Lecture 18: Verification, Validation and Testing: Overview

Adriana Sejfia

School of Informatics, University of Edinburgh

Last Lectures

- Requirements engineering
- Design
- Construction
- Refactoring

This Lecture: Verification, Validation and Testing (VV&T)

- Motivation
- Definitions
- Essence of testing
- Terminology of what can go wrong
- Approaches to testing, kinds of tests
- How to test:
  - Test-first development
  - Test-driven development
  - Behaviour-driven development
- Evolving tests
- Limitations of testing

VV&T: Motivation

- High quality code does what it is supposed to do.

VV&T: Motivation

- High quality code does what it is supposed to do.
- What it is supposed to do means:
  - Meets stated requirements
  - Meets wider expectations (of whoever asked for its development, and ideally of stakeholders)

VV&T: Motivation

- High quality code does what it is supposed to do.
- What it is supposed to do means:
  - Meets stated requirements
  - Meets wider expectations (of whoever asked for its development, and ideally of stakeholders)
- Problems:
  - How can we know this is the case?
  - When it is not, how can we isolate the cause?

VV&T: Definitions

- VV&T generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).

VV&T: Definitions

- VV&T generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).
- Verification: are we building the software right?
  - Does software meet requirements?

VV&T: Definitions

- VV&T generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).
- Verification: are we building the software right?
  - Does software meet requirements?
- Validation: are we building the right software?
  - More general. Does software meet expectations?

VV&T: Definitions

- VV&T generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).
- Verification: are we building the software right?
  - Does software meet requirements?
- Validation: are we building the right software?
  - More general. Does software meet expectations?
- Testing is a useful (but not the only) technique for both.

VV&T: Definitions

- VV&T generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).
- Verification: are we building the software right?
  - Does software meet requirements?
- Validation: are we building the right software?
  - More general. Does software meet expectations?
- Testing is a useful (but not the only) technique for both.
- Other techniques for verification: reviews/inspections/walkthroughs, static analysis

VV&T: Definitions

- VV&T generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).
- Verification: are we building the software right?
  - Does software meet requirements?
- Validation: are we building the right software?
  - More general. Does software meet expectations?
- Testing is a useful (but not the only) technique for both.
- Other techniques for verification: reviews/inspections/walkthroughs, static analysis
- Other techniques for validation: prototyping, early releases

Essence of Testing

- Generating stimulus for component

Example

```
public double calculateShippingCost(double orderTotal, String membershipLevel,
                                     boolean expressDelivery) {
    double baseCost;

    if (orderTotal >= 50.0) {
        baseCost = 0.0;
    } else {
        baseCost = 5.99;
    }

    if (membershipLevel.equals("PREMIUM")) {
        baseCost = 0.0;
    } else if (membershipLevel.equals(”STANDARD")) {
        baseCost *= 0.5;
    }

    if (expressDelivery) {
        baseCost += 9.99;
    }

    return baseCost;
}
```

Example

1. (orderTotal=60.0, membershipLevel="STANDARD", expressDelivery=false) → expects 0.0

2. (orderTotal=30.0, membershipLevel="PREMIUM", expressDelivery=true) → expects 9.99

Essence of Testing

- Generating stimulus for component
- Collecting outputs from component

Essence of Testing

- Generating stimulus for component
- Collecting outputs from component
- Checking if actual outputs are as expected

Essence of Testing

- Generating stimulus for component
- Collecting outputs from component
- Checking if actual outputs are as expected
- Often hard **to fully test a component in isolation**
  - Component test environment constructed using mock objects

Mock Object Example

```
public boolean placeOrder(String customerEmail, double amount) {
  boolean paymentSuccess = paymentGateway.charge(customerEmail, amount);
  if (paymentSuccess) {
     emailService.sendConfirmation(customerEmail);
               return true;
   }
   return false;
}
```

Mock Object Example

```
public boolean placeOrder(String customerEmail, double amount) {
  boolean paymentSuccess = paymentGateway.charge(customerEmail, amount);
  if (paymentSuccess) {
     emailService.sendConfirmation(customerEmail);
               return true;
   }
   return false;
}
```

Mock Object Example

```
public boolean placeOrder(String customerEmail, double amount) {
  boolean paymentSuccess = paymentGateway.charge(customerEmail, amount);
  if (paymentSuccess) {
     emailService.sendConfirmation(customerEmail);
               return true;
   }
   return false;
}
```

Using Mockito:

```
PaymentGateway mockPayment = mock(PaymentGateway.class);
EmailService mockEmail = mock(EmailService.class);
when(mockPayment.charge("alice@example.com", 49.99)).thenReturn(true)
```

Terminology of What Can Go Wrong

- Mistake: Human behaviour that produces fault(s)

Terminology of What Can Go Wrong

- Mistake: Human behaviour that produces fault(s)
- Fault: An incorrect step, process, or data definition in a computer program. A.k.a defect or, informally, a bug

Terminology of What Can Go Wrong

- Mistake: Human behaviour that produces fault(s)
- Fault: An incorrect step, process, or data definition in a computer program. A.k.a defect or, informally, a bug
- Error: A discrepancy between some computed value and the correct value; captured by tests

Terminology of What Can Go Wrong

- Mistake: Human behaviour that produces fault(s)
- Fault: An incorrect step, process, or data definition in a computer program. A.k.a defect or, informally, a bug
- Error: A discrepancy between some computed value and the correct value; captured by tests
- Failure: The termination of an intended behavior due to a fault manifestation

Terminology of What Can Go Wrong

- Mistake: Human behaviour that produces fault(s)
- Fault: An incorrect step, process, or data definition in a computer program. A.k.a defect or, informally, a bug
- Error: A discrepancy between some computed value and the correct value; captured by tests
- Failure: The termination of an intended behavior due to a fault manifestation
- Faults do not necessarily lead to errors

Terminology of What Can Go Wrong

- Mistake: Human behaviour that produces fault(s)
- Fault: An incorrect step, process, or data definition in a computer program. A.k.a defect or, informally, a bug
- Error: A discrepancy between some computed value and the correct value; captured by tests
- Failure: The termination of an intended behavior due to a fault manifestation
- Faults do not necessarily lead to errors
- Errors do not necessarily lead to failures

Terminology Example

```
public double calculateAverage(int[] scores) {
   int total = 0;
   for (int i = 0; i <= scores.length; i++) {
      total += scores[i];
   }
   return total / scores.length;
}
```

Terminology Example

```
public double calculateAverage(int[] scores) {
  int total = 0;
  for (int i = 0; i <= scores.length; i++) {
    total += scores[i];
  }
  return total / scores.length;
}
```

scores = {10, 20, 30}?

Terminology Example

```
public double calculateAverage(int[] scores) {
  int total = 0;
  for (int i = 0; i <= scores.length; i++) {
    total += scores[i];
  }
  return total / scores.length;
}
```

scores = {0, 0, 0, 0}?

Approaches to Testing: Black Box

- Black box testing: focuses on requirements while treating the system as a black box (not looking into its code)

Approaches to Testing: Black Box

- Black box testing: focuses on requirements while treating the system as a black box (not looking into its code)
- Advantages:
  - Helps conduct verification
  - When refactoring, tests do not need to be changed

Approaches to Testing: Black Box

- Black box testing: focuses on requirements while treating the system as a black box (not looking into its code)
- Advantages:
  - Helps conduct verification
  - When refactoring, tests do not need to be changed
- Disadvantages:
  - May not thoroughly exercise the different ways to execute the code

Approaches to Testing: White Box

- White box testing: considers software code; tests that the system does what the developer intended

Approaches to Testing: White Box

- White box testing: considers software code; tests that the system does what the developer intended
- Advantages:
  - Helps developers check their work; more thorough

Approaches to Testing: White Box

- White box testing: considers software code; tests that the system does what the developer intended
- Advantages:
  - Helps developers check their work; more thorough
- Disadvantages:
  - Will miss misinterpreted requirements; refactoring will require updating the tests

Approaches to Testing: Regression

- Repeat some/all tests after modifications
- Can help identify bugs and their location quicker

Approaches to Testing: Regression

- Repeat some/all tests after modifications
- Can help identify bugs and their location quicker
- Run them overnight

Approaches to Testing: Regression

- Repeat some/all tests after modifications
- Can help identify bugs and their location quicker
- Run them overnight
- Leverage version control + build tools

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors
- Integration tests: test that components interact properly

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors
- Integration tests: test that components interact properly
- System tests: at the level of the whole system, check if requirements met

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors
- Integration tests: test that components interact properly
- System tests: at the level of the whole system, check if requirements met
- Acceptance tests: check that system meets user/customer needs (validation); done in real environment with real data

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors
- Integration tests: test that components interact properly
- System tests: at the level of the whole system, check if requirements met
- Acceptance tests: check that system meets user/customer needs (validation); done in real environment with real data
- Stress tests: push system to its limits to check that performance degrades gracefully

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors
- Integration tests: test that components interact properly
- System tests: at the level of the whole system, check if requirements met
- Acceptance tests: check that system meets user/customer needs (validation); done in real environment with real data
- Stress tests: push system to its limits to check that performance degrades gracefully
- Performance tests: checking other performance requirements

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors
- Integration tests: test that components interact properly
- System tests: at the level of the whole system, check if requirements met
- Acceptance tests: check that system meets user/customer needs (validation); done in real environment with real data
- Stress tests: push system to its limits to check that performance degrades gracefully
- Performance tests: checking other performance requirements
- Regression tests: repeat tests after modifications

Kinds of Tests

- Module (unit) tests: for each class in OO software, with subset of tests for each method; isolate causes of errors
- Integration tests: test that components interact properly
- System tests: at the level of the whole system, check if requirements met
- Acceptance tests: check that system meets user/customer needs (validation); done in real environment with real data
- Stress tests: push system to its limits to check that performance degrades gracefully
- Performance tests: checking other performance requirements
- Regression tests: repeat tests after modifications
- (Large area — whole third-year course on testing. Basics only here. See SWEBOK for more.)

How to Test

- Desirable that tests are:
  - Repeatable

How to Test

- Desirable that tests are:
  - Repeatable
  - Documented (both the tests and the results)

How to Test

- Desirable that tests are:
  - Repeatable
  - Documented (both the tests and the results)
  - Precise

How to Test

- Desirable that tests are:
  - Repeatable
  - Documented (both the tests and the results)
  - Precise
  - Done on configuration-controlled software

How to Test

- Desirable that tests are:
  - Repeatable
  - Documented (both the tests and the results)
  - Precise
  - Done on configuration-controlled software
- Ideally, tests should be written at the same time as the requirements — now standard practice

How to Test

- Desirable that tests are:
  - Repeatable
  - Documented (both the tests and the results)
  - Precise
  - Done on configuration controlled software
- Ideally, tests should be written at the same time as the requirements — now standard practice
  - Tests and requirement features can be cross-referenced

How to Test

- Desirable that tests are:
    - Repeatable
    - Documented (both the tests and the results)
    - Precise
    - Done on configuration controlled software
- Ideally, tests should be written at the same time as the requirements — now standard practice
    - Tests and requirement features can be cross-referenced
    - Use cases can suggest tests

How to Test

- Desirable that tests are:
  - Repeatable
  - Documented (both the tests and the results)
  - Precise
  - Done on configuration controlled software
- Ideally, tests should be written at the same time as the requirements — now standard practice
  - Tests and requirement features can be cross-referenced
  - Use cases can suggest tests
- Helps to ensure testability of requirements.

Test-First Development (TFD)

- Basic idea:
  - Write tests informed by requirements before writing the code (but still having requirements!)

Test-First Development (TFD)

- Basic idea:
  - Write tests informed by requirements before writing the code (but still having requirements!)
  - Write code to pass the tests

Test-First Development (TFD)

- Basic idea:
  - Write tests informed by requirements before writing the code (but still having requirements!)
  - Write code to pass the tests
  - Iteratively run tests as code is written

Test-First Development (TFD)

- Basic idea:
  - Write tests informed by requirements before writing the code (but still having requirements!)
  - Write code to pass the tests
  - Iteratively run tests as code is written
- Tests implicitly define the interface and specification of behaviour for the functionality being developed

Test-First Development (TFD)

- Basic idea:
  - Write tests informed by requirements before writing the code (but still having requirements!)
  - Write code to pass the tests
  - Iteratively run tests as code is written
- Tests implicitly define the interface and specification of behaviour for the functionality being developed
  - Bugs found at the earliest possible point

Test-First Development (TFD)

- Basic idea:
  - Write tests informed by requirements before writing the code (but still having requirements!)
  - Write code to pass the tests
  - Iteratively run tests as code is written
- Tests implicitly define the interface and specification of behaviour for the functionality being developed
  - Bugs found at the earliest possible point
  - Bug location is relatively easy

Further Advantages of TFD

- **Clarifies requirements**: trying to write a test often reveals you don't completely understand what the code should do
  - Discover issues more quickly than if coding first
  - Makes coding easier

Further Advantages of TFD

- **Clarifies requirements**: trying to write a test often reveals you don't completely understand what the code should do
  - Discover issues more quickly than if coding first
  - Makes coding easier
- **Avoids poor ambiguity resolution**: if coding first, ambiguities might be resolved based on what's easiest to code, leading to user-hostile software

Further Advantages of TFD

- **Clarifies requirements**: trying to write a test often reveals you don't completely understand what the code should do
  - Discover issues more quickly than if coding first
  - Makes coding easier
- **Avoids poor ambiguity resolution**: if coding first, ambiguities might be resolved based on what's easiest to code, leading to user-hostile software
- **Ensures adequate time for test writing**: if coding first, testing time might be squeezed or eliminated — very risky

Test-Driven Development (TDD)

- A subtly different term from TFD
- In Extreme Programming, detailed tests replace requirements

Test-Driven Development (TDD)

- A subtly different term from TFD
- In Extreme Programming, detailed tests replace requirements
- Disadvantage: communication with stakeholders is affected

Behaviour-Driven Development (BDD)

- More recent term
- Writing use cases in a more stylised language which can be parsed by a machine and at least partially turned into tests

Behaviour-Driven Development (BDD)

- More recent term
- Writing use cases in a more stylised language which can be parsed by a machine and at least partially turned into tests
- Advantages:
  - More interpretable by stakeholders
  - Produce tests automatically

Behaviour-Driven Development (BDD)

- More recent term
- Writing use cases in a more stylised language which can be parsed by a machine and at least partially turned into tests
- Advantages:
    - More interpretable by stakeholders
    - Produce tests automatically
- Disadvantages:
    - Still not ideal for stakeholder communication
    - May go deeper into design/implementation and lose sight of higher-level needs

Evolving Tests When a New Bug Is Identified

- Assume an implementation passes all current tests.

Evolving Tests When a New Bug Is Identified

- Assume an implementation passes all current tests.
- When a new bug is identified by users or code review, good discipline is:
    1. Fix or create a test to catch the bug

Evolving Tests When a New Bug Is Identified

- Assume an implementation passes all current tests.
- When a new bug is identified by users or code review, good discipline is:
    1. Fix or create a test to catch the bug
    2. Check that the test fails

Evolving Tests When a New Bug Is Identified

- Assume an implementation passes all current tests.
- When a new bug is identified by users or code review, good discipline is:
  1. Fix or create a test to catch the bug
  2. Check that the test fails
  3. Fix the bug

Evolving Tests When a New Bug Is Identified

- Assume an implementation passes all current tests.
- When a new bug is identified by users or code review, good discipline is:
  1. Fix or create a test to catch the bug
  2. Check that the test fails
  3. Fix the bug
  4. Run the test that should catch this bug: check it passes

Evolving Tests When a New Bug Is Identified

- Assume an implementation passes all current tests.
- When a new bug is identified by users or code review, good discipline is:
    1. Fix or create a test to catch the bug
    2. Check that the test fails
    3. Fix the bug
    4. Run the test that should catch this bug: check it passes
    5. Rerun all the tests, in case your fix broke something else

Limitations of Testing

- Writing tests is time-consuming

Limitations of Testing

- Writing tests is time-consuming
- Coverage almost always limited: may happen not to exercise a bug

Limitations of Testing

- Writing tests is time-consuming
- Coverage almost always limited: may happen not to exercise a bug
- Difficult/impossible to emulate live environment perfectly
  - e.g. race conditions that appear under real load conditions can be hard to find by testing

Limitations of Testing

- Writing tests is time-consuming
- Coverage almost always limited: may happen not to exercise a bug
- Difficult/impossible to emulate live environment perfectly
  - e.g. race conditions that appear under real load conditions can be hard to find by testing
- Can only test executable things (mainly code, or certain kinds of model) — not high-level design or requirements

Reading

- Essential: SWEBOK v4 Ch 5 — Software Testing
- Essential: Sommerville SE Ch 8
- Suggested: Stevens Ch 19