Software Engineering — Dynamic Testing, Part Two

Arkadiusz Chrobot

Department of Information Systems, Kielce University of Technology

Kielce, January 26, 2026

Outline

Motto

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

— Brian W. Kernighan

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

— Martin Golding

"To tell somebody that he is wrong is called criticism. To do so officially is called testing."

— Gaurav Khurana

Introduction

The testing levels, described in the previous lecture, were devised in times when the procedural-oriented programming was the predominant way of creating software. For object-oriented programming, they need to be slightly adjusted, which is the first topic of this lecture. We also discuss the metrics used in testing. They are used for assessing the progress and quality of tests, but also the quality of the software under testing.

Testing Object-Oriented Code

Object-oriented code is different from procedural-oriented code in several aspects:

- An object, as an independent software component, is usually bigger than a single subroutine.
- Objects integrated into subsystems are typically loosely coupled, they don't have a common "entry point".
- If the objects are reused, then testers may not have access to the source code of objects' classes, and they cannot analyse it.

Testing Object-Oriented Code

Testing Levels

In object-oriented code testing the levels of tests may be defined as follows:

unit testing The methods of objects should be tested in isolation, if feasible. The functional and structural methods can be applied for that.

class testing It is an over-level of unit testing. The sequences of interdependent object operations should be verified. Both testing methods may be applied.

integration testing The clusters of objects that are used together should be tested. The tests should be based on usage scenarios.

system testing The way of testing object-oriented software at this level is the same as described in previous lecture.

acceptance testing At this level, there are also no changes.

It should be noted, that the boundary between unit testing and integration testing in case of object-oriented software is fuzzy.

Testing Object-Oriented Code
Class Testing

The *Class Testing* has three objectives:

1. Examination of methods of the object in isolation — this is the basic requirement of unit testing. To ensure the isolation the test dummies are used in unit tests.
2. Verification of setting and accessing all attributes of the object — necessary only in some cases.
3. Checking the object in all possible states — all events that result in changing the object state should be tested.

Testing Object-Oriented Code

Class Testing

Let's assume, that a class, that implements the following interface, has to be tested.

```
interface LightControl {
    void switchOn();
    void makeBrighter();
    void makeDimmer();
    void switchOff();
}
```

Testing Object-Oriented Code

Class Testing

To verify the class's object in all possible states, the following sequences of method invocations have to be verified:

`switchOn()` → `switchOff()`

`switchOn()` → `makeBrighter()` → `switchOff()`

`switchOn()` → `makeDimmer()` → `switchOff()`

`switchOn()` → `makeBrighter()` → `makeDimmer()` → `switchOff()`

`switchOn()` → `makeDimmer()` → `makeBrighter()` → `switchOff()`

Testing Object-Oriented Code

Integration Testing

The integration testing of object-oriented code can be performed using the following approaches:

Use cases or scenarios testing The use cases or scenarios describe how the software should be used, so the test cases can be derived from these descriptions.

Thread testing The object-oriented software is usually event-driven. In this method, the reactions of software to events are verified. The tester needs to know how the events are internally processed by the software to test it correctly.

Objects interaction testing This is a similar method to thread testing, however, this time, the test cases should verify the *method* — *message* paths that start with an input event and end with an output event.

Metrics

Dynamic testing is a resource consuming activity. To assess its progress and quality, special *metrics* were developed. Some of them are also used for measuring the quality of tested software. Metrics are also designed specifically for a given testing method or level.

Code Coverage Metrics

The *Code Coverage Metrics* inform what proportion of the tested code is run when the test cases are preformed. The usual recommendation is to execute about 80% of the code while testing. However, this figure is ambiguous. Code coverage should be specified in context. For critical software it may be even higher, while in non-critical systems could be less. The Code Coverage Metrics allow testers to find redundant test cases, i.e. such that do not detect new defects. There are three such metrics:

- *Block Coverage Metric*,
- *Decision Coverage Metric*,
- *Path Coverage Metric*.

Code Coverage Metrics

Block Coverage Metric

The *block of code* is a sequence of statements that don't contain any statements that change the flow of control (like loops). The *Block Coverage Metric* is defined as the ratio of the number of tested blocks of code to the total number of blocks in the verified component. The *Line Coverage Metric* is a special case of the Block Coverage Metric, where the blocks are one line long. These metrics are easy to understand and may be applied both to source and executable code. However, they don't make it possible to verify how well the conditional expressions in control flow statements are tested.

Code Coverage Metrics

Block Coverage Metric — Example

```
if(a==3) {
    function_1(...);
} else {
    function_2(...);
}
```

This code snippet requires two test cases to get the 100% code coverage — one for each branch of the conditional statement.

Code Coverage Metrics

Block Coverage Metric — Example

```
int a=-2;
if(b>0)
    a=1;
double x=sqrt(a);
```

This code snippet requires only one test cases to get the 100% code coverage — for `b` greater than 0. However, this test case won't detect a serious defect that manifests only when $b \leq 0$.

Code Coverage Metrics

Decision Coverage Metric

This *Decision Coverage Metric* allows testers to verify how well the control flow statements have been checked. It is defined as the ratio of number of tested branches of control flow statements to the total number of branches of control flow statements. The metric offers as good verification of code blocks as the Block Coverage Metric, and better verification of conditional expressions in control flow statements. However, it requires designing a larger number of test cases, and it doesn't take into consideration the short-circuit evaluation of expressions.

Code Coverage Metrics

Decision Coverage Metric — Example

```
if(a>0&&(b<0 || function(a,3)>0)) {
        statement_1;
} else {
        statement_2;
}
```

According to the Decision Coverage Metric, the code snippet can be fully tested with the use of only two test cases. Unfortunately, they don't have to invoke the function in the condition.

Code Coverage Metrics

Path Coverage Metric

The *Path Coverage Metric* is closely related to prime paths, discussed in the previous lecture. It takes into account the short-circuit evaluation of conditional expressions, but requires designing even larger number of test cases than the Decision Coverage Metric. If the tested software has many loops, then the test cases should be defined in such a way, that only a limited number of their iterations should be performed.

Requirements Coverage Metrics

The *Requirements Coverage Metrics* apply to the functional testing method, and are used in system and acceptance testing. Their objective is to check how well the software requirements have been verified in the tests. Basically, these metrics measure the ratio of the number of tested requirements to the number of all requirements. However, the requirements may be described differently, depending of the applied method of requirements elicitation. Some of them may require more than one test case to be verified properly.

Requirement Coverage Metrics

Error Coverage Metric

The *Error Coverage Metric* is defined as the number of errors handled by the software during testing to the total number of errors that the software is supposed to react to. The errors in this context mean exceptions like a wrong data format or unavailability a of network connection or other resources.

Requirement Coverage Metrics

Use Cases Coverage Metrics

The use case is a collection of scenarios that describe typical interaction with the software. There are two categories of use cases: business use cases that are related to the high-level requirements and system use cases, that specify the technical details of processes inside the software. Metrics can be defined for both of them.

Miscellaneous Metrics

In this section are introduced metrics that don't fall into any of the previous categories. They may be used for simultaneously assessing the code and tests quality.

Miscellaneous Metrics

Number of Detected Defects

The *Number of Detected Defects Metric* or the *Cumulated Number of Detected Defects Metric* allows testers to estimate the initial quality of code and its increment after subsequent sessions of testing and debugging. Using it, the quality control engineers may also assess the effectiveness of testing methods they apply, and the defect removal techniques used by programmers. The Cumulated Number of Detected Defects Metric requires recording the number of discovered defects in each testing session. When the Cumulated Number of Detected Defects stabilizes, the testing of the software may be finished.

Miscellaneous Metrics

Number of Component Defects

The *Number of Component Defects Metric* is just the number of discovered defects in each of the software components. It is used to estimate the quality of every component. While the metric is easy to calculate, its interpretation depends on many factors. A component with numerous defects can be a complex one or may be created by a team of overworked engineers. It may also happen, that the test cases used for testing this component are more efficient in finding defects than other. Such test cases may be used in the maintaining phase of the software life cycle, to detect regression.

Miscellaneous Metrics

Density of Defects

The *Density of Defects Metric* is defined as the number of defects per line of code or kilo lines (1000 lines) of code. It allows the software engineers to assess the increment of the software quality in relation to the increment of its lines of code. Although there is no formal definition of the concept of line of code, the metric is widely used.

Miscellaneous Metrics

Percentage of Detected Defects

The *Percentage of Detected Defects Metric* is used to evaluate the testing efficiency. It is defined as the ratio of number of detected defects to the estimated total number of defects. To estimate the latter, some defects are *seeded* in the software, i.e. they are artificially introduced to the code. The total number of defects is estimated with the use of the following expression: $N_t = N_s \cdot \frac{n_t}{n_s}$, where N_s is the total number of seeded defects, N_t is the estimated number of true defects, n_t is the number of true defects detected by tests, and n_s is the number of seeded defects detected by tests. The more the types of seeded defects match the types of true defects, the more reliable is this metric.

Metrics

Ending

There are many more metrics used in Software Engineering, than these described in this lecture. However, using the metrics discussed here usually should be enough for estimating the quality of software and tests.

Questions

?

The End

Thank You for Your attention!