CSE 403, Project Phase 4b: TEST2 (60 points)
Testing Resources #2

For your final "test resources" deliverable, you should submit any unit test files, automated testing facilities, testing plan documents, and other testing tools or artifacts you have created as part of the quality assurance of your project.

Wiki Page for TEST2:

Please create a **wiki page** on your GitHub that contains information to the grader about your TEST2 phase. Generally speaking this wiki page will be some details and instructions to the grader about what kinds of testing you have performed. The following sections of this spec will describe the details of exactly what information should appear on that wiki page.

Unit Tests:

Coverage: Submit additional unit tests so that your total test coverage as reported by a coverage tool such as Emma is at least **60% coverage** of the lines of code in your project. We will verify this by running your code coverage in your automated build on the Jenkins server and/or through a local build and coverage test on your source code, so this should be properly set up before submission.

Automated Build: Your automated build on the course Jenkins server is still expected to run properly. The setup you created in your Beta phase should still run and should still check out and compile/build your project, perform static analysis on your code, run your unit tests, check your coverage, and automatically email the team if anything fails. Any new tests added in this TEST2 phase should be run by your Jenkins automatic build.

Unit Test Metrics/Grading: Your unit tests will be graded on whether they follow the criteria described in the Test1 document, such as breadth, depth, and having a good mix of black/white box tests. Your tests should also now follow the design guidelines described in Osherove's *Art of Unit Testing*, such as: keeping each test method small with few assertions and minimized logic and control flow; descriptive naming; avoiding redundancy with helpers and/or parameterized tests; effectively using timeouts; testing expected errors and exceptions; using well-structured assertions with descriptive messages; and ensuring that your tests run in an unconstrained order without dependency on each other.

System/Integration Tests:

To receive full testing credit your project must include at least **two (2) areas of significant additional system/integration testing**. These are considered separate from your unit tests, though some of these tests are technically implemented as unit test files. Part of your grade for this item will depend on your choosing a reasonable subset of functionality to test and choosing effective test cases (in number, scope, and coverage) to ensure code quality. If you already did any of the kinds of testing below, your previous work can count toward the completion of this phase.

The kinds of testing you can choose from are the following. **Choose two** from the following list:

1. Integration Testing
2. Functional/UI Testing
3. Usability Testing
4. Performance Testing
5. Security Testing/Auditing
6. Reliability Testing

You can choose any pair of testing types you want, **except for the pair of #2 UI testing + #3 Usability testing**. Both of these kinds focus heavily on the UI, so we would like you to choose at least one kind of testing outside of that pair.

Your TEST2 **wiki page** should briefly describe which kinds of tests your team performed, along with a few details about each kind of test such as what exactly you did, what evidence you are submitting of this, how the customer/grader can view or run the tests, etc. Specific details to include for each kind of testing are listed in the following paragraphs.

The next pages give details about each kind of testing. We have listed activities that you could perform for each kind of testing that we would consider sufficient to get credit. There may be other ways to earn credit within a given category of testing. If you have a different idea, please consult your customer/grader to get approval.

1) Integration Testing

Turn in a set of integration tests implemented as unit test files. Turn in at least **2 nontrivial integration tests,** with a stub and mock to dodge dependencies and test states and interactions.

- at least 1 **"stub"** class/unit
- at least 1 **"mock"** class/unit

For creating mock objects, we suggest using EasyMock (which is well integrated with Android) or JMock or a similar framework, though this is not required; you may write them from scratch if you like. (These mock tools can also help create stubs.) If you do these tests, make sure that your **automated build** on Jenkins also runs your integration tests (either together with, or separately from, your other unit tests). In your wiki document, describe which parts you performed integration testing on, which test files are related, and how to run the tests.

2) Functional/UI Testing

Write functional tests to simulate usage of your product's user interface to ensure proper functionality.

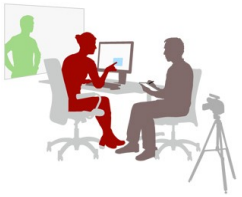
- ≥ **2 automated walkthroughs** (e.g. **Robotium** tests) of your UI that span multiple activities
- wiki descriptions of ≥ **2 ad-hoc UI tests** that a manual tester can perform on the UI

The idea of Robotium and Selenium tests is that they run outside of your entire app and automate tests on your entire app's UI, including being able to span multiple activities or web pages in one test. If you do this kind of testing, integrate the UI tests into your **automated build** on the Jenkins server.

Ad-hoc UI testing (where a user simply tries clicking things manually to see if they work) can be a part, but not all, of your work here. If you do ad-hoc testing, you must document what kind of functionality was tested (which UI controls are clicked in what order, what result is expected, etc.), and the ad-hoc tests should be related to specific user goals or use cases such as those described in your SRS document.

Submit at least 2 saved UI walkthroughs / tours / scripts that each perform a significant non-trivial task with your product. In your document, describe what parts of UI are tested, what tours/walkthroughs were done, what file(s) are related, and how to run your UI tests. A better functional UI test is one that uses its assertions appropriately (proper assertion method, etc.) and that is resilient against minor unrelated changes to your UI over time.

3) Usability Testing



Perform activities to see that your application is easy for users to interact with and understand.

- an informal **usability study** as generally described in the *"Don't Make Me Think"* reading
- a **user survey** of users' reactions to your product, its features, and its usability

Perform a user study where at least **3 users** outside your team are brought in to use your application and monitor their relative success in doing so. The more structured and specific such a user study is, the more likely it is to receive full credit. A well-run user study involves giving the user sets of specific tasks that he/she is to perform and then observing the results, making specific notes and observations about aspects that were easy or difficult for the user, amount of time required to perform the task, number of clicks or missteps along the way, etc. For full credit, give each user at least **2-3 specific and non-trivial tasks** or scenarios to perform and monitor their progress. Your study should produce some kind of artifact or documents about the test as evidence that it occurred, such as those described in the *Don't Make Me Think* reading.

You could additionally conduct a **user survey** to discover issues about the usability of your product or card sorting to discover ways that users perceive relationships between your product's features. In your wiki you should describe in moderate detail what happened in your study; along with your wiki you should submit any artifacts created during your study, such as what tasks the user was asked to perform, notes, timings, results, etc.

4) Performance Testing

Conduct a sets of tests and profilings to gauge and improve the performance of your application.

- a **performance audit** (saved logs from profiler runs, along with notes about bottlenecks)
- non-trivial **refactoring** and code changes to help ease these bottlenecks

Use a profiling tool such as hprof, VisualVM, JProfiler (Java), or ruby-prof (Ruby) to conduct your profiling. Save logs from profiler runs in your repo as evidence of your run. Identify bottlenecks in your app's performance and document these as though you were doing a performance-based code review by annotating the relevant source code. Then perform non-trivial refactoring of the code to help improve the performance around these bottlenecks. In your wiki, describe what automated tool(s) were used, which parts of code were bottlenecks, and how the customer/grader can run them to verify their results. If you make code changes based on a performance profile, also indicate the changes with comments in the code and on the version control checkin(s) related to these changes.

5) Security Testing/Auditing

Perform a set of activities to help ensure the security of your application.

- a comprehensive **security audit** of your code
- some sort of **automated security check** that can be run from the command line

For starters, perform a **security audit** of your code, which is similar to a code review but with a focus on security issues. Use an audit checklist with a set of possible vulnerabilities and issues, and fill out this checklist as you go along in your audit(s).

After your audit, fix any known major security issues in your code. For any code changes you make based on the security audit, indicate the changes with comments in the code as well as on the version control checkin(s) related to these changes. In particular, if you are doing security revisions for this phase, you should change your project to use **secure HTTPS / SSL connections** for any web service requests your app makes to a remote web server, and you should also **encrypt any personal or sensitive data** that your app stores on the mobile device, in its local databases, or any remote databases you are using to save such data. (Non-personal and non-sensitive data does not need to be encrypted.)

Also set up and run at least one **automated security checking** tool such as Tarantula, Acegi, Rats, Wapiti, Oedipus, Nikto, etc. to look for vulnerabilities in your application or server. If you do these tests, make sure that your **automated build** on Jenkins also runs your automated security test (either together with, or separately from, your other unit tests).

In your wiki, tell us what audits you performed, what kind of attacks or issues you examined, and what code was changed. In your wiki, also describe what automated tool(s) were used and how the grader can run them to verify their results.

6) Reliability Testing

Perform a set of testing activities to help ensure the robustness of your application.
Include at least **two of the following items** (you do *not* need to do all six of them):

- implementing substantial **logging** in your code to be better aware of the causes of problems
- setting up **site monitoring** software to check if your app is "live" and report an alert if not
- performing **load tests** to see how your product, site, or data back-end handles heavy traffic
- adding **instrumentation** and/or **performance counters** to your code to help you better self-diagnose bottlenecks
- setting up a **load monitoring tool** such as JMeter
- performing specific tests with subsystems crippled and/or disabled, such as your database or network connection

In your wiki, describe what metrics you are evaluating or monitoring (uptime? performance under load? handling subsystem failure? etc.) and how the customer can examine your reliability testing code to evaluate and grade it.

(Since performance testing and reliability testing are closely related, you can choose to do a combination of some parts of each if you like. Consult your customer/grader to make sure you have chosen a sufficient set of testing tasks to perform.)