Topic 8

Software Testing

Part 1

Introduction to Software Testing

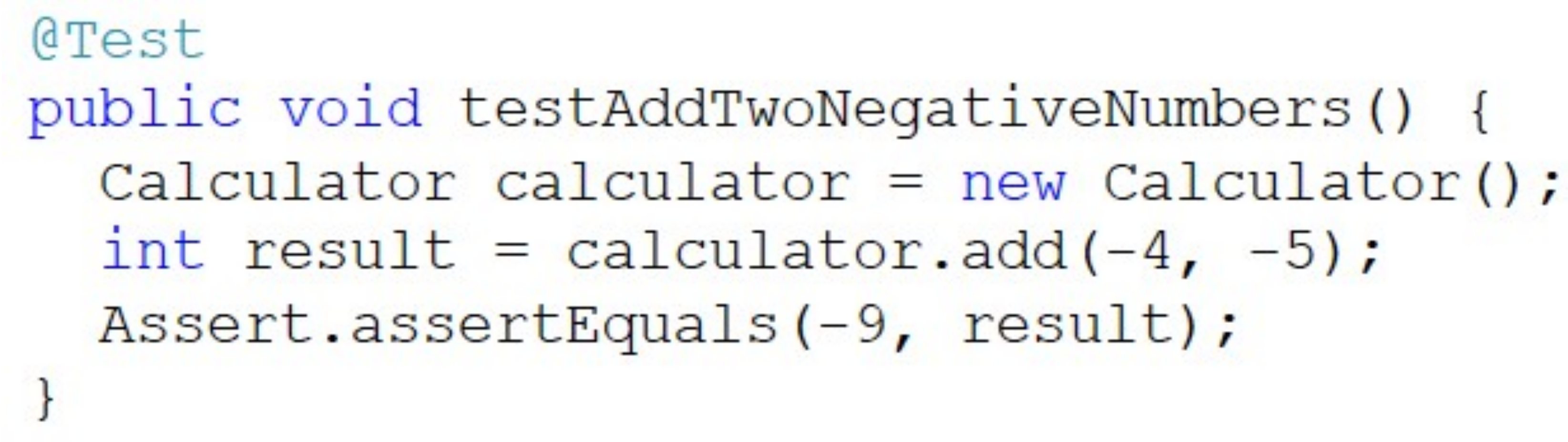
- Terminology
- Testing Types
  - Unit
  - Integration
  - System
- Continuous Integration
- Test-Driven Devlopment

What is the purpose of testing?

- Historical View
  - Testing is done to show the system works
  - Tend to go easy on the program
  - Programmers use same logic to test as they did to code
  - Some (many) bugs do not get caught
- Modern View
  - Testing is done to uncover bugs
  - We purposely take the attitude of trying to break the program
  - Result: more bugs caught, more reliable system

Terminology

- Test case
  - A set or sequence of inputs used to test a program, along with an expected output
  - JUnit Ex:



```
@Test
public void testAddTwoNegativeNumbers() {
  Calculator calculator = new Calculator();
  int result = calculator.add(-4, -5);
  Assert.assertEquals(-9, result);
}
```

Terminology

- Test suite
  - A set of test cases

Testing builds confidence in code

- Good test cases
  - one we think is likely to uncover a bug
- Good test suite
  - contains enough good test cases to test the requirements thoroughly
  - Having software that consistently passes a good test suite is more likely to be reliable upon release

Terminology

- Bug
  - Informal term that can mean several different things
  - Sometimes, it is more useful to use precise terminology
- Failure
  - Something the program does wrong (crashing, incorrect result)
- Fault
  - The incorrect code causing the failure (= instead of ==)
- Error
  - Mistake the programmer made leading to the fault
    - e.g. made a typo or didn’t realize that == was needed

Can we catch them all?

- Software errors tend to follow the Pareto Principle (80-20 rule)
  - 80% of the failures caused by 20% of the faults
    - Easier to find – failures occur frequently
  - 20% of the failures caused by 80% of the faults
    - Less frequent and therefore harder to find

Can we catch them all?

- Some failures may be very hard to find
  - Timing issues (race conditions)
  - Complex interactions with external systems
- In a large system, it is likely we will never find all the bugs
  - Avoid, find, eliminate as many bugs as possible
  - Build failsafe checks to alleviate effects of faults

Can we catch them all?

- Create systems to have failures reported
  - Error reports
    - automatically submitted upon failure
  - Bug reports
    - submitted by developers and/or users
  - Beta-testing
    - Release “beta” version for users to try

Testing vs. Debugging

Testing: running test cases, finding failures

- Can often be done without looking at the code
- Can be automated or partially automated

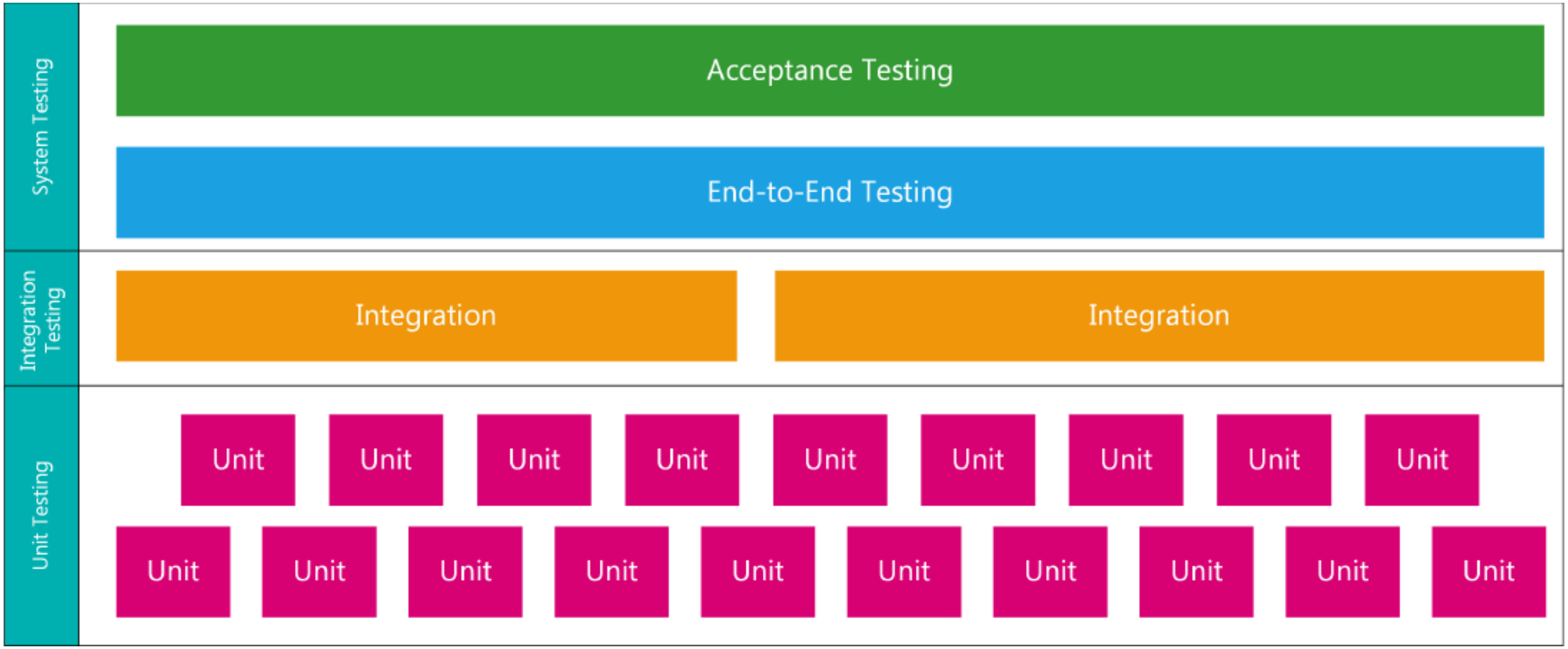
Debugging: finding and correcting faults

- Need to work with the code
- Use a debugger
- Cannot generally be automated
- Static analysis tools exist for finding certain faults (ex. findbugs)

Automated Testing

- Whenever possible!
- Automating tests:
  - Use a “driver”
    - simple program existing solely to test a function or module
  - Testing with a driver
    - Write the driver
    - Compile the driver together with the module
    - Run the driver
  - In practice, this should be used only for the simplest of programs. In reality, we use a test framework like JUnit, which acts as the driver of our tests.

Types of Testing

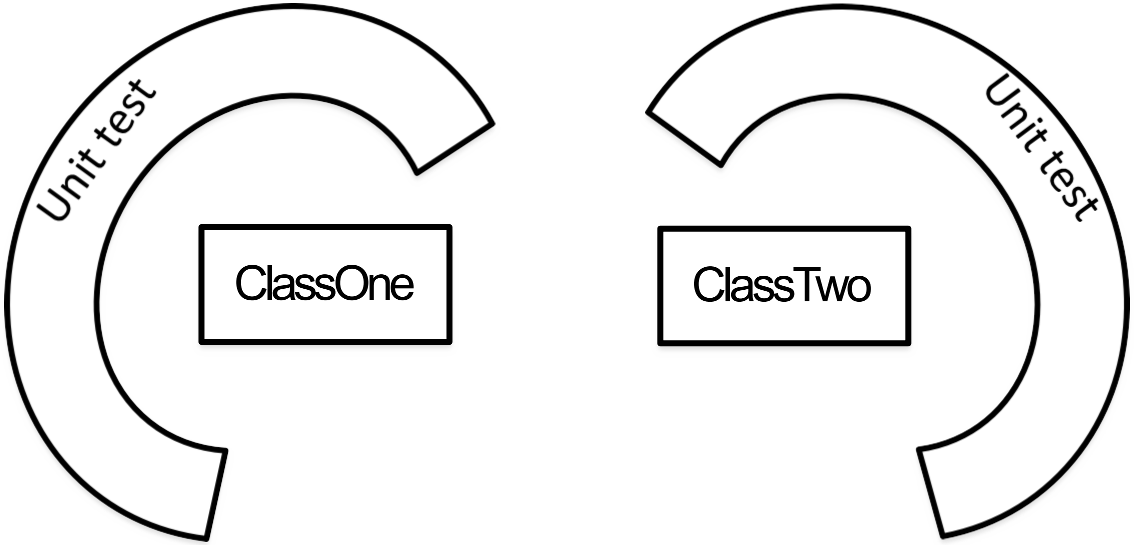


Unit Testing

- Test individual methods and classes
- Test components in isolation from each other

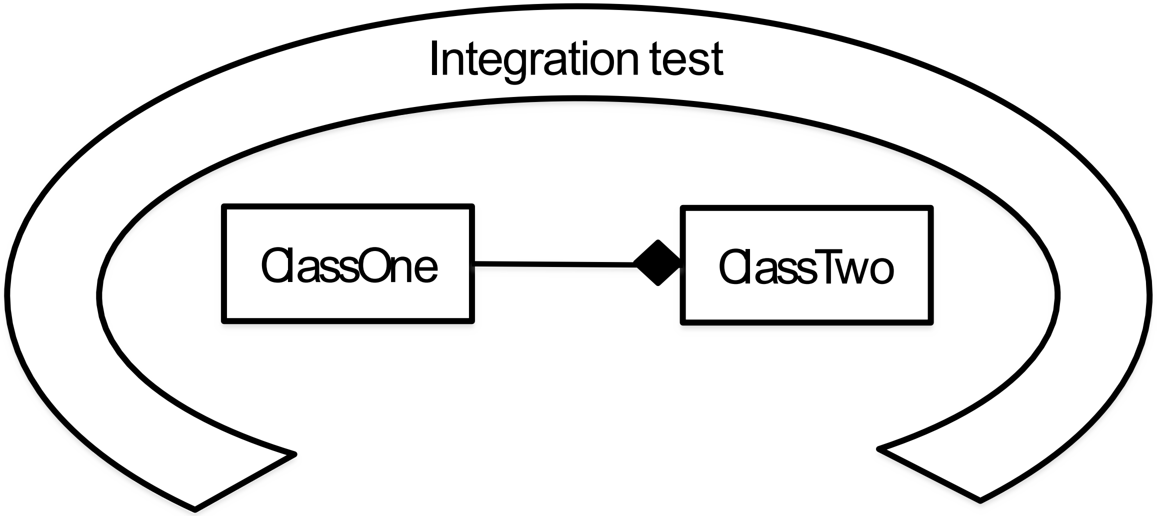
Unit Testing

- Test each class independently
  - Or each software component



Integration Testing

- Ensure that modules compile and interoperate correctly



Wait…

- How do we unit test ClassTwo if it requires ClassOne?
- We need some kind stand-in
- Some options:
  - Stub methods
  - Fake classes/objects
  - Mock objects

Stub

- Stand in for a function not yet written/integrated
- Simple, usually returning known value
- “stubbed out” methods are invoked to run testing

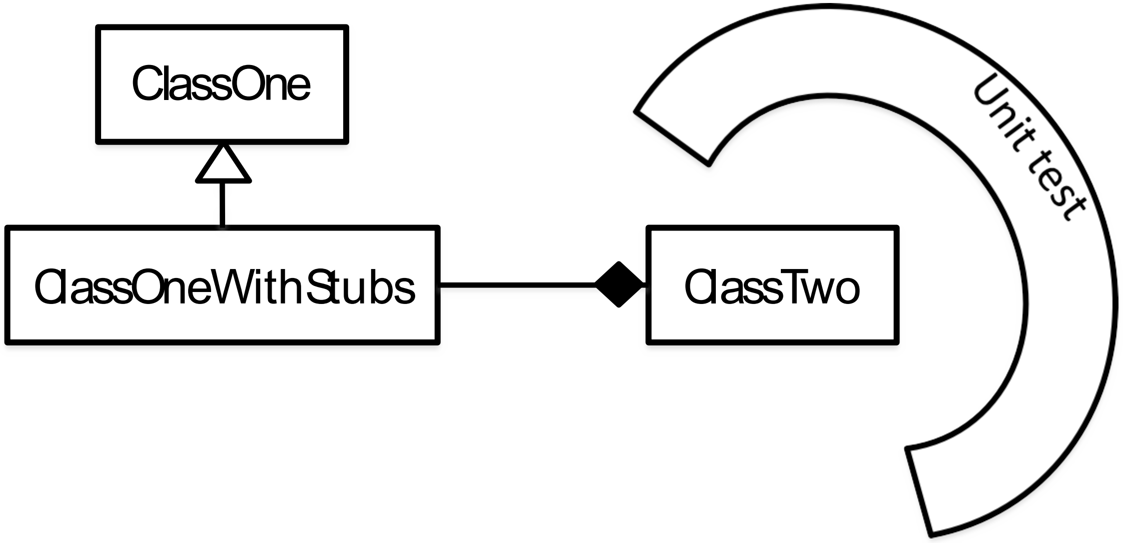
Stubs

We’ll usually use a stub method if a method we need to test

- Uses other methods/classes that aren’t finished yet
  - Stub out the unfinished methods/classes
- Uses methods/classes that work with external sources
  - (files, database, network)
  - Unit tests need to be fast (don’t want to wait for i/o when not necessary)
  - Unit tests need to be isolated (code + database = integration test)
  - Stub out these methods/classes
- Uses methods that return different values based on date/ time
- Uses stochastic (non-deterministic) methods

Stubbing Methods

- Can be done manually
  - Ex. Can create a subclass of ClassOne with some methods implemented/overridden as stubs, and use this for tests



Stubbing Stochastic Methods

- Stubs are easy to create for deterministic methods.
- What about non-deterministic methods that exhibit unpredictable behaviour?
  - Always return the same value
  - Always return the same sequence of values
    - Use a static variable to track how many times the method has been called
    - On the first invocation, we'll return 5
    - On the second invocation, we'll return 99
    - ...

Fake Objects

- Several definitions exist
  - Object that has all methods implemented as stubs
  - Object that takes some kind of shortcut making it unsuitable for the final product

Unit vs. Integration Tests

- With only integration tests, can’t definitively say
  - The problem is in your code
  - The problem is in the database

We waste time finding the bug

- Unit + integration tests means
  - My code works
  - My code works with the database
  - My code works with your code

Mocks

- Can often be confused with stubs
  - Mocks do allow us to stub methods
  - Also allow us to:
    - Verify that specific methods were called
    - Verify that specific arguments were passed
    - Thus, we can record and verify the interactions between the class and its collaborators

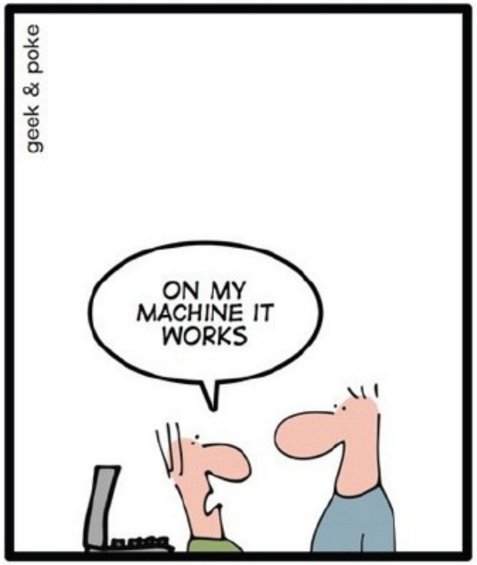
Continuous Integration

(Common) Scenario

- Joe and Jane are working on a project
  - They each implement a few classes
  - Code them
  - Ensure they are well tested
- When they're each done, they integrate them
  - Everything breaks

Integration Pain

That awkward moment near the end of a project when everyone realizes that none of their classes interoperate correctly



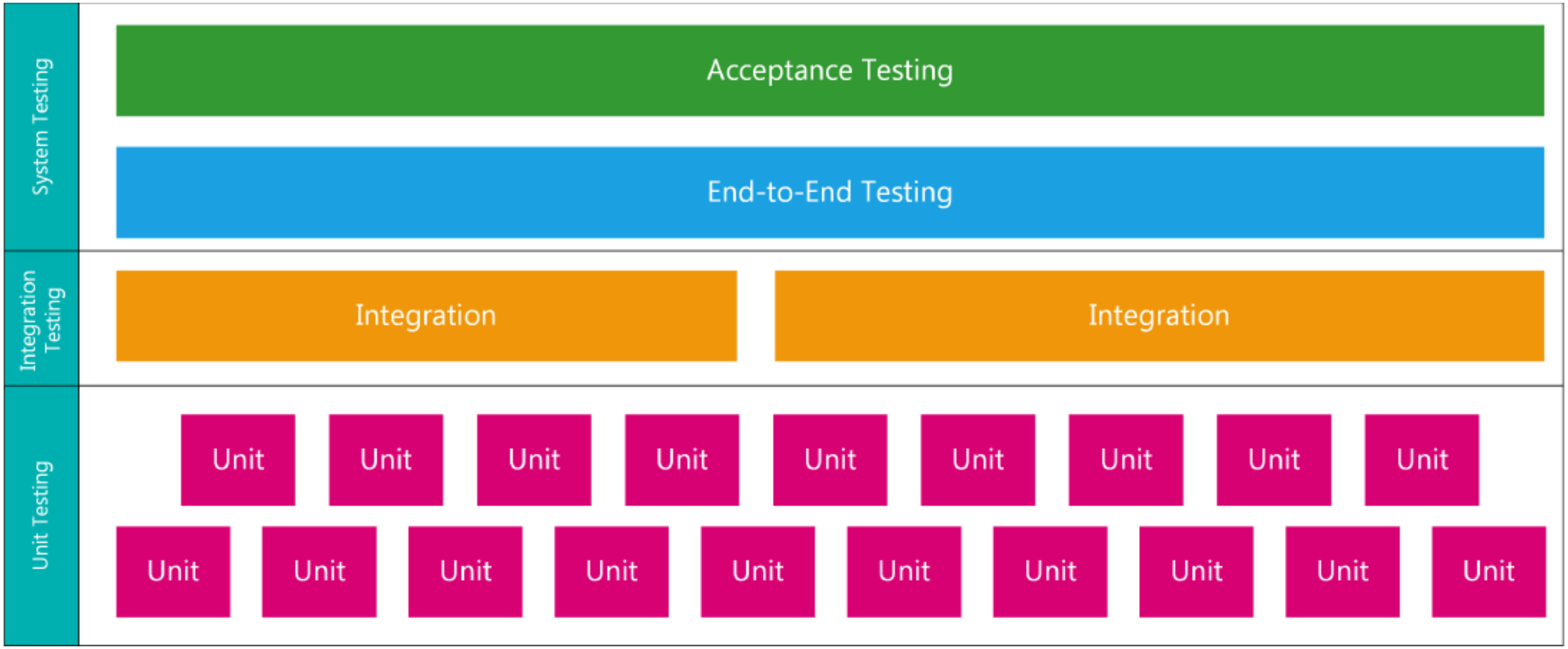
Integration Pain

- Extremely risky for a project
- Difficult to determine time required to resolve the integration problems
  - May (vastly) exceed our budget/expertise
  - May (vastly) exceed our schedule

Continuous Integration

- Mitigates risks associated with integrating software
- Avoids integration pains
- Integrate early and integrate often
  - i.e. on every change

Types of Testing



System Testing

Acceptance Testing

End-to-End Testing

Integration Testing

Integration

Integration

Unit Testing

Unit Unit Unit Unit Unit Unit Unit Unit Unit

Unit Unit Unit Unit Unit Unit Unit Unit Unit Unit

System Testing

- Testing the entire system
  - End-to-end
    - Tests workflows or paths
      - happy paths and unhappy paths
  - Acceptance
    - Tests done by the client in “accepting” that the requirements of the contract are met (so they pay you)
      - ...or by tester acting as such
    - Suite of tests defining when a requirement or user story is “done”

Automating Acceptance Tests

Benefit of MVC-like system designs

- Substitute a test driver class for the GUI view.
- Interact with the controller and model as the GUI would
- In a REST-based application, testing back and front ends can initially be done in isolation