Software Engineering — Dynamic Testing, Part One

Arkadiusz Chrobot

Department of Information Systems, Kielce University of Technology

Kielce, December 9, 2024

Outline

Motto

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald E. Knuth

Introduction

This lecture discusses dynamic testing in more details. The topics of test levels and test methods are covered here.

Testing Levels

There are four *levels* of dynamic tests:

unit tests apply to the smallest components of code, like functions;

integration tests check the interaction of components;

system tests verify the completed *version* of software;

acceptance tests prove to stakeholders, that the software meets requirements.

Acceptance Tests

In the case of custom software the *Acceptance Tests* may be performed by the software developers, but more often they are made by the customer (so-called *User Acceptance Tests* or *UATs*). If the customers lack required assets to do such tests or want to be objective, they may outsource acceptance tests to a third party company.

For generic-purpose software, the *Alpha*, and *Beta* tests may be applied. The *Alpha Testing* is performed on-premise by a group of selected clients of the software company. The *Beta Testing* is done on clients' computers with the use of a time-limited or otherwise restricted version of the software.
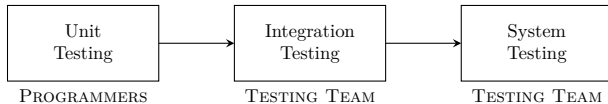
Testing Levels

Figure: Testing levels and responsible parties

Testing For Defects

Dynamic testing, especially testing for defects, is about conducting experiments that verify if the software behaviour and its outcomes adhere to its *specification*. The (formal or informal) description of such an experiment is called a *test case*. It defines the input data for a test and specifies the expected behaviour and outcomes of the software under the test (see Figure 2).

The main difficulty in testing is designing the smallest possible number of relevant test cases (see Figure 3). The quality of testing can be estimated using *metrics*, that will be discussed in the next lecture. However, good quality tests can be only achieved by using a systematic method of designing test cases. The two most commonly used methods for testing are *functional testing* and *structural testing*.
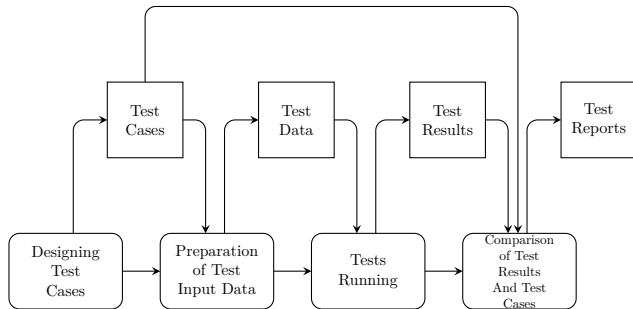
Testing For Defects

Figure: Testing process

Testing For Defects

I — input data causing software misbehaviour
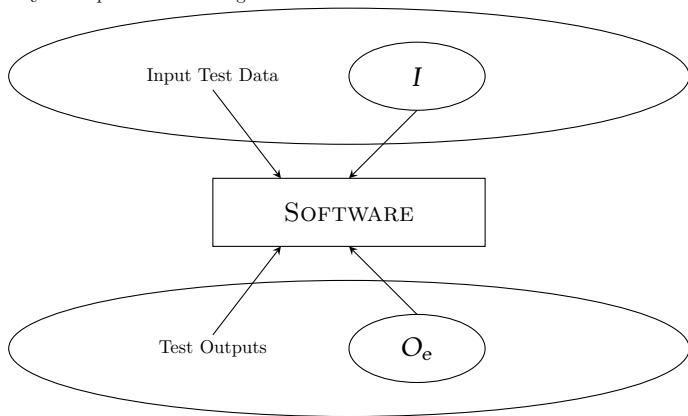O_e — output data revealing defects

Figure: Model of Testing

Functional Testing
Equivalence Class Partitioning (ECP)

The *Functional Testing* is also known as *Black-Box Testing*. In this method, the source code of tested software is either unavailable, or too complex to analyse. Only the specification of the software is known, that describes what the software does. Functional testing may be applied for every level of tests. The first step in designing appropriate test cases for black-box testing is partitioning the set of input data into subsets called *Equivalence Classes*. Each equivalence class includes data of similar characteristics, thus processed similarly by the software (see Figure 4).

Functional Testing
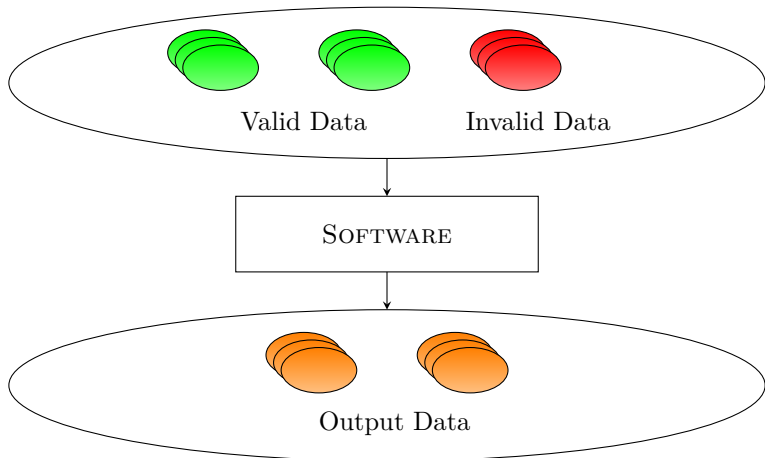## Equivalence Class Partitioning (ECP)

Figure: Equivalence Class Partitioning (ECP)

Functional Testing
Boundary Testing

The designer of test cases should not only choose input data from the "middle" of equivalence classes, but also data that "lie" on its boundaries (so-called *boundary values* or *edge cases*), are close to the boundaries or are just behind these boundaries (see Figure 5).

Functional Testing
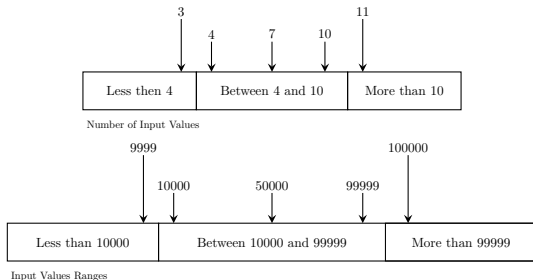## Boundary Testing

Figure: Boundary Testing

Functional Testing
Example

Let's suppose that we want to test a software component, that tries to locate a given value in a sequence of numbers. The formal specification of the component is given in the next slide.

Functional Testing
Example

Specification

procedure Search (Key:ELEM; T:ELEM ARRAY; Found: **in out** BOOLEAN; L: **in out** ELEM_INDEX);

Pre-condition
-- The array has at least one element.
T'FIRST <=T'LAST

Post-condition
-- Value is found and its location is in L.
(Found **and** T(L)=Key)

or
-- Value is not in the array.
(**not** Found **and not**(**exists** i, T'FIRST <=i<=T'LAST, T(i) = Key))

Functional Testing
Example

By analysing the specification and by following the test case designing methods described earlier, we can define the following equivalence classes:

Array	Values(s)
Only one element.	Is in array.
Only one element.	Not in the array.
More than one element.	Is in the first element.
More than one element.	Is in the last element.
More than one element.	Is in the middle element.
More than one element.	Is not in the array.

Functional Testing
Example

Based on the equivalence classes we can design the following test cases:

Input Sequence (T)	Key	Result (Found, L)
17	17	true, 0
17	0	false, ??
17,29,21,23	17	true, 0
41,18,9,31,30,16,45	45	true, 6
17,18,21,23,29,41,38	23	true, 3
21,23,29,33,38	25	false, ??

Structural Testing

In *Structural Testing* (also known as *White-Box Testing* or *Clear Box Testing*, *Transparent Box Testing*, *Glass Box Testing*) the source code of tested component is available. This allows the test designer to derive additional test cases from the code (see Figure 6). This method may be applied to unit tests and to some extent to integration tests.
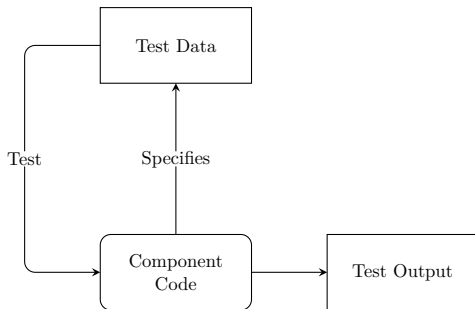
Structural Testing

Figure: Structural Testing

Structural Testing
Example — Binary Search

In the next slide is presented method in Java, that tries to locate a value (`key`) in an array (`elemArray`) applying the binary search algorithm. The sole fact of using such an algorithm defines an additional pre-condition: the values in the array should be in a non-descending order.

Structural Testing
Example — Binary Search

```java
public static Result search(int key, int elemArray[]) {
    int bottom = 0;
    int top = elemArray.length-1;
    while(bottom<=top) {
        int middle = bottom + (top-bottom)/2;
        if(elemArray[middle]==key)
            return new Result(middle,true);
        if(elemArray[middle]<key)
            bottom=middle+1;
        else
            top = middle-1;
    }
    return new Result(-1,false);
}
```

Structural Testing
Equivalence Class For Binary Search

Knowing that the tested method uses the binary search algorithm, we can define two more equivalence classes — sequences of numbers where the value to find is stored in the left and the right neighbour of the middle element (see Figure 7).
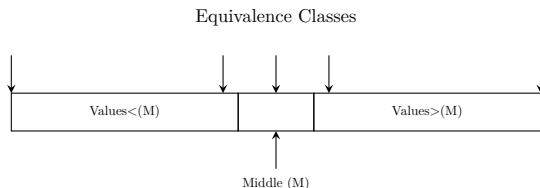
Equivalence Classes

Figure: Equivalence Classes For Binary Search

Structural Testing
Test Cases For Binary Search

The following test cases may be defined for the `search()` method:

Input Sequence (T)	Key	Result (Found, L)
17	17	true,0
17	0	false,??
17,21,23,29	17	true,0
9,16,18,30,31,41,45	45	true,6
17,18,21,23,29,38,41	23	true,3
17,18,21,23,29,33,38	21	true,2
12,18,21,23,32	23	true,3
21,23,29,33,38	25	false,??

Additionally, an large sequence of values (up to 2147483647 elements), with a value to find located at the end of it, should be used.

Structural Testing

The structural testing allows the testers to check each *prime path* in the tested component. A *path* is a sequence of statements performed together. It starts with an *entry point* and ends with an *exit point*. A *prime path* is a path that is different from others by at least one statement.

When all prime paths in the component are tested, then it means that each of the statements in the component has been performed at least once and each *simple condition* has been verified when it is true and false. A simple condition is a part of the condition expression that doesn't include logical operator (OR, AND). The combinations of prime paths are *not* tested, because it would be too expensive.

To find all prime paths in the code the *Control-Flow Graph* (*CFG*) can be applied. The CGF for `search()` method is given in the Figure 8.
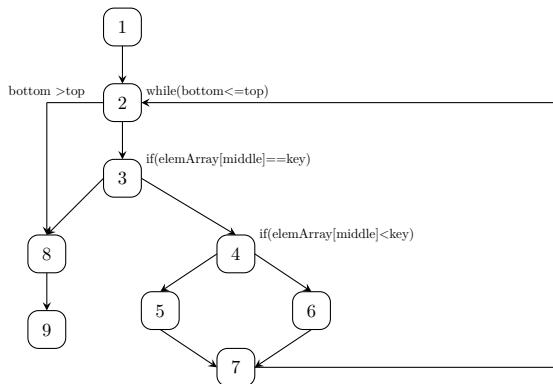
Structural Testing
Control-Flow Graph

Figure: Control Flow Graph

Structural Testing

There are 4 prime paths in the CGF for `search()` method, e.g.:

$1, 2, 3, 8, 9$
$1, 2, 3, 4, 6, 7, 2$
$1, 2, 3, 4, 5, 7, 2$
$1, 2, 3, 4, 6, 7, 2, 8, 9$

The number of such paths is defined by the *Cyclomatic Complexity*, by Thomas J. McCabe Sr. It is calculated the following way:

$$CC(G) = \#edges - \#nodes + 2$$

or it can be calculated by counting all the simple conditions in the code and adding one to the sum. Both the methods are useless when the code is multithreaded, or recursive or contains the infamous `goto` statement. The cyclomatic complexity specifies the minimum number of test cases needed to verify the code under tests.

Integration Testing

The main objective of *Integration Testing* it to verify the interaction of cooperating software components. That means that the integration tests can be performed when at least two such components are available. Integration Testing is an incremental process (see Figure 9). When only two components are ready for tests, then three sets of test cases can be prepared: two that verifies the components in isolation (e.g. T1 and T2) and one that checks how they cooperate (e.g. T3). If more components are added, then the existing test cases need to be updated and new sets of them should be included to verify the interaction of new elements with the rest of the system.

Integration Testing
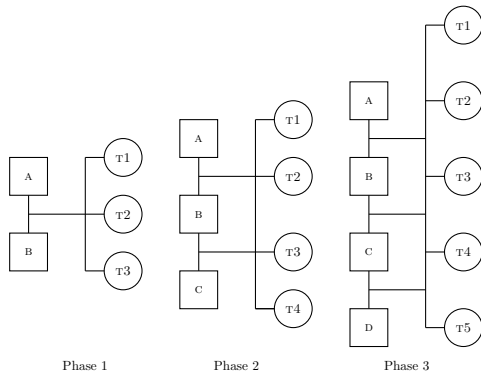Incremental Integration Testing

Figure: Incremental Integration Testing

Integration Testing

Integration Testing is usually not as straightforward, as shown in the Figure 9. The newly added test cases may reveal some defects in the interaction of previously integrated components or in the components themselves, that were not detected by the earlier used test cases.

It may also happen that the integration of just two components is impossible and more of them have to be integrated at once. Such an approach is called a *Big Bang Integration Testing*. However, two others approaches are more common: the *Top-Down* and the *Bottom-Up Integration Testing*. They are strictly related to the methods of constructing the software.

In the Top-Down approach, first are build and tested the high-level components. It means that the lower-level ones, on which depend the high-level ones, are unavailable and have to be replaced in testing by *test doubles*. Later, when the missing components are finally available, they use other test doubles when tested.
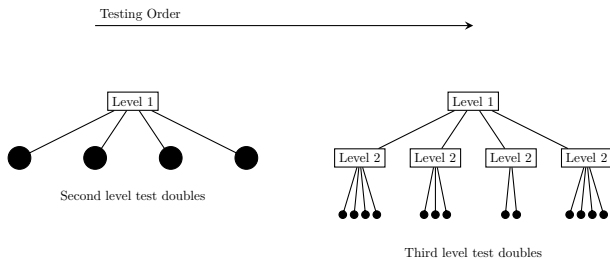
Integration Testing
Top-Down Testing

Figure: Top-Down Testing

Integration Testing
Test Doubles

A *Test Double* is a software component used in testing. It usually pretends to be another component, which final version is not available at the time of tests, however the verified software depends on that component. The double may perform additional tasks during a test, like recording how many times it was invoked by the tested software. The ▸ classification of test doubles was proposed by Gerard Meszaros, an employee of Microsoft:

dummy A test double that is passed as an argument to a subroutine. It does nothing.

fake It has some working implementation, but not suitable for production environment (i.e. an in-memory database).

stub It provides responses, but only to some specified requests. It doesn't react to anything else.

spy It is a stub that records some limited information about how it was invoked in tests.

mock A mock *verifies* how it is invoked. It may throw exception is it is called incorrectly. Like the spy it records and verifies how many times it was called in testing.

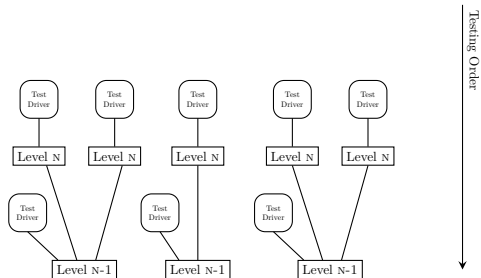# Integration Testing
Bottom-Up Testing

Figure: Bottom-Up Testing

Integration Testing

In the *Bottom-Up* approach the lowest-level components are built and verified first. However, they are not independent, so a special *Test Drivers* must be prepared in order to test them. When the lowest-level components are finally integrated in highest-level ones, then another set of *test drivers* is needed to test them.

Integration Testing
Comparison of Top-Down And Bottom-Up Methods

- In Top-Down approach, the software architecture can be validated earlier, than in Bottom-Up.
- The proof of feasibility can be delivered early in both approaches.
- Tests implementation is equally hard in both approaches.
- Tests observation requires additional measures in both cases.

In real-life software projects, both methods of developing and testing are applied simultaneously, so these cons and pros don't matter that much.

Integration Testing
Interface Testing

Most test cases in Integration Testing are designed to verify the *interfaces* of the components, rather then their internal implementations (see Figure 12). The most common types of interfaces are as follows:

parameters used to pass data or function references from one component to another,

shared memory it is a memory area used by two or more concurrent threads or processed to exchange data,

procedural interface one of components provides services called by other components,

message passing one of the components requests services of other component by passing a message.

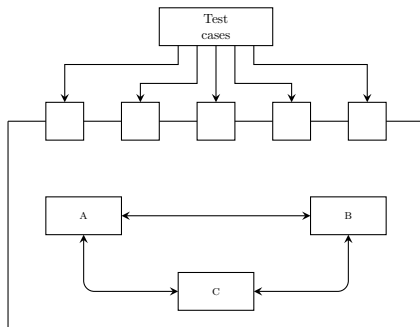# Integration Testing
Interface Testing

Figure: Interface Testing

Integration Testing
Interface Testing — Recomentations

Each of the interface types needs a different approach when designing test cases:

1. For parameters, the boundary testing method may be applied. Special care need the reference and pointer parameters. Tests should be performed with their values equal `null`.

2. The shared memory should be tested in such a way, that the processes or threads, that use it, are activated in different order.

3. Some of the test cases that verify the procedural interface should make the component equipped with it fail. This allows testers to check if the programmers haven't made wrong assumptions about how the failure is announced.

4. *Stress Testing* should be applied to verify the message passing interface.

System Testing

System testing takes place when the development of at least an early version of the software is finished. Some experts consider it to be a natural extension of integration testing, but tests at this level have a different character. They not only check the functionality of the software, but also verify if it meets its non-functional requirements:

functional tests check if the software correctly provides its services,

efficiency tests examine how the software is working under a regular workload,

stress tests verify the behaviour of the software for significantly higher workload than it was designed for,

security tests check if the software appropriately protects its assets,

compliance tests verify if the software meets required standards,

portability tests check if the software runs correctly on different system or hardware platforms,

reliability tests verifies if the software works reliably.

Questions

?

The End

Thank You for Your attention!