

# Computer Science 331

## Introduction to Testing of Programs

Mike Jacobson

Department of Computer Science  
University of Calgary

Lecture #3-4

## Outline

- 1 Definitions
- 2 Principles
- 3 Stages and Types of Testing
  - Stages of Testing
  - Types of Tests
- 4 Implementation and Evaluation
- 5 Debugging
- 6 References

## Definitions

### What is Testing?

#### Testing:

- is the process of examining or running a program in order to find errors.
- provides *some* evidence that software meets its specifications

#### A **Test Plan** (or “Testing Strategy”)...

- is a systematic approach to testing software
- includes
  - deciding how software will be tested
  - deciding when tests will occur
  - deciding who will do the testing
  - deciding what test data will be used — and what the expected output should be for each input

## Definitions

### Well-Designed Test Plans

Four main characteristics of well-designed test plans:

- 
- 
- 
-

## What is Defensive Programming?

### Defensive Programming...

- is a style of programming intended to ensure that software continues to function (or, at least, does not cause harm) in spite of unforeseeable use of the software
- includes the use of code that detects unexpected or invalid input data values — one way of “preparing for testing” as you write your code

See

[http://en.wikipedia.org/wiki/Defensive\\_programming](http://en.wikipedia.org/wiki/Defensive_programming)

for more information about defensive programming.

One advantage of developing a test plan *early* is that it makes defensive programming easier.

## Is This the Objective of Testing?

Assuming that we are testing *complex* software including an extremely *large number of lines of code*

**Q:** Do we test in order to *prove that a program is correct*?

**A:** No!

### Explanation:

- 
- 

## What is Debugging?

**Debugging** is a methodical process of finding and removing defects in a program.

### General process:

- Recognize that a bug exists (eg. ideally, via testing)
- Isolate the *source* of the bug
- Identify the *cause* of the bug
- Determine a *fix* for the bug
- Apply the fix and *test it*

### A Common Error in Debugging:

- 

## More About the Objective of Testing

### Objective of Testing:

- We test in order to prove that a program is *incorrect*!

### Explanation:

- It is *extremely* unlikely that long and complex software is free of errors
- It is generally cheaper and easier to correct an error if it is detected *early* in software development
- Adversarial mindset (goal is to *try* to make the program fail) improves chances of locating errors

## Who Should Test Your Software?

It is frequently a good idea to have someone else test the software you have designed and implemented (if possible!).

### Explanation:

- We all have “blind spots.” Frequently, other people can more easily see problems with our work that we don’t notice ourselves
- It is easy (and human) for us to be overly “protective” of our own work — we’d like to think it is perfect! This is not helpful, considering that “the goal of testing is to prove that your software is incorrect”

## Why Prove Correctness *and* Test?

We need to prove correctness (or, at least, know about a proof of correctness) because...

- you can’t “test in” quality or use testing to repair a method based on an incorrect algorithm — or debug code effectively unless you know what it is supposed to do

We need to test because...

- proofs of correctness tend to be “sketched” instead of developed in detail, or skipped altogether, if correctness seems “obvious”
- sometimes the *proofs* are faulty... and they tend to rely on idealistic and unrealistic assumptions (e.g.: arithmetic is exact); testing provides a “reality check”
- a variety of errors can be introduced during the coding phase, even if you are starting with an algorithm that really is “correct.”

## A Limitation of Testing

You cannot use testing to improve **software quality**, ie,

- readability
- complexity
- maintainability
- efficiency

**Q:** When do we try to achieve these desirable properties?

**A:**

## Principles of Testing

Remember what kind of software we are testing (large, complex)!

Summary:

- A test *succeeds* if it finds an error.
- It is (almost always) impossible to test *completely*.
- Development of a test plan can — and should — begin *early on* in software development.
- Ideally, you should not test your own program.
- Testing can be effective in detecting and removing (some) errors from well-designed software. It is generally *not* effective if used to improve low-quality software.  
If you find lots of errors, there are probably lots more!
- Testing takes time and hard work but is worth it!

## Unit Testing

During **Unit Testing** ...

- each “module” (class or function) is tested individually.
- goal is to show that each module meets its specifications
- ignores interaction between modules

This is the *first* stage of software testing

- later stages consider groups of modules, and are simpler if we can be confident that each module works correctly by itself

Well-written unit tests serve as important *documentation*

- describes the *expected behaviour* of the module on a variety of inputs (ideally including both “valid” and “invalid” inputs)

## Integration Testing

**Integration Testing** ...

- is performed after unit testing.
- Individual modules (that separately seem to be acceptable) are combined to form and test progressively larger subsystems.
- Multiple methods of an object might be tested in combination as part of this process.

Overall idea — “building block” approach

- gradually add and test new modules to a tested base
- after testing the integration of a new module, it is added to the tested base and the process is repeated with a new module, until all have been included

## Regression Testing

**Regression Testing:**

- If an error is found and corrected then testing of the affected modules and subsystems should be **repeated**, to be sure no new errors were introduced!
- This is one reason why it is important to *document* tests — you may need to use them more than once!

**Note:** bugs can also be *reintroduced* via:

- poor revision control practices (eg. when two people work on the same code)
- inadequate documentation of testing (so that, eg., bug #1 gets reintroduced when recoding to eliminate bug #3)

## Validation Testing

**Validation (Acceptance) Testing** ...

- is performed after integration testing.
- Previous testing is generally conducted by software developers (possibly including testing specialists).
- Validation testing also involves potential users of the software (or current users, if an existing system is being changed or replaced).
- Idea is to test completed program using test cases and environments as close to (and as extreme as) input from actual users.

*This type of testing is beyond the scope of CPSC 331.*

## System Testing

### System Testing ...

- is performed after validation testing (if it is needed).
- Used when the software being developed is part of a large system with other components (possibly including other software as well as specialized hardware, people, etc... ). This larger system is tested.
- Analogous to integration testing, where the “module” to be integrated into a larger system is the entire *software* system (now being integrated into a system with other kinds of components)

*This type of testing is also beyond the scope of CPSC 331.*

## Static Testing

### Static Testing (structured walkthrough):

- involves examination of source code without execution.
- often first stage of *unit testing*
- is a “reality-check” on code before proceeding to more detailed or complicated testing

### Two types:

- Desk checking: read through code, look for errors
- Hand Executions: trace code execution on small inputs with known outputs by hand

### Support Tools:

- pencil, paper, time, patience, ...

## Dynamic Testing

### Dynamic Testing:

- tests the behaviour of a module or program during execution.

### Two types:

- **Black Box Testing** (also called **Functional Testing**)
- **White Box Testing** (also called **Structural Testing**)

*Both* black box and white box testing are useful for all phases of testing

## Black Box Testing

### Black Box Testing ...

- includes tests designed using *only* the problem specification (*not* the code)
- tests both *valid* and *invalid* input
- tests typical cases and *boundary conditions* (special, rarely-occurring cases)
- is useful for finding
  - incorrect or missing functions,
  - interface errors (involving functions),
  - interface errors for data structures or external data bases,
  - initialization and termination errors.
- is generally used in later testing states, but certainly *can* and *should* be used during unit testing too.

## Example

Consider an object's method with the following **signature**:

```
public void removeMe ( Object[] array );
```

and with

- **Pre-Condition:** input array is not null
- **Post-Condition:** input has been modified by a removal of the *first* instance of `this`, closing the gap and setting the last entry of the input to `null`, if `this` was found as an array entry; otherwise, the input is unchanged and a `NoSuchElementException` is thrown
- **Exceptions:**
  - `NoSuchElementException`
  - `NullPointerException`

## Example Test Cases

Example test case inputs for `x.removeMe()`:

## White Box Testing

Includes tests designed using the internal workings of a module (including source code).

- goal is to test every line of code and every execution path

Tests typically try to ensure that:

- every *statement* in code is executed in one or more tests
- each “if” and “else” branch of every *conditional* statement is tested
- each *loop* is iterated zero, one, several, and as many times as possible (if these situations are feasible)
- each *exit condition* causing a loop or function to terminate is executed
- all *exception handling* is tested

## Why White Box Testing is Useful

Use white box testing to test paths not covered by black box tests:

- parts of code (unit testing)
- paths/interfaces between units (integration testing)
- interactions between systems (system testing)

Two reasons why this is useful (may be more!):

- 1
- 2

## Important Note About Test Design

Tests must be designed *completely* before tests are carried out.

In particular, a test's *expected results* must be determined and *documented*, so that they are available for comparison with the values that are actually generated when a test is carried out.

The design and execution of tests can begin *before* coding and be carried out *during* and *after* coding:

- Black box tests can be designed using specifications of requirements *before* coding begins.
- Unit tests can be executed once individual modules are completed (and before others have).
- Integration tests can be carried out gradually, while coding continues, as well.

## Additional Code for Unit and Integration Testing

**Stub:** piece of code that simulates the activity of a missing component (that is called by whatever you are testing)

- could be simple as something that echoes the input it receives and prompts for, and returns, appropriate data to the module being tested
- could be as complex as an alternate (perhaps, resource-inefficient) fully functional implementation of another part of the system

**Driver:** piece of code that emulates a calling function (supplying test data to whatever you are testing and reporting test results)

## Test Harness

**Test Harness:** combination of a software *test engine* and a test *data repository*

- automates testings (running tests and monitoring results)
- since *it will often be necessary to repeat tests* the overhead associated with the use of this is generally worthwhile!

**Note:** You will be using a test harness (including the test engine JUnit) in this course.

## Write Your Code to Make Testing Easier

This is part of “defensive programming.”

- Document your code appropriately!
  - Include preconditions and postconditions for methods, including in javadoc comments for all public methods
  - Include loop invariants and loop variants for loops
  - Include assertions describing expected program state at critical code segments
- Two helpful mechanisms provided by Java:
  - Exceptions — discussed in Section 2.3 of the text
  - Assertions — information available on the course web site

## Advice for Debugging

### Recommended Steps:

- Reproduce the error (what inputs and execution environments cause the error?)
- Simplify the error (use the simplest possible input that causes the error when debugging)
- Locate the error (divide and conquer — isolate class, then function, code block, ...)
- Know what the program *should* do (compare against what the program *does* do)
- Look at all details (keep an open mind!)
- Make sure you understand the bug *before* you “fix” it (no quick-fixes to make the particular input work)

## Further Reading

Chapter 2 of the Koffman and Wolfgang reference has additional information including a JUnit case study/tutorial

**Wikipedia** has an extensive series of helpful articles on software testing as well as debugging.

Sun's documentation on programming with assertions in Java including the `assert` class (see course web page for URL)

Self-study and lab exercises on the course web page.