**Document 04 – Development Requirements**

1. **Project Management**
2. You will use a GitHub *Project* to manage your development. The project name is *Sprint 1*. You will have these columns: *To Do, In Progress, Complete.*
3. For your project you are required to break work down into small tasks that 1-2 people will work on (some tasks may involve the whole group). These tasks should be estimated to take between 0.5 hours and 4 hours in length. Think about this carefully keeping in mind that you must work as a team, and that all team members should contribute to the development tasks. Tasks will be entered as *Issues* in GitHub and added to the *To Do* column. From there, they will be moved to an *In Progress* column and finally to the *Done* column.

This will be new activity and it might be a bit of a struggle to do this. Almost all software companies use this approach. Many times a project manager will develop these, sometimes in conjunction with developers. In some companies, a developer will simply go to the *To Do* list and “pull a ticket.” When complete, they pull another one.

Task/Issue breakdown is iterative. Your group will identify some initial tasks to get started on development. As issues are in progress or completed you will learn more about what you need to do. If a task turns out to be much bigger, or involve things you didn’t initially think of, then simply add them as new tasks, possibly closing out the initial task. Thus, Task/Issue breakdown is ongoing.

Sometimes a task may be to figure out what needs to be done, a planning session, which will result in concrete tasks that contribute to the development.

1. **Testing**
2. You are required to have JUnit tests for each class following the conventions discussed in class. Note the following about unit tests:

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when such test cases fail, it may be unclear which component is causing the failure. Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher-quality unit that is also more maintainable.

<https://en.wikipedia.org/wiki/Unit_testing>

1. You are required to have *integration test(s)* class(es) that test two or more classes that have an association to test that the collaboration works correctly. Name these tests beginning with: *Int* and follow by the names of the classes involved. Note the following about integration tests:

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

<https://en.wikipedia.org/wiki/Integration_testing>

1. **Coding**
2. You are required to use best practices for Java coding. There are way to many to mention. Minimally, these include: meaningful class and member names, proper class and member names (camelcase), consistent indentation and use of braces, line length (<120 characters, generally).
3. You should develop Javadoc for all class members and a website should be generated and saved in your GitHub repository along with the code in the master branch.
4. **Version Control**
5. Only working, tested code should be in your master branch of GitHub and it should be in a package named *sprint1*
6. Each person is required to work in their own branch. Do NOT delete branches after merging/rebasing. Branches are required to be named: *lastname\_1, lastName\_2,* etc.
7. Commit early and commit often. When you add a method, commit. When you change something, commit. When you rename some files, commit.
8. The title of all commit messages are required to begin with one of these: Fix *short title*, Add *short title*, Change  *short title*. If you find you need another prefix, then use it. Another two may be: Refactor *short title*, Merge *short title*. Titles should meaningfully summarize what was committed.

A commit message should detail why something was done. A commit message is not always necessary.

1. When you have code on a branch ready to merge with master, you should use the rebase workflow and then open a pull request. Pull requests are required to be reviewed by at least one other member before merging.
2. **Individual Team Members**
3. All time spent on the project is entered into the Time Log as soon as you end working.
4. You should work consistently on the project.
5. **Grading Criteria**
6. Use of Issues and Project to track and manage progress
7. Appropriate unit and integrations tests
8. Use of Javadoc
9. Appropriate use of GitHub
10. Quality of code
11. Number of use cases completed
12. Quality of Document 03
13. Effort