006
    ▼ Passenger ship: Mottled 30006
        38%   Job: Job_98_54_58 60059 78.54 stevedore mechanic
        43%   Job: Job_96_78_32 60058 68.87 cleaner
        100%  Job: Job_55_86_52 60056 27.68 electrician inspector
        40%   Job: Job_92_88_55 60057 74.07 janitor inspector carpenter
        27%   Job: Job_50_97_85 60055 109.03 craneOperator
  ▼ Dock: Pier_7 20007
    ▼ Passenger ship: Trammels 30007
        26%   Job: Job_79_59_19 60061 115.11 cleaner stevedore
        68%   Job: Job_24_66_42 60064 44.0
        46%   Job: Job_69_89_38 60060 64.5 crew electrician
        25%   Job: Job_96_10_53 60063 117.94 janitor
```

Figure 3. Running Jobs

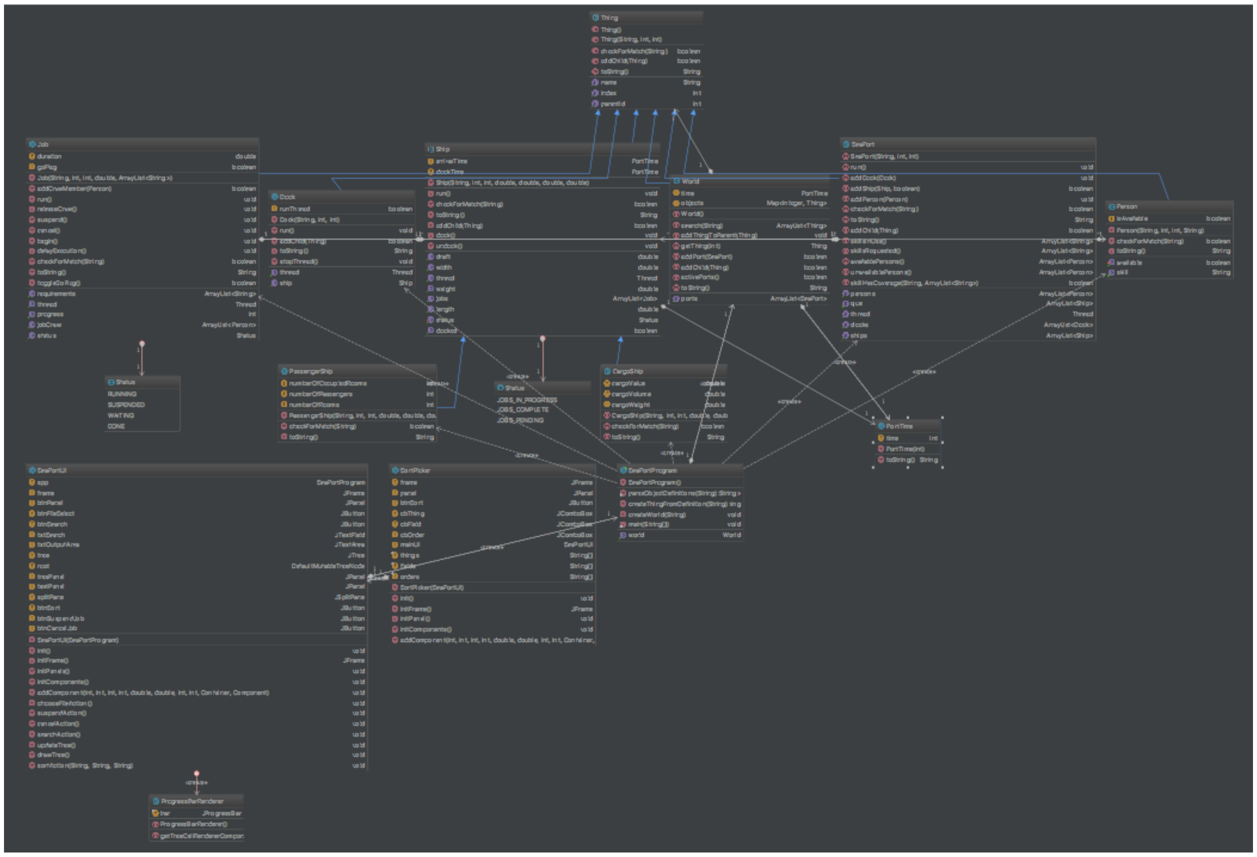


Figure 4. UML Diagram