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 Changes Document

**1. Class, Collaboration, and Responsibility Changes**

1. Model and UI Dependencies for User Classes *(applies to Login and LoginUI as well)*.
   1. Start: ParkManager was dependent on ParkManagerUI, and the same was true for Administrator and Volunteer.
   2. Change: ParkManagerUI is now dependent on ParkManager, with the same change made to Administrator and Volunteer.
   3. Reason: We learned that it is an important Object-Oriented Heuristic that interface classes should always depend on the model class, and not the other way around.
2. DataPollster and Schedule changed to Singletons.
   1. Start: DataPollster and Schedule were non-static classes that would be constructed by various other classes.
   2. Change: DataPollster and Schedule are now static classes, and any class that wants to access its methods must call DataPollster.getInstance() [same for Schedule].
   3. Reason: It is important for Login and the User classes to be accessing the same DataPollster and Schedule instances. As such, our User UI classes had unnecessary dependencies on DataPollster/Schedule, as they had to be passed them by Login so that they could pass them to the User model classes. By making DataPollster and Schedule static (via the Singleton design pattern), we removed these dependencies.
3. User abstract class added.
   1. Start: Administrator, Volunteer, and ParkManager were similar, but unrelated classes.
   2. Change: The three User model classes now extend the User abstract class.
   3. Reason: This allowed for a plethora of opportunities to make use of polymorphism in our code. We were able to greatly simplify our methods-- and reduce the total number of methods-- accordingly.
4. UI abstract class added.
   1. Start: AdministratorUI, VolunteerUI, and ParkManagerUI were similar, but unrelated classes.
   2. Change: The three User UI classes now extend the UI abstract class. LoginUI does as well.
   3. Reason: This helped us to cut down on redundancy in our code, as methods shared by all three of the User UI classes-- getUserInt, getUserString, and calendarToString, can be inherited from UI instead.
5. Saving scheme changed to Serializable.
   1. Start: Our saving and loading processes relied on text files with a hard-coded representation of our JobList and UserList classes.
   2. Change: Our saving and loading processes now rely on serializable files, and as such, are able to more or less save JobList and UserList directly. We updated many of the classes related to JobList and UserList (User model classes, DataPollster, Job, UserList, JobList, and Schedule) to implement Serializable accordingly.
   3. Reason: Serializable is an overall better and cleaner way to locally handle save data. Furthermore, the requirements were changed to where it was now necessary.
6. Isolated logic from User UI classes.
   1. Start: User UI classes had some logic in them, such as sorting through lists returned by DataPollster, and making checks on data.
   2. Change: This logic now exists within the User model classes.
   3. Reason: We decided that User UI classes should have almost zero logic. This way, changes to the interface (such as from console to GUI) would be much easier to make, and to further compartmentalize our code base.
7. Changed ArrayList<ArrayList<String>> declarations to List<List<String>>.
   1. Start: ArrayList<ArrayList<String>>.
   2. Change: List<List<String>>.
   3. Reason: The original way was quick and hacky, so that we could get it done. The new way allows us the opportunity to use different kinds of lists, if the need ever arises.

**2. Changes to Existing Implementing Code**

1. Comparison logic moved from Administrator to User.
   1. Start: Administrator contained comparison logic used to sort all Volunteers by their First and/or Last Names.
   2. Change: This comparison logic now exists in User.
   3. Reason: The comparison process relied on overriding the compareTo method for Users. This was previously done using anonymous inner classes, but was simplified by just overriding the method directly in User.
2. ParkManager subset getter logic moved to DataPollster.
   1. Start: ParkManager contained a lot of logic dealing with getting subsets of jobs, parks, and volunteers.
   2. Change: This logic now exists in DataPollster.
   3. Reason: We decided that DataPollster should not just return the fields of JobList, but should also be in charge of returning subsets of those fields as well. This made these methods more accessible to other user classes, and because they are all together, made our program as a whole easier to understand.
3. DataPollster.getPendingJobs() greatly simplified.
   1. Start: DataPollster.getPendingJobs() was very lengthy and complex.
   2. Change: Much of the logic in getPendingJobs() was isolated into two new helper methods, shareSameDate() and visibleToVolunteer(). Furthermore, the logic in all three methods was simplified.
   3. Reason: getPendingJobs() was originally obtuse and very hard to understand. These changes made DataPollster more readable and reliable.
4. DataPollster and Schedule now use emails as indexing keys rather than User objects.
   1. Start: DataPollster and Schedule methods would take in User objects as arguments and use them to index JobList and UserList for relevant information.
   2. Change: DataPollster and Schedule now use the unique User email Strings instead as keys to index JobList and UserList.
   3. Reason: This cut down on dependencies on the User model classes, and indexing with an unique String key was much cleaner than doing so with an entire object.
5. SaveLoad methods now take in filename strings.
   1. Start: SaveLoad methods were hardcoded to always save to jobList.ser and userList.ser.
   2. Change: SaveLoad methods now take in the filenames of files to save to.
   3. Reason: This made our saving processes much more malleable. Now, we can have multiple different save files on hand, which made testing SaveLoad much more feasible.
6. Complete overhaul to SaveLoad methods.
   1. Start: SaveLoad had methods that were very hard to understand.
   2. Change: Text parsing methods were removed, and the remaining methods are now thoroughly explained and better written.
   3. Reason: SaveLoad was one of our hardest classes to understand, and as such, it was volatile. It is now much safer, because our group is confident in how it is supposed to function.
7. New JobIDs are generated dynamically.
   1. Start: Job had a static int that was incremented every time a new Job was created.
   2. Change: When a new Job is created, we now look through JobList to generate a new unique JobID.
   3. Reason: By making a static process dynamic instead, we made our interactions with JobList more flexible and less prone to bugs.
8. A Job now contains data indicating whether it is in the past.
   1. Start: Previously, to tell if a Job was in the past, such data had to be recalculated constantly.
   2. Change: Now, Job has a boolean field to tell if the job is in the past.
   3. Reason: Eliminating constant recalculation of this data improves the speed of the program.
9. Volunteers can now be directly added to a Job.
   1. Start: We used to have the Schedule class facilitate adding a Volunteer to a Job.
   2. Change: The Job class has the ability to directly add Volunteer information to its list of Jobs.
   3. Reason: It logically made more sense to have the Job facilitate adding a Volunteer to itself rather going through an intermediary class like Schedule.
10. JobList has an initial capacity of 10000.
    1. Start: JobList used to have to resize itself dynamically.
    2. Change: JobList is now made arbitrarily large to prevent data structure resizing.
    3. Reason: This optimization upon initialization allows for greater efficiency of the system.
11. An abstract User superclass was created to allow the UserList class to treat all users polymorphically.
    1. Start: Volunteers, Administrators, and Park Managers used to all be treated differently, meaning our UserList class used to have three separate lists for each type of User.
    2. Change: All users are now stored in one master list that contains Park Managers, Administrators, and Volunteers.
    3. Reason: This fix was crucial in creating a more flexible code base that utilizes the benefits of polymorphism.
12. List getters for specific types of Users in UserList employ reflection.
    1. Start: Beforehand, each specific type of User had a String field specifying which type of User they were.
    2. Change: The getter methods in UserList that return lists of specific types of Users now determine user type through reflection, passing that information into another helper method.
    3. Reason: The code base is more robust and abstract by utilizing reflection in Java through generics and the “instanceof” keyword.
13. Login took the place of Main.
    1. Start: Login used to be Main, which included the main method of our program.
    2. Change: Login now contains the functional components controlling the login process for LoginUI (its main method got moved to the new Runner class).
    3. Reason: Consistency and organization of the code were paramount reasons for this decision. The group decided that, like all of the Users and their respective UI classes, Login should be associated with a LoginUI class.
14. Addition of regular expression usage to validate strings being in the correct email format.
    1. Start: All strings were previously accepted as valid email addresses.
    2. Change: Now, all strings entered when the user is prompted for an email address (either to login or to register) are checked with this robust regular expression to determine whether it matches a valid email format (ie. includes the @ symbol, etc.).
    3. Reason: The usage of regular expressions for checking email addresses is done in most applications. Therefore, we wanted to conform to industry standard for this program.
15. Changed MainUI to LoginUI.
    1. Start: MainUI used to contain not only our program’s main method but also many functional routines that needed to move out of the UI class.
    2. Change: After removing its main method and renaming it to LoginUI, we’ve put only essential logic and printing in this class.
    3. Reason: We wanted to separate login logic and the main method of the program into different classes for ease of testing and increased modularization.
16. Removed Park class.
    1. Start: The Park class stored information that was unnecessary for our application (ie. zip code and city name).
    2. Change: Once we realized that all we needed was the park name, we edited our program to treat all programs as strings rather than Park objects.
    3. Reason: This small change made a big impact in terms of reducing our program’s memory efficiency.
17. Encapsulated functionality in all User UI classes.
    1. Start: User UI classes had many functional, public methods.
    2. Change: All methods but commandLoop() in all User UI classes are private.
    3. Reason: Encapsulating the UIs’ behavior in this way limits the ability for the client to tamper with the program as it is running.

**3. New Implementing Code (“start” is not applicable)**

* Classes for all business rules.
  + Change: For each of the business rules, we were using private methods to keep track of them. We took those methods and put them in individual classes.
    - BusinessRule1
      * Added, tested.
    - BusinessRule2
      * Added, tested.
    - BusinessRule3
      * Added, tested.
    - BusinessRule4
      * Added, tested.
    - BusinessRule5
      * Added, tested.
    - BusinessRule6
      * Added, tested.
    - BusinessRule7
      * Added, tested.
    - BusinessRule8
      * Added, tested.
  + Reason: We created separate classes for each business rule so that we could test them individually and thoroughly.
* New Runner class containing main method of program.
  + Change: Moved the main method from the Main class (became Login) to a separate Runner class that only contains this method and is defined for this sole purpose.
  + Reason: Since starting the application from inside a model class seemed inappropriate, we created this class in the startup package.
* Improved inheritance through the creation of a UI abstract class.
  + Change: Similar to how we decided to treat all types of Users more generally, the group created an abstract UI class to allow all of the Users’ UI classes to share the same API.
  + Reason: Since we realized that all of our UI classes seemed to share similar fields, we moved this data (ie. Scanner) up the inheritance hierarchy.
* Treated all Users generally with the User abstract class (also affected UserList class).
  + Change: Implementing this class allows other classes dependent on the various users to become more concise, like UserList. UserList became a much more logical bit of code as we changed it to only have a single master list of Users rather than individual lists for specific types of Users.
  + Reason: User was made to represent the fact that all Users were identified by their email and full name. Furthermore, it made sense to compare Users for equality here as well as to perform the sorting previously done in Administrator.

**4. Changes to Tests**

* Pre-existing model tests: DataPollsterTest, JobTest, and ScheduleTest.
  + Start: Due to time constraints, these tests were never submitted as passing. Also, the group ended up changing the API without updating its associated unit tests. The unit test methods were also too large and encompassed many input partitions.
  + Change: Fixed these tests given new API. Furthermore, we ended up breaking up individual unit test methods along input partitions of the given methods that they were testing.
  + Reason: We waited to fix this test because we wanted to focus on refining its associated class’ API because, as noted in the class responsibilities section, the division of computational labor changed from deliverable 2 to deliverable 3.
* New model tests: AdministratorTest, JobListTest, LoginTest, ParkManagerTest, SaveLoadTest, UserListTest, and VolunteerTest.
  + Change: We needed to test the logic in these model classes! Additionally, by doing so, we found many bugs in them and we were able to better define the scope of each class in terms of its interaction with other classes as well as its role within the larger system.
  + Reason: After working on these classes’ APIs, we wrote the associated unit tests for those methods (repeating this process while there was more functionality to add).
* New model.businessRules tests: BusinessRule1Test, BusinessRule2Test, BusinessRule3Test, BusinessRule4Test, BusinessRule5Test, BusinessRule6Test, BusinessRule7Test, and BusinessRule8Test.
  + Change: We wrote separate classes for all business rules to more effectively test the constraints that they are to enforce. So, these test classes were written to test that the business rules were being checked properly. Having every business rule as its own classes allowed other classes to employ the functionality of these classes to do its own checks.
  + Reason: To be sure that these classes were operating properly, we tested the one or two methods of each of these classes with at least as many unit testing methods. Note that, since most of these classes only had one very simple method, their corresponding unit test class also had a minimal number of test methods.