

# CMSC 132: Object-Oriented Programming II

---

## Program Testing



**Department of Computer Science**  
**University of Maryland, College Park**

# Program Testing

## ■ Empirical testing

- Test software with selected test cases
- More scalable than verification
- Test failures frequently indicate software errors
  - Absence of failures doesn't prove software correct
- If code isn't exercised by any test, hard to have confidence in it
  - Even if it has been “formally verified”

# Kinds of testing

## ■ Automated testing

- The software is tested by a completely automatic process
  - e.g., JUnit or submit server testing
- Can be expensive or difficult to construct, but fairly cheap to repeat

## ■ Manual testing

- A person uses the software, perhaps guided by a script, and notes bugs
- Often easier to conduct than writing test cases, but very expensive to repeat

# Test size

## ■ Small

- Unit test – test individual components

## ■ Medium

- Integration tests
- Test subsystems containing several components
- Can test interactions between components, properties that are only demonstrated in larger systems

## ■ Large

- System or acceptance tests
- Test entire system, including non-software components

# Types of Testing

- **Clear box testing**
  - Allowed to examine code
  - Attempt to improve thoroughness of tests
- **Black box testing**
  - No knowledge of code
  - Treat program as “**black box**”
  - Test behavior in response to inputs

# Testing – Terminology

- **Test case**
  - Individual test
- **Test suite**
  - Collection of test cases
- **Test harness**
  - Program that executes a series of test cases
- **Test framework**
  - Software that facilitates writing & running tests
  - Example – JUnit

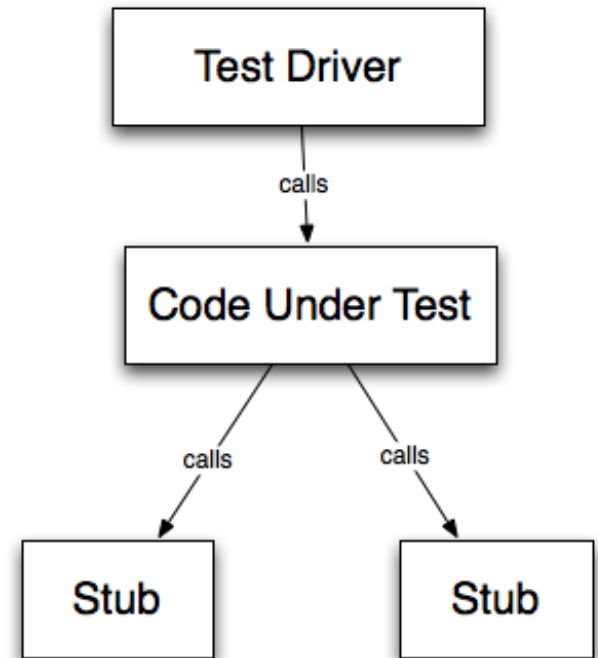
# Testing – Terminology

## ■ Test driver

- Program to create environment for running tests
- Declares variables, creates objects, assigns values
- Invokes tested code, checks results, reports failures

## ■ Stub

- Skeleton code in place of unfinished method / class
- Implements minimal functionality to allow test to occur
  - Allows software testing to begin



# Mock Objects

- **Similar to a stub**
- **But they record the calls made to them**
- **If the wrong calls are made to them, the test fails**
- **Can prerecord the sequence of expected calls**
  - **Also eliminates need for mock objects to contain any logic**
- **Or the test driver can query the calls after the test**
  - **Useful if calls aren't deterministic and need more careful logic to check**

# When to use mock objects

- If you want to test the calls made to other objects, rather than the return values or output of the methods under test
- You need to use mock objects
- Mock objects can also be easier to use than creating functional stubs
- Mock objects can simulate situations that might be hard to test on real code
  - e.g., does the code recover if the network fails?

# EasyMock example

```
warehouseControl = MockControl.createControl(Warehouse.class);
warehouseMock = (Warehouse) warehouseControl.getMock();
Order order = new Order(TALISKER, 50);
//setup – record expected calls and return values
warehouseMock.hasInventory(TALISKER, 50);
warehouseControl.setReturnValue(true);
warehouseMock.remove(TALISKER, 50);
warehouseControl.replay(); // put mock into replay mode
//exercise – execute code under test
order.fill(warehouseMock);
//verify
warehouseControl.verify();
assertTrue(order.isFilled());
```

# Unit Test

- **Test individual units extensively**
  - **Classes**
  - **Methods**
- **Central part of Extreme Programming (XP)**
  - **Extensive unit testing during development**
    - **Pair programming**
  - **Design unit tests along with specification**
- **Approach**
  - **Test each method of class**
  - **Test every possible flow path through method**

# Test Coverage

- **How do you know if your tests are any good?**
  - **In general, you can know if they are bad or insufficient, harder to tell that they are good**
- **Do they handle and check all the situations described in the specification and use cases?**
- **Do they invoke all the methods?**
- **Do they test all of the code?**

# Flow Path

- Unique execution sequence through program

- Example

```
S1
while (B1) {
  if (B2)
    S2
  else
    S3
}
```



| Flows      |
|------------|
| S1         |
| S1, S2     |
| S1, S3     |
| S1, S2, S2 |
| S1, S2, S3 |
| S1, S3, S2 |
| S1, S3, S3 |
| ...        |

# Test Coverage

- **Not possible to test all flow paths**
  - Many paths by combining conditionals, switches
  - Infinite number of paths for loops
  - New paths caused by exceptions
- **Test coverage**
  - Whether code is executed by **some** test case
  - Alternative to flow path
  - Ensure high % (if not all) of lines of code tested
  - Does not capture all possible flow paths
    - Even if all lines of code tested by some test case

# Test Coverage, continued

- **Branch coverage is stronger than statement coverage**
  - **Generally achievable**
- **Can be tricky to cover all exceptions and error cases**
- **Control flow coverage doesn't tell you about data coverage**
  - **Did you try it with negative integers, or with non-ASCII characters?**
- **Coverage won't tell you about functionality you forgot to implement or test**

# When to test

- **If code has never been tested, you have no idea if it ever worked**
- **But it is also important to perform regression testing**
  - **Check to see if some functionality that used to work stops working**
  - **The faster a regression is identified, the cheaper it is to fix, at any scale**
    - **Within a minute is better than within an hour**
    - **Within a day is better than within a week**

# Why regression test?

- **Bits don't rot**
- **But running regression tests give developer much more freedom to change existing code**
  - **“I need to rewrite this component to support new functionality – I wonder if anything might be depending on the details of how it works now?”**
- **This freedom is key to agile development, and important even in more structured development methodologies**

# Selecting regression tests

- **Big, well tested systems will have too many tests to run all of them every time you compile**
- **Prioritize tests by size**
  - **Ones that take only a few seconds**
  - **Ones that need to run over the weekend**
- **And by proximity to code changed**
  - **After changing some code, you only need to rerun the tests that executed the code that was changed**
- **Research work on prioritizing tests**

# Developing Quality Test Cases

- **Useful to have someone else write test cases**
  - **One person might make the same incorrect assumption in both their code and in their tests**
- **Tips on developing test cases**
  - **Develop test data during analysis & design phases**
    - **Use cases → Test cases**
    - **Pay close attention to problem specification**
  - **Check boundary conditions**
    - **1<sup>st</sup> and last iterations of loop**
    - **1<sup>st</sup> and last values added to data structure**
  - **Improve code coverage**