Testing & Test-cases

Software Engineering for Dependable Systems

Tomas Bures
bures@d3s.mff.cuni.cz
Testing
Validation and Verification (V&V)

- **Validation**: Building the right product.
  - Does the software meet the expectations of the customer?

- **Verification**: Building the product right.
  - Does the software conform to its specification?

- **When to check quality:**
  - In some software development processes, V&V is done as early as possible (e.g., prototyping, agile).

- It is understood that problems discovered early are easier and less expensive to fix.

- However, there are parts of the specification that can be checked only when the system is ready to be deployed.
Functional and Nonfunctional Properties

- Functional properties are related to what a system (or a part of it) is supposed to do.
  - Use cases, use-case diagrams (UML)

- Nonfunctional (or extrafunctional) properties are related to how the system carries out an operation.
  - Performance; e.g., response time or throughput.
  - Security.
  - Availability; e.g., uptime 99.999%.
  - Some nonfunctional properties are more difficult to check during early stages of the development process.
Product Qualities

- Internal qualities
  - Maintainability, extensibility, portability, testability, ...

- External qualities
  - usefulness qualities:
    - usability, performance, security, interoperability
  - dependability
    - correctness, reliability, safety, robustness
Dependability Qualities

• Correctness:
  ▣ A program is correct if it is consistent with its specification
    • seldom practical for non-trivial systems

• Reliability:
  ▣ likelihood of correct function for some “unit” of behavior
    • relative to a specification and usage profile
    • statistical approximation to correctness (100% reliable = correct)

• Safety:
  ▣ preventing hazards

• Robustness
  ▣ acceptable (degraded) behavior under extreme conditions
Example of Dependability Qualities

- **Correctness, reliability:** let traffic pass according to correct pattern and central scheduling

- **Robustness, safety:** Provide degraded function when possible; never signal conflicting greens.
  - Blinking red / blinking yellow is better than no lights; no lights is better than conflicting greens

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Tools for Validation and Verification

- **Software inspection** analyses requirement documents, designs, and source code (the latter, often automatically)
  - It is a **static** method: It does not require an executable artefact, hence it can be applied throughout all the stages of software development.

- **Software testing** uses an executable representation of the system
  - It is a **dynamic** method: The product is exercised with test input data
  - The resulting output is checked against the specification.
  - If there is no agreement, an error is found which must be fixed.
  - Different forms according to the knowledge assumed for the system under study: black-box or white-box.
V&V and the Development Process

Software inspections

- Requirements specification
- High-level design
- Formal specification
- Detailed design
- Program

Prototype

Program testing

Figure taken from http://www.cs.st-andrews.ac.uk/ifs/Books/SE7/Presentations/index.html
Important Point

- Software inspections can only check the agreement between a program and its specification.
- They cannot show that the software is operationally useful.
- Nor can they check nonfunctional properties (but may give hints).
- Software testing can only detect errors, not prove their absence.
- Testing all possible execution paths for nontrivial programs is impossible.
- They are not competing techniques, rather they are complementary.
Related Activity: Debugging

- Defect testing and debugging are distinct processes.
- Verification and validation is concerned with establishing the existence of defects in a program.
- Debugging is concerned with locating and repairing these errors.
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.
The Debugging Process

Key activity: regression testing

- Re-run the tests (or a subset of them) after a problem is fixed.
- It is not uncommon that a fix introduces errors elsewhere!
Software Qualities and Process

- Qualities cannot be added after development
  - Quality results from a set of inter-dependent activities
  - Analysis and testing are crucial but far from sufficient.

- Testing is not a phase, but a lifestyle
  - Testing and analysis activities occur from early in requirements engineering through delivery and subsequent evolution.
  - Quality depends on every part of the software process

- An essential feature of software processes is that software test and analysis is thoroughly integrated and not an afterthought
<table>
<thead>
<tr>
<th>Requirements Elicitation</th>
<th>Requirements Specification</th>
<th>Architectural Design</th>
<th>Detailed Design</th>
<th>Unit Coding</th>
<th>Integration &amp; Delivery</th>
<th>Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify qualities</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plan acceptance test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plan system test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plan unit &amp; integration test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitor the A&amp;T process</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Planning &amp; monitoring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation of specs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate specifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyze architectural design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inspect architectural design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inspect detailed design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code inspection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generation of tests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate system test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate integration test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate unit test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate regression test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update regression test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generation of tests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate regression test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update regression test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test case execution and SW validation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collect data on faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analyze faults and improve the process</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For instance, in an object-oriented design:

classes → components → overall system
Structure of a Software Test Plan

- Testing process
- Requirements traceability
  - Tests should cover at least all the requirements provided by the users.
- Tested items
  - Complete coverage of all artefacts is in general very difficult (too expensive). Items to be tested should be listed here.
- Testing schedule
- Test recording procedures
  - Results must be recorded to give the possibility of checking later whether tests have been done correctly.
- Hardware and software requirements
- Constraints
  - For example, staff shortages, deadlines, . . .
Software Inspections

• Empirical studies have shown that they are effective in detecting large amounts of errors in software.

• Many errors may be detected in a single inspection.
  ▪ Recall, it is a static methods which does not require a running system.
  ▪ With software testing, usually only one defect at a time may be discovered: the system usually crashes when an error occurs.

• They reuse domain and programming language knowledge: reviewers are likely to have seen the types of error that commonly arise.
Program Inspection

- It is a formal methodology for reviewing documents.
- It looks for defects such as logical errors, anomalies in the code, or non-compliance with standards.
- The process may have different variants according to the organisation in which it is performed.

Typical pre-conditions
- Availability of a precise specification.
- Availability of syntactically correct code (or design).
- An error check-list.
  - This is dependent on the programming language.
  - The weaker the typing, the longer the list.
Composition of the Reviewing Team

- Author
  - Responsible for fixing defects discovered during the review.

- Inspector

- Reader
  - Paraphrases the code during an inspection meeting.

- Scribe
  - Records the outcome of the inspection meeting.

- Moderator
  - Manages the process. Responsible for scheduling possible follow-up meetings.
The Program Inspection Process

- Planning is the responsibility of the moderator: choose a team, fix dates, ...
- At the overview the author presents the program under inspection.
- At the inspection meeting errors are reported. Meetings should be kept relatively short (e.g., under 2 h).
- Rework is the author's responsibility.
- Follow-up may be needed to assess the code in case of major changes required.

Figure taken from http://www.cs.st-andrews.ac.uk/ifs/Books/SE7/Presentations/index.html
Typical Checks

- Data faults
  - Base indices for arrays? Possibility of buffer overflows?

- Control faults
  - For each conditional statement, is the condition correct? Are loops guaranteed to terminate? Are compound statements correctly bracketed?

- Input/output faults
  - Are all input variables used? Are output variables used? Can unexpected inputs cause corruption (e.g., null pointers)?

- Exception management
  - Have all possible error conditions been taken into account?
Automated Static Analysis (for code)

- Performed by software tools which process the source code in search of potentially dangerous situations.
  - E.g. FindBugs

- Does not replace program inspection by humans, as it checks for more mechanical errors:
  - Variables used before initialisations, variables declared but never used, variables never used between two successive assignments.
  - Unreachable code.
  - Return values of functions/methods that are not used.

- Static analysers are typically available in Integrated Development Environments.

- Much more useful for weakly typed languages.
Software Testing

- Component (or unit) testing
  - Testing of individual program components. The notion of component depends on the programming language under consideration.
  - Usually under the responsibility of the authors.
  - Tests are based on the developers' experience.

- System testing
  - Testing of integrated components that form a (sub-)system.
  - Usually under the responsibility of an independent team.
  - Tests are based on a system specification.
Goals of Software Testing

- **Validation testing**
  - Demonstrates that the software meets the requirements.
  - It is successful when the system operates as intended.
  - The system is exercised using typical input data.
  - Does not reveal the absence of faults though!

- **Defect testing**
  - Discover faults that may lead to unintended behaviour or failure.
  - It is successful when the test makes the system perform incorrectly.
  - Reveals the presence, not the absence of faults!

- **Guidelines on what to test**
  - Functionality accessed from menus.
  - Combinations of functions accessed through the same menu (e.g., text formatting).
  - User input forms with correct and incorrect input.
Sources of Test Obligations

- **Functional (black box, specification-based):** from software specifications
  - Example: If spec requires robust recovery from power failure, test obligations should include simulated power failure

- **Structural (white or glass box):** from code
  - Example: Traverse each program loop one or more times.

- **Model-based:** from model of system
  - Models used in specification or design, or derived from code
  - Example: Exercise all transitions in communication protocol model

- **Fault-based:** from hypothesized faults (common bugs)
  - Example: Check for buffer overflow handling (common vulnerability) by testing on very large inputs
Functional testing

- Functional testing: Deriving test cases from program specifications
  - Functional refers to the source of information used in test case design, not to what is tested

- Also known as:
  - specification-based testing (from specifications)
  - black-box testing (no view of the code)

- Functional specification = description of intended program behavior
  - either formal or informal
Black-Box Testing

- The system (or component) is treated as a black box.
- Behaviour understood by relating inputs to outputs.
- It is only concerned with the functionality, not its actual implementation.

Figure taken from http://www.cs.st-andrews.ac.uk/ifs/Books/SE7/Presentations/index.html
Why functional testing?

- The base-line technique for designing test cases
  - Timely
    - Often useful in refining specifications and assessing testability before code is written
  - Effective
    - finds some classes of fault (e.g., missing logic) that can elude other approaches
  - Widely applicable
    - to any description of program behavior serving as spec
    - at any level of granularity from module to system testing.
  - Economical
    - typically less expensive to design and execute than structural (code-based) test cases
Early functional test design

- Program code is not necessary
  - Only a description of intended behavior is needed
  - Even incomplete and informal specifications can be used
    - Although precise, complete specifications lead to better test suites

- Early functional test design has side benefits
  - Often reveals ambiguities and inconsistency in spec
  - Useful for assessing testability
    - And improving test schedule and budget by improving spec
  - Useful explanation of specification
    - or in the extreme case (as in XP), test cases are the spec.
Functional vs. structural: Classes of faults

- Different testing strategies (functional, structural, fault-based, model-based) are most effective for different classes of faults

- Functional testing is best for missing logic faults
  - A common problem: Some program logic was simply forgotten
  - Structural (code-based) testing will never focus on code that isn’t there!

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Functional vs. structural: Granularity levels

- Functional test applies at all granularity levels:
  - Unit (from module interface spec)
  - Integration (from API or subsystem spec)
  - System (from system requirements spec)
  - Regression (from system requirements + bug history)

- Structural (code-based) test design applies to relatively small parts of a system:
  - Unit
  - Integration
Steps: From specification to test cases

1. Decompose the specification
   - If the specification is large, break it into independently testable features to be considered in testing

2. Select representatives
   - Representative values of each input, or
   - Representative behaviors of a model
     - Often simple input/output transformations don’t describe a system. We use models in program specification, in program design, and in test design

3. Form test specifications
   - Typically: combinations of input values, or model behaviors

4. Produce and execute actual tests
Systematic vs. Random Testing

• Random (uniform):
  - Pick possible inputs uniformly
  - Avoids designer bias
    • A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
  - But treats all inputs as equally valuable

• Systematic (non-uniform):
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are apt to fail often or not at all

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Why Not Random?

- Non-uniform distribution of faults
- Example: Java class “roots” applies quadratic equation

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

- Incomplete implementation logic: Program does not properly handle the case in which \( b^2 - 4ac = 0 \) and \( a = 0 \)
- Failing values are sparse in the input space — needles in a very big haystack. Random sampling is unlikely to choose \( a = 0 \) and \( b = 0 \)
Functional testing: exploiting specification

- Functional testing is systematic testing
- Functional testing uses the specification (formal or informal) to partition the input space
  - E.g., specification of “roots” program suggests division between cases with zero, one, and two real roots
- Test each category, and boundaries between categories
  - No guarantees, but experience suggests failures often lie at the boundaries (as in the “roots” program)
Partitioning

- Selecting relevant input data for testing.
- Based on the assumption that some inputs are somewhat similar: if one is troublesome, so will be all the others belonging to the same class.
- Example:

```java
class Account {
    public float getBalance() { ... }
    public void withdraw(float amount) { ... }
}
```

- Partition the floats into:
  - Negative values
  - Zero
  - Positive values:
    - < getBalance()
    - == getBalance()
    - > getBalance()
  - Another dimension: more than two decimal digits!
Other General Testing Guidelines

- Design inputs that cause buffers to overflow
- Force invalid outputs to be generated
- Force computation results to be too large or too small
Structural Testing

- Also called white-box testing.
- Test cases are inferred from the program structure, which is required to be known.
- Can be done incrementally, knowledge of the program can be used to add further test cases.
- The objective is to test all program statements (not all path combinations).

Figure taken from http://www.cs.st-andrews.ac.uk/ifs/Books/SE7/Presentations/index.html
Why structural (code-based) testing?

- One way of answering the question “What is missing in our test suite?”
  - If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
  - But what’s a “part”?
    - Typically, a control flow element or combination:
      - Statements (or CFG nodes), Branches (or CFG edges)
      - Fragments and combinations: Conditions, paths

- Complements functional testing: Another way to recognize cases that are treated differently
  - Recall fundamental rationale: Prefer test cases that are treated differently over cases treated the same

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
No guarantees

- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure

- What is the value of structural coverage?
  - Increases confidence in thoroughness of testing
    - Removes some obvious inadequacies
Structural testing complements functional

• Control flow testing includes cases that may not be identified from specifications alone
  ▪ Typical case: implementation of a single item of the specification by multiple parts of the program
  ▪ Example: hash table collision (invisible in interface spec)

• Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  ▪ Typical case: missing path faults
Structural testing in practice

- Create functional test suite first, then measure structural coverage to identify see what is missing

- Interpret unexecuted elements
  - may be due to natural differences between specification and implementation
  - or may reveal flaws of the software or its development process
    - inadequacy of specifications that do not include cases present in the implementation
    - coding practice that radically diverges from the specification
    - inadequate functional test suites

- Attractive because automated
  - coverage measurements are convenient progress indicators
  - sometimes used as a criterion of completion
    - use with caution: does not ensure effective test suites
Statement testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

- Coverage:
  \[
  \frac{\text{#executed statements}}{\text{#statements}}
  \]

- Rationale: a fault in a statement can only be revealed by executing the faulty statement
```java
String decode(String encoded) {
    int encIdx = 0;
    int encLen = encoded.length();
    StringBuilder decoded = new StringBuilder();
    while (encIdx < encLen) {
        char c = encoded.charAt(encIdx++);
        if (c == '+') {
            decoded.append(' ');
        } else {
            if (c == '%') {
                int digitHigh = charTable.getValueOfHex(encoded, encIdx++);
                int digitLow = charTable.getValueOfHex(encoded, encIdx++);
                if (digitHigh == -1 || digitLow == -1) {
                    return null;
                } else {
                    decoded.append((char)(16 * digitHigh + digitLow));
                }
            } else {
                decoded.append(c);
            }
        }
    }
    return decoded.toString();
}
```

Example

\[ T_0 = \{"", "test", "test+case%1Dadequacy"\} \]
14/15 = 93% statement coverage

\[ T_1 = \{"adequate+test%0Dexecution%7U"\} \]
15/15 = 100% statement coverage

\[ T_2 = \{"%3D", "%A", "a+b", "test"\} \]
15/15 = 100% statement coverage
Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to basic block coverage or node coverage
  - Difference in granularity, not in concept

- No essential difference
  - 100% node coverage \(\iff\) 100% statement coverage
    - but levels will differ below 100%
  - A test case that improves one will improve the other
    - though not by the same amount, in general

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Coverage is not size

- Coverage does not depend on the number of test cases
  
  $T_0, T_1: T_1 >_{\text{coverage}} T_0 \quad \quad T_1 <_{\text{cardinality}} T_0$
  
  $T_1, T_2: T_2 =_{\text{coverage}} T_1 \quad \quad T_2 >_{(\text{cardinality})} T_1$

- Minimizing test suite size is seldom the goal
  
  - small test cases make failure diagnosis easier
  
  - a failing test case in $T_2$ gives more information for fault localization than a failing test case in $T_1$
String decode(String encoded) {
    int encIdx = 0;
    int encLen = encoded.length();
    StringBuilder decoded = new StringBuilder();
    while (encIdx < encLen) {
        char c = encoded.charAt(encIdx);
        if (c == '+') {
            decoded.append(' ');
        } else {
            if (c == '%') {
                int digitHigh = charTable.getValueOfHex(encoded, encIdx);
                int digitLow = charTable.getValueOfHex(encoded, encIdx);
                if (digitHigh == -1 || digitLow == -1) {
                    return null;
                } else {
                    decoded.append((char)(16 * digitHigh + digitLow));
                }
            } else {
                decoded.append(c);
            }
        }
    }
    return decoded.toString();
}

"All statements" can miss some cases

Complete statement coverage may not imply executing all branches in a program
Example:
- Suppose block C were missing
- Statement adequacy would not require true branch from B to A

T₃ = {"", "a%0D%4J"}
100% statement coverage
But no true branch from B
Adequacy criterion: each branch (edge in the CFG) must be executed at least once

Coverage:

\[
\frac{\text{#executed branches}}{\text{#branches}}
\]

- **T3** = {“”, “a%0D%4J”}
  - 100% Stmt Cov. 88% Branch Cov. (7/8 branches)

- **T2** = {“%3D”, “%A”, “a+b”, “test”}
  - 100% Stmt Cov. 100% Branch Cov. (8/8 branches)

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Statements vs. branches

- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true (see $T_3$)
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)
“All branches” can still miss conditions

- Sample fault: missing operator (negation)
  
  \[
  \text{digit\_high} == 1 \; || \; \text{digit\_low} == -1
  \]

- Branch adequacy criterion can be satisfied by varying only `digit\_low`
  - The faulty sub-expression might never determine the result
  - We might never really test the faulty condition, even though we tested both outcomes of the branch

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Condition testing

- Branch coverage exposes faults in how a computation has been decomposed into cases
  - intuitively attractive: check the programmer’s case analysis
  - but only roughly: groups cases with the same outcome

- Condition coverage considers case analysis in more detail
  - also individual conditions in a compound Boolean expression
    - e.g., both parts of digit_high == 1 || digit_low == -1

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Basic condition testing

- Adequacy criterion: each basic condition must be executed at least once

- Coverage:

  \[
  \frac{\text{#truth values taken by all basic conditions}}{2 \times \text{# basic conditions}}
  \]
Basic conditions vs branches

- Basic condition adequacy criterion can be satisfied without satisfying branch coverage

\[ T_4 = \{ \text{"first+test\%9Ktest\%K9"} \} \]
- satisfies basic condition adequacy
- does not satisfy branch condition adequacy

- Branch and basic condition are not comparable
  - neither implies the other

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Covering branches and conditions

- Branch and condition adequacy:
  - cover all conditions and all decisions

- Compound condition adequacy:
  - Cover all possible evaluations of compound conditions
  - Cover all branches of a decision tree

```
digit_high == -1
  true
  digit_low == 1
    true
    false
    FALSE
  false
  FALSE
```

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
Compound conds.: exponential complexity

Short-circuit evaluation often reduces this to a more manageable number, but not always.

\[((a \lor b) \land c) \lor d \land e\]

<table>
<thead>
<tr>
<th>Test Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>—</td>
<td>T</td>
<td>—</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>—</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>—</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>F</td>
<td>—</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>—</td>
<td>T</td>
<td>—</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>—</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>—</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>F</td>
<td>—</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>—</td>
<td>F</td>
<td>F</td>
<td>—</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>—</td>
</tr>
<tr>
<td>13</td>
<td>F</td>
<td>F</td>
<td>—</td>
<td>F</td>
<td>—</td>
</tr>
</tbody>
</table>
Modified condition/decision (MC/DC)

- Motivation: Effectively test important combinations of conditions, without exponential blowup in test suite size
  - “Important” combinations means: Each basic condition shown to independently affect the outcome of each decision

- Requires:
  - For each basic condition C, two test cases,
  - values of all evaluated conditions except C are the same
  - compound condition as a whole evaluates to true for one and false for the other

Taken from http://ix.cs.uoregon.edu/~michal/book/ (© Mauro Pezzè & Michal Young)
**MC/DC: linear complexity**

\[((a \mid b) \&\& c) \mid d) \&\& e\]

<table>
<thead>
<tr>
<th>Test Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>13</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

- N+1 test cases for N basic conditions
- Red values independently affect the output of the decision
- Required by the RTCA/DO-178B standard
Comments on MC/DC

- MC/DC is
  - basic condition coverage
  - branch coverage
  - plus one additional condition: every condition must independently affect the decision’s output

- It is subsumed by compound conditions and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage

- A good balance of thoroughness and test size (and therefore widely used)
Path Testing

- Ensures that each test input covers a different path in the control flow of the system.

- May use a high-level representation with a graph where nodes represent statements, and arcs denote the flow of control.

- Exhaustive path coverage may be expensive to guarantee in realistic scenarios.
Path Testing

Figure taken from http://www.cs.st-andrews.ac.uk/ifs/Books/SE7/Presentations/index.html