

Testing and Test Driven Development

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 10

Before we start: Career Fairs

COMPUTING AND SOFTWARE

Career & Internship Fair

Tuesday, Feb. 8 | 11 am - 4 pm

VIRTUAL: On Handshake

COMPLETE YOUR PROFILE THEN SIGN UP!



Alarm.com
CACI
Caterpillar Inc.
Credit One Bank
DISH
Epic
Fluid Truck Share
Garmin
Goldman Sachs
L3Harris Technologies
Lucid Software
MaXentric Technologies LLC
MojoTech
National Geospatial-
Intelligence Agency
National Security Agency (NSA)
NetApp
Raytheon Technologies
Ricoh USA, Inc
RSM US LLP
Sandia National Laboratories
Southwest Research Institute
Workiva



Career Services
UNIVERSITY OF COLORADO BOULDER



colorado.edu/career

Career Fairs

- Three fairs are recommended for CS students:
- Feb 8: Computing and Software Virtual Career and Internship Fair
- Feb 10: STEM Industries Virtual Career & Internship Fair
- Feb 17: STEM Industries In Person Career & Internship Fair
- Employer interest includes all student levels (freshmen through PhDs) for a mix of internships and full-time openings.
- For more information, contact:
 - Amy L. Richards, M.S.
 - Amy.L.Richards@colorado.edu
 - Professional Development and Industry Relations Program Manager

FAQ

- Register and get information about Career Fairs in Handshake
 - Virtual fairs are hosted within the Handshake platform.
 - Registration and employer information for all fairs can be found in Handshake.
- Virtual fairs offer group info sessions (with chat) and one-on-one sessions (video or audio) with company representatives.
 - You can book sessions in advance according to your schedule.
 - Remember to cancel if your availability changes.
- Update your Handshake profile with recent projects and skills so employers can see your accomplishments.
 - All students have a Handshake profile but it will only show basic information to employers.
 - This is an opportunity to present some experience and show your skills, so populate your profile for best results.
- Contact Career Services with questions about Handshake, the fairs, how to prepare, and resume review.
 - You can schedule an appointment with Career Services in BuffPortal Advising.
 - More info about Engineering Career Services at <https://www.colorado.edu/engineering/engineering-career-services>
- Employers are hiring for internships, part-time, and full-time positions.
 - Employers are interested in freshmen through PhD, so all levels are welcome! You can even filter by “School Years” to see who is hiring different levels.
 - It’s smart to attend career fairs even if you aren’t looking for an immediate opportunity. Practice, learn, explore.

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Quick look into Software Testing

Testing effectively may be just as important a part of designing and developing well, so let's give it a bit of attention...

Definitions

- Verification – ensuring your code meets engineering requirements
 - Validation – ensuring your code meets application expectations
 - Testing – running a program on selected inputs and checking the results
-
- Formal verification – constructing a proof that a program is correct
 - Z notation – a formal specification for describing and modeling business requirements for a program with relational algebra – allows code to be “correct by construction”
-
- Reference [1]

Software Quality Expectations

- 1 - 10 defects/kloc: Typical industry software
- 0.1 - 1 defects/kloc: High-quality validation - the Java libraries might achieve this level of correctness
- 0.01 - 0.1 defects/kloc: The very best, safety-critical validation, NASA
- Kloc = 1000 lines of code

So a 100,000 lines of typical industry source code (at the low end of 1 defect/kloc), it means you missed 100 bugs!

Reference [2]

Why Software Testing is Hard

- **Exhaustive testing** is infeasible. The space of possible test cases is generally too big to cover exhaustively. Imagine exhaustively testing a 32-bit floating-point multiply operation, $a * b$. There are 2^{64} test cases!
- **Haphazard testing** (“just try it and see if it works”) is less likely to find bugs, unless the program is so buggy that an arbitrarily-chosen input is more likely to fail than to succeed. It also doesn’t increase our confidence in program correctness.
- **Random or statistical testing** doesn’t work well for software. Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole lot. Physical systems can accelerate tests, e.g. opening a refrigerator 1000 times in 24 hours instead of 10 years. This assumes continuity or uniformity across the space of defects that’s only true for physical artifacts.

Reference [2]

Why Software Testing is Hard

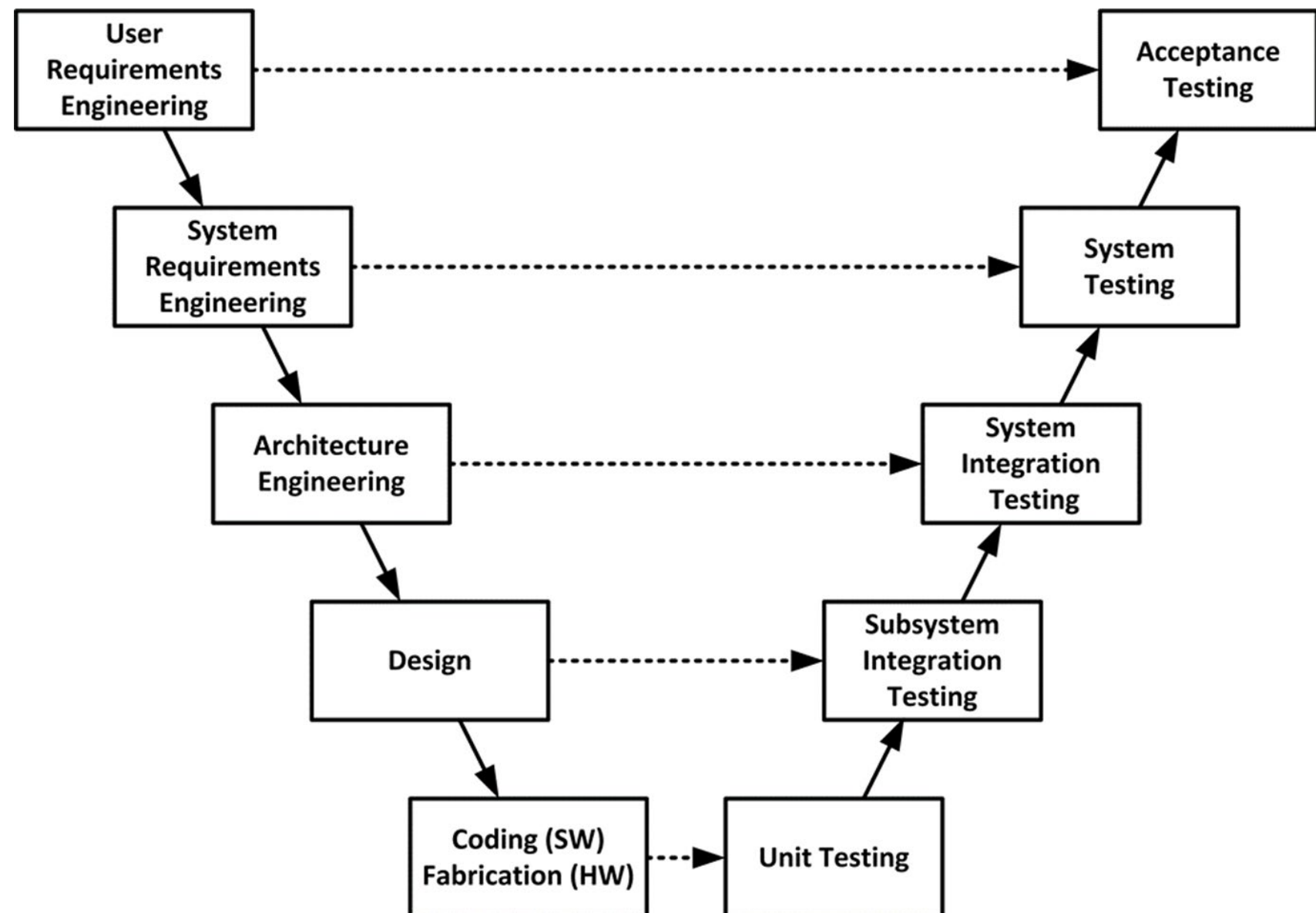
- Software behavior varies discontinuously and discretely across the space of possible inputs
 - The system may seem to work fine across a broad range of inputs, and then abruptly fail at a single boundary point
 - The famous Pentium division bug affected approximately 1 in 9 billion divisions
- In physical systems, there is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point
 - In a physical system, statistical testing will observe some failures even before the point is reached

Reference [2]

Multiple Levels of Testing for Systems

We have to consider how we will prove out our code at each level – from units, to sub systems, to integration, to full system tests, and to acceptance testing...

Reference [3]



Making Software Testing Work

- Test cases must be chosen carefully and systematically
- As a developer, our goal is to make the program work
- As a tester, you want to make it fail
- This is often a difficult transition for a developer
- Testers must identify and seek out vulnerabilities in order to eliminate them
- Reference [2]

Test-after/Test-with

- Often, we develop code in a “**test-after**” fashion. We develop the code, get it running, and then start to look for issues.
 - One of the issues with test-after goes back to the cost of finding and fixing defects, which increases as we get closer to finalizing our system
- A step improved from this is “**test-with**” development, where each element of code we write is written along with test cases to unit test that particular code.
 - At least here, we are taking the time to consider how the code should be tested and including some test infrastructure
 - We can also use these test cases in automated testing when new builds are made to see if changes to code have broken our expected behaviors

Test Driven Development

- A recognized best practice is “**test-first**” development, which is the basis of **Test Driven Development or TDD**
- In TDD, we start with a specification, develop tests that exercise that specification, and then write the code for the functionality – once your code passes the tests, you’re done
- The specification indicates the input and output behavior of system elements
 - Types of parameters and any constraints
 - Type of return value and how inputs relate to that return value
- Reference [2]

Do TDD and OOAD mix?

- Generally, OOAD is a conceptual exercise that leads to detailed specification and implementation
- TDD is details first development, developing tests prior to writing code
- I have been getting more concerned about how these two things mix, especially if one is given priority over the other
- The answer is, they can, but you must consider it carefully
- There's a nice example of a blended approach here:
<https://towardsdatascience.com/object-oriented-programming-and-the-magic-of-test-driven-development-d377acae85fa>
- Suggested approach
 - Write the story (what is the problem)?
 - Draw the problem
 - Identify the problem's entities (the classes)
 - Create the entity hierarchy/connections
 - Identify entity responsibilities (and communication/messaging)
 - Write the tests for these entities
 - Check that they fail as expected
 - Write your code
 - Repeat/refactor/refine
- OOAD – Conceptual foundation
- TDD – Specification/implementation technique

Writing Tests Strengthens Specifications

- When you start to develop tests, you begin to question the specification
- Is anything incorrect, incomplete, ambiguous?
- Are there odd corner cases to consider?
- Writing a test early to prepare for and address these issues prevents wasted time implementing code from a buggy specification.
- Reference [2]

Partitioning for Test Cases

- Developing a test suite is in itself a challenging and interesting design problem – finding a small set of cases to run quickly that still validates the code
- One approach to this is dividing the input space into subdomains with sets of inputs
 - Our goal is to have subdomains that cover all of input space, so any possible input is present in at least one subdomain
 - This allows us to develop specific tests for each subdomain
 - This ensures testing coverage, that we're not missing parts of the input space that random tests might not reach
- In some cases, we may have to consider the subdomains of the output space as well to ensure coverage, but usually inputs are the focus
- Reference [2]

Subdomain example

- Consider a simple multiply function for two large integer values – a, b
- Typical partitions might include
 - a and b are positive
 - a and b are negative
 - one or the other are negative and positive
- Special cases might include a or b becoming 0, 1, or -1
- Should consider boundaries – a and b small, absolute value of a or b bigger than maximum integer sizes
- Covering the space of 7 variations for 2 parameters is 49 tests!
- Reference [2]

Black Box vs. White Box Testing

- Black Box testing – decisions are made only based on the specification; we do not consider any knowledge of how a function works
- White Box (aka Clear Box or Glass Box) testing – testing with knowledge of code internals
 - For instance, if you know certain combinations of inputs will force different algorithms or code sections to be visited, you should partition tests and subdomains to visit them as needed
 - Care should be taken not to test cases that are not part of the specification
- Reference [2]

Test Coverage

- Coverage - How thoroughly does a test set exercise a program
- Types (in increasing strength – i.e. takes more tests to achieve):
 - Statement coverage: is every statement run by some test case?
 - Branch coverage: for every if or while statement in the program, are both the true and the false direction taken by some test case?
 - Path coverage: is every possible combination of branches – every path through the program – taken by some test case?
- 100% path coverage is usually infeasible, requiring exponential-size test suites to achieve
- 100% statement coverage is rare due to unreachable defensive code (like “should never get here” assertions)
- 100% branch coverage is highly desirable
- Reference [2]

Test Coverage

- One standard approach to testing is to add tests until the test suite achieves adequate statement coverage
- That is, every reachable statement in the program is executed by at least one test case
- In practice, statement coverage is usually measured by a code coverage tool, which counts the number of times each statement is run by your test suite
- With such a tool, white box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed
- Reference [2]

Unit test best practices

- Arrange, Act, Assert
 - Arrange – get things ready to test – instantiate an object or generate data
 - Act – decide what one action you'll perform to check
 - Assert – Using an appropriate check (like `IsEqual`) see if your code did what you expected
- One assert vs. multiple asserts per test method
 - Debatable – one assert is usually preferred for failure visibility
- Avoid test interdependence
 - Ideally each test should handle any set up or tear down needed
- Short and visible tests
 - Make it easy to see what went wrong
- If the Arrange step is hard, that may be a code smell
 - May point to a non-modular design
- Run tests with every build
 - Part of a standard DevOps/build process that includes any tests added
- <https://stackify.com/unit-testing-basics-best-practices/>

Automated Tests, Regression Tests

- If you're using TDD, you will have suites of unit tests for your code that could be run at every build in your DevOps cycles – this is automated testing – of the test cases you and your team developed by hand
 - As opposed to automatic test generation which is a hard problem, and an subject of active computer science research
- Regression testing refers to test cases added to verify a modification was successful (to keep your code from regressing, i.e. getting worse)
- Suites of unit and regression tests grow as code matures
- The greatest value of these tests is when the tests are run often and automatically for builds
- Reference [2]

Three General Goals for Good Software

- Response to change
 - Readiness for change is supported by writing tests that depend on behavior in a specification
 - Automated regression testing helps keep bugs from coming back when changes are made
- Safe from bugs
 - Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, immediately when you introduced them
- Easy to understand
 - Readability, maintainability,...
 - More part of documentation, code standards, and code review
- Reference [2]

Test Frameworks

- Unit Test Harness

- Software package to assess code – provides:
 - A common language to express test cases
 - A common language to express expected results
 - Access to the features of the production code programming language
 - A place to collect the unit test cases for the project, system, or subsystem
 - A mechanism to run the test cases, either in full or in partial batches
 - A concise report of the test suite success or failure
 - A detailed report of any test failures

- Reference [4]


Test Frameworks

- Mocks
 - The mock object (or simply the mock) is a test double
 - It allows a test case to describe the calls expected from one module to another
 - During test execution the mock checks that all calls happen with the right parameters and in the right order
 - The mock can also be instructed to return specific values in proper sequence to the code under test
 - A mock is not a simulator, but it allows a test case to simulate a specific scenario or sequence of events
- Reference [4]

Test Frameworks

- Frameworks for Java [5]

- JUnit
- JBehave
- Serenity
- TestNG
- Selenium



```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass(); // MyClass is tested

        // assert statements
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
    }
}
```

- Frameworks for Python [6]

- Robot
- PyTest
- unittest/PyUnit
- Behave
- Lettuce
- Nose

From Tutorial at [7]

JUnit Notes

- Typical Method Annotations
 - @Test – makes a public method into a test case
 - @Before, @After – methods to run before/after every test case
 - @BeforeClass, @AfterClass – methods to run before/after any/all test cases run
- Typical Assertions
 - assertTrue(test), assertFalse(test) – check Boolean tests
 - assertEquals, assertEquals, assertEquals – check equality
 - assertNull, assertNotNull – check for null elements/objects
 - fail – force test to fail
- Best practices
 - Write the test to know EXACTLY what failed if test doesn't pass
 - Tests should be self contained and not dependent on each other
 - Focus on boundary, empty, and error cases and combined behavior

<https://courses.cs.washington.edu/courses/cse331/11sp/sections/section4-cheat-sheet.pdf>

Help for JUnit

- Junit.org is the home site for the tools (both 4 and 5):
 - <https://junit.org/junit5/> or <https://junit.org/junit4/>
- I very much like this cycle of tutorials, presented step by step:
 - <http://tutorials.jenkov.com/java-unit-testing/simple-test.html>
 - These tutorials are for JUnit 4.8, JUnit 5.8 is also in use but has some minor differences – you can use either
 - Summary of JUnit 4 vs 5 here: <https://howtodoinjava.com/junit5/junit-5-vs-junit-4/>
 - Also a good tutorial: <https://howtodoinjava.com/junit-5-tutorial/>
- A nice best practice discussion
 - <https://phauer.com/2019/modern-best-practices-testing-java/>
- Another nice cheat sheet
 - <https://www.jrebel.com/blog/junit-cheat-sheet>
- Applying design patterns (observer, singleton, factory, template) to JUnit:
 - JUnit Recipes book - <https://livebook.manning.com/book/junit-recipes/chapter-14/>

Running jUnit from inside IDEs

- Running jUnit from inside IDEs:
 - <https://www.jetbrains.com/help/idea/performing-tests.html>
 - <https://code.visualstudio.com/docs/java/java-testing>
- Running jUnit from a command line:
 - <https://www.lambdatest.com/blog/run-junit-from-command-line/>
- For assignments, when we say capture test results, you can cut/paste the results from a testing window in the IDE or from the reporting the tool provides
- To clarify: For classroom use of JUnit, I'm really interested in your exploring the tool, not in comprehensive industrial strength test cases... Simple tests of object instantiation or value equality are fine!
 - Having said that, how deep you go in using it past project requirements is up to you

Summary

- Like OOAD, TDD is a learned approach, it's not something most developers do by default
- If you can't do "test-first" at least consider "test-with" where developing repeatable test cases for your code becomes part of your normal development
- Don't be surprised to find an industry environment where TDD or test-with is part of the development cycle because of automated DevOps processes that want to apply tests to every build
- I will eventually be requiring some test case development in your projects going forward, although I won't be asking for as extensive a test suite as you might do in an industry environment

Next Steps

- Assignments

- The next participation discussion topic is up on Piazza (about testing), a new one will go up Monday, please respond – don't get too far behind on these!
- Project 2 Part 2, the first team programming project, is due Wednesday 2/9
 - We'll introduce Project 3 on 2/9
- Next quiz up this weekend, due Thursday 2/10
- The first part of the Graduate Project, the Proposal, is due today
 - I will review the topics and proposals this weekend
 - Next part is the Outline, due on 2/22
- Readings from the Head First Design Patterns textbook
 - Start with Lecture 11 on Design Patterns – which lines up with Chapter 1
 - Lecture 12 on Observer and Lecture 13 on Decorator are supported by Chapter 2 and 3

- Coming up

- Next up: Patterns/Strategy, Observer, Decorator
- Please come find us for any help you need or questions you have!

Test Frameworks References

- [1] <https://www.hillelwayne.com/post/why-dont-people-use-formal-methods/>
- [2] <https://ocw.mit.edu/ans7870/6/6.005/s16/classes/03-testing/>
- [3] https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html
- [4] Test Driven Development for Embedded C, Grenning, 2011, Pragmatic Bookshelf
- [5] <https://dzone.com/articles/top-5-java-test-frameworks-for-automation-in-2019>
- [6] <https://www.lambdatest.com/blog/top-5-python-frameworks-for-test-automation-in-2019/>
- [7] <https://www.vogella.com/tutorials/JUnit/article.html>