Verification

Any activity that is undertaken to determine if the system meets its objectives or not.

- Every product should be verified (e.g., code, design documentation, user documentation).
- Every quality should be verified (e.g., behaviour, modifiability, robustness, usability).
- Some qualities or products will not yield yes/no verification results
  - Impossible/difficult to measure (e.g., correctness)
  - Subjective (e.g., modifiability)
- Implicit qualities should be verified.

Approaches to verification

1. Testing
2. Static Analysis (e.g., of design).
3. Symbolic execution
4. Model checking

Testing

Execute the system and observe the behaviour to determine if it is acceptable.

> *"Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." (E. W. Dijkstra)*

The goal of testing is to find bugs.

1. What *test cases* (input values) will be used?
2. How many test will be run?
3. How will we do the testing (testing structure)?
4. How do we know if the behaviour is correct?

Test case selection 1: Black-Box Testing

- Based on externally observable behaviour of a component.
- No reference to implementation.
- Normally divide *input domain* (possible inputs) into *equivalence classes* — sets of inputs for which the future behaviour is the same.
- Choose some test cases from each (or as many as possible) of the equivalence classes.
- Try to choose some where errors are likely (e.g., boundaries of the equivalence classes).
- Assumes that the implementation chooses the same classes.
- Number of classes may be very large.

Test case selection 2: Clear-Box Testing

- Based on examination of code.
- Choose test cases so that all parts of the code are tested.
  - Lines
  - Conditions
  - Paths
- Danger of the tester missing the same cases as the implementer.
- Line coverage is very hard.
- Path coverage is practically impossible.

Test case selection 3: Random Testing

- Randomly choose test cases according to some probability density function (usage profile).
- Typically requires more test cases to find faults.
- May find cases that were overlooked.
- Can be used to estimate reliability (likelihood of fault occurring in practice).
- Validity of reliability is very dependent on the validity of the usage profile.

How many Tests?

- *Exhaustive testing* — Try every possible input.
- Until you're confident that all bugs have been found.
- Until you stop finding bugs.
    - Track rate of fault detection (faults / hour of testing).
    - Set a threshold for acceptable rate.
- As many as you have time for.

How good is Random Testing?

Consider this simple (wrong) program to compare equal length strings:

```
bool stringcmp(string s1, string s2)
{
  bool eq = true;
  unsigned i = 0;
  while (i < s1.length()) {
    eq = (s1[i] == s2[i]);
    i++;
  }
  return(eq);
}
```

What's the probability of finding this error by testing?

Probability of finding bug

$= \Pr(\text{two unequal test strings have the same last character})$
$= 1 - \Pr(\text{strings differ in their last character})^n$
where n is the number of test cases.
Assume random strings from an alphabet of 100 characters.
$= 1 - \frac{99}{100}^n$

n	Pr(detecting error)
1	0.010
5	0.049
10	0.096

So how many test cases to be 99% sure of detecting the error?
$0.99 \geq 1 - \frac{99}{100}^n$
$0.99^n \leq 0.01 \Rightarrow n \geq 459$

Testing Structure 1: Unit Testing

- Test each ‘unit’ (class/module/package) independently.
- If the parts all work then the whole should work.

Bottom-up Test the units at the bottom of the *uses* hierarchy first.

- Requires *driver functions* to call the units.
- Tested units can be used when testing higher level units.

Top-down Test the units at the top of the uses hierarchy first.

- Requires *stub functions*.

Testing Structure 2: Integration Testing

- Test the interaction between components.
- May require driver or stubs on either side.
- Will help find places were developers didn't have the same understanding of the design. (Fix the documents, they're probably ambiguous.)

Testing Structure 3: System Testing

- Test overall system behaviour.
- Very hard to isolate bugs.
- Can only be done late in the process, so cost of fixes is high.
- Typically used for acceptance testing (customer, regulatory body).

Checking Correct Behaviour: Oracles

An *oracle* is a means of determining if the observed behaviour is correct or not.

- Most common form: human observation.
  - Time consuming
  - Expensive
  - Error prone
- Automated oracles — use a program to check.
  - Fast, cheap, accurate.
  - Must be coded somehow (can be generated from spec. if spec. is written formally).
  - Could itself have errors.
- Partial oracles — don't check all required properties.
  - Check those that are easiest to check.
  - Check those that are likely to be source of faults.

Static Analysis

Peer review — ask a colleague to look it over.

Inspection/Walk-through/Structured Review — structured meeting to review each product.

- Reader: leads the review, paraphrases each section
- Recorder: makes detailed notes
- Inspectors: look for problems
- Preparation is essential
- Just find problems, don't fix them
- Emperically shown to be more efficient than testing
- Will find problems that won't be found in testing (e.g., documentation, confusing code, unusual error cases).

Static Analysis (cont'd)

Formal verification — prove that design/implementation satisfies its specification.

- Requires formal specification.
- Requires high level of mathematical precision.
- Only practicable if automated proof checking tools are used.
- Very high effort for any non-trivial system.

Symbolic Execution

- Trace through program using symbolic expressions (i.e., algebraic manipulation) for all variables.
- Represent symbolic execution as symbolic state triple: (variable symbolic values, path, path condition)
- For each step in the program, update the symbolic state
    - Read/input creates new symbol
    - Assignment creates symbolic value expression
    - Conditions add constraints to path condition.
- Must consider all possbile paths.

Model Checking

Construct (usually automatically) a (finite state) model of the design, use tools to check properties, for example:

- non-reachability of error/failure states
- absence of deadlock
- reachability of desired states (liveness)
- properties on sequences of states, e.g.,
  - forall next states/exists next state satisfying P
  - In every/a sequence from S, P_1 holds until P_2 holds
  - In every/a sequence from S, P_1 will eventually hold

Verification Artifacts

- Verification plan — what will you do to verify the system?
  - Outline of test cases. (e.g., "$0 < \texttt{spin} < 45$")
  - Structure of testing.
  - Outline of harnesses/stubs required (consider JUnit).
- Verification report
  - Actual test cases. (e.g., "$\texttt{spin} = 37$")
  - pass / fail for each test.
  - Test Harness/stubs