Software Testing

A mini-lecture series

CSE498 Collaborative Design (W) - Secure and Efficient C++ Software Development

02/04/2026

Kira Chan

https://cse.msu.edu/~chanken1/

Testing is exhaustive

- Last lecture, we showed that
  - it is mathematically impossible to build a *test oracle*
  - Halting problem
- Similarly, for a given software program (or function), it is infeasible to test it against all possible inputs and configurations
  - Input space scales based on the number of parameter and possible input
  - How do you specify all the possible parameters?
  - Computational resource?
  - Space explosion

Broad types of testing

- Unit Testing
- Integration Testing
- Acceptance Testing
- Regression Testing
- Compatibility Testing
- Performance Testing

Unit Testing

- Break down software into smallest units (think Legos)
- Test each block individually
- What is the smallest unit of a software program that you can test?

Iclicker poll (not graded)

- Consider test driven development (where test cases are generated before coding), what should be considered a good test case?
  a) One that leads to a failure in the program
  b) One that always leads to a tautology (e.g., REQUIRE(0 == 0))
  c) One that leads to a success (passing) in the program
  d) One that can automatically fix errors in the code
  e) One that includes both passing and failing test cases

Iclicker poll

- Consider test driven development (where test cases are generated before coding), what should be considered a good test case?
  a) **One that leads to a failure in the program**
  b) One that always leads to a tautology (e.g., REQUIRE(0 == 0))
  c) One that leads to a success (passing) in the program
  d) One that can automatically fix errors in the code
  e) One that includes both passing and failing test cases

What is a good test case

- A good test case is one that fails
  - Otherwise, you cannot tell if your test case is bad or your code is bad

1. Validates that your testing process is working
2. Effectively highlights bugs
3. Highlights missing functionalities
4. Acts as a form documentation
   a) Highlights expected input (or range)
   b) Highlights use cases

Testing is often a daunting task

- There are so many permutations of inputs to test!

- How do you know where to start?
- How do you know when to stop?
- How do you know if you are testing the right thing?

How do we even start?

- I find it generally easier to take a Test Driven Development (TDD) approach
  - Write the test case first, then write the function
  - If you write your functions first, you have an idea of what inputs you can handle already
    - Introduces inherent testing bias into your code
    - “Test your own code gently”

- *If you do pair programming, one person can write the test cases while the other partner can “solve the puzzle”

High-level TDD Pipeline

1. Understand the problem space and inputs (behaviour of the sys)
2. Develop the test cases
3. Run the tests, make sure they should fail
4. Write simplest code that passes the tests (building functionality)
5. Be sure all test cases pass
6. Refractor as needed, while ensuring the test still pass
    a) Move code to where it belongs
    b) Remove duplicated code
    c) Split into multiple methods, etc.

Does that sound familiar?

1. Understand the problem space and inputs (behaviour of the sys)
2. Develop the test cases
3. Run the tests, make sure they should fail
4. Write simplest code that passes the tests (building functionality)
5. Be sure all test cases pass
6. Refractor as needed, while ensuring the test still pass
   a) Move code to where it belongs
   b) Remove duplicated code
   c) Split into multiple methods, etc.

High-level TDD Pipeline

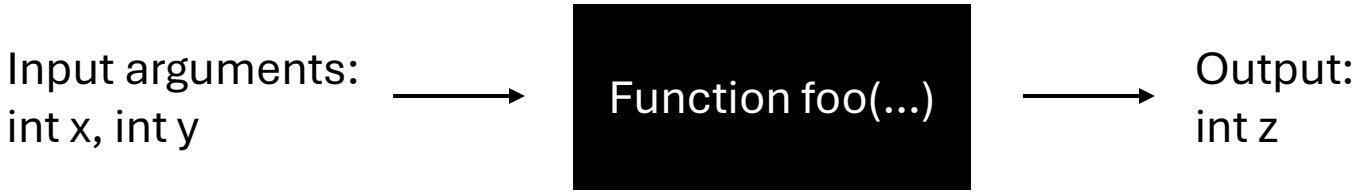
1. Understand the problem space and inputs (behaviour of the sys)
2. Develop the test cases
3. Run the tests, make sure they should fail
4. Write simplest code that passes the tests (building functionality)
5. Be sure all test cases pass
6. Refractor as needed, while ensuring the test still pass
    a) Move code to where it belongs
    b) Remove duplicated code
    c) Split into multiple methods, etc.

How do we even go about testing?

- Focusing on the Orange above

1. Understand the problem and the function's inputs and outputs
   - Consider the function to be tested as a "black-box"

Black-box model of a function



Some magic occurs here
(we will build the magic)

Functions are instead input transformers

How do we even go about testing?

1. Understand the problem and the function's inputs and outputs
   - Consider the function to be tested as a "black-box"
   - Ignore all the implementation details
   - We know what the function is supposed to do
   - Focus on the input space and the expected output space

How do we even go about testing?

1. Understand the function's inputs and outputs
2. Start with some expected (correct) inputs – good paths
   - Start small (very obvious example)
   - Add more difficult inputs
3. Identify what might be a reasonable incorrect path
   - Test to make sure that the function indeed breaks
4. Add edge cases
   - Usually where the "bugs" of the program lie
   - Divide by zeroes
   - Unaccepted Inputs (e.g., index -1 for a remove function)
5. Run the tests and make sure they fail!
   - Remember, a good test case is one that fails

Identifying test cases

Testing Extremes

- On one end, you can test as many inputs as you can think of
- Testing the same ``type'' of input also does not help
  - Tests should test different things
- Testing just the immediate functionality is also incomplete
- Testing all possible inputs is simply impossible
- Good testing is about balancing the two ends, picking the most effective test cases.

Mentality

- Writing test cases is like trying to find ways to break the program
- Come up with as many “whacky” ways to find incorrect behavior in the program

Example: Vector insert

- Suppose you are testing the function that pushes back an item
- Class Vector {
  - Private:
    - unsigned int size = 0;
    - Tree<int> container;
  - Public:
    - Void Vector() {…} // assume correct constructor
    - Void pushback(int item) {
      - container.push_back(item);
      - size++; }
    - Int size() { return size; }
- }

- Test case:
- vector.init() // assume empty vector
- REQUIRES(vector.size() == 0)
- vector.pushback(42)
- REQUIRES(vector.size() == 1)

Example: Vector Insert

- Will the following snippet pass our test case?
- Class Vector {
  - Private:
    - unsigned int size = 0;
    - Tree<int> container;
  - Public:
    - Void Vector() {…} // assume correct constructor
    - Void pushback(int item) { size++; }
    - Int size() { return size; }
- }

Example: Vector insert

- Suppose you are testing the function that pushes back an item

- vector.init() // assume empty vector
- REQUIRES(vector.size() == 0)
- vector.pushback(42)
- REQUIRES(vector.size() == 1)

Example: Vector Insert

- Yes, make sure that the size is updated correctly
- Should probably check if the item is actually 42 in the vector!
- Should probably test if we can push again (multiple times)
- Should probably test if the item is actually pushed to the end
  - If you only insert one item, this will appear functionally correct
- Should probably test what happens if you push something unexpected!
  - REQUIRE_THROWS(vector.pushback(“42”))

Example: Vector Insert (more complete)

- vector.init() // assume empty vector
- REQUIRES(vector.size() == 0)
- vector.pushback(42)
- REQUIRES(vector.size() == 1) // make sure size actually increases
- REQUIRES(vector[0] == 42) // make sure 42 is actually inserted
- vector.pushback(20)
- REQUIRES(vector.size() == 2) // check size again
- REQUIRES(vector.back() == 20) // make sure that the last item is 20
- REQUIRES(vector == {42, 20}) // make sure that the vector looks right
- REQUIRES_THROW(vector.pushback("0")) // make sure it breaks on invalid inputs

Example: Vector insert - Summary

- Do not test only the immediate property of your function
- If you can test the correctness of a given function multiple ways, you should do so!
  - Size
  - The position the new item is inserted at
  - The items in the whole vector

Example: Vector size

- Suppose we want to test the vector.size() function

- Vector.init() // assume empty vector
- Vector.pushback(10, 20, 42)
- REQUIRES(vector.size() == 3)

- Are we done after the test pass?

Example: Vector size

- Suppose this is the code below for my vector wrapper class
  - Will our previous test case correctly catch this?
- Class Vector {
  - Private:
    - unsigned int size = 0;
    - tree<int> container;

  - Public:
    - Void Vector() {…} // assume correct constructor
    - Void pushback(int item) { size++; container.push_back(item); }
    - int popback() { return container.pop_back(); }
    - Int size() { return size; }
- }

Example: Vector size

- Does the function still correctly work after you:
  - Insert?
  - Remove?
  - Append?
  - Extend?
  - Clear?
  - Std::move?

Example: Vector size (more complete)

- Vector.init() // assume empty vector
- REQUIRES(vector.size == 0) // check to make sure initialised vector is empty

- Vector.pushback(10, 20, 42)
- REQUIRES(vector.size() == 3) // check to make sure size updates correctly after pushing back

- Vector.remove(10) // remove the first element: 10
- REQUIRES(vector.size() == 2) // checking size correctness after remove

- Vector.clear()
- REQUIRES(vector.size == 0) // Make sure that we have an empty vector after clear()

Example: Vector size – summary

- If your function will update after other function calls, check that it still works correctly after the other functions are called.

Example: Vector clear

- Suppose we want to test the vector.clear() function

- Vector = {1,2,3,4,5}
- Vector.remove(3)
- Vector.remove(4)
- REQUIRES(vector.size == 3)
- Vector.clear()
- REQUIRES(vector.size == 0)

- Are we done?

Example: vector clear

- Will the following snippet pass our test case?
- Class Vector {
  - Private:
    - unsigned int size = 0;
    - tree<int> container;
  - Public:
    - Void Vector() {…} // assume correct constructor
    - Void clear() {container.delete(); size = 0;} //container.delete() iterates the tree and deletes each element one at a time
    - Int size() { return size; }
- }

Example: Vector clear

- Suppose we want to test the vector.clear() function

- Vector = {1,2,3,4,5}
- Vector.remove(3)
- Vector.remove(4)
- REQUIRES(vector.size == 3)
- Vector.clear()
- REQUIRES(vector.size == 0)

- Vector.pushback(0) // crashes because we just deleted the data structure

Example: Vector clear - summary

- This specific example is a little nuanced
  - But it could very much happen, especially if there are some “gotchas” in other libraries that you are using

- Check to make sure that your data structure still work as expected after you modify it (especially after you delete or remove stuff)
- Check the side effects of your functions

Summary from examples

- Testing is more than just checking the immediate effect of your function
  - Do not check only size() after insert
- Check the same function multiple times
  - Do not only insert() once and accept a “correct” output
  - Try multiple input values, try the same value multiple times, try combinations of parameters
- Do not only test if your test case pass only if you use the “right input” only
  - Try some unexpected inputs (different type, out of bounds, etc.)
- Check if the function unexpectedly modifies input
  - Probably good to check if pushback() might have accidentally corrupted the input if you pass by reference
- Check to make sure that your code did not have “side effects” on things it should not affect
  - Check other functions like insert() again after you have called clear() on your vector
  - Check related functions

Questions and Discussion

Poll

- Which of the following best describe test case design?

A. Test all the possible input combinations
B. Search for common edge cases online (e.g., input ‘0’, ‘-1’, etc.) and test
C. Identify the correct behaviours (w.r.t. inputs and outputs) of the function and test
D. Analyze the written code to produce the right test cases

Poll

- Which of the following best describe test case design?

A. Test all the possible input combinations
B. Search for common edge cases online (e.g., input ‘0’, ‘-1’, etc.) and test
C. Identify the correct behaviours (w.r.t. inputs and outputs) of the function and test
D. Analyze the written code to produce the right test cases

Some other high-level notes

Testing for input const-ness

- If your function do not modify the input (and should not). It is good practice to makes sure that your input copy is the same after you call the function
- Make a deep copy of the original object at the start of the testing, and make sure they are identical after the test cases

Edge cases testing

- This one might be a little challenging
  - Requires a lot of expertise in the problem and some thoughts
  - Think of it as a puzzle, and your job is to break the function
- Check for common issues:
  - Out of bounds
  - Input with “0” or null or nullptr
- If your problem implements an existing problem, then you can use a lot of existing inputs to test your program
  - Example: knapsack problem

Testing only one instance of an input

- Suppose we have the following code to test a random generator

- Num = randint(0, 100)
- Check(0 <= num <= 100)

- Num2 = randint(0, 100)
- Check(num != num2)

- Set a different seed and test if the first number generated is the same?
- Issues?

Testing only one instance of an input

- Int Randint(int x, int y)
  - Return x + std::rand() % (y – x);

- What is wrong with this function?
- This program will never produce [y]
  - You also cannot discover this issue with testing

- Int Randint(int x, int y)
  - Return x + std::rand() % (y – x + 2);
- This program will show the incorrect output 1/101 times

Change the way you think about testing

- Computers are fast now, use their computation to your advantage.
- Use many inputs (random insertion and deletion)
- Randomness testing example
- Generate 1 million numbers
- Test the average
- Split the values into “buckets”, then test if the buckets fall within the expected variance.

Preset objects

- Some classes and functions may operate over some data
  - Serializers
  - World classes (input as a 2D matrix)
- You should create a way to load a sample “playground” for testing
- Store a simple maze or world object as a text file
  - Load this object fresh each time you instantiate your test(s)
  - Test different parts of your code with the maze
    - Move right, make sure your agent actually moved right
    - Add cell, check to make sure a new cell of the type is actually added
    - Etc.

Another mentality to write better test

- As a tester: how can you break the programs?
  - Programs might “look” correct on one input, and may fail again on the same input
  - Think about what the program is supposed to do, not the implementation

- As a developer: how can you sneak around the test cases?
  - Can you come up with an incorrect program that can avoid detection
  - Write these test cases as if you were creating a leetcode problem

Why is testing important

- Software importance
  - A good test document ensures
    - That new builds do not break old features
    - Serves as a possible documentation for your code
    - What kinds of inputs may you (as a developer of the class or function) expect to be able to handle

Preview for the next episode

- Testing is hard

- We will discuss non-conventional techniques that can automatically help us with testing
- We will also discuss testing at other cycles

Person of the day
Margaret Hamilton

- Developed the on-board flight software for NASA's Apollo program with her team
  - See the code that was written by her and her team ->
- Director of the Software Engineering Division of the MIT Instrumentation Laboratory
- Established rigorous testing practices while she was there