Weekly Homework 10

Miriam Gaetano, Yevgeniy Terentyev
Softwaretechnik

June 24, 2025

10-1

a) Difference between Verification and Validation

- **Verification:** Are we building the product **right**?
  This checks whether the software has been correctly developed according to technical specifications using unit tests and code reviews during development.
- **Validation:** Are we building the **right** product?
  This checks whether the software meets customer or real-world requirements at the end of development through user testing.

b) What is a test case? When is it successful?

A test case consists of:

- **Input:** data provided to execute the test,
- **Preconditions:** conditions that must be true before execution,
- **Expected result:** what the system is expected to do,
- **Execution context:** the environment or state in which the test is run.

A test case is considered successful if the actual result matches the expected result.

c) Relationship between failure, error, and defect

- **Defect (or bug):** an error in the source code,
- **Error:** an incorrect program state caused by a defect,
- **Failure:** a visible incorrect behavior.

Defect → Error → Observable Failure

d) Similarities and differences between the following pairs

- **Structural testing vs Code review:** Structural testing executes the code; code review is only a manual check. Both aim to improve quality.
- **Load testing vs Stress testing:** Load testing simulates realistic heavy usage, stress testing pushes the system beyond its limits.
- **Testing vs Debugging:** Testing finds errors, debugging locates and fixes them.
- **Functional testing vs Acceptance testing:** Functional testing checks technical behavior, acceptance testing checks whether user needs are met.
- **Top-Down vs Bottom-Up:** Top-Down tests high-level modules first, Bottom-Up starts from the low-level ones.

10-2

`classifyTriangle(int side1, int side2, int side3)`
This function determines whether a triangle is equilateral, right-angled, isosceles, or normal.

a) OCL Precondition for classifyTriangle

What conditions must be true before the function can be executed?
Three positive integers must be provided, and the sum of any two sides must be greater than the third.

```
context Math::classifyTriangle(side1: Integer, side2: Integer, side3
   : Integer)
pre:
  side1 > 0 and side2 > 0  and side3 > 0
  and side1 + side2 > side3
  and side2 + side3 > side1
  and side1 + side3 > side2
```

b) Black-box Testing: Functional Testing

Create at least 7 test cases based on only the interface description of the function.
Consider different cases where you expect different behaviors of the system.

Black-box testing is a technique where we do not look at the code but we rely only on what the function should do

Input (s1, s2, s3)	Expected Type	Reason
(3, 3, 3)	Equilateral	All sides equal
(3, 4, 5)	Rectangle	Right triangle ($3^2 + 4^2 = 5^2$). Pythagoras.
(5, 5, 3)	Isosceles	Two equal sides. Base case for isosceles.
(4, 6, 5)	Normal	No equal sides
(0, 5, 5)	Error	A side is zero. cannot exist. Invalid input test.
(-1, 4, 4)	Error	A side is negative
(1, 2, 3)	Error	Does not respect the triangle inequality (1+2=3 and not greater). Limit case test.

Table 1: Black-box test cases

c) White-box Testing: Branch Coverage (C1)

White-box testing is the opposite of black-box. Here we look inside the code and after understanding how it works we test all the logical decisions (if, else) for both true and false.

```
class Math {
  enum TriangleType { RightAngled, Isosceles, Equilateral, Normal }

  public TriangleType classifyTriangle(int side1, int side2, int
     side3) {
      if ((side1 == side2) || (side2 == side3) || (side1 == side3))
          return TriangleType.Isosceles;

      if ((side1 == side2) && (side2 == side3))
          return TriangleType.Equilateral;

      int sq1 = side1 * side1;
      int sq2 = side2 * side2;
      int sq3 = side3 * side3;

      if ((sq3 + sq2 == sq3) || (sq1 + sq3 == sq2) || (sq3 + sq2 ==
         sq1))
          return TriangleType.RightAngled;

      return TriangleType.Normal;
  }
}
```

Find places in your program where there are branching (‘if‘ conditions) and write them down.

- if ((side1 == side2) or (side2 == side3) or (side1 == side3))
- if ((side1 == side2) and (side2 == side3))
- if ((sq3 + sq2 == sq3) or (sq1 + sq3 == sq2) or (sq3 + sq2 == sq1))

Then, create test cases such that each branch of the program is executed at least once. Present the cases in a table and explain why you chose each one.

Input	Expected Output	Explanation
(3, 3, 3)	Isosceles (but Equilateral)	Bug: returns early as Isosceles but 1 and also 2 is true
(3, 3, 4)	Isosceles	Two equal sides 1 is true. 2 is false
(3, 4, 5)	Right-angled	1 false, 2 false, 3 true Pythagorean triple
(4, 5, 6)	Normal	No condition matches 1,2,3 all false

Table 2: White-box test cases

d) Combining Tests and Analyzing Bugs

Tests like (0, 5, 5), (-1, 4, 4), (1, 2, 3) return a triangle type instead of raising an error because no preconditions are enforced in the code.
The test (3, 3, 3) returns `Isosceles` instead of `Equilateral`. The logical condition for Equilateral should come **before** the Isosceles check.

e) Statement Coverage (C0)

Have we achieved C0? What is the usefulness of C0 and C1?
Yes, we have achieved C0 because every line of code has been executed by at least one test. However, **C1 (branch coverage)** is more powerful and precise, because it ensures that all logical decisions are tested both as true and false.

f) Evaluation of the Two Testing Techniques

Both **functional (black-box)** and **structural (white-box)** testing are useful.

- Functional testing helps define what the software should do from the user's perspective.
- Structural testing reveals internal logic errors.

Used together, they complement each other and improve reliability.
Used together they are certainly better because the functional test only looks at what the program should do listing the main requirements to be respected without going to see the code so it gives us greater clarity of what all the possible cases to take into consideration could be, while the second is strictly tied to the code so we can more easily find logical bugs but used alone without previous functional tests it could lose its usefulness because by focusing on the functioning of the code we could lose sight of what are the main requirements to be respected that the functional test instead lists.

g) Undiscovered Failures and Discovery Techniques

We discovered lack of checks for invalid inputs (lati¡=0) through black-box testing but also logical bugs in the code discovered through white-box testing .
There may be additional failures that we did not discover. A bug may only trigger with a very specific combination of numbers that we did not test.
This is why we call it Exploratory Testing where you simply try random inputs or use your intuition and curiosity without a precise plan.
Some bugs might only appear with very specific input combinations that were not included in our planned test cases.

To find such failures, we can use **Exploratory Testing**, where:

- We try unexpected or random inputs,
- We explore the system without a rigid plan,
- We rely on curiosity and experience to spot odd behaviors.

Exploratory testing helps find edge cases not covered by structured methods.

10-3

a

Not following the rule that "a programmer should not test his/her own code" can lead to several significant problems:

Inadequate Error Detection

The programmer may not identify subtle bugs or logical errors that a fresh set of eyes could catch. Changes made to the code might introduce new bugs that the original programmer fails to notice.

Limited Perspective

The programmer might not consider all possible use cases or edge cases, leading to incomplete testing.

b

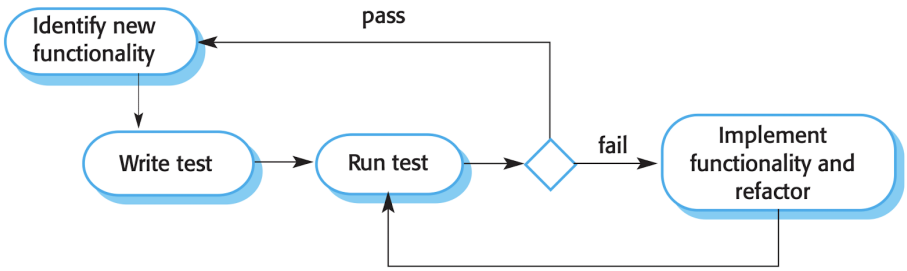
Early Bug Detection

TDD often allows for immediate feedback on code correctness and introduces Regression testing on the go, which usually is very tedious and expensive to do afterwards.

Comprehensive Coverage

By emphasizing the creation of tests before writing the actual code, TDD ensures that all requirements are clearly defined and considered. If there are some uncovered cases left they usually derive from lack of understanding of the problem.

TDD



1. You start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
2. You write a test for this functionality and implement it as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.
3. You then run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

Source: Software Engineering Pearson 10th Edition