CSC 309 Lecture Notes Week 4

Formal Specs in Testing
Intro to Testing Techniques

I. Deriving and refining method specs.

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.

II. 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

III. Formal specs in formal verification

A. To verify formally, two specs needed:

1. formal spec of given program

2. formal spec of programming language

Formal specs in verification, cont’d

B. Program spec is "entry ticket" to verification.

C. Details in later lectures.

IV. **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 will discuss exception handling further in upcoming lectures.

V. Details of deriving method specs.

A. Start with Spest specs for 308.

B. Update and expand based on design refinements done in 309.

C. For some details, see M3 example.

Now on to General System Testing Techniques

VI. General Concepts

VI. **General Concepts**

A. Components are independently testable.

VI. **General Concepts**

A. Components are independently testable.

B. Testing is thorough and systematic.

VI. General Concepts

A. Components are independently testable.

B. Testing is thorough and systematic.

C. Testing is repeatable.

VII. Overall system testing styles

VII. Overall system testing styles

A. Top-down

VII. Overall system testing styles

A. Top-down

1. Top-level methods tested first.

VII. **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

Testing styles, cont’d

B. Bottom-up

1. Lower-level methods tested first.

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

Testing styles, cont’d

C. Object-oriented

   1. Methods for particular class are tested.

Testing styles, cont’d

C. Object-oriented

   1. Methods for particular class are tested.

   2. Stubs and drivers written as necessary.

Testing styles, cont’d

D. Hybrid

Testing styles, cont’d

D. Hybrid

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

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

Testing styles, cont’d

E. Big-bang

   1. All compiled in one huge executable.

Testing styles, cont’d

E. Big-bang

   1. All compiled in one huge executable.

   2. Cross fingers and run it.

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.

VIII. 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.

Independently testable designs, cont’d

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

Independently testable designs, cont’d

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

   1. A *stub* is also known as a *mock*.

Independently testable designs, cont'd

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

1. A *stub* is also known as a *mock*.

2. *Drivers* generally supplied by testing framework, as part of its typical use.

IX. General approaches to testing

IX. General approaches to testing

A. Black box testing

IX. General approaches to testing

A. Black box testing

1. Each method viewed as black box.

IX. 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

General approaches, cont’d

B. White-box testing

   1. Testing based on method code.

General approaches, cont’d

B. White-box testing

1. Testing based on method code.

2. Inputs that fully exercise code logic.

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 pre/postcond enforcement

General approaches, cont’d

C. Runtime pre/postcond enforcement

   1. Code added to methods to enforce pre/postconds at runtime.

General approaches, cont’d

C. Runtime pre/postcond enforcement

1. Code added to methods to enforce pre/postconds at runtime.
2. E.g., input range checking.

General approaches, cont’d

C. Runtime pre/postcond enforcement

1. Code added to methods to enforce pre/postconds at runtime.
2. E.g., input range checking.
3. Function returns (or throws) error if condition is not met.

General approaches, cont’d

C. Runtime pre/postcond enforcement

1. Code added to methods to enforce pre/postconds at runtime.
2. E.g., input range checking.
3. Function returns (or throws) error if condition is not met.
4. Crudely, function could use *assert*.

General approaches, cont’d

D. Formal verification

General approaches, cont’d

D. Formal verification

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

General approaches, cont’d

D. Formal verification

1. Pre/post conds treated as math’l theorems.
2. Function body treated as math’l formula.

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.

X. Functional unit test details

X. Functional unit test details

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

X. **Functional unit test details**

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

B. This constitutes the *unit test plan*.

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

Unit test details, cont’d

C. Note that

Unit test details, cont’d

C. Note that

1. Must specify all input parameters and data fields.

Unit test details, cont'd

C. Note that

1. Must specify all input parameters and data fields.

2. Must specify all ref parms, return val, modified fields.

Unit test details, cont’d

C. 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

D. One test plan for each method.

Unit test details, cont’d

D. One test plan for each method.

E. Unit test plans included a javadoc comments.

XI. Module, i.e., class testing

XI. Module, i.e., class testing

A. Write unit test plans for each method.

XI. Module, i.e., class testing

A. Write unit test plans for each method.

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

XI. 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.

Class testing, cont’d

1. Start with unit tests for constructors.

2. Next, unit test other constructive methods.

Class testing, cont'd

1. Start with unit tests for constructors.

2. Next, unit test other constructive methods.

3. Unit test selector methods.

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.

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/write a test driver that:

Class testing, cont'd

D. Use/write a test driver that:

   1. executes each method test plan,

Class testing, cont’d

D. Use/write a test driver that:

1. executes each method test plan,
2. compares actual with expected output,

Class testing, cont'd

D. Use/write a test driver that:

1. executes each method test plan,

2. compares actual with expected output,

3. reports the differences, if any,

Class testing, cont'd

D. Use/write a test driver that:

1. executes each method test plan,
2. compares actual with expected output,
3. reports the differences, if any,
4. optionally records output results.

Class testing, cont’d

E. Concrete examples:

```
projects/work/calendar/testing/
    implementation/source/java/
        caltool/schedule/
            ScheduleTest.java

projects/work/calendar/testing/
    implementation/source/java/
        caltool/caldb/
            UserCalendarTest.java
```

Class testing, cont’d

F. Java details

Class testing, cont'd

F. Java details

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

Class testing, cont’d

F. Java details

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

2. Test class may extend class it tests.

Class testing, cont'd

F. Java details

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

2. Test class may extend class it tests.

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

Class testing, cont'd

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

Class testing, cont’d

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

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

Class testing, cont'd

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

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

5. Each tested class implements `dump` method for dumping test values as String.

XI. Integration testing

XI. Integration testing

A. Once tested, modules are integrated.

XI. **Integration testing**

A. Once tested, modules are integrated.

B. Stubs replaced with actual methods.

XI. 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.

XI. 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.

D. **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:

```
projects/work/calendar/testing/
  implementation/
    integration-test-plan.html
```

1. Integrate `schedule` + `caldb`

1. Integrate `schedule` + `caldb`
2. Add `view` to `schedule+caldb`

1. Integrate `schedule` + `caldb`
2. Add `view` to `schedule+caldb`
3. Add `admin` to `schedule+view+caldb`

1. Integrate `schedule` + `caldb`
2. Add `view` to `schedule+caldb`
3. Add `admin` to `schedule+view+caldb`
4. Integrate `caldb` + `caldb.server`

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 + ...`

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 + ...`

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 + ...`

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 + ...`

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

XII. Black box testing heuristics

XII. Black box testing heuristics

A. Provide inputs where the precondition is true, varying inputs to exercise precond logic.

XII. **Black box testing heuristics**

A. Provide inputs where the precondition is true, varying inputs to exercise precond logic.

B. Provide inputs where the precond is false, ***if not a by-contract method.***

Black box heuristics, cont’d

B. For data ranges:

Black box heuristics, cont’d

B. For data ranges:

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

Black box heuristics, cont'd

B. 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

Black box heuristics, cont’d

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

Black box heuristics, cont’d

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

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

Black box heuristics, cont’d

C. 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

D. Provide selected combinations of inputs.

Black box heuristics, cont’d

D. Provide selected combinations of inputs.

   1. Combinatorially explosive in general.

Black box heuristics, cont’d

D. Provide selected combinations of inputs.

1. Combinatorially explosive in general.
2. Pairwise combination is practical approach.

Black box heuristics, cont’d

D. Provide selected combinations of inputs.

1. Combinatorially explosive in general.
2. Pairwise combination is practical approach.
3. Used by Spest generator.

Black box heuristics, cont’d

D. Provide selected combinations of inputs.

1. Combinatorially explosive in general.
2. Pairwise combination is practical approach.
3. Used by Spest generator.
4. See pairwise.org

Black box heuristics, cont’d

E. For collection classes:

Black box heuristics, cont’d

E. For collection classes:

1. Test empty collection.

Black box heuristics, cont’d

E. For collection classes:

1. Test empty collection.
2. Test with one, two elements.

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.

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.

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.

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.

XIII. Function paths

A. Control flow through method body.

B. Branching defines path separation point.

C. An old-school ***flow chart*** show paths clearly.

D. Each path is labeled with a number.

XIV. 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

XV. 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 or dead code.

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

D. Complete "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*

XVI. 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.

XVII. 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. Automated in

```
projects/work/calendar/testing/
    implementation/source/java/Makefile
```

Template in

```
classes/309/lib/csl-Makefiles/
    testing-Makefile
```

D. Perform tests initially using debugger.

XVIII. 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 in a couple weeks.

XIX. 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.

XX. Testing with large data.

A. Suppose we have

```
class SomeModestModel {
    ...
}

class HumongousDatabase {
    ...
}
```

Large-data requirements, cont’d

B. Modest amount of test data can be built programmatically, i.e., by calling constructive methods

C. Large amount of (persistent) data can be stored external from program, built by external means if appropriate.

D. The latter are external ***test fixtures***.

XXI. 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 CSC 309, oracle is defined by implementation of Spest 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:

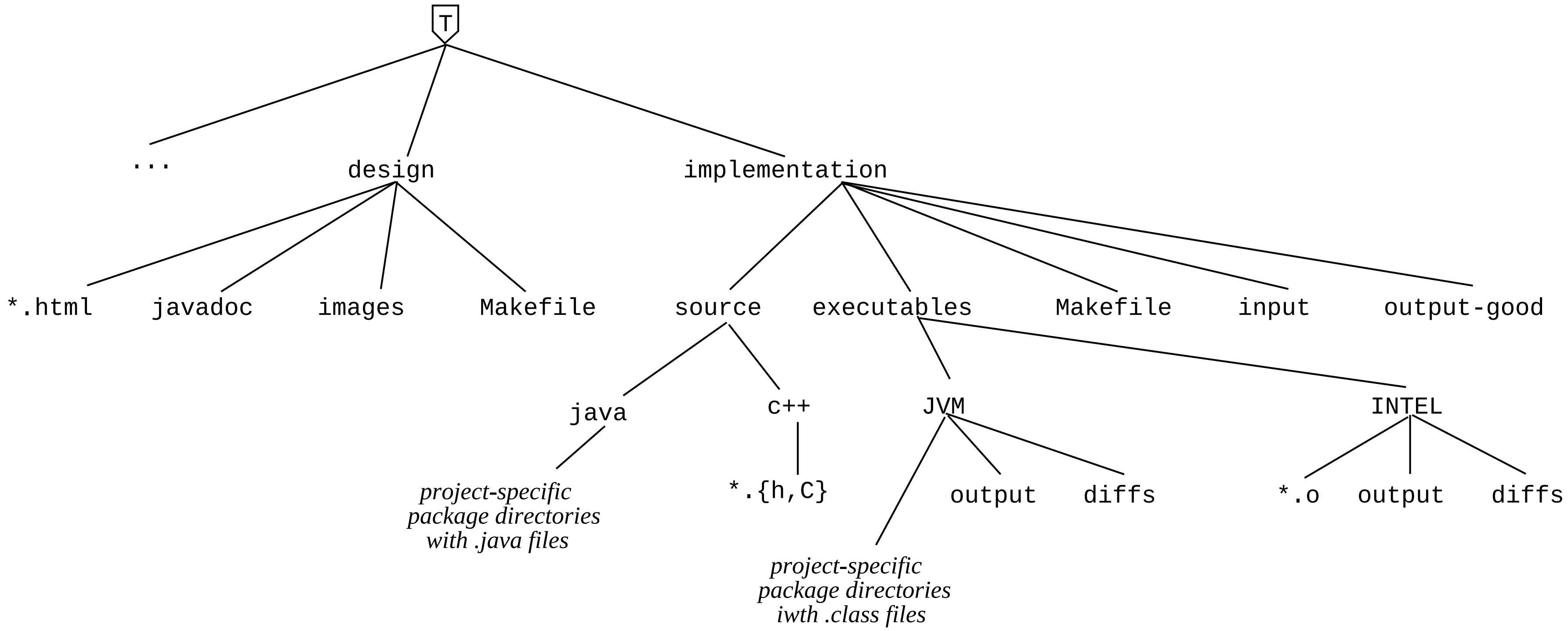
Terminology, cont’d

a. Provide measure of testing effectiveness.

b. Compare different testing strategies.

XXII. 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.