Taylor Gorman, Michael Overby, Reid Thompson, and Arshdeep Singh

Dr. Josh Tenenberg

TCSS 360 - Section A

June 1st, 2015

Rage Against the Waterfall Model Deliverable 3 Project State Document

**User Stories**

1. As a Park Manager I want to submit a new job request.
   1. When the user is prompted by ParkManagerUI, he/she can select to Create a New Job, and is then requested to input various job details by ParkManagerUI.createNewJob(), including the Park, number of volunteers, and the job dates. The Job is then constructed and passed to ParkManager.addJob(), which makes a quick check on the attached park before passing it to Schedule.receiveJob(). Schedule then submits the job to a barrage of tests, and if it passes them all, it is added to the JobList.
   2. Jobs are heavily scrutinized before they are added to the JobList. We ensure that the Park is one that the ParkManager manages; that there is at least one viable Volunteer slot; that there are not currently 30 pending jobs; and so on, so that all Business Rules-- and several other exceptions-- are satisfied. All of these checks are verified within ParkManagerTest via extensive unit testing.
   3. The User Interface asks for each job detail as clearly as possible. If the Job is successfully added, the User is told so; otherwise, the User is told that the Job could not be added, and is given an exact reason why. For instance, if the User enters in an End Date that comes before the Start Date, they are explicitly told: “Sorry, but the Start Date must come before or on the same date as the End Date.” A similar printout is ready for all foreseeable problems.
2. As a Volunteer I want to view upcoming jobs I can sign up for.
   1. Implemented with the use of three classes - Volunteer, VolunteerUI and DataPollster. When the user is prompted, he can enter in the correct value to view the jobs that can be signed up for. When he enters the value, a method is called in Volunteer class which returns the updated list of jobs. This list of jobs is then printed out (but only the jobs which have not already passed).
   2. The UI is given a list which contains updated jobs. Only some of the jobs will be printed because not all of them are taking place in the future. The size of the list of future jobs is tested to make sure that it is equal to all the jobs minus the jobs in the past.
   3. Jobs that have not passed will be printed to the console. The user can verify that only future jobs are shown by looking at the start date of each printed job.
3. As a Volunteer I want to volunteer for a job.
   1. Implemented with the use of three model classes (Volunteer, Schedule, and JobList), one UI class (VolunteerUI), and three business classes (3, 6, and 7). When the user asks to sign up for a job, he is asked for the job ID number of the job he would like to sign up for. Once he has entered a value, he will be shown the levels for that job which have open positions. Then he will be asked to enter in the level of the job that he would like to work. Once he has done this, the values he has entered will be checked for validity using the business rules.
   2. Each of the possible input values (that the user might enter) is tested for, including border cases. Depending on what the user enters, different things will be printed out.
   3. Once the user has successfully signed up for a job, a statement will be printed to the console stating so. If there was a problem with signing up, the user will be told what the problem was.
4. As a Volunteer I want to view the jobs I am signed up for.
   1. Implemented with the use of JobList, Volunteer, DataPollster and VolunteerUI. When the user asks to view the list of jobs that he is already signed up for, Volunteer will get the list of all jobs from DataPollster. Then a new list of jobs will be created which consists only of jobs that this volunteer is signed up for. This new list will be handed over to the UI which will print out the information on each job in the list.
   2. This is tested with multiple cases. One case is to test before the volunteer has signed up for any jobs at all (the size of the list should thus be zero). Another case is after he has signed up for one or multiple jobs (the size of the list would thus be equal to the number of jobs that the user has signed up for.
   3. If the user has not signed up for any jobs but still asks for what jobs he has signed up for, then he will be told via the console that he has not signed up for any jobs yet. Once the user has signed up for any number of jobs, the information on each job will be sequentially printed out.
5. As a Park Manager I want to view upcoming jobs in the parks that I manage.
   1. When the user is prompted by ParkManagerUI, he/she can select to view all of their upcoming jobs. ParkManagerUI.displayJobs() is called, which then calls Parkmanager.getJobs(). ParkManager.getJobs() makes a call to DataPollster.getManagerJobs(), which gets the complete Job List from JobList and creates a subset of Jobs that are attached to the Parks that the Parkmanager manages. This list is then returned, and ParkManagerUI.displayJobs() formats the list and prints it out to the User.
   2. If the ParkManager does not have any Jobs, then they are told that they do not have any upcoming jobs to display. Furthermore, because Jobs are rigorously tested by Schedule before being added to the JobList, and we set up the design to where all new Jobs must go through Schedule, we have ensured that all Jobs that the ParkManager winds up viewing are valid.
   3. Once the User selects to view all of their upcoming Jobs, they are immediately printed out in a very orderly way. The details of the Jobs are clearly displayed, in a way that they both offer a lot of information while remaining easy to process in the case that the ParkManager has a lot of Jobs to sort through.
6. As a Park Manager I want to view the Volunteers for a job in the parks that I manage.
   1. When the user is prompted by ParkManagerUI, he/she can select to view all of the Volunteers for a Job that they manage. Then, ParkManagerUI.viewJobVolunteers() requests the ID of the Job in question. It passes this on to ParkManager.getJobVolunteerList(), which in turn calls DataPollster.getJobVolunteerList(). This method gets the Job from JobList, and looks through the VolunteerList for that Job. It assembles a list of Volunteer emails, which it then passes back to ParkManager. ParkManager returns this list to ParkManagerUI, which parses the list and displays it to the console.
   2. If the ParkManager is not the manager for the Job ID, then they are told so in the console. If there are no Volunteers attached to the Job, then none are printed. Because a Volunteer’s request to be added to a Job is rigorously tested by Schedule (as are newly submitted Jobs), and we have set up our software design so that Schedule is staunchly in between Users and the Job List, we ensure that the Volunteers being returned-- and the Jobs that they are signed up with-- are valid.
   3. When the User selects to view all Jobs for a volunteer, they are directly asked for the Job ID. Because Job IDs are displayed when the ParkManager displays all of their Jobs, there shouldn’t be any confusion here. Then, Volunteers signed up for the Job are displayed in a neat and organized fashion. The ParkManager can easily read the output, and in the case that they are not the manager of the provided Job ID, they are told so immediately.
7. As an Administrator, I want to search volunteers by last name.
   1. When the user is prompted by the AdministratorUI, they can choose option 2, which sends them to AdministrarorUI.searchByLastName(), which calls AdministrarorUI.promptForVolsLastName. The last name is then passed to Administrator.getMatchingVolunteers, which returns the list of users with that last name. If there are none, there is a message for that. If there are some, the AdministrarorUI.displayVolunteers() method is called to actually print all of the Volunteers and their information.
   2. There are no error checks associated with this business rule, since the only thing involved is getting a list of Volunteers. Whether the list is populated or not, that is a valid result.
   3. The volunteers are printed out in alphabetical order, along with what jobs (if any) they are signed up for.

**Business Rules**

1. A job may not be added if the total number of pending jobs is currently 30.
   1. This code is written in it’s own class which holds a test method.
   2. The test method asks for a list of pending jobs from the joblist which is passed in as a parameter. JobList gets the entire list of jobs and only passes back the number of jobs which have not yet passed. This number will be tested against the number 30 (which is the max number of jobs that can be pending).
   3. If the park manager attempts to add a job, the list of jobs will be checked. If the number of pending jobs is 30 or greater than 30, then the job will not be added. The user will be told so via the console. If the job is added successfully, a success message will be printed.
2. A job may not be added if the total number of pending jobs during that week (3 days on either side of the job days) is currently 5. In other words, during any consecutive 7 day period there can be no more than 5 jobs.
   1. This code is written in it’s own class which holds a test method and a conversion method (which converts to days from milliseconds).
   2. This business rule is tested in multiple ways. It is called when the park manager tries to add a job. Before he can add the job, the week of the job must be checked to make sure that no more than 5 jobs are held that week. We determine a week to be 3 days before the start date of the job and 3 days after the end of the job. One test case is when a manager tries to add a job during a week in which all the other jobs occur during the front half of that week. Another test case is when the second half of the week has all of the jobs for that week. A third case is when the day of the job has all of the jobs in that week.
   3. If the user is able to successfully create the job then a success message will be printed to the console (the manager can also see that he has created the job via one of the menu options). If the job could not be added then a message will be printed to the console stating so.
3. Volunteer may not sign up for a work category on a job if the maximum number of Volunteers for that work category has already been reached.
   1. This code is written as its own class with a test method.
   2. It takes in a value for which job level the user wants to sign up for, if that value is invalid then a thrown and caught. If that value is valid then it is tested against the capacity of the level, then a boolean is returned dependent on whether or not there were open positions.
   3. If the volunteer attempts to sign up for a level which is full, then a statement is printed to the console stating so (and the volunteer is prevented from signing up for the job).
4. A job may not be scheduled that lasts more than two days.
   1. This code is written as it’s own class with a test method.
   2. The test method takes in the job that the park manager wants to add. It creates a calendar that is based off the starting date of the job. It then adds two full days to this job. If the job’s end date comes after this new calendar date, then false will be returned because the job lasts for too many days. To test this, we can make multiple cases. One case is to create a job which has its start and end date on the same day. This job should be successfully created. A second test case is when the job’s end date is the day after its start date (so the job lasts 2 days); this should also give a success message because its okay. What is not okay is when the job lasts more than 2 days (so try making a job that lasts 3 days - an exception should be thrown and caught).
   3. The park manager will know that he successfully added the job to the job list if he gets a success message (printed on the console). If the user instead gets an error message, then he will know why that error was given (because it’ll state the reason in the message).
5. A job may not be added that is in the past or more than three months in the future.
   1. The code is written as its own class with 2 test methods. The first test method tests for whether the job is in the past. The second test method tests for whether the job is more than three months into the future.
   2. It takes in the job that is to be created as a parameter in both tests. A calendar will be created for the current date in real life time and compared to the time of the job; if the job comes before it, then it should throw an exception which will be caught.) ( The other test is for the future: A calendar will be created for the current date and then increased by 90 days; If the job comes in after that date, then an exception will be thrown.
   3. If the park manager attempts to create a job that is not within the valid date range, an exception will be thrown and caught (and a message stating so will be printed to the console.
6. A Volunteer may not sign up for a job that has passed.
   1. This code is written in its own class. An instance of it is created in Schedule class when a volunteer attempts to sign up.
   2. It takes in the the job that the volunteer wishes to sign up for, in its test method. A calendar is created for the current date in time, it will be compared with the job’s start date to make sure that the job is in the future. Also, the job’s myIsPast field will be checked to see if the job has passed.
   3. If the user tries to sign up for a job that is in the past, then an exception will be thrown and caught (and a message stating the reason for the exception will be printed to the console).
7. A Volunteer may not sign up for two jobs on the same day.
   1. This code is written in its own class. An instance of it is created in the Schedule class when a volunteer attempts to sign up for a job.
   2. The test method takes in three parameters. The first is the email of the user. The second is the job that the user wants to sign up for. The third is the joblist which contains all of the jobs. When the user attempts to sign up, the joblist will be searched for all jobs which contain the user. If the job that the user wants to sign up for takes place on the same date as any of the other jobs that the user is already signed up for, then the user will be prevented from signing up.
   3. If the volunteer tries to sign up for a job that takes place on a date that he is already working, then an exception will be thrown and caught - and a message will be printed to the console stating that there is a job conflict.
8. A Park Manager can create jobs only for those parks that he/she manages.
   1. The code for this is written in its own class. An instance of the class is created in schedule when a park manager attempts to create a new job.
   2. The test method in this business rule’s class takes in two parameters. The first parameter is the job that the manager wants to create. The second parameter is the list of parks that the manager manages. A boolean will be returned depending on if the list contains the park’s location.
   3. If the location of the job is not listed within the manager’s list of managed parks, then false will be returned with will cause an exception to be thrown and caught. Then a message will be printed to the console stating why the exception was thrown.