CSE 403, Winter 2010
PHASE 7 (60 points): Testing Resources (TEST)
due Sat March 13, 11:59pm

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. Submit or provide the following three (3) major items:

1. Unit Tests

Unit Test Metrics: Your unit tests will be graded on whether they follow the criteria described in the TPS 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.

Coverage: You are required to administer unit testing over a significant portion of your application's model code using a framework such as JUnit. Specifically, you should have at least **40% code coverage** by your unit tests (as recommended by Desikan/Ramesh's *Software Testing* p62). In order to prove that you have accomplished this, you should submit a script that the grader/customer can execute that will run a code coverage tool to verify the percentage of coverage. You must set up this tool in your repository or project space.

Instead of overall coverage, if you like, you may set up a tool that weights importance of coverage by complexity (e.g. Crap4j) and add test cases to your code until you have covered what it reports as **50%** of the "more complex" code in your project. Speak to your customer if you plan to do this.

Build/Check-in Tests: You must submit some sort of automated unit testing facility. That is, you must have some sort of build test, check-in test, or automatic nightly build test that runs without user intervention. You should submit a script that your customer can use to run all of your unit tests (or as many as can be run from a terminal) with a single command. You may want to learn about a build tool such as Ant that can help to connect your builds to unit tests, coverage tests, etc.

2. System/Integration Tests

As described in the TPS document, to receive full testing credit your project must include at least **three (3) areas of significant additional system testing and/or integration testing**. These are considered to be 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 the quality of your implementation.

Submit a **brief (at least 1-2 page) document** describing which 3 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 following are the kinds of testing that you could perform for this deliverable. 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.

Integration Testing: Turn in a set of integration tests implemented as unit test files. Turn in at least **3 nontrivial integration test files, using at least 2 "stub" classes/units and at least 1 "mock" class/unit** to dodge dependencies and test states and interactions. For creating mock objects, we suggest using JMock or a similar framework, though this is not required; you may write them from scratch if you like. (See the web site for links about mocks. JMock can also help create stubs.) Turn in a script so that the customer can run your integration tests (separately from other unit tests). In your document, describe which parts you performed integration testing on, which files are related, and how to run the tests.

Security Testing: Perform a set of activities to help ensure the security of your application. (This option is available for web and mobile applications only, or client apps with a heavy amount of network/internet activity.) For starters, perform a comprehensive **security audit** of your code, 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). You should also set up and run at least one **automated security checking** tool such as Tarantula, Acegi, Rats, Wapiti, Oedipus, Nikto, etc. to test whether it is able to find any vulnerabilities in your application or server. In your document, tell us what audits if any you performed, what kind of attacks (automated or otherwise) that you tried, which attacks if any succeeded, and what code was changed. If you make code changes based on a security audit, indicate the changes with comments in the code as well as comments on the SVN checkin(s) related to these changes. In your document, also describe what automated tool(s) were used and how the customer/grader can run them to verify their results.

Functional/UI Testing: Write a set of automated tests that simulate usage of your product's user interface to ensure its proper functionality. **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.

In addition to any ad-hoc testing, you must also submit some form of **automated UI testing**. This should be done with some sort of framework that simulates or automates usage of a user interface, such as Selenium (web apps) or Abbot/Costello (Java). Submit at least **6 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.

Usability Testing: Perform a set of activities to help ensure that your application is easy for common users to interact with and understand. For example, perform a user study where 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. For example, 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.

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 document you should describe in moderate detail what happened in your study; along with your document you should submit any artifacts created during your study, such as what tasks the user was asked to perform, notes, timings, results, etc.

Performance Testing: Conduct a sets of tests and profilings to gauge and improve the performance of your application. Use a profiling tool such as hprof, JProfiler, JMeter (Java), ruby-prof (Ruby), or Xdebug (PHP) to conduct your profiling. Save logs from your profiler runs to turn in as evidence that you conducted such a performance test. Identify bottlenecks in your app's performance and perform a non-trivial amount of refactoring to help ease these bottlenecks. In your document, describe what automated tool(s) were used, which parts of code were found to be bottlenecks, and how the customer/grader can run them to verify their results. If you make code changes based on a performance profile, indicate the changes with comments in the code as well as comments on the SVN checkin(s) related to these changes.

Reliability Testing: Perform a set of testing activities to help ensure the robustness and reliability of your application. (This option is available for web and mobile applications only.) Such activities should include a nontrivial combination of the following: Setting up **site monitoring** software to check whether your app is "live" and reporting an alert if not; Performing **load tests** to see how your product handles heavy traffic; implementing effective **logging** in your code to be better aware of the causes of problems later; Add **instrumentation** and/or **performance counters** to your product's code to help you better self-diagnose bottlenecks and problems; set up a **server / load monitoring tool** such as JMeter; and performing specific tests with subsystems crippled and/or disabled, such as your database or network connection speed. In your document, 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.

Localization/Internationalization Testing: Make your application's user interface capable of displaying itself in at least one major spoken/written language besides English. Modify your project's UI code to make it properly localized, separating out strings into resource files, stopping any English-specific text, strings, or HTML content from being hard-coded into the application. In your document, describe how localization was done, who wrote the localized text and how its correctness was verified, what files(s) of your product were affected, and how the customer can run your app in its other language.

3. Bug Tracker

Since a key goal of testing is to reveal the presence of bugs, we will check your bug tracker to see that your group has been using it throughout the project to report bugs, issues, unfinished features, and other issues. You should be filing bugs in your tracker for any bugs uncovered via testing, even if the bug is fixed immediately upon its discovery. A performance bottleneck, poorly usable UI feature, etc. should also be considered a "bug" and included in your bug tracker.

As we are grading we will examine your bug-tracker to look for an adequate number of reported bugs. The bugs should be thoroughly filled out with information such as: description, steps to reproduce the bug, severity, who the bug is assigned to, its current status (open, resolved, won't fix, etc.)

Since this turnin coincides with the end of your work on the project, the vast majority of the bugs in your tracker should be resolved in some way, either as being "Fixed" or that you "Won't Fix" the bug along with a reason, etc.

There is not a specific exact number of bugs that must be present in the system to get full credit here. But there should be a significant number of bugs of various kinds in the system. There should also be evidence that you have been using the tracker throughout a significant portion of the dates of your project (not just a bunch of bugs added in the last few days before this turnin is due). The bugs should also be well distributed between your various developers on your team.