Unit and Integration Testing

Unit Testing

- Executes a small portion of an application (typically a function) and verifies its behavior independently of other parts
- Unit tests generally follow three stages
  - Arrange Phase: Initialize a portion of the application you want to test
  - Act Phase: Apply some stimulus to that portion
  - Assert Phase: Observe and verify the resulting behavior
- Generally falls into one of two categories
  - State-based: verifying that the test produces correct results or that the resulting state is correct
  - Interactions-based: verifying that the test properly invokes certain methods

Unit Test Example

```
TEST(palindromeTests, isPalindromeReturnsTrue) {
    // Arrange Phase: we create and set up a system under test, which could be a
    // method, object, or system of related objects. This phase could be empty if
    // testing a static method
    PalindromeDetector detector = new PalindromeDetector();

    // Act Test: we exercise the system under test, usually by invoking a method
    // We either check the return of the method or we check for expected byproducts
    bool is_palindrome = detector.is_palindrome("kayak");
    // We should also test the false case, as well as corner cases (“”, etc.)

    // Assert Phase: causes the test to pass or fail, verifies that execution meets
    // expectation
    ASSERT_TRUE(is_palindrome)
}
```

Writing Good Unit Tests

- **Easy to write**: you usually need to write multiple tests for each unit, so they need to be easy to write, update, and extend
- **Readable**: the intent of the test should be clear. Tests are often thought of as another form of documentation because a test suite efficiently describes how the unit is expected to act
- **Reliable**: tests should only fail when there is a bug in the program. It should not depend on the order the tests are executed, the running environment, or utilize randomness.
- **Fast**: tests need to be run repeatedly and often, usually during the development process as well as part of any continuous integration or deployment system.

Unit Testing Exercise

```
vector<int> prime_sieve(int N) {
    // searches for all prime numbers between 1 and N
    // and returns them as in a single vector
}
```

When Unit Testing is Hard

- Tightly Coupled to Concrete Implementations: the unit relies on hard-coded, concrete instances such as language streams (`cout`), local filesystem, current time, etc.
- Violates the Single Responsibility Principle: either the system has multiple responsibilities (consuming information and processing it) or it has more than one reason to change (the internal logic changes or non-parameter external influences change such as the file system, time, etc.)
- Lies about it’s dependencies: the function signature doesn’t specify every parameter the unit uses, such as user input within a function
- Hard to Understand and Maintain: the unit depends on a large and/or complex set of systems being run before it

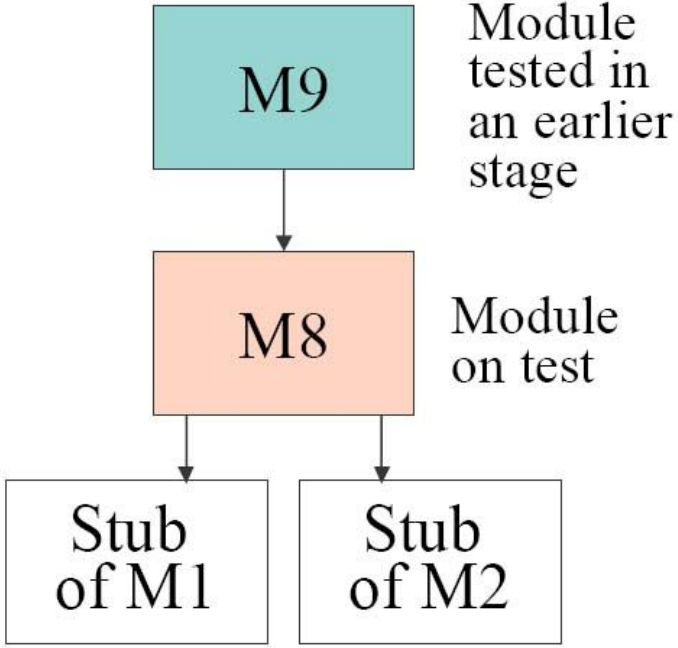
Integration Testing

- Verifies that systems work together (integrate) in a real-life environment
- Tests for defects in interfaces and interactions between components
- This often involves external resources like databases and web servers to be present or mocked
- Often tries to mimic system in the real-world including environments, install strategies, outside failures, etc.
- Often tries to simulate the user, often by mimicking their interactions with the system

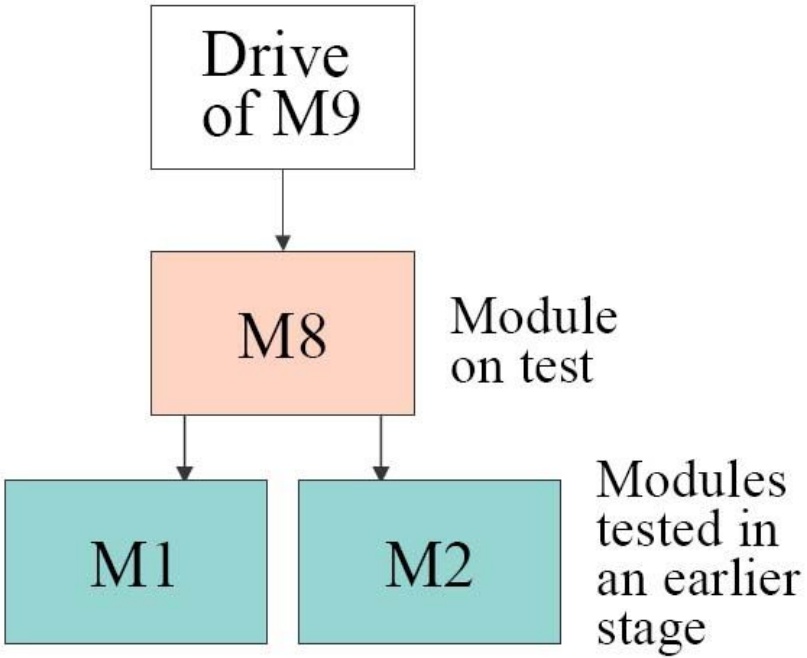
Stubs and Drivers

- You don’t need every module to be developed before you can perform an integration test, you can instead use stubs and drivers to simulate some interactions
  - Stubs: a simulation of a system that the module in question is going to call, which simulates a response
  - Drivers: a simulation of a system that is going to be making a call to the module
- Drivers are typically used in a bottom-up approach, where you create the innermost modules and need to simulate a higher level system call them
- Stubs are typically used in a top-down approach, where you create the highest level modules first and the systems that module calls doesn’t exist

Top-down testing of module M8



Bottom-up testing of module M8



Bottom Up vs. Top Down

Bottom Up:

+ fault localization is easier
- critical top level modules which control application flow are tested last
- impossible to create an early prototype

Top Down:

+ Can create early prototypes with lower level modules stubbed
- Requires a large number of stubs since there are more lower level modules than higher level ones

Testing for your CS 100 Assignments

- Creating unit tests early and practicing test driven design can be a useful development strategy when creating different functions and classes for your assignment
- Create a large variety of integration tests using bash to simulate inputs is extremely effective for testing how your system handles different inputs (and is essentially how we test your program)
- Running regressions on both these test suites gives you high confidence you aren’t breaking old features when developing new ones

Questions?

References

https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters

https://www.softwaretestinghelp.com/what-is-integration-testing/

http://www.cs.nott.ac.uk/~pszcah/G53QAT/Report08/zhw05u-WebPage/meth.html