CSC 307 Lecture Notes Week 8
Use of Formal Method Specification in Testing
Introduction to System Testing Techniques
Testing Implementation, in TestNG and JUnit

I. **Detailed summary of milestones 7-8 deliverables**

A. ***Design and Implementation***

1. Package design refinement.
2. Javadoc.
3. Java implementation of model classes.
4. Necessary data structures.
5. View classes refined to use real model data.
6. Bottom line -- some stuff runs for real.

B. ***Testing***

1. One integration test plan per team.
2. One class test plans per team member.
3. Three methods Spest spec'd, per member (should already be done from Milestones 5-6).
4. Three unit tests per member (for the methods that have Spest).
5. Code reviews.

C. ***Administration***

1. Revised `HOW-TO-RUN.html`
2. Separate `m8-duties.html`
3. Updated `work-breakdown.html`
4. Design reviews, wed & fri week 9.

II. **The Testing "Big Picture"**

A. This was on the chalkboard or whiteboard during lecture on Friday.

B. The picture will be here in these notes in a couple days.

III. **Deriving and refining method specifications.**

A. The validation of user inputs requires that we know exactly what constitutes valid versus invalid input values.

1. The purpose of operation pre- and postconditions is to answer just this question.
2. In addition to input validation, pre- and postconditions are used for formal system testing to inform the development of of unit tests.

B. Here is a recap from 307 of what pre- and postconditions mean:

1. A precondition is a boolean expression that must be true before the method executes.
2. A postcondition is a boolean expression that must be true after the method has completed execution.

IV. **How formal specification is used in testing.**

A. As we'll discuss below, a formal function test consists of the following elements:

1. Test inputs within legal ranges, and expected output results.
2. Test inputs outside of legal ranges, and expected output results.
3. Test inputs on the boundaries of legal ranges, and expected output results.
4. Test different combinations of multiple inputs.

B. The formal preconditions are used to determine what the inputs should be.

C. The formal postconditions are used to determine the expected results for the given inputs.

V. **How formal specification is used in formal verification.**

A. In order to verify a program formally, two forms of specification must be provided:
   1. A formal specification of the given program
   2. A formal specification of the language in which the program is written in

B. Hence, a formal program specification is an integral part of formal verification; that is, a formal specification is the "entry ticket" to program verification, that that one cannot verify a program without its formal spec.

C. We will discuss formal verification details in an upcoming lecture.

VI. **Precondition enforcement -- "by contract" style versus "defensive programming" style.**

A. At the specification level, failure of an operation precondition renders the operation simply "undefined".
   1. For an abstract specification, this is a reasonable definition of precondition failure.
   2. However, at the design and implementation level, precondition failure must be dealt with more concretely.
   3. There are two basic approaches to such concretization.

B. *Approach 1:* A precondition is a guaranteed assumption that will always be true before a method to which it is attached is executed.
   1. This approach is can be called the "programming by contract" approach.
   2. In this approach, the code of the method does not enforce its own precondition, but rather the precondition must be enforced by all callers of the method.
   3. Such enforcement can be formally verified or implemented with runtime checks at the *calling* site.
   4. Bottom line is that the method being called assumes that it's precondition is true at all times, and does no checking of the precondition itself.

C. *Approach 2:* A precondition must be checked by the method to which it is attached.
   1. This approach can be called the "defensive programming" approach.
   2. In this approach, the code of the method includes logic to enforce its precondition.
   3. The enforcement can:
      a. Assert unconditional failure on any precondition violation.
      b. Return an appropriate "null" value as the method return value or in an output parameter.
      c. Output an appropriate error report to stderr or the user view screen.
      d. Throw an appropriate exception (see below for further discussion).

D. In Model/View communication, it is useful to use the exception handling approach, as illustrated in the example in the Week 4 notes.

E. We will discuss further the issue of when and how to use exception handling in design in upcoming lectures.

VII. **Details of deriving and refining formal method specifications.**

A. Start with the Spest specs you developed for the abstract model.

B. Update and expand these specs based on design refinements you make for the implementation.

C. As discussed in Lecture Notes Week 5, formal preconditions and postconditions appear in the Spest specification in a reasonably standard form of predicate logic.
   1. In the abstract specification, the point of the preconditions and postconditions was to help us fully understand the operations from the end user's point of view, so we could be sure that what the user sees in the HCI is backed by a complete and consistent specification of what the program does.
   2. When the abstract specification is refined into the design, the formal logic will be refined to include implementation-level considerations, in addition to the user-level considerations of the abstract spec.
   3. In addition, as new methods are designed, they too are formally specified.
   4. The formal specifications are then used to develop the functional test plans for the implementation.

D. Here's a recap of the key aspects of the Spest notation covered in the Week 5 notes.
   1. The `return` keyword is used as a value not as a control construct. E.g., if a method returns the sum of its two inputs `x` and `y`, the postcondition looks like this:

```
post: return == x + y;
```

2. The `if`, `then` and `else` keywords are used as expression operators not as control constructs.
   a. To be precise, the following Java if-then-else expression
      ```
      if X then Y else Z
      ```
      is semantically equivalent to the following standard Java expression
      ```
      X ? Y : Z
      ```
   b. The following if expression
      ```
      if X then Y
      ```
      is semantically equivalent to *no* standard Java expression, since Java always requires the "else" part of and if-then-else expression.
   c. The else-less if-then expression above is equivalent to the following Java
      ```
      X ? Y : null
      ```
      for all types for which null is a legitimate value.
3. In Spest, the "prime" notation is used to denote the output value of an identifier; e.g.,
   a. If *x* is a parameter or instance variable, then *x* without the prime denotes its value before the method run.
   b. The value of *x* with the prime, i.e., *x'*, denotes its value after the methods runs.
   c. The prime suffix can only be appended to method parameters or to accessible data members within a method.
4. Universal and existential quantifiers are supplied, with the following syntax:
   ```
   forall (T x;constraint;predicate)
   exists (T x;constraint;predicate)
   ```
   a. The universal quantification form is read "for all values *x* of type *t*, such that *constraint* holds, *predicate* is true."
   b. The existential form is read similarly as "there exists at least one value *x* of type *t*, such that *constraint* holds, and *predicate* is true."
5. Lecture Notes Week 5 has further discussion and examples.

-- Now onto System Testing Techniques --

VIII. **General concepts of system testing.**

A. Software components are independently testable.

B. Testing is thorough and systematic.

C. Testing is be repeatable, with the same results each time.

IX. **Overall system testing styles**

A. Top-down
   1. Top-level functions in the function calling hierarchy are tested first.
   2. Function "stubs" are written for lower-level functions that are called by functions above.

B. Bottom-up
   1. Lower-level functions in a function calling hierarchy are tested first.
   2. Function "drivers" are written for upper-level functions that call functions below.

C. Object-oriented
   1. All functions for a particular class are tested, independent of how they may be used in overall system.
   2. Stubs and drivers are written as necessary.

D. Hybrid

1. A combination of top-down, bottom-up, and object-oriented testing is employed.
2. This is a good practical approach.

E. Big-bang
1. All functions are compiled together in one huge executable (typically the night before it's due).
2. We cross our fingers and run it.
3. When the big-bang fizzles, we enter the debugger and keep hacking until things appear to work.

X. **Practical aspects of independently testable designs.**

A. For all modules to be separately designed and implemented, modular interfaces should be designed cleanly and thoroughly.
1. Don't fudge on function signature details or pre/postcondition logic; i.e., think clearly about these details *before* the implementation begins.
2. Be clear on what needs to be public and what protected.

B. Be prepared to write *stubs* and *drivers* for other people's modules so that independent testing can be achieved.

XI. **General approaches to testing and verification**

A. Black box testing
1. Each function is viewed as a black box that can be given inputs to produce its outputs.
2. The function is tested from the outside (specification) only, without looking at the code inside.

B. White-box testing
1. Each function is viewed as a "white" (i.e., transparent) box containing code.
2. The function is tested by supplying inputs that fully exercise the logic of its code.
3. Specifically, each logical control path through the function is exercised at least once by some test.
4. This is the kind of testing that is done informally during the course of system debugging.

C. Runtime pre-condition enforcement
1. Code can be added to functions to enforce preconditions at runtime.
2. For example, if a precondition states that a certain input must be within a certain range, then code is added to the beginning of the function to check this condition
3. The function returns (or throws) an appropriate error if the condition is not met.

D. Formal verification
1. The pre- and postconditions of each function are treated as formal mathematical theorems.
2. The body of the function is treated as a form of mathematical formula, given certain formal rules of program interpretation for the language in which the function is written.
3. Verification entails proving that the precondition theorem implies the postcondition theorem, with respect to the mathematical interpretation of the function body.

XII. **Functional unit test details**

A. For each method, a list of *test cases* is be produced.

B. This list of cases constitutes the *unit test plan* for each method.

C. A unit test plan is defined in the following general tabular form, as show in Table 1.

D. Note that
1. The inputs for each case specify values for all input parameters as well as all referenced data fields for that case.
2. The outputs for each case specify values for all reference parameters, return value, and modified data fields for that case.
3. In any case, data fields that are not explicitly mentioned in the inputs or outputs are assumed to be "don't

Case No.	Inputs	Expected Output	Remarks
1	parm 1 = ... parm m = ... data field a = ... data field z = ...	ref parm 1 = ... ref parm n = ... return = ... data field a = ... data field z = ...	
n	parm 1 = ... parm m = ... data field a = ... data field z = ...	ref parm 1 = ... ref parm n = ... return = ... data field a = ... data field z = ...	

Table 1: Unit test plan.

care" -- i.e, not used as an input or not modified on output.

E. One such test plan is written for each method in each class.

F. In an object-oriented testing strategy, unit test plans are referenced in then class test plans.

XIII. **Module, i.e., class testing**

A. Write unit test plans for each class method.

B. Write a *class test plan* that invokes the unit test plans in a well-planned order.

C. General guidelines for class testing are the following:

1. Start the class test plan by invoking the unit tests for the constructors, so that subsequent tests have field data values to work with.
2. Next, unit test other constructive methods (i.e., methods that add and/or change field data) so that subsequent tests have data to work with.
3. Unit test selector methods (i.e., methods that access but do not change data) on the results produced by constructive methods.
4. Test certain method interleavings that might be expected to cause problems, such as interleaves of adds and deletes.
5. Stress test the class by constructing an object an order of magnitude larger than ever expected to be used in production.

D. Once the plan is established, write a test driver for all methods of the class, where the driver:

1. executes each method test plan,
2. records the results,
3. compares the results to the previous test run,
4. reports the differences, if any

E. A couple concrete examples of class test plans are in the Calendar Tool testing directory:

- unix3:~gfisher/work/calendar/testing/implementation/source/java/caltool/model/caldb/UserCalendarTest.java
- unix3:~gfisher/work/calendar/testing/implementation/source/java/caltool/model/caldb/UserCalendarTest.bjava

F. In terms of Java details:
   1. Each class *X* in the system design has a companion testing class named *XTest*.
   2. A test class is a subclass of the class it tests.
   3. Each method *X.f* has a companion unit test method named *XTest.testF*.
   4. The comment at the top of each test class describes the test plan for that class.
   5. The comment for each unit test method describes the unit test plans for the testing method.
   6. Each tested class provides a specialization of `java.lang.Object.toString`, which is used to dump the values of tested class objects.

XIV. **Integration testing**

A. Once class test plans are executed, classes are integrated.

B. Specifically, stub methods used in a unit or class test are replaced with the actual methods.

C. Subsequently, the test plan for the top-most method(s) in a collection is rerun with the integrated collection classes.

D. The integration continues in this manner until the entire system is integrated.

E. A concrete example of an integration test plan is in the Calendar example testing directory:

```
unix3:~gfisher/work/calendar/testing/implementation/source/java/caltool/integration-test-
```

XV. **Black box testing heuristics**

A. Provide inputs where the precondition is true.

B. Provide inputs where the precondition is false.
   1. These form of inputs do examples/***not*** apply to by-contract methods that do not check their on precondition.
   2. These form of test inputs ***do*** apply to methods with a defensive implementation, where the method explicitly checks the precondition and throws an exception or otherwise returns an indication that the precondition is violated.

C. For preconditions or postconditions that define data ranges:
   1. Provide inputs below, within, and above each precondition range.
   2. Provide inputs that produce outputs at the bottom, within, and at the top of each postcondition range.

D. For preconditions and postconditions with logically and'd and or'd clauses, provide test cases that fully exercise each clause of the logic.
   1. Provide an input value that makes each clause of the and/or logic both true and false.
   2. This means 2^n test cases, where n is the number of logical terms.

E. For methods that take multiple inputs, test different combinations of inputs.
   1. For example, suppose we use the preceding rules to establish that
      a. test values for input *x* are *a*, *b*, and *c*
      b. test values for input *y* are *d*, *e*, and *f*
   2. The pairwise combination of test cases involve providing input value parts *[a,d]*, *[a,e]*, *[a,f]*, *[b,d]*, *[b,e]*, *[b,f]*, *[c,d]*, *[c,e]*, and *[c,f]*.
   3. For methods of three or more inputs, combinations are done on all input pairs, rather than across all three or more variables.
   4. Research has shown that pairwise combination can be a very effective way of choose inputs and it is far less combinatorially explosive than all possible combinations of three or more inputs.
   5. See for example www.pairwise.org/

F. For classes that define some form of collection:
   1. Test all operations with an empty collection.
   2. Test all operations with a collection containing exactly one element and exactly two elements.
   3. Add a substantial number of elements, confirming the state of collection after each addition.

4. Delete each element, confirming state of collection after each delete.
5. Repeat addition/deletion sequence two more times.
6. Stress test by adding and deleting from a collection of a size that is an order of magnitude greater than that ever expected to be used in production.

XVI. **Function paths**

A. A path is defined in terms of control flow through the logic of a method body.

B. Each branching control construct defines a path separation point.

C. By drawing the control-flow graph (i.e., flow chart) of a method, its paths are clearly exposed.

D. To ensure full path coverage, each path is labeled with a number, so it can be referenced in white box tests.

XVII. **White box testing heuristics**

A. Provide inputs that exercise each method path at least once.

B. For loops
   1. provide inputs that exercise the loop zero times (if appropriate),
   2. one time
   3. two times
   4. a substantial number of times
   5. the maximum number of times (if appropriate).

C. Provide inputs that can reveal flaws in the implementation of a particular algorithm, such as:
   1. particular operation sequences
   2. inputs of a particular size or range
   3. inputs that may cause overflow, underflow, or other abnormal behavior
   4. inputs that test well-known problem areas in particular algorithm

XVIII. **Testing Implementation -- the anatomy of a unit test method.**

A. ***Class and method under test:***

```
class X {

  // Method under test
  public Y m(A a, B b, C c) { ... }

  // Data field inputs
  I i;
  J j;

  // Data field output
  Z z;

}
```

B. ***Testing class and method:***

```
class XTest {
  public void testM() {

    // Set up
    X x = new X(...);
    ...

    // Invoke
    Y y = m(aVAlue, bValue, cValue);
```

```
    // Validate
    assertEqual(y, expectedY);
  }
}
```

C. The common core for a unit testing method is the same for all test implementation frameworks:
   1. ***Setup*** -- set up the inputs necessary to run a test
   2. ***Invoke*** -- invoke the method under test and acquire its actual output
   3. ***Validate*** -- validate that the actual output equals the expected output

D. Summary of where test specification and planning fits in:
   1. The `javadoc` comment for the method under test contains the Spest spec that specifies what must be true for inputs and outputs used in the tests.
   2. The `javadoc` comment for the testing class specifies the major phases of the testing, including the order in which the unit tests are executed.
   3. The `javadoc` comment for the testing method defines the unit test plan in tabular form; the plan has inputs, expected outputs, and remarks for each test case.

XIX. **A testing example using TestNG.**

A. TestNG is the recommended functional testing framework for 307.
   1. The "NG" in the name stands for "Next Generation", indicating it is a evolution of earlier testing frameworks, such as JUnit.
   2. Overall, it is very similar to JUnit, but has some features that improve on what is offered in JUnit.
   3. If your team has members who are familiar with JUnit, or some comparable testing framework, your team may use that instead of TestNG.
   4. To be used in 307, the requirements for a testing framework are these:
      a. It must support method/function-level unit testing.
      b. It must support class-level inter-method testing.
      c. It must support regression testing.

B. There is a very good how-to document for TestNG at http://testng.org/doc/documentation-main.html

C. Examples of using TestNG to implement class and unit tests are in the 307 Milestone 8 example for the Schedule Test

D. We'll go over these examples in class, in particular the example for Calendar Tool scheduling:
   • the Schedule.java model class
   • and its companion ScheduleTest.java testing class.
   • a Makefile that builds the tests
   • the simple TestNG configuration file that defines the testing components
   • the command-line execution script that runs the compiled tests

E. ***Important Note:***: For 307 Milestone 8, you need to implement three unit tests per team member, but the do not need to execute; test execution will be required for the final project deliverable.

XX. **Reconciling path coverage with purely black box tests.**

A. In CSC 307, we will use a purely black box testing style.

B. To ensure that all paths are covered, black box tests can be executed under the control of a *path coverage analyzer* (though we will not use such an analyzer in 307).

C. If the analyzer reports one or more paths not being covered, the coverage results are analyzed to see if new black box tests cases need to be added.
   1. When uncovered paths contain useless or dead code, the code can be removed and no further test cases are required.
   2. When uncovered paths are legitimate code, new test cases are added to the black box tests to ensure full path coverage.

D. A complete "grey box" test plan can have an additional column that indicates the path each black box test case covers, as in:

Test No.	Inputs	Expected Output	Remarks	Path
i	parm 1= ... parm m =	ref parm 1 = ... ref parm n =		*p*

where *p* is the number of the method path covered by the test case *i*.

XXI. **Specifying large inputs and outputs in functional tests**

A. For collection classes, inputs and outputs can grow large.

B. For convenience, such inputs and outputs can be specified as file data, instead of the result of calling a series of constructor methods in the context of a class test.

C. When external test data files are used, they can be referred to in test plans and used during test execution.

XXII. **Test drivers for test execution**

A. Once a test suite is defined, it must be executed.

B. To automate the testing process, and ensure that it is repeatable, a *test driver* is written as a stand-alone program.

1. The test driver executes all tests defined in the system test plan.
2. It records all results in an orderly manner, suitable for human inspection.
3. The test driver also provides a *test result differencer* that compares the results of successive test runs and summarizes differences.

C. For 307, this process is automated in a Makefile, as exemplified in

```
unix3:~gfisher/work/calendar/testing/implementation/source/java/Makefile
```

D. To perform tests initially, before all tests are executed via the Makefile, a symbolic debugger such as jdb can be used to execute individual methods.

XXIII. **Testing concrete UIs**

A. With a UI toolkit such as Swing, concrete UI tests are performed in the same basic manner as other functional tests.

B. User input, such as button pressing, is simulated by calling the interface method that is associated with the particular form of input, e.g., SomeButtonListener.actionPerformed.

C. Outputs that represent screen contents are validated initially by human inspection of the screen.

D. Ultimately, some machine-readable form of the screen output must be used to compare test results mechanically.

E. Note that we will NOT do this level of testing in 307, but rather test the GUIs via human interaction.

XXIV. **Unit testing is a "dress rehearsal" for integration testing.**

A. One might think if we do a really thorough job of method and class tests, integration should not reveal any further errors.

B. We know from experience that integration often does reveal additional flaws.

1. In this sense, failures of integration testing can be viewed as unit test failures.
2. That is, a flaw revealed by an integration test indicates an incompleteness of the test cases for some individual method.
3. The flaw is remedied by updating of the appropriate method test plan.

C. In so doing, individual tests become stronger.

XXV. **Testing models with large process data requirements.**

A. Suppose we have the following

```
class SomeModestModel {

    public void doSomeModelThing(String name) {
       ...
       hdb.doSomeProcessThing(...);
       ...
    }

    protected HumongousDatabase hdb;

}

class HumongousDatabase {

    public void doSomeProcessThing(...) {
       ...
    }

}
```

B. In such cases, it may be quite time consuming to implement a stub for the method HumongousDatabase.doSomeProcessThing.

C. This is a place where bottom-up testing is appropriate.

XXVI. **On really bright coders who don't need to do systematic testing.**

A. There are a few of these floating around at various institutions.

B. They do informally what mere mortals need to do in a more systematic way.

C. Ultimately, even the brightest hack will not be able to do all testing informally.

D. As programs are built in larger teams, no single person can know enough about the entire system to test it alone.

E. Therefore, team-constructed software must be team tested, in a systematic manner.

XXVII. **Other testing terminology**

A. The testing oracle

1. A test oracle is someone(thing) who(that) knows the correct answer to a test case.
2. The oracle is used in test plan generation to define expected results.
3. The oracle is also used to analyze incorrect test results.
4. For the style of development we have used in CSC 307, the oracle is defined by human interpretation of the requirements specification.
   a. When using a formal specification such as Spest, the oracle for a method is defined precisely as the method's postcondition.
   b.
5. When building a truly experimental piece of code for which the result is not yet known, specification-based oracle definition may not always be possible.
   a. These are cases such as artificial intelligence systems where the code is designed to tell us something we don't already know the answer to.
   b. To test such systems requires some initial prototype development, inspection of the results, and then definition of the tests.

B. Regression testing

1. This is the name given to the style of testing that runs all tests in a suite whenever any change is made to any part of the system.
2. Typically full regression tests are run at release points for the system.

3. There is ongoing research aimed at "smart" regression testing, where not all tests need to be run if it can be proved that a given change cannot possibly affect certain areas of the system.

C. Mutation testing
   1. This is a means to test the tests.
   2. The strategy is to *mutate* a program and then rerun its tests.
   3. For example, suppose an if statement coded as "if (x < y)" is mutated to "if (x >= y)".
   4. When such a mutation is made and a previously successful set of tests are run, the tests should fail in the places where the mutated code produces an incorrect result.
   5. If a set of previously successful tests do not fail on a mutated program, then one of two possibilities exists:
      a. The tests are too weak to detect a failure that should have been tested, in which case the tests need to be strengthened.
      b. The mutated section of code was "dead" in that it did not compute a meaningful result, in which case the code should be removed.
   6. Generally, the first of these to possibilities is the case.
   7. Mutation testing can be used systematically in such a way that mutations are made in some non-random fashion.
      a. Such systematic mutation provides a measure of testing effectiveness.
      b. This measure can be used to test the effectiveness of different testing strategies.

XXVIII. **Testing directory structure**

A. Figure 1 shows the details of the testing directory structure in the context of a normal project directory (without package subdirectories).

B. The contents of the testing subdirectories are shown in Table 2.

C. In the table, the variable *$PLATFORM* refers to the one or more subdirectories that contain platform-specific testing files (e.g., JVM, INTEL).

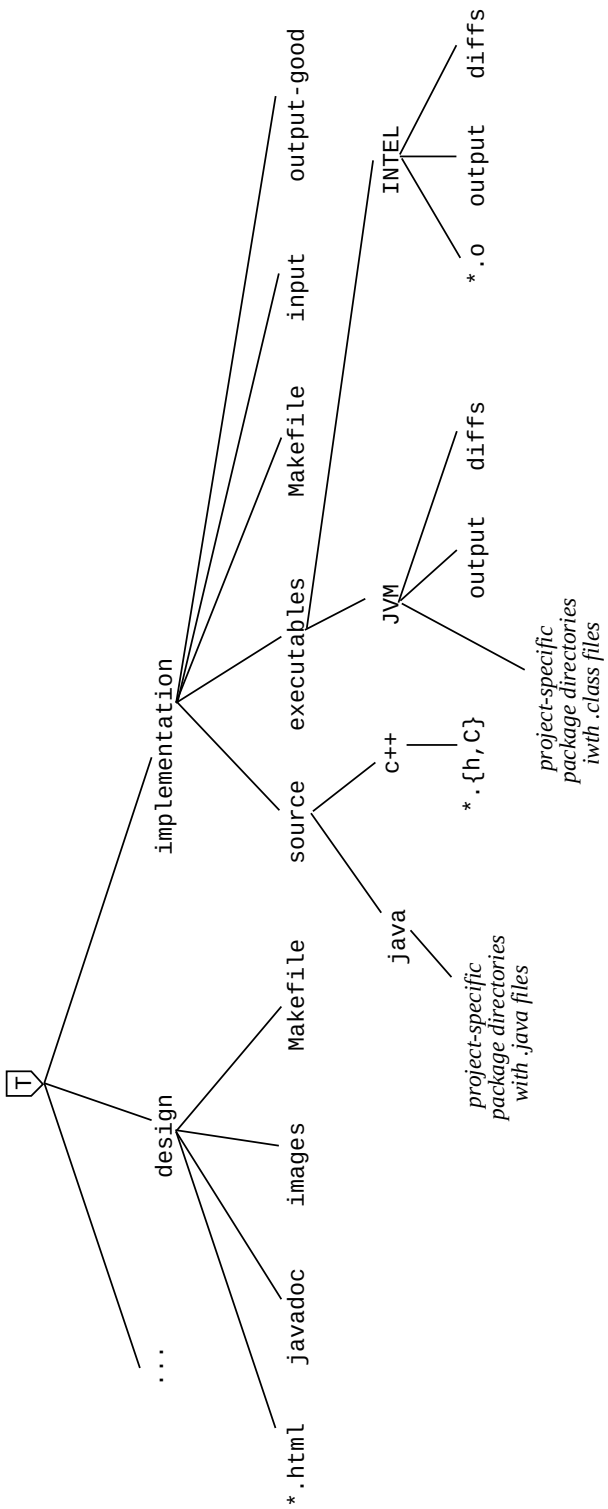

Figure 1: Testing directory structure.

Directory or File	Description
`*Test.java`	Implementation of class testing plans. Per the project testing methodology, each testing class is a subclass of the design/implementation class that it tests.
`input`	Test data input files used by test classes. These files contain large input data values, as necessary. This subdirectory is empty in cases where testing is performed entirely programatically, i.e., the testing classes construct all test input data dynamically within the test methods, rather than inputing from test data files.
`output-good`	Output results from the last good run of the tests. These are results that have been confirmed to be correct. Note that these good results are platform independent. I.e., the correct results should be the same across all platforms.
`output-prev-good`	Previous good results, in case current results were erroneously confirmed to be good. This directory is superfluous if version control of test results is properly employed. However, this directory remains as a backup to avoid nasty data loss in case version control has not been kept up to date.
$PLATFORM`/output`	Current platform-specific output results. These are the results produced by issuing a make command in a platform-specific directory. Note that current results are maintained separately in each platform-specific subdirectory. This allows for the case that current testing results differ across platforms.
$PLATFORM`/diffs`	Differences between current and good results.
$PLATFORM`/Makefile`	Makefile to compile tests, execute tests, and difference current results with good results.
$PLATFORM`/.make*`	Shell scripts called from the Makefile to perform specific testing tasks.
$PLATFORM`/.../*.class`	Test implementation object files.

Table 2: Test file and directory descriptions.