CSC 307 Lecture Notes Week 8

ormal Specs in Testing
Intro to Testing Techniques

I. Milestones 7-8

1. Due Mon 23 November
2. Refined model/view designs
3. High-level testing design
4. Approx 75% of implementation operational
5. Data validation for 3 model methods
6. Unit tests for 3 model methods

II. The Testing "Big Picture"

-- on the board for today ...

III. Refining method specs for testing

A. Testing requires that we know exactly what constitutes valid versus invalid inputs.

1. Pre- and postconds answer this question.

2. Used to inform unit test development.

Overview, cont’d

B. Recap of what pre/postconds mean.

1. ***Precondition*** is one boolean expression that is true before method executes.

2. ***Postcondition*** is one boolean expression that is true after method completes.

IV. Formal specs used in testing

A. Formal method test consists of:

1. Inputs within legal ranges, expected output

2. Inputs outside legal ranges, expected output

3. Inputs on boundaries, expected output

Formal specs in testing, cont’d

B. Preconds used to determine inputs.

C. Postconds used to determine expected output

V. Formal specs used in verification

A. Programs can be formally verified, that is *proved* correct.

B. A formal spec is necewsary to do this.

C. We’ll discuss further during last week of class.

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

A. Precond failure means an op is "undefined".

1. For abstract spec, this is enough.

2. At imple’n level, precond must be dealt with more concretely.

3. Two basic approaches.

Precond enforcement, cont’d

B. *Approach 1:* Precond is guaranteed true, before method call.

1. This is "***programming by contract***".
2. Precond enforced by callers.
3. Verified or checked at *calling* site.
4. Bottom line -- called method assumes its precond is always true.

Precond enforcement, cont'd

C. *Approach 2:* Precond is checked by method being called.

1. This is ***"Defensive programming"***.

2. Method includes logic to enforce its own precondition.

3. Enforcement can:

Precond enforcement, cont’d

a. Assert unconditional failure.

b. Return "null" value.

c. Output error report.

d. Throw an exception.

Precond enforcement, cont’d

D. In Model/View comm’n, we use exception handling approach.

E. We’ll discuss further next week.

VII. **General concepts of functional testing.**

A. Components are independently testable.

B. Testing is thorough and systematic.

C. Testing is repeatable.

VIII. **Overall system testing styles**

A. Top-down

1. Top-level methods tested first.

2. "Stubs" written for lower-level methods.

Testing styles, cont’d

B. Bottom-up

1. Lower-level methods tested first.

2. Function "drivers" written for upper-level methods.

Testing styles, cont’d

C. Object-oriented

1. All methods for a particular class are tested.

2. Stubs and drivers written as necessary.

Testing styles, cont’d

D. Hybrid

1. A combination of top-down, bottom-up, and object-oriented testing is employed.

2. This is a good practical approach.

Testing styles, cont’d

E. Big-bang

1. All compiled in one huge executable.

2. Cross fingers and run it.

3. When big bang fizzles,
   enter debugger and hack.

IX. Independently testable designs

A. Modular interfaces designed thoroughly.

1. Don’t fudge on method signatures, pre/post logic.

2. Be clear on public and protected.

B. Write *stubs* and *drivers* as necessary.

X. General approaches to testing

A. Black box testing

1. Each method viewed as black box.

2. Function tested using spec only.

General approaches, cont’d

B. White-box testing

1. Testing based on method code.

2. Inputs that fully exercise code logic.

3. Each control path is exercised at least once by some test.

General approaches, cont'd

C. Runtime precond enforcement

1. Code added to methods to enforce preconds at runtime.

2. E.g., input range checking.

3. Function returns (or throws) error if condition is not met.

General approaches, cont'd

D. Formal verification

1. Pre/post conds treated as math'l theorems.

2. Function body treated as math'l formula.

3. Verification entails proving precond implies postcond, *through* method body.

XI. Functional unit test details

A. List of *test cases* produced for each method.

B. This constitutes the *unit test plan*.

C. Tabular form:

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

Unit test details, cont’d

D. Note that

1. Must specify all input parameters and data fields.
2. Must specify all ref parms, return val, modified fields.
3. Not mentioned assumed "don’t care".

Unit test details, cont'd

E. One test plan for each method.

F. Unit test plans included in module test plan for complete class.

XII. Module, i.e., class testing

A. Write unit test plans for each method.

B. For class as whole, write *class test plan*.

C. Guidelines:

Class testing, cont’d

1. Start with unit tests for constructors.

2. Next, unit test other constructive methods.

3. Unit test selector methods.

4. Test certain method interleavings.

5. Stress test.

Class testing, cont'd

D. Use a test driver that:

1. executes each method test plan,

2. checks the results,

3. reports any erroneous results

Class testing, cont'd

E. Concrete milestone 8 examples:

```
caltool/testing/
    implementation/source/java/
        caltool/model/schedule/
            ScheduleTest.java

caltool/testing/
    implementation/source/java/
        caltool/model/caldb/
            UserCalendarTest.java
```

Class testing, cont’d

F. Java details

1. Each class *X* has companion testing class named *XTest*.

2. Test class is subclass of class it tests.

3. Each method *X.f* has a companion unit test method named *XTest.testF*.

Class testing, cont’d

4. Comment at top of test class describes the module test plan.

5. The comment for each unit test method describes unit test plan.

6. Unit test details coming up.

XIII. Integration testing

A. Once tested, modules are integrated.

B. Stubs replaced with actual methods.

C. Test plan for top-most method(s) rerun with integrated modules.

D. Continues until entire system is integrated.

Integration testing, cont’d

E. Concrete example:

```
caltool/testing
  implementation/source/java/caltool/
    integration-test-plan.html
```

1. Integrate `schedule` + `caldb`
2. Add `view` to `schedule+caldb`
3. Add `admin` to `schedule+view+caldb`
4. Integrate `caldb` + `caldb.server`
5. Add `caldb.server` to `schedule + ...`
6. Add `options` to `schedule + ...`
7. Add `file` to `schedule + ...`
8. Add `edit` `schedule + ...`
9. Add top-level `caltool` classes

XIV. Black box testing heuristics

A. Provide inputs where the precondition is true.

B. Provide inputs where the precondition is false.

Black box heuristics, cont’d

C. For data ranges:

1. Provide inputs below, within, above each precond range.

2. Provide inputs that produce outputs at bottom, within, at top of each postcond range.

Black box heuristics, cont’d

D. With and/or logic, provide test cases that fully exercise logic.

1. Provide an input that makes each clause both true and false.

2. This means 2^n test cases, where n is number of logical terms.

Black box heuristics, cont’d

E. For collection classes:

1. Test empty collection.
2. Test with one, two elements.
3. Add substantial number of elements.
4. Delete each element.
5. Repeat add/del sequence.
6. Stress test with order of magnitude greater than expected size.

XV. Function paths

A. Control flow through method body.

B. Branching defines path separation point.

C. Flow chart show paths clearly.

D. Each path is labeled with a number.

XVI. **White box testing heuristics**

A. Exercise each path at least once.

B. For loops:

1. zero times (if appropriate),
2. one time
3. two times
4. a substantial number of times
5. max number times (if appro)

White box heuristics, cont'd

C. Provide inputs to reveal imple'n flaws:

1. particular operation sequences
2. inputs of particular size or range
3. inputs that may cause overflow, underflow, other abnormal behavior
4. inputs that test well-known problems in algorithm

XVII. 307 Testing Approach

A. Write black-box tests from spec.

B. Verify white-box cases using coverage tool

XVIII. Testing Implementation -- core of unit a testing method

1. ***Setup*** -- set up inputs necessary to run test

2. ***Invoke*** -- invoke the method *under test* and acquire its output

3. ***Validate*** -- validate that actual output equals expected output

XIX. Testing Implementation -- detailed 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 inputs
    X x = new X(...);
    aValue=...; bValue=...; cValue=...;

    // Invoke method
    Y y = m(aValue, bValue, cValue);

    // Validate output
    expectedY=...; expectedZ=...;
    assertEqual(y, expectedY);
    assertEqual(z, expectedZ);
  }
}
```

C. Summary of test artifact locations:

1. `javadoc` comment for method under test has
   JML spec

2. `javadoc` comment for testing method has
   unit test plan

3. `javadoc` comment for testing class has
   class test plan

Test artifact locations, cont’d

4. code in method under test
   does useful work

5. code in testing method
   calls method under test, checks results

6. code in testing class has
   testing methods, supporting data fields

XX. A testing example using TestNG.

A. TestNG recommended for 307.

1. "NG" stands for "Next Generation".

2. Very similar to JUnit, interoperable.

3. You ***may use*** JUnit, or comparable.

TestNG or equivalent, cont’d

4. testing framework requirements:

   a. must support method-level unit testing

   b. must support class-level testing

   c. Must support regression testing.

TestNG or equivalent, cont’d

B. Good TestNG how-to doc linked from `307/examples/milestone8`

C. Also TestNG usage examples.

D. We’ll go over these examples now.

TestNG or equivalent, cont'd

- ***`Schedule.java`*** model class
- ***`ScheduleTest.java`*** testing class
- *plus these support files (for Milestone 10):*
    - Makefile to build and run
    - simple TestNG config file
    - command-line execution script

TestNG or equivalent, cont'd

E. ***Important Note:***
For Milestone 8, you need to

- implement three unit tests per team member
- they do ***not*** need to execute for M8
- test execution required for the final project

XXI. Reconciling path coverage

A. Write purely black box tests.

B. To ensure coverage, execute under path coverage analyzer.

C. If analyzer reports paths not being covered, strengthen black box tests.

Reconciling path coverage

1. Uncovered paths may contain useless code.

2. When legitimate code, add new black box test cases.

D. "Grey box" test plan can have path column:

Reconciling path coverage

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

XXII. Large inputs and outputs

A. For collections classes, i/o can grow large.

B. Can be specified as file data.

C. Referred to in test plans.

XXIII. Test drivers

A. Once defined, test must be executed.

B. *Test driver* written as stand-alone program.

1. Executes all tests.
2. Records results.
3. Provides *result differencer*.

Test drivers, con’td

C. Makefile-based example in

```
caltool/testing/
    implementation/source/java/Makefile
```

Template in

```
classes/307/lib/unix3-Makefiles/
    testing-Makefile
```

D. Perform tests initially using debugger.

XXIV. Testing concrete UIs

A. Performed in the same basic manner.

B. User input is simulated.

C. Output screens validated initially by human.

D. Machine-readable form of screen to compare results mechanically.

Testing concrete UIs, cont'd

E. We'll look at mechanized GUI testing briefly next week.

F. No time to implement it in 307.

XXV. Unit test is "dress rehearsal" for integration testing ...

A. Integration *"should not"* reveal further errors.

B. From experience, it often does.

C. In so doing, individual tests become stronger.

XXVI. Testing with large data.

A. Suppose we have

```
class SomeModestModel {
    ...
}

class HumongousDatabase {
    ...
}
```

Large-data requirements, cont’d

B. May be time consuming to implement stub.

C. Bottom-up testing is appropriate.

XXVII. **Other testing terminology**

A. The testing oracle.

1. Someone(thing) who knows correct answers.
2. Used to define expected results.
3. Also used to analyze incorrect test results.
4. In 307, oracle defined as implementation of method postcondition.

Terminology, cont’d

5. When building truly experimental code, spec-based oracle may not be possible.

   a. E.g., AI systems.

   b. Need initial prototype development.

Terminology, cont’d

B. Regression testing

1. Run ***all*** tests whenever any change is made.

2. Must happen before release.

3. Ideally happens much more often.

4. Ongoing research on "smart" regression.

Terminology, cont’d

C. Mutation testing

1. It’s a way to test the tests.

2. Strategy -- ***mutate*** program, then rerun tests.

3. E.g., "if (x < y)" is mutated to "if (x >= y)".

Terminology, cont’d

4. With such mutation, tests should fail where the mutated code produces bad result.

5. If previously successful tests do ***not*** fail, ... ?

Terminology, cont’d

a. The tests are too weak and need to be ***strengthened***.

b. The mutated section of code was "dead" and ***should be removed***.

Terminology, cont’d

6. Generally, the first of these is the case.

7. Mutation can be used systematically to:

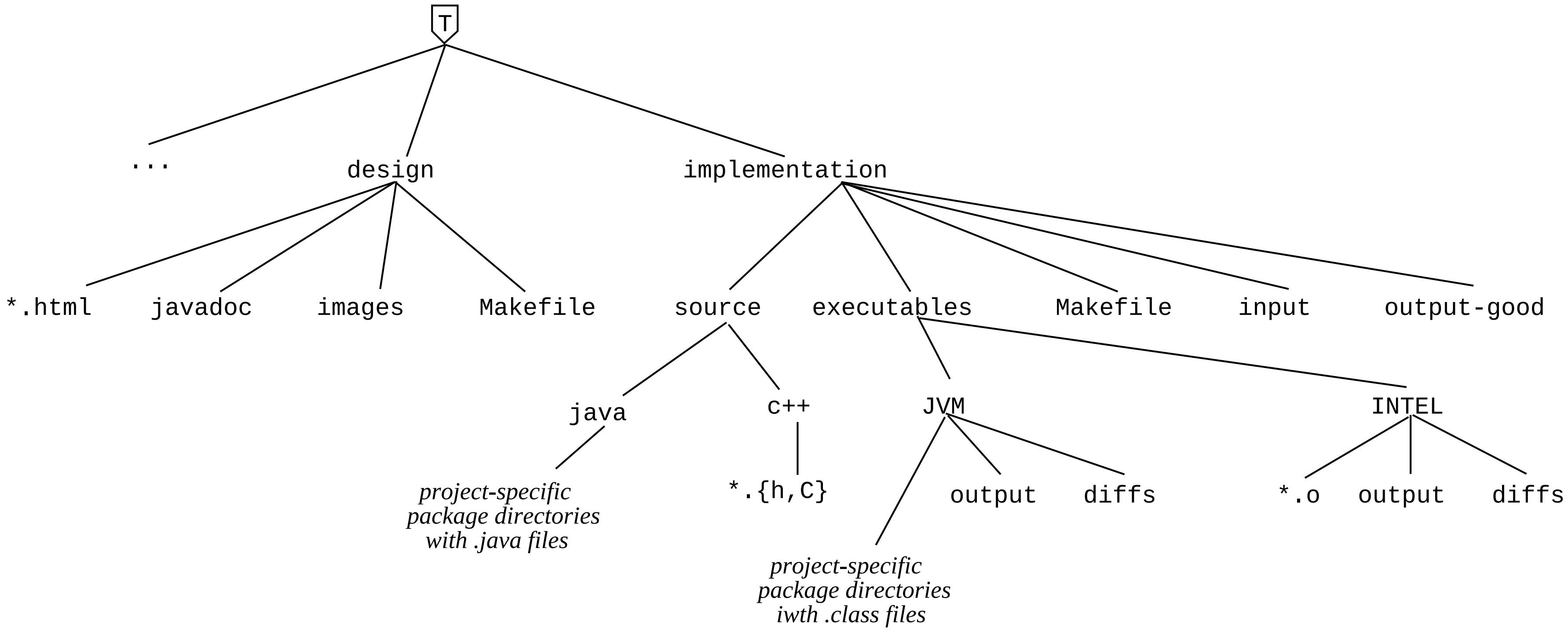
Terminology, cont'd

a. Provide measure of testing effectiveness.

b. Compare different testing strategies.

XXVIII. Testing directory structure

A. Figure 1 in notes ...



T

...

design

implementation

*.html

javadoc

images

Makefile

source

executables

Makefile

input

output-good

java

c++

JVM

INTEL

project-specific
package directories
with .java files

*.{h,C}

output

diffs

*.o

output

diffs

project-specific
package directories
iwth .class files

Test dir structure, cont’d

B. Contents of testing subdirs:

Directory or File	**Description**
`*Test.java`	Implementation of class testing plans.
`input`	Test data input files used by test classes.
`output-good`	Output results from last good run of the tests.
`output-prev-good`	Previous good results, in case current results were erroneously confirmed to be good.
$PLATFORM/`output`	Current platform-specific output results.
$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.