CSE 403, Project Phase 3b: TEST1 (50 points)
Testing Resources 1

For this phase you will submit two major deliverables:

1. A set of **unit test cases** that test part of the functionality of your "Beta" code.
2. An **automated testing** setup that periodically builds your project, runs tests and static analysis on your code, and alerts you if any problems arise, which helps with constantly ensuring that new code integrates properly ("Continuous Integration").

Unit Test Cases

Submit unit test cases that combine to **cover at least 50%** of the total **lines** of source code of your project. (To verify that you have achieved this coverage, you will need to set up a coverage checking tool such as EMMA or Cobertura.) Each test case file must examine a major class or subsystem of your project.

Framework: Write these tests using the appropriate *"-unit"* framework for your language. For example, if you are using Java, use JUnit, TestNG, or similar. (We highly recommend JUnit.) In Ruby, use Test::Unit or RSpec. (We recommend using Ruby's built in unit testing facilities rather than external libraries.)

Breadth: Each test file should test only one class/subsystem; its area of testing should be commented clearly. Each test file contains several test cases (test methods) that each test a small part of the functionality of the class(es) under test. We will check that you have chosen a broad set of non-trivial functionality to test. We suggest that your unit tests focus their coverage on the "model" and/or data-driven aspects of the project, since these are often conducive to the unit testing process; but you can write tests for any part you like.

Depth: For full credit, each test class should be fairly comprehensive over the surface of the class(es) that it is testing. That is, there should be at least one testing method/case that covers each major method, constructor, etc. of the class(es) under test. It is likely that some of these methods are more complex than others; more complex methods (such as those that accept several parameters, or those that perform complex operations on the object's state) may need to be covered by multiple test methods in order to be considered well covered. For example, if you were testing a method that adds an element to a list in sorted order, you would want one test case that adds to the front of the list, another that adds to the end, another to the middle, another to an empty list, another to a one-element list, another to a large list, and so on.

Black/white box: At least one of your unit tests must be a black-box test (where the tester is *not* fully aware of all of the internal details of the code under test) and at least one other must be a white-box test (where the tester looks at the code and uses this information to guide the creation of test cases). Indicate this by labeling (commenting) every test case as to whether it is black-box test or white-box. The tests should fit their given category. That is, if the test is black-box, it should focus on externally discoverable behavior such as boundary cases, parameter values, unexpected/edge cases, and combinations of calls. If the test is white-box, it should focus on covering each path and statement of the code in the class under test.

Bug-tracking: If any of your test cases are related to bugs or issues in your bug tracker, the tracker should mention what test case was written to address the issue, and the commit message in which that test was checked in should refer to the related bug/issue.

Test-driven Development: At least one of your tests must be written in a partial "test-driven development" (TDD) style. This means that the test must be written to cover a class or subsystem that has not been written or finished yet. Generally the class under test is created with mostly empty or trivial "stub" methods that do not yet produce the proper expected behavior. It is expected that this test case will fail when it is run. The idea is that now the developer must write the class such that it will pass all of the test cases. The test case should be comprehensive enough that once the code under test is written and passes it, the developers have reasonable assurance that the class under test has been well implemented. At the time at which your TEST1 is submitted, either the code/class under test by your TDD test(s) should not be completed yet, or if you have ended up writing it by then, there should be a point in your version control log at which the code under test was not complete but the TDD unit test file(s) did exist. Ask your primary customer TA if you are unsure how to meet this requirement.

Internal Correctness: The unit tests will be graded on both external and internal correctness. Unit test code is not exempt from the constraints normally applied to your project code, such as being commented and meeting your style guidelines. We will grade your tests on whether they have the qualities listed in this spec, as well as whether they demonstrate intelligent usage of the testing framework (understanding which `assert` method to call, or using timeouts and exception checking properly, or setting up before/after code that runs before each test, etc.), and whether they follow the qualities of effective unit testing as described in the readings in class. For example, tests should not depend on being run in a certain order, should not call each other, should test boundary cases, and should use helpful assertion messages for when tests fail. Each testing method should be concise and contain a minimal number of assertions and minimal amount of complex logic such as loops and if/else statements. Similar tests should minimize redundancy; it is often wise to concoct various test "helper" methods that allow you to describe tests at a high level as calls to another helping method. This way your test program does not become unwieldy or redundant, and it is easier to add more tests.

Automated Testing / Continuous Integration:

Set up an automated build and test job that will monitor your source code and other resources. We have set up a Continuous Integration (CI) server running the **Jenkins** CI software, available at the following URL *(note the **"n"** in front of "webster")*:

- **http://nwebster.cs.washington.edu:8080/cse403/**

The Jenkins software is ready and waiting to accept new Jobs to be added by your group. You should set up a job that does all of the following tasks at least **once per day**:

- **Checks out the current build** of your project from the version control system, including all source code and resources.
- **Compiles/builds** (if necessary) any source code into binary format, along with running any related scripts to prepare any necessary resources or files that require pre-processing.
- Performs some form of **static analysis** of your code, such as running Checkstyle or FindBugs or PMD over the project.
- Runs all of your **unit tests**, noting whether they passed or failed.
- Checks the **coverage** of your unit tests as a percentage of the overall code size (e.g. lines of code), to ensure that the coverage is over the threshold described in this document. You can find your test coverage using a tool such as EMMA or Cobertura.
- If any **issues or failures** are found during any of the above steps, the Job should **send email** to your group as well as to your customer TA to alert them that something has gone wrong. For example: if the code does not compile for any reason; if the static analysis reveals a violation in the code; if any unit test fails to pass; or if your coverage is below the required percentage.

You won't lose points if your automated CI test reports a failure, so please feel free to set it up even before you have each item fully "passing." For example, if you know that your unit tests cover only 30% of your code, and therefore will fail the coverage check, you can still set up Jenkins to perform that check and alert you that your test coverage is insufficient. The point is not to always avoid these messages but to let them help remind you to stay on top of these issues and not let them linger for long without repairing them.

If you feel that the provided NWebster / Jenkins CI setup is not sufficient for your needs, please contact your customer TA. We can try to provide additional resources on that server for you if necessary. Or if you have a strong reason why a different CI server setup of your own is more appropriate than our provided server, we will be willing to discuss this with you. But we prefer to have every group using the provided server if possible.

Submission and Grading:

Submit your documents and resources online by checking them into your version control system on **GitHub** by the due date. The automated testing system must be present and actively running in the given Jenkins CI server by the due date. The automated testing is more useful the earlier it is set up, so we generally do not recommend postponing it until the very end of this phase.

FYI: Testing will also be a large part of the next phase of the project. In that phase, we will ask you to raise the minimum percentage of your code that must be covered under unit tests, as well as asking you to perform additional kinds of testing besides unit tests (such as functional/UI testing, performance testing, reliability testing, load/stress testing, etc.).

This document and its contents are copyright © University of Washington. All rights reserved.