CS 4500
Software Development

Unit Testing

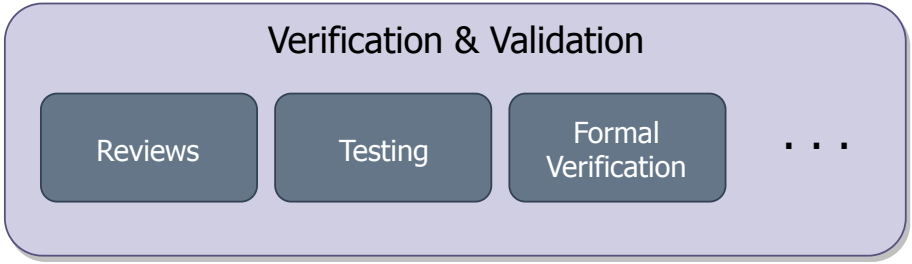
Ferdinand Vesely

October 4, 2019

The Big Picture

Software Verification & Validation

- A wider process – various activities



Software Verification & Validation

Goal

Ensure that the system meets expectations.

Software Verification & Validation

Validation

*Are we building the **right product**?*

- Does the system deliver functionality expected by stakeholders?
- E.g., acceptance testing

Verification

*Are we building the **product right**?*

- Does the system meet its specification?
- Formal verification, testing

Verification: Approaches

Formal Verification

With respect to a (formal) specification:

1. Model checking
   - enumerate all states
   - show that each state satisfies desired properties
2. Deductive verification
   - specification + implementation $\Rightarrow$ proof obligations

Caveat: How do we know the spec is adequate?

Verification: Testing

- Show program behaves as intended
- Discover defects
- Execute program with artificial data
- Check the results: Errors? Anomalies?

Limitations

"Testing can only show the presence of errors, not their absence"

- Dijkstra

Stages

1. **Development Testing**
   - system is tested during development to discover bugs and defects
2. Release Testing
   - complete version of the system before release
   - separate team
3. User Testing
   - Alpha testing
   - Beta testing
   - Acceptance testing

Development Testing

Levels

Unit

- individual program units
- functionality of routines and components

Integration

- gradually integrate components
- test as each new component integrated

System

- test the system as a whole
- closest to user's experience of the system

Unit Testing

- Test individual units in isolation
- Defect testing process:
  - Discover faults or defects
  - Where behavior incorrect / not conforming to spec
  - Success: bug discovered
- Units:
  - Individual functions / methods
  - Modules / object classes

Module / Class Testing

Complete coverage:

- Test all operations associate with a module / object
- Set / interrogate all object attributes
- Exercise all possible states
  - simulate all events which cause a state change

Inheritance

- Information is not localized
- Not enough to test in parent class and assume operation works in subclasses

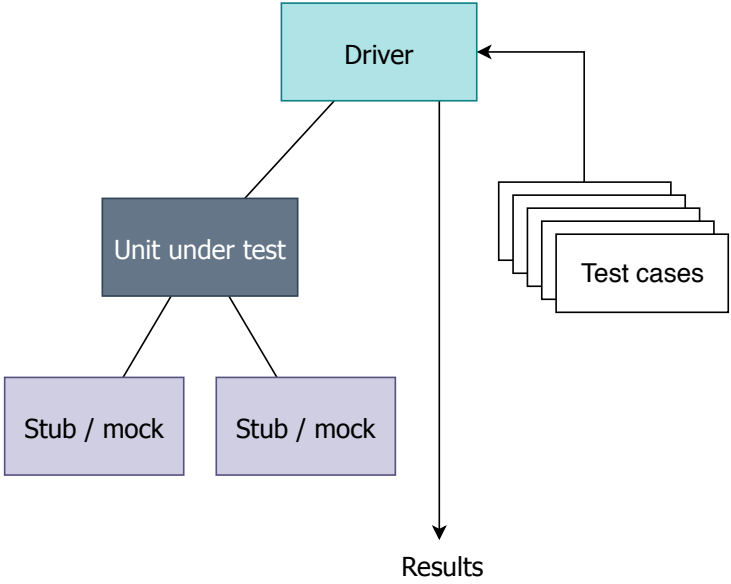
Manual Testing?

- Possible to test manually…
- Usually not necessary at unit level
- Slows down development

Unit Test Automation

- Whenever possible: automate
- Tests run and checked without manual intervention
- Automation frameworks, e.g., the xUnit family

Unit Test Setup



Test Case Execution

1. Setup
   - set up environment / context for the test
2. Test
   - invoke the unit under test
   - check assertion on the result
3. Teardown
   - tear down the environment

Test Case Execution

```
Stack<int> testStack;
...
void testNewStackEmpty() {

  testStack = new Stack<int>();  // setup

  assertTrue(testStack.empty()); // test

  delete testStack;              // teardown
}

void tests() {
  testNewStackEmpty();
  testPushThenPopEmpty();
  ...
}
```

xUnit

- unit test automation frameworks
- simplifies administration of tests
- origin: Smalltalk
- JUnit, unittest (Python), cppunit, OUnit, …

xUnit

Concepts:

1. Fixture / context
   - provides environment for each test
   - set up and teardown
2. Test case
   - executes a scenario
   - checks assertions
3. Test suite
   - collections of test cases with a common fixture
   - order of test cases *should not* matter
4. Runner (driver)
   - run test suites, report results

Mock Objects

```
class EmptyMockStack extends Stack<int> {
  public boolean empty() {
    log.println("empty()");
    return true;
  }

  public void push(int elt) {
    log.println("push(" + elt +")");
  }
  ...
}
```

Choosing Test Cases

- Exhaustive tests for routines – usually not feasible
- E.g., a numeric function with two 32-bit integer arguments: Total number of combinations?

Choosing Test Cases

- Exhaustive tests for routines – usually not feasible
- E.g., a numeric function with two 32-bit integer arguments: Total number of combinations?

$$2^{32} \times 2^{32} = 2^{64} = 18,446,744,073,709,551,616$$

- Do we *need to* test all of those cases?

Choosing Test Cases

- Test cases should:
  (a) show that component does what it's supposed to
  (b) reveal defects if there are any
- Corresponding types of unit test cases:
  (a) exercise / exhibit normal operation
  (b) check problem cases, check abnormal inputs – do they cause a crash

Strategies for Choosing Test Cases

Partition testing

- Identify groups of input with common characteristics
- These should be processed the same way by SUT
- Choose from each group

Guideline-based testing

- Based on previous experience with common errors

Partition Testing

- Inputs and outputs – fall into different classes
- Where members of a class are related – equivalence classes
- Each class – equivalence partition / domain
- Program behaves equivalently for each member of the same class
- Test cases from each partition

Partition Testing

- Consider `bool validPassword(String pass)`
- Should return true if:
  (a) $8 \leq$ `pass.length()` ≤ 15
  (b) contains at least one digit
- Otherwise false
- Exception on non-latin1 characters

Partition Testing

`validPassword` input partitions:

(a) valid: only contains latin1 and

1. both (a) and (b)
2. (b) and `pass.length()` < 8
3. (b) and `pass.length()` > 15
4. (a) and `pass` does not contain a digit
5. `pass.length()` < 8 and does not contain a digit
6. `pass.length()` > 15 and does not contain a digit

(b) invalid:

1. `pass` contains a Latin1 character

Partition Testing

Depending on need and input type:

1. Choose a normal value from each partition (“middle”)
2. Choose boundary values – below and above

E.g.,

- `""` (false),
- `"1234"` (false),
- `"1234567"` (false),
- `"12345678"` (true)

Testing Guidelines

General, e.g.,:

- Choose inputs that force the system to generate all error messages
- Design inputs that cause input buffers to overflow
- Repeat the same input or series of inputs numerous times
- Force invalid outputs to be generated
- Force computation results to be too large or too small

Testing Guidelines

Specific, e.g., for sequences:

1. Test with singleton sequences
   - Programmers sometimes think of sequences as containing more than one value
2. Use different sequences of different sizes in different tests
   - Reduce chance of hiding errors because of accidental characteristics of the input
3. Ensure that first, middle, and last elements of the sequence are accessed
   - Reveals problems at partition boundaries

Property-based Testing

- Generative testing
- Not supplying specific inputs and expected outputs
- Write properties about code
- Engine generates *random* inputs
- Check if properties hold
- Originally: QuickCheck in Haskell
- Java: junit-quickcheck (https://pholser.github.io/junit-quickcheck/)
- Python: Hypothesis (https://hypothesis.readthedocs.io/en/latest/)

QuickCheck in Haskell

For example:

```
prop_commutativeAdd :: Int -> Int -> Bool
prop_commutativeAdd x y = x + y == y + x

prop_reverseReverse :: [Int] -> Bool
prop_reverseReverse xs = reverse (reverse xs) == xs
```

```
> quickCheck prop_commutativeAdd
+++ OK, passed 100 tests.
> quickCheck prop_reverseReverse
+++ OK, passed 100 tests.
```

QuickCheck in Java

```
@RunWith(JUnitQuickcheck.class)
    public class StringProperties {
        @Property public void concatenationLength(String s1, String s2)
            assertEquals(s1.length() + s2.length(),
                         (s1 + s2).length());
        }
    }
```