

Software Quality

Lecture 8

Ruzanna Chitchyan, Jon Bird, Pete Bennett
TAs: Alex Elwood, Alex Cockrean, Casper Wang

Overview

- Software quality and how to get to it
- Test-driven development
 - White box testing
 - Black box testing

Software Quality

Why is Software Quality relevant?

- Reputation
- Cost of Product and Maintenance
- Software Certification
- Organizational Certification
- Legality
- Moral/ethical codes of practice

Software Quality is Multi-dimensional

- Subjective or “fitness for use”: as perceived by an individual user (e.g., aesthetics of GUI, missing functionality...)
- Objective or “conformance to requirements”: can be measured as a property of the product (e.g., detailed documentation, number of bugs, compliance with regulations)
- Practical: what does it mean to your team and your clients?

Quality Models: ISO/IES25010

- **Functional Suitability**

- Functional Completeness
- Functional Correctness
- Functional Appropriateness

- **Performance Efficiency**

- Time Behaviour
- Resource Utilisation
- Capacity

- **Compatibility**

- Co-existence
- Interoperability

- **Usability**

- Appropriateness
- Realisability
- Learnability
- Operability
- User Error Protection
- User Interface Aesthetics
- Accessibility

- **Reliability**

- Maturity
- Availability
- Fault Tolerance
- Recoverability

- **Security**

- Confidentiality
- Integrity
- Non-repudiation
- Authenticity
- Accountability

- **Maintainability**

- Modularity
- Reusability
- Analysability
- Modifiability
- Testability

- **Portability**

- Adaptability
- Installability
- Replaceability

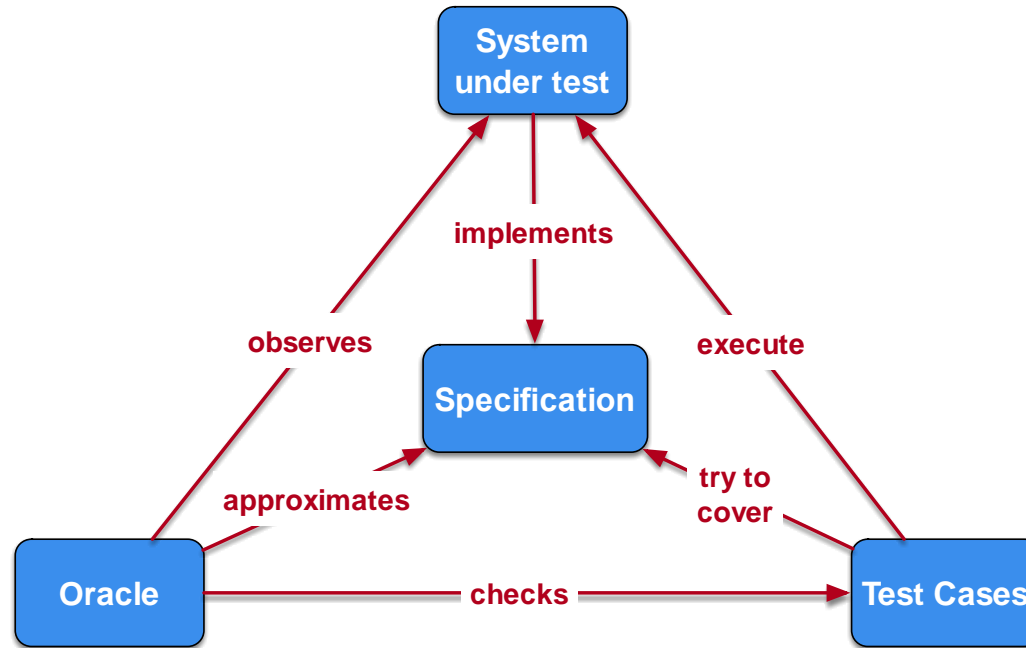
Steps Towards Software Quality:

- Use a standard development process
- Use a coding standard
 - Compliance with industry standards (e.g., ISO, Safety, etc.)
 - Consistent code quality
 - Secure from start
 - Reduce development costs and accelerate time to market
- Define and monitor metrics (defect metrics and complexity metrics)
 - High complexity leads to higher number of defects
- Identify and remove defects
 - Conduct manual reviews
 - Use Testing

Testing

Mauro Pezze and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

Testing process: key elements and relationships



From: M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Software Engineering (ICSE)*, 2011 33rd International Conference on, pages 391–400. IEEE, 2011.

Testing: White Box

Mauro Pezze and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

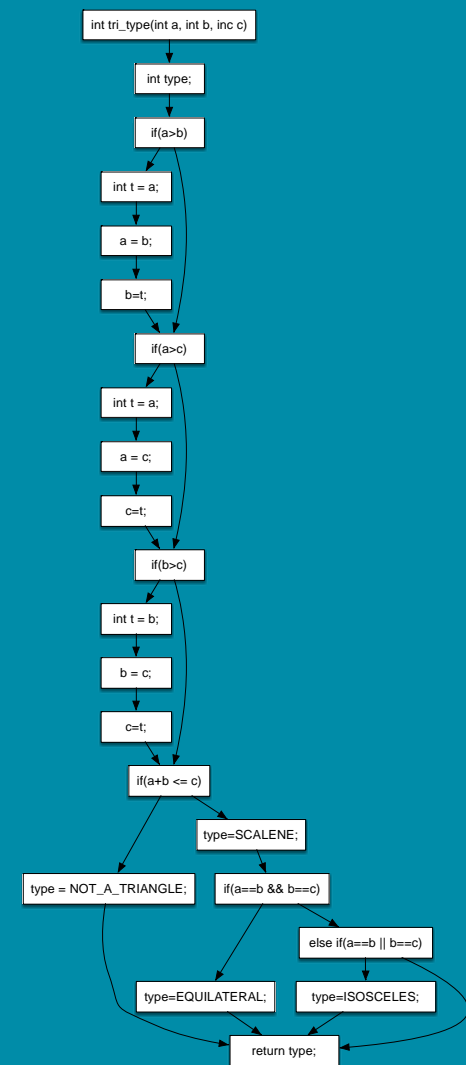
White Box Testing

- Access to software “internals”:
 - Source code
 - Runtime state
 - Can keep track of executions.
- White box testing exploits this to
 - Use code to measure coverage
 - Many different ways
 - Drive generation of tests that maximise coverage

```
1  int tri_type(int a, int b, int c) {
2      int type;
3      if (a > b)
4          { int t = a; a = b; b = t; }
5      if (a > c)
6          { int t = a; a = c; c = t; }
7      if (b > c)
8          { int t = b; b = c; c = t; }
9      if (a + b <= c)
10         type = NOT_A_TRIANGLE;
11     else {
12         type = SCALENE;
13         if (a == b && b == c)
14             type = EQUILATERAL;
15         else if (a == b || b == c)
16             type = ISOSCELES;
17     }
18     return type;
19 }
```

White Box Testing

- Access to software “internals”:
 - Source code
 - Runtime state
 - Can keep track of executions.
- White box testing exploits this to
 - Use code to measure coverage
 - Many different ways
 - Drive generation of tests that maximise coverage.



White-Box Testing

- Coverage Metrics:
 - Statement coverage
 - Branch coverage
 - Def-Use or Dataflow coverage
 - MC/DC (Modified Condition / Decision Coverage)
 - Mutation coverage...
- Prescribed metrics, e.g., DO178-B/C standard for civilian aircraft software
 - non-critical - statement coverage
 - safety-critical - MC/DC coverage

Statement Coverage

- Test inputs should collectively have executed each statement
- If a statement always exhibits a fault when executed, it will be detected
- Computed as:

$$\text{Coverage} = \frac{|\text{Statements executed}|}{|\text{Total statements}|}$$



Branch Coverage

- Test inputs should collectively have executed each branch
- Subsumes statement coverage
- Computed as:

$$\text{Coverage} = \frac{|\text{Branches executed}|}{|\text{Total branches}|}$$

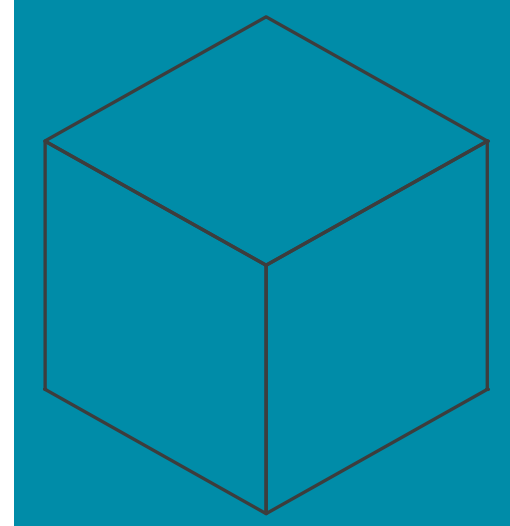


Testing: Black Box

Mauro Pezze and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

Black Box Testing

- No access to “internals”
 - May have access, but don’t want to
- We know the interface
 - Parameters
 - Possible functions / methods
- We may have some form of specification document



Testing Challenges

- Many different types of input
- Lots of different ways in which input choices can affect output
- An almost infinite number of possible inputs & combinations

Equivalence Partitioning (EP) Method

Identify tests by analysing the program interface

1. Decompose program into “functional units”
2. Identify inputs / parameters for these units
3. For each input
 - a) Identify its limits and characteristics
 - b) Define “partitions” - value categories
 - c) Identify constraints between categories
 - d) Write test specification

Example – Generate Grading Component

The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:

greater than or equal to 70 - 'A'

greater than or equal to 50, but less than 70 - 'B'

greater than or equal to 30, but less than 50 - 'C'

less than 30 - 'D'

Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.

EP – 1. Decompose into Functional Units

- Dividing into smaller units is good practice
 - Possible to generate more rigorous test cases.
 - Easier to debug if faults are found.
- E.g.: dividing a large Java application into its core modules / packages
- Already a functional unit for the Grading Component example

EP – 2. Identify Inputs and Outputs

- For some systems this is straightforward
 - E.g., the Triangle program:^[L]_[SEP]
 - Input: 3 numbers,
 - Output: 1 String
 - E.g., Grading Component
 - Input: 2 integers: exam mark and coursework mark
 - Output: 1 String for grade
- For others less so. Consider the following:
 - A phone app.
 - A web-page with a flash component.

EP – 3.a Identify Categories

Category	Description
Valid	valid exam mark
	valid coursework mark
	valid total mark
Invalid	invalid exam mark
	invalid coursework mark
	Invalid total mark

EP: 3.b Define “Partitions” - value categories

- Significant value ranges / value-characteristics of an input

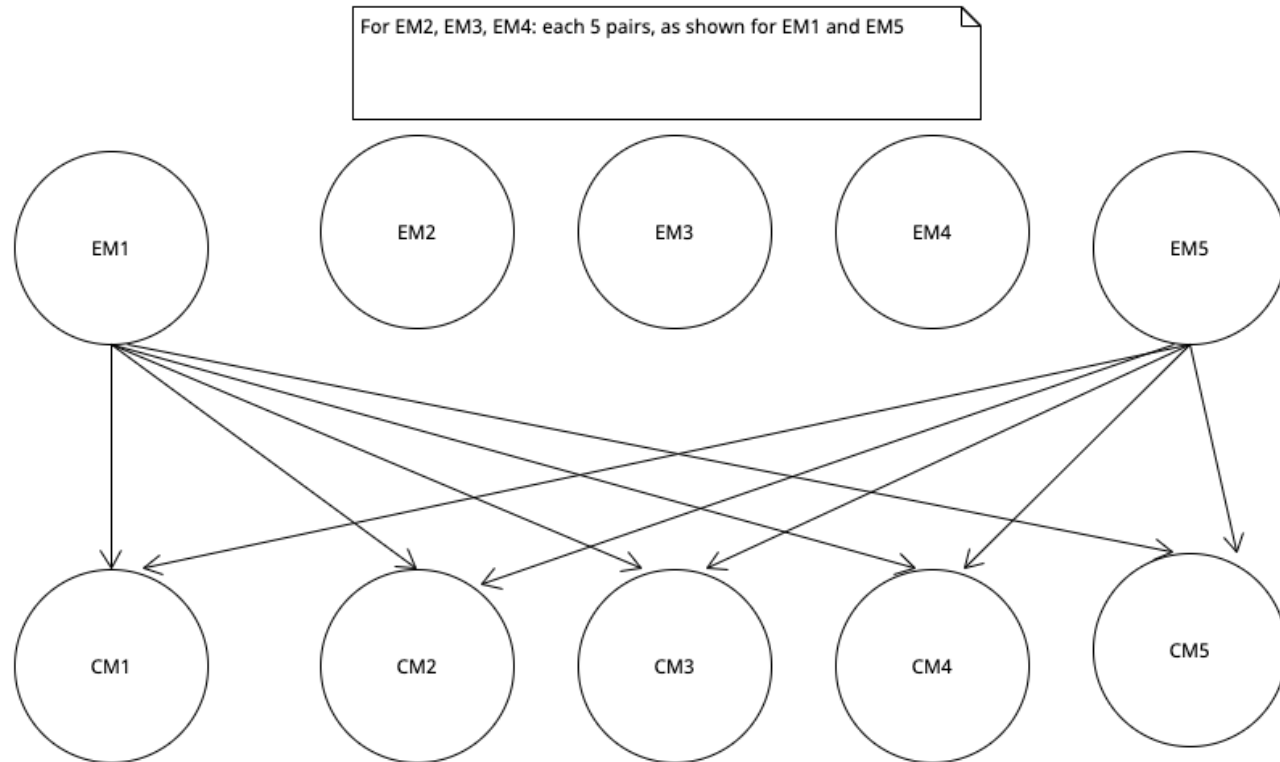
Category	Description	Partition
Valid	EM_1 valid exam mark	$0 \leq \text{Exam mark} \leq 75$
	CM_1 valid coursework mark	$0 \leq \text{Coursework mark} \leq 25$
Invalid	EM_2 invalid exam mark	Exam mark > 75
	EM_3 invalid exam mark	Exam mark < 0
	EM_4 invalid exam mark	alphabetic
	EM_5 invalid exam mark	Other real number (outside of EM_1)
	CM_2 invalid coursework mark	Coursework mark > 25
	CM_3 invalid coursework mark	Coursework mark < 0
	CM_4 invalid coursework mark	alphabetic
	CM_5 invalid coursework mark	Other real number (outside of CM_1)

EP – 3. c Identify Constraints between Categories

- Not all categories can combine with each other

Category		Condition
valid exam mark	EM_1	$0 \leq \text{Exam mark} \leq 75$
invalid exam mark	EM_2	Exam mark > 75
invalid exam mark	EM_3	Exam mark < 0
invalid exam mark	EM_4	alphabetic
invalid exam mark	EM_5	Other real number
valid coursework mark	CM_1	$0 \leq \text{Coursework mark} \leq 25$
invalid coursework mark	CM_2	Coursework mark > 25
invalid coursework mark	CM_3	Coursework mark < 0
invalid coursework mark	CM_4	alphabetic
invalid coursework mark	CM_5	Other real number

EP – 3. d Write Test Specifications



Example: Inputs and Expected Outputs

The test cases corresponding to partitions derived from the input exam mark are:

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	$0 \leq e \leq 75$	$e < 0$	$e > 75$
Exp. Output	'B'	'FM'	'FM'

Boundary Values

- Most frequently errors occur in "edge" cases
 - Test just under boundary value
 - Test just above the boundary value
 - Test the boundary value

How do we go about using this?

- Testing applied in Java: Use JUnit
 - uses “Assertions” to test the code
 - Allow us to state what *should* be the case
 - If assertions do not hold, JUnit’s logging mechanisms reports failures
 - Various types of assertion are available, e.g., assertEquals(expected, actual); assertTrue(condition); assertFalse(condition); assertThat (value, matchingFunction)

Review

- What is Software Quality?
- What are key elements and relationships for test specifications?
- How do we carry out white-box testing?
- How do we carry out black-box testing?

