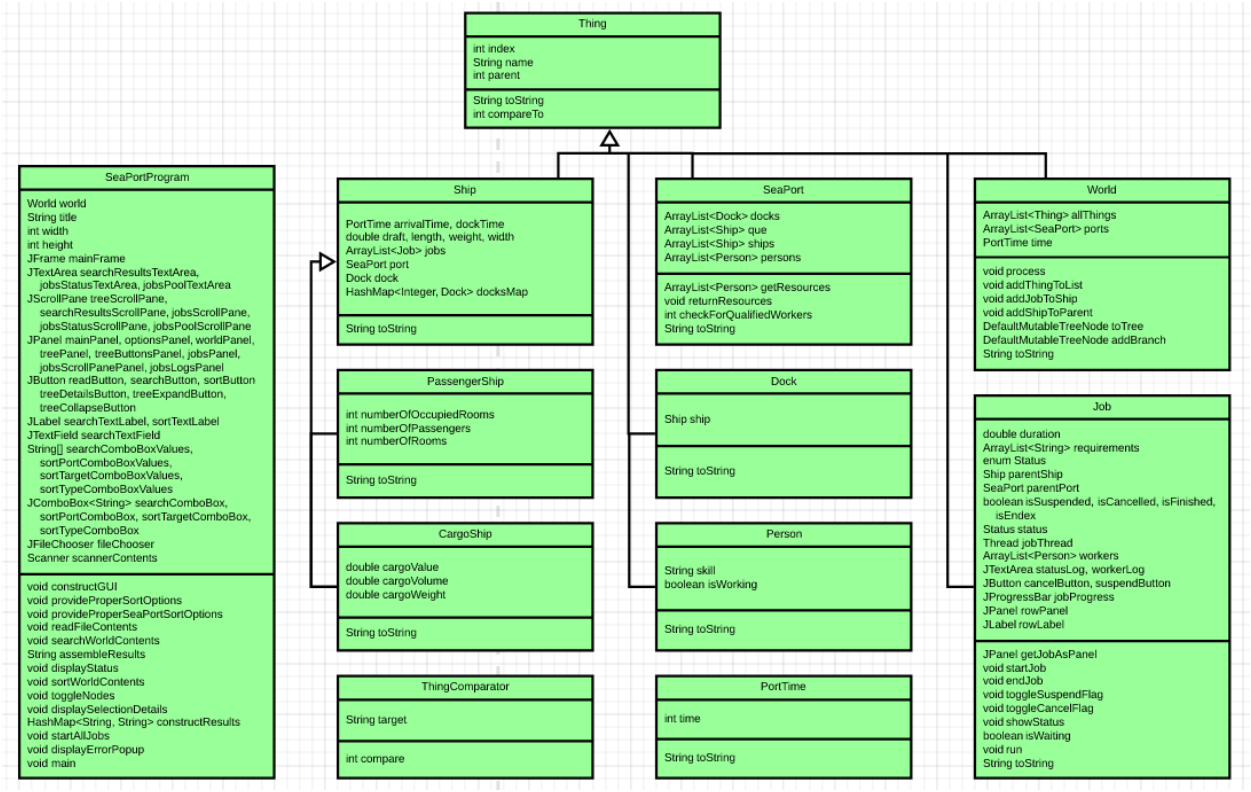
Project 3 Documentation

02/12/18

Andrew Eissen

UMUC CMSC 335

Part I: Design



(Author note: Getters/setters not included in UML diagram due to a lack of space and an inability to make the entire diagram fit in one frame to screen capture. Relatedly, all images, diagram and screenshots included, have been included in a zip file for convenient HD viewing.)

As per the Project 3 design rubric, the author assembled all classes as they appeared in the first project, augmented via the required expansion of the extant `Job` class for use running jobs from ships. As per the first and second projects, the only unused class was `PortTime` which was mentioned nowhere in the rubric and was simply included due its mention on the first page. The author is beginning to wonder if the class is used for anything, as the Project 4 overview concerns the allocation of `SeaPort` dock workers to specific jobs, and has nothing to do with time at all, apart from the job's local duration value.

Part II: User's Guide

Users unfamiliar with the process of running Java programs may have some difficulty running Project 3 without some prior experience and a set of qualifications. Users must have Java installed on their computer and must have administrator privileges on their chosen machine. Pointing the Windows command prompt to the folder containing the files, users must first compile the classes by typing `javac SeaPortProgram.java`. Once compiled, users may enter `java SeaPortProgram` to run the program and begin interacting with the GUI. Alternatively, if users possess an IDE like NetBeans, they can open `SeaPortProgram.java` and run the program from within after creating a new package.

To prevent invalid input, the program will not open non-text files or ill-formatted text files, so users must make an effort to open a file that conforms to the organizational layout of the sample data files. It should be noted that the use of search bar necessitates the inclusion of proper input. Case and spelling are both integral to retrieving the desired results, as the program will not find results that are improperly formatted as they appear in the text file.

Part III: Test Plan

From the outset, the author decided a revamp of the user interface was required to ensure that all necessary information was displayed in the most logical and visually pleasing manner possible. The three equally-sized panel-based design of the previous project, while logical in that case, failed to deliver with the inclusion of the jobs rows, as per the Project 3 rubric. As such, the `World JTree` and search/sort log were stacked and both sidelined to a `BorderLayout.WEST` position to make space for all the panels associated with `Job`

progression. For maximum readability, the jobs progression pseudo-table was placed in the `BorderLayout.CENTER` position and allowed to span the greater part of the interface to ensure that the job and ship names, progress bar, and associated buttons were all accessible and readable from the start. The associated status log and the area for the future dock worker `JTable` were placed below in equally sized panels.

Relatedly, the author has seen fit to remove the `toString()` raw text output text area due to a lack of space and a subsequent expansion of the `JTree`. Now, upon the selection of a legitimate `World` node contained in the tree and a press of the new “More info” button, a `JOptionPane` will appear displaying the class-specific attributes of that instance in a `JTable`. As a result, the removal of the raw text area ultimately results in no loss of data, as all information can still be accessed, but in a new and more user-friendly/readable format.

In reference to the Project 2 documentation’s future expansions section, the author has gone about implementing a pair of “expand all” and “collapse all” buttons to aid users in more easily traversing the `JTree`. Originally, in the course of implementing this idea, the author employed a `JToggleButton`-driven approach that changed the text of the button based upon whether or not the user had elected to expand or collapse the tree. However, the author found that if the user sought to expand the tree by hand over the course of exploring its contents, then hit the button to collapse the tree, two presses of the button would be required, the first to expand the remaining nodes and the second to actually collapse the tree. As this was mildly inconveniencing, the author reverted to the use of a pair of buttons with separate functions instead.

Part III b: Test Cases

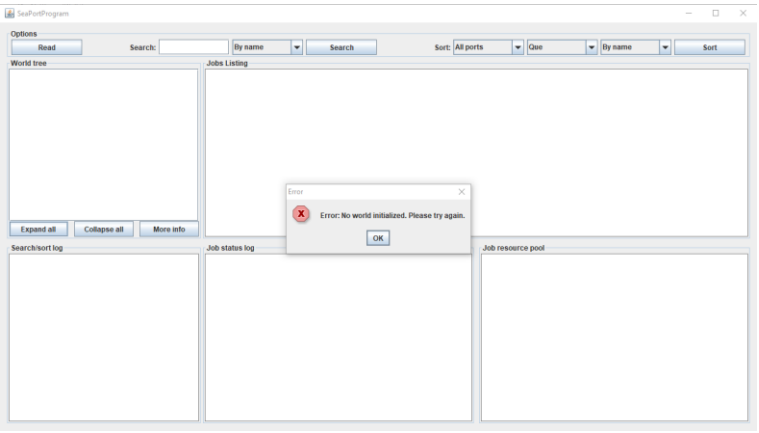
Text File	Input	Expected Output	Output
N/a	Sort button	Error: No world initialized. Please try again.	Error: No world initialized. Please try again.
`aSPaa.txt`	"Que" node "More info" button	Error: Improper selection. Please select a different node.	Error: Improper selection. Please select a different node.
`aSPaa.txt`	"Suspend" button on all running jobs	Progression is paused, button color/text is changed.	Progression is paused, button color/text is changed.
`aSPaa.txt`	"Cancel" button for Gallinules	Ship's progression is halted, button color/text is changed.	Ship's progression is halted, button color/text is changed.
`aSPaa.txt`	No input	Differs depending on which job gets SeaPort access first; see associated discussion	Differs depending on which job gets SeaPort access first; see associated discussion
`aSPae.txt`	No input	Differs depending on which job gets SeaPort access first; see associated discussion	Differs depending on which job gets SeaPort access first; see associated discussion

The test files employed in the aforementioned test cases differ slightly from those used in the previous two projects. Though `aSPaa.txt` makes a reappearance, its contents have since been augmented by a handful of jobs attached to both moored and queued ships that allow the program to display its waiting process for ships that have not yet been moored at a `Dock`. Furthermore, the new file, `aSPae.txt`, is an author-handwritten variation of `aSPaa.txt` that adds an additional number of ports, ships, and jobs, some borrowed and modified from the former `aSPad.txt` file. The reason why `aSPad.txt` itself was not included as a test file was related to its sheer size. It was inordinately difficult to keep track of what was happening where with nearly a dozen ports and presumably hundreds of ships and jobs floating about. From the author's perspective, it was easier to create a new, smaller, more manageable file and chart out the progression of `Job`s by hand prior to program initialization than it would be to sit down and work through a file containing a multiplicity of objects.

As was the case with the first and second project documentation files, the author will not include tests of previously tested functionality, such as the search/sort functions, as no changes have been made to the underlying methods. Only tests of the `Job` progression-related features and functionality will be included. For details regarding the other functionality, the author recommends a look back at the specific cases included in the Projects 1 and 2 documentation files.

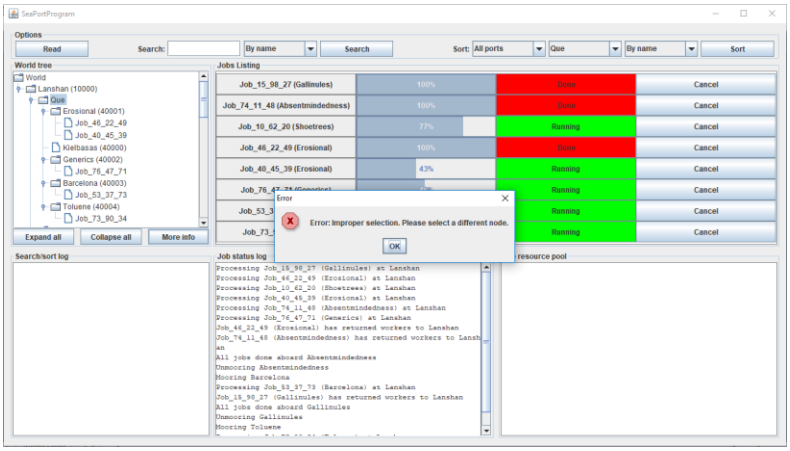
Lastly, due to the inherent complexity of displaying images of processes that occur in real time and would be better suited to `.gif` files, the author has included a bit of verbal discussion charting the specific operations and actions of the various progressions that may be encountered for each of the text files.

Test Case 1



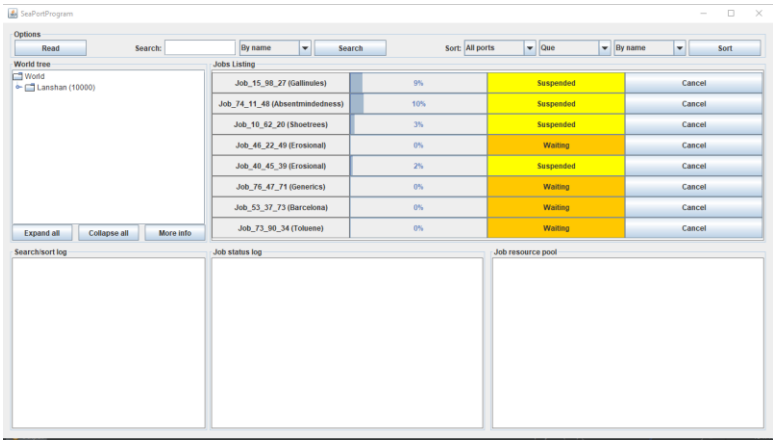
The first case tests the program's reaction to presses of the `JTree` buttons prior to any `World` initializations. As is the case with all UI buttons included herein, the program displays a `JOptionPane` indicating that the user needs to first build a world.

Test Case 2



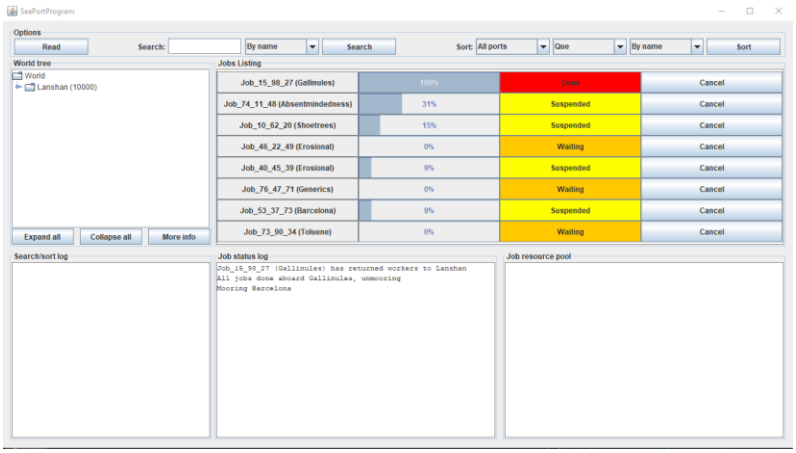
The second case tests instances wherein the user has selected a non-content navigation header node on the `JTree` and pressed the “More info” button. As these headers are simply for organizational purposes, the program displays a modal informing the user of this.

Test Case 3



The third and fourth test cases experiment with the individual job-based `JPanel`’s assorted buttons. This case tests the suspend button, which is pressed in the example on all running jobs. Doing so pauses the job progression until clicked again.

Test Case 4



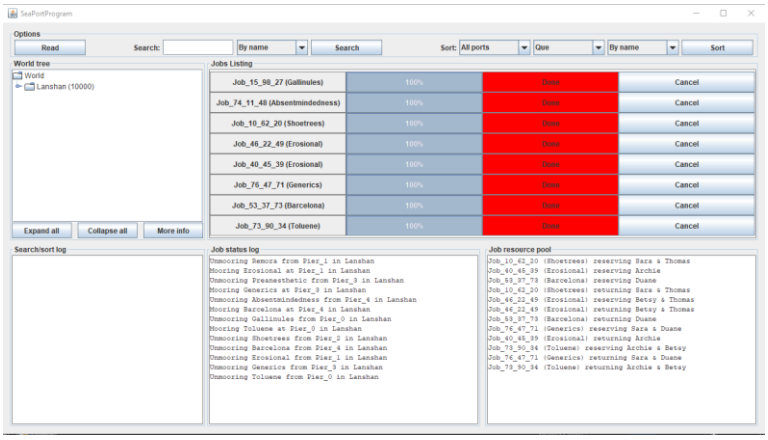
The fourth test case, like the third, tests the various job `JPanel` buttons, this time focusing on the “Cancel” button. When pressed, the associated job is marked as completed by the port in question, releasing its workers and unmooring the ship if it is the last job.

Test Case 5: Extensive discussion

The first run-through of the program discussed herein employs the file previously used in the preceding two projects and included by default in the class’s ancillary materials `.zip` file, namely `aSPaa.txt`.

This file, now augmented with the inclusion of a few new ship-bound `Jobs`, contains a single `SeaPort` and a total of four `Dock`s, a fact which keeps the job progression simple and easy to follow. As with all test runs, the first thread to gain access to the port’s dock workers may differ, resulting in different progressions but the same ultimate conclusion/result. A number of related alternative progression have been included as well to illustrate the various paths the multithreading process may take.

Test Case 5A



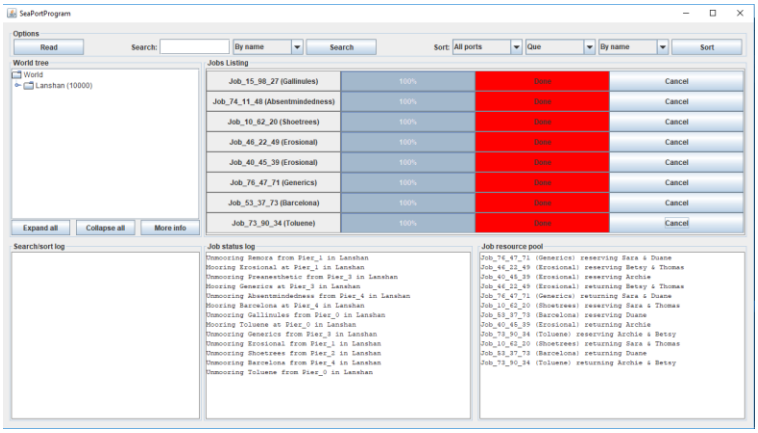
In the case of the first progression, the process begins with the unmooring of the SS *Remora* and SS *Preanesthetic*, ships moored at `Docks` at the start of the program but bereft of any jobs in need of completion. These vessels are replaced by the SS *Erosional* and SS *Generics*, ships selected from the queue due to their jobs in need of processing. The first moored vessel to gain access to Port Lanshan's worker resource pool, the SS *Shoetrees*, requires an electrician and a clerk to complete its sole job, and such requests the services of Sara and Thomas and pulls them from the available worker pool. *Erosional* requires only a captain to complete the first of its two jobs, and thus pulls Archie from the pool. In addition to these, two vessels, SS *Gallinules* and SS *Absentmindedness*, do not require any workers to complete their tasks, and run their jobs in the background. All other jobs are kept waiting until either their ships are moored or until required workers become available.

Eventually, the completion of *Gallinules* and *Absentmindedness'* jobs free up their piers for other vessels in the queue, namely SS *Toluene* and SS *Barcelona*. *Barcelona* requires an inspector to complete its sole job, and thus requests the services of Duane, who is removed

from the resource pool, leaving only cleaner Betsy in the pool. However, with the conclusion of *Shoetree's* job, it returns Sara and Thomas and unmoors itself, allowing the docked *Erosional* to begin its second job with Betsy and Thomas. Shortly after *Erosional* returns them back to the pool, *Barcelona* returns Duane and unmoors itself, having completed its task. With Betsy, Sara, Thomas, and Duane in the pool, *Generics* may stop waiting and begin its task requiring electrician Sara and inspector Duane.

Meanwhile, *Erosional* finally completes its remaining job, returns Captain Archie, and unmoors from its dock. With Archie and cleaner Betsy now available, *Toluene* may begin its job. Finally, *Generics* returns Sara and Duane and unmoors, after which *Toluene* releases Archie and Betsy and unmoors. With this, all jobs are completed and all ships removed from the port's docks.

Test Case 5B



Alternate arrangements are also possible, depending on which job thread is granted access to its port's resources first. In this included alternate example, the SS *Generics* gains access before SS *Shoetrees*, and SS *Erosional* begins both its jobs simultaneously. Both events

block all other ships from progressing until two of these three jobs are completed. However, despite the switch up, the program operates as intended, with *Barcelona*, *Toluene*, and *Shoetrees* eventually gaining access to their needed workers and completing all tasks as required. Interested readers may scan the included screenshot's status/worker logs for details of the progression as it was undertaken by the program.

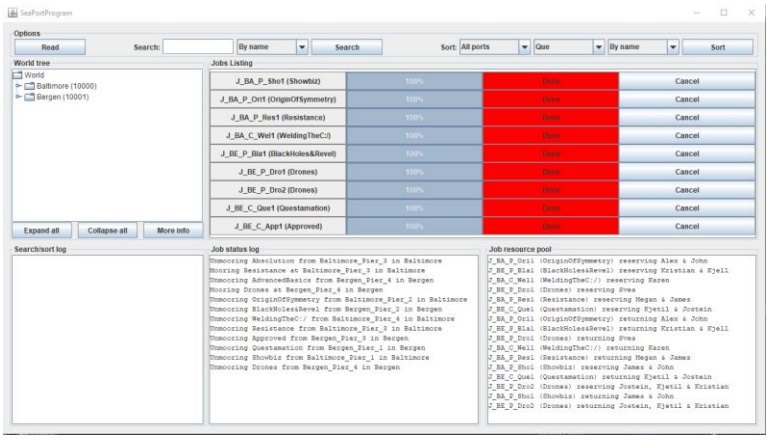
Test Case 6: Extensive discussion

The second run-through of the program discussed herein employs an author-created file that was hand-coded for simplicity and one wherein each item was named in such a way as to allow the author to easily distinguish between types of `Ship`, locations of `Docks`, and the associated `Ships` to which the included `Jobs` are attached.

Unlike the previous example, this file includes a pair of `SeaPorts` with four `Docks` apiece. `CargoShips` are provided the names of albums by Ubiquitous Synergy Seeker and `PassengerShips` are named after Muse albums, `Docks` are named with the name of their `SeaPort` in the title, `Persons` are provided fitting names related to their `SeaPort` geographic locations, and `Jobs` titles are uniformly formatted with the first two letters of their port, subclass type of their ships, first four letters of their ship, and position in that ship's list of jobs.

Though perhaps less detailed in terms of ship progression and mooring/unmooring actions, this file enabled the author to test the program's response to multiple ports and ensure all jobs are progression properly in tandem as expected.

Test Case 6A



As this example file includes two ports, in Baltimore, Maryland and Bergen, Norway, the respective progressions of each port's ships while be traced separately to limit possible confusion.

Starting first in Baltimore, the program begins by removing the SS *Absolution* from its pier, as this vessel has no jobs to complete and is simply hogging up dock space. The ship is replaced by the SS *Resistance*. Job progression begins first with the SS *OriginOfSymmetry*, which requires an engineer and cook for the duration of its job. As such, workers Alex and John are extracted from the job pool and assigned to this vessel. Meanwhile, the SS *WeldingTheC:/* requires service of navigator Karen, removing her from the pool. The newly moored *Resistance*, in need of an inspector and security guard to complete its sole job, requests Megan and James, leaving the Baltimore worker pool empty. In the background, SS *Approved* progresses its job without the aid of any workers due to its lack of required skills.

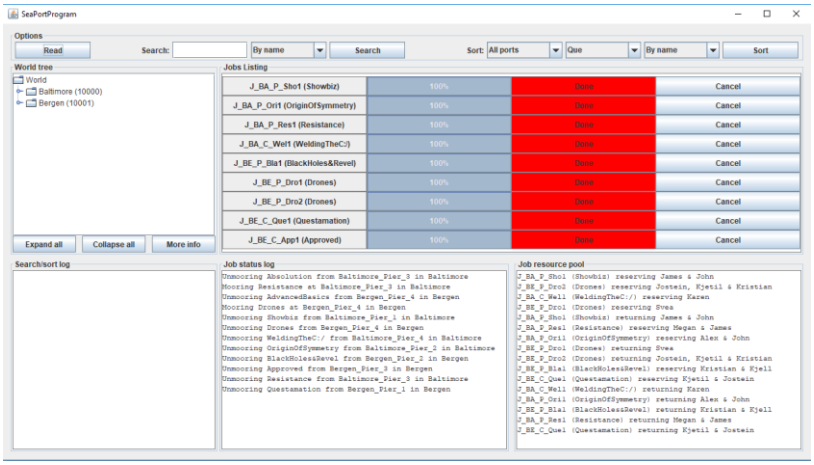
Workers are returned to the pool first by *OriginOfSymmetry*, which releases Alex and John and unmoors from its pier. As no other vessels remain in the queue, the pier is left open

(perhaps in future projects for vessels from international waters in need of certain worker skills to complete their jobs). Soon after, *WeldingTheC:/* releases Karen and unmoors, and *Resistance* returns Megan and James and itself unmoors. The SS *Showbiz*, which has been patiently docked since the program's start in preparation for receiving the services of an inspector and cook, requests James and John. Upon its returning of these workers to the pool and its unmooring from the pier, the Baltimore-based jobs are completed.

Meanwhile in Norway, job progression in Bergen begins with the initial unmooring of SS *AdvancedBasics*, a ship devoid of jobs, and its replacement by the queued SS *Drones*. Job progression begins with SS *BlackHoles&Revel*'s request for an inspector and security guard, resulting in the removal of Kristian and Kjell from the Bergen worker pool. SS *Drones*, newly arrived from the queue, requires only an engineer for its first of two jobs, and removes Svea from the job pool while its other job is kept on standby. The SS *Questamation* requires a cook and navigator, and as such removes the remaining two workers in the pool, Jostein and Kjetil, leaving the pool empty.

BlackHoles&Revel is the first to release its workers at its sole job's completion, re-adding Kristian and Kjell to the pool and unmooring itself from its Bergen pier. *Drones* releases Svea soon thereafter and waits to leave its pier until the start and completion of its remaining job. This second *Drones* job does not start until the completion of *Questamation's* job and subsequent release of Jostein and Kjetil. With the Bergen worker pool completely full, *Drones* may now finally require the services of Jostein, Kjetil, and Kristian and complete its final job. With this job's completion, all of *Drones'* jobs are complete, allowing it to unmoor and end the port's job progression.

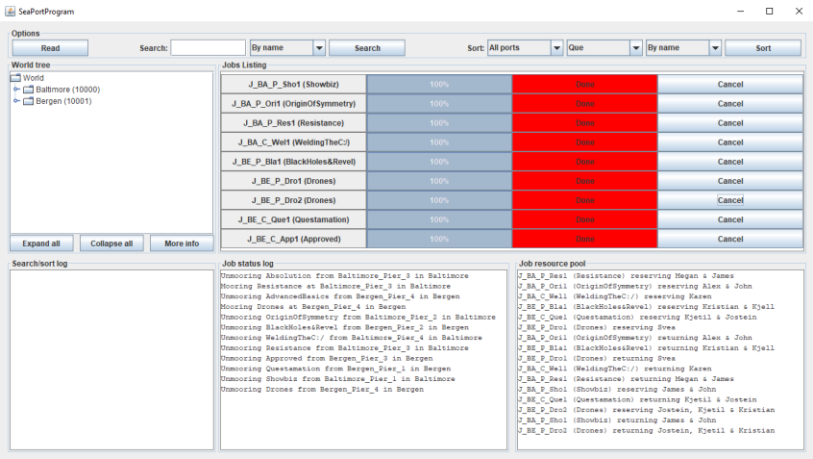
Test Case 6B



As was the case with the first file, `aSPaa.txt`, variations are possible, depending upon which of the moored ships' `Job` threads gain mutually exclusive access to their respective ports' dock worker pools. In the case of one such observed alternate, the SS *Showbiz* is not forced to wait in Baltimore for its required workers until the conclusion of the progression, but takes them from the start before *OriginOfSymmetry*. This forbids all but the SS *WeldingTheC:/* from progressing their jobs until *Showbiz* completes and unmoors. In Bergen, the SS *Drones* manages to begin both its jobs simultaneously, requiring the use of four out of the five available workers in the resource pool and forcing all other jobs to pause until both are finished to completion.

However, all jobs are eventually completed as expected, with waitlisted vessels using the workers as they become available in the worker resource pool. Interested readers are encouraged to trace the progression in the included screenshot as outlined in the worker/status logs to see this process unfold for themselves.

Test Case 6C



An additional observed alternate progression begins in a slightly different manner to those listed above, while maintaining a series of similarities with the main progression traced above. In Baltimore, much like the first process, the SS *Resistance*, SS *OriginOfSymmetry*, and SS *WeldingTheC:/* monopolize the entirety of the worker pool, though in this case, the *Resistance's* job is run first. In Bergen, the SS *Questamation*, SS *Drones*, and SS *BlackHoles&Revel* all request first, as in the main progression, though the order is switched slightly.

As expected with all alternate approaches, however, the progression completes with all jobs being fulfilled by the workers as they come available in their respective `SeaPorts`. As always, readers are encouraged to trace the individual progressions as outlined in the logs to see for themselves the manner in which the jobs are completed.

Part IV: Comments

As far as future modifications and updates are concerned, the author has a plan in mind for the contents of the dock worker resource pool panel. Though initially the author considered

a `JTextArea` approach similar to the current Project 3 implementation, he has since leaned towards a `JTable` approach wherein each `Person's` name, location, skill, and current job are displayed in a row, and updated upon being returned to the `SeaPort`'s worker `ArrayList`. This idea would in theory make use of a separate class extending `AbstractTableModel`, though the author has not attempted the implementation yet and thus cannot definitively state which approach may be best given the specific constraints of the project itself.

Regarding design weaknesses, the author is displeased with the fact that the `Job` class possesses a fair few GUI elements and requires a fair few "janky" parameters to assist in placing the rows properly in the GUI. The author is a proponent of the MVC (model-view-controller) paradigm of design and generally likes to keep his GUI components in a single class dedicated to the user experience, while keeping computation code separate and in the background. As such, he wasted a fair bit of time initially attempting to get the `Job` threads to synchronize properly from within a `for` loop housed within the `SeaPortProgram`, wherein `JPanel` rows could be constructed and added within the body of the class that defines their intended targets.

Instead, however, when the attempt did not work, the author implemented the next best thing, keeping job thread initialization within the body of the main `SeaPortProgram` class so as to allow the GUI class pass the required Swing logging components to the `Jobs` and place assembled job `JPanel`s itself. The additional benefit was the fact that assembling each `Job` and starting each job thread after the completion of file reading prevents errors in which jobs erroneously tell their ships to remove themselves from their docks before all jobs have

been read and completed. This approach was suggested on a page on the resource website linked in the Project 3 rubric, namely Ideas: “an alternate approach is to create the job threads but not start them until the data file is completely read, using a loop at the end of the read file method.” Further discussion of these changes can be found in the associated JavaDoc blocks of the `SeaPortProgram.startAllJobs` method and its associated methods in other classes.

The author discovered an issue with data text files assembled with the included `CreateSeaPortDataFile.java` program, one wherein some jobs belonging to ships anchored somewhere in the harbor of various ports could sometimes be indefinitely waiting for workers due to a lack of any workers possessing the necessary skills to complete the job. The author thinks that the only way to rectify this issue is to move the ship from its original port to one possessing the workers necessary to complete the job. This will likely be explored further in the next project.

Part V: Lessons Learned

The author learned much about synchronization due to a massive amount of external research undertaken to more easily wrap his head around the specifics of the process. Though the concept remains a bit fuzzy in practice, the author was able to ensure that the associated `Job` threads are all properly loaded through the application of a pair of `synchronized` blocks housed within the body of the `run()` method.

Furthermore, in the course of replacing the raw world contents section with a `JTable`-equipped `JOptionPane` used to display object properties on button click, the author learned

a fair bit about how to design and implement a `JTable`. Having used HTML tables and wikitables frequently in the past, the author expected `JTables` to behave in a similar manner, and was initially overcome by the sheer complexity inherent in the construction of a proper `JTable`. However, after some research, the author was able to overcome to initial design curve and formulated a decent first attempt. The knowledge gained from that attempt will likely prove useful in the next project, wherein the author plans to display the available dock workers in each `SeaPort` in a `JTable` listing each worker's name, location, skill, and current job per row.