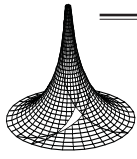

16/2 (2018), 215–232
DOI: 10.5485/TMCS.2018.0451
tmcs@science.unideb.hu
http://tmcs.math.unideb.hu

Teaching
Mathematics and
Computer Science

How to teach testing?

Tibor Gregorics, Krisztián Mózsi and Rudolf Szendrei

Abstract. Testing methodology is an important part of IT education. It is desired to show the beginner programmer students the advantage of testing by having them do only a small amount of work. In this paper, we will show how to make testing as a part of programming in simple exercises. These exercises are solved with the analogous programming technique, which is based on programming theorems over enumerators. We have elaborated grey-box test cases for the programs which have been developed based on programming theorems. These test cases can be taught together with the programming theorems, and they can serve as a standard testing procedure for programmers. We also suggest a test tool to automatize test runs, and we will discuss its usage in a short case study.

Key words and phrases: programming technology, program testing.

ZDM Subject Classification: P50.

Introduction

In the education of informatics, it is necessary for the student to know and to be able to use software development methodology. An important part of the methodology is to manage the complete life cycle of the software being developed. This includes requirement analysis, the selection of the proper architecture, the design of the software, the selection of milestones, implementation, software testing (testing in short) and maintenance and, finally, documentation. [7]

Since we consider the software as a product, we must keep in mind its cost, quality, and deadline. Quality assurance is important at all phases of software development, and there are proper tools for it. One of the most important tools

Copyright © 2018 by University of Debrecen

of quality assurance is testing, which also appears explicitly as a development phase.

Nowadays, testing becomes more important when the size and the complexity of a software are getting more and more enormous. In an industrial environment, several test tools are applied to do the unit and integration test of the software. Moreover, developer teams use version controls, issue trackers, and test environments to provide traceability to the development. Among development methods, the agile method and the test-driven development method are getting more and more popular in the industry. Because of this, it is particularly important for students to learn these methods during their studies.

In the ELTE Computer Science MSc, there is a special course related to the topic, which students should accomplish when they are close to the end of their studies. However, it would be also important to teach BSc students the necessity of testing and the related approach as soon as possible because this could get students to consider testing as a useful tool.

In the next chapters, we introduce a feasible methodology for learning testing in the first-year programming courses in the ELTE Computer Science BSc. The basics of this methodology are gathered from the programming theorems [1] [2] used to solve simple programming problem. These theorems are not only the basics of the special programming technology which is called as analogous programming [1] [4], but they also serve as a good starting point to collect the potential test cases easily. We discuss this in chapter 2. In chapter 3, we show the typical grey-box test cases [7] related to programming theorems. In chapter 4, we show an example how to design testing based on the specification if we use the methodology of analogous programming. In chapter 5, we collect those C++-related test tools which can be integrated and used easily in the existing programs. Out of these tools, we also choose the one which best fits our needs. In chapter 6, we discuss the possibilities how to adapt the suggested methodology into practice by using the selected tool. For this tool, we also give a deeper insight into its usage in this chapter. Finally, we summarize our experience about the suggested methodology and the selected testing tool.

Programming theorems and testing

Programming theorems are frequently used to plan algorithms. A programming theorem is a pattern, a task-algorithm pair where the algorithm solves the task. Most programmers consider programming theorems as sample solutions.

When they want to construct a program to solve a problem which is like the task of a theorem, they try to repeat the same activities which created the algorithm of the theorem. Therefore, in this case, algorithmic thinking is supported by programming theorems.

However, there exists another method to create programs based on programming theorems. This is analogous programming. [1] [4] When students start studying programming, this can be a good alternative to the learning of algorithmic thinking. In this context, the problem is decomposed into subproblems so that each subproblem corresponds to the task of a programming theorem. After recognizing the similarities between a subproblem and the task of a theorem, the algorithm of the theorem can be transformed to solve the actual problem. This concept does not need algorithmic thinking because during its application it is needless to understand how the algorithm of the programming theorem works. We only need to reveal the differences between the subproblem and the task of the selected programming theorem, and then change the corresponding parts of the algorithm. At the end, the final program can be built up from the solving programs of the subproblems.

This kind of program designing technique helps not only to design algorithms, but it also affects the whole life cycle of the development. This life cycle can be divided into four steps (see Figure 1).

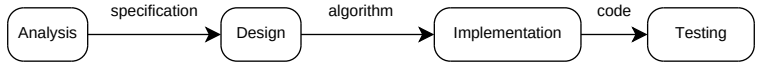

Figure 1. Steps of program development: a) Analysis, the result of which is the specification, b) Design, which gives the required algorithm, c) Implementation, which consists of coding the algorithm, the I/O tasks, and it also creates the proper structure of the program, d) Testing

During problem analysis, we can start thinking about which programming theorems can be used in combination to solve the new problem. The result of the analysis is a specification, which gives us not only the formally specified problem, but it also refers to the solution. This executable specification is a well-known technique used in requirement analysis of complex software. [7]

In the design phase, we must take care of only two things. We must adapt precisely the programming theorems to the given problem parts [5], and we must link the resulting algorithms to get the complete abstract program [3].

The implementation follows the same logic described above. In this step, a traditional procedural program is made, where the corresponding subroutines of the program are implemented as procedures or functions.

Traditionally, a simpler program can be tested under black-box strategy and white-box strategy. [6] The black-box strategy is unaware of the internal structure of the application to be tested and based on the specification while the white-box strategy has access to the internal structure of the application. When the internal structure is partially known, these strategies are mingled and we can call it a grey-box strategy. (see Figure 2)

The black-box test cases based on the specification. Among black-box test cases, invalid test cases form another group. This group contains those cases where input data do not satisfy the precondition of the program. In this case, the program should give a preprogrammed error message according to the invalid data.

The white-box testing analyzes the program code. It can be separated in two parts: testing the codes of the input and output handling and testing the code of the algorithm code.

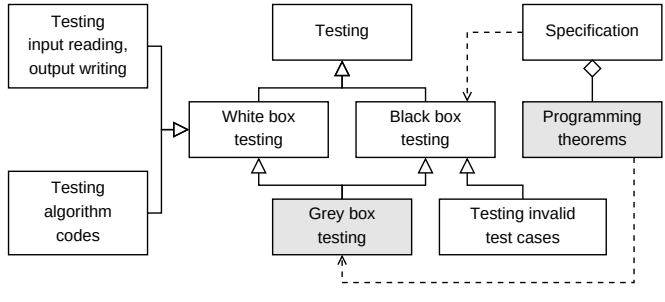

Figure 2. Test case groups and their relations to the specification of the problem, and the programming theorems used.

If somebody as a teacher has already dealt with the problem of how to get students to make precise test plans, then they must know how hard it can be for beginner programmers (and even for professional ones) to be the judges of their own programs. Up to now, program testing has not been in our focus in the first-year programming courses of our syllabus. We included testing strategies, but we cannot require their application in practice.

The novelty of this paper is our discovery that the solving program of a problem can be tested with the typical test cases of the programming theorems

which are applied to create the program. The test cases of a programming theorem can be defined precisely and can be taught together with the theorem itself.

Moreover, this testing plan can be assembled during analysis since specification introduces the theorems that help to solve the actual problem. Not only does this kind of specification describe a problem, but it also mentions a possible solution with programming theorems. This is called executable specification. [7] On the one hand, the test cases deriving from the applied programming theorems count as white-box test cases because they are based on the structure of the algorithms of theorems. On the other hand, these test cases act like black-box test cases since they are created upon specification and it is also imaginable that the final program will not be constructed with the programming theorems referred to in the specification. These test cases integrate the specialties of the black-box and white-box testing: they are grey-box test cases.

If we collect and teach typical test cases for each programming theorem, then we can establish a standard for programmers, so they will not need to make test cases on their own. In our teaching experience, when students need to plan their programs based on analogous programming technology and they make a specification referring to programming theorems, they must also assemble the test cases deriving from the applied programming theorems. This process is demanded in lessons and in assignments, as well

Standard grey-box test cases

As we can see, it is exciting to discover and use grey-box test cases of the corresponding programming theorems. We can get these test cases in the same way as we get the solution during the analogous programming, in other words, we must adapt test cases to the concrete task. The programming theorems, such as summation, counting, maximum selection, linear (sequential) search, selection (when the searched element must exist) and conditional maximum search accepted by the informatics community [1] [2] have a common property: they are all built upon an enumeration of a collection. This enumeration is mostly an interval of integer numbers, or a set, or in many cases, it is an array, sequence, or maybe a sequential input file. Because of this, the test cases given to the processed enumeration are common for all programming theorems. Perhaps, there are special test cases which are specific to one or more programming theorems.

Test cases of enumeration	
Boundary check of the enumeration	Is first item processed?
	Is last item processed?
	Is an intermediate item processed?
Based on enumeration size	empty enumeration
	enumeration with one item
	enumeration with many items

Table 1. Grey-box test cases related to the enumeration

It must always be checked how a solution works for enumerations of different sizes (see Table 1). It sounds unbelievable, but a lot of programs fail on examinations because they do not work properly on the empty enumeration. An enumeration which contains one or two items can be an interesting test case, but we also need to test enumerations which contain an arbitrary number of items. In case of the counting theorem, it helps evaluating test results when all the items satisfy the condition of counting. It is important to notice that the precondition of the programming theorem of maximum selection requires a non-empty enumeration. Despite this fact, testing the program against an empty enumeration is still required. For now, it will test the precondition of the program against an invalid test case instead of doing a valid test of the programming theorem on a valid input.

It is important to check whether the program processes the first and last item of the enumeration. This is a general testing aspect which affects all programming theorems, but we must also consider the specialties of each theorem. For example, only one enumeration with two different items is needed for summation. Counting also needs one enumeration with two items, where both satisfy the condition of counting. The maximum selection theorem needs two enumerations: where the maximum is the first item and where this is the last item. The linear search theorem needs three inputs, where the item which we are searching for is the first or the last one. Selection also needs two inputs, where the first or an inner item is what we are seeking, respectively. To test a conditional maximum search, we must check the test cases of the linear search and the maximum selection theorem.

When applying the summation programming theorem, it is necessary to do a stress test: the maximum size of the collection must be measured, which can still be processed by the program without any errors. Counting needs cases where

Test cases of programming theorems	
Summation	an empty enumeration
	an enumeration which contains two different items
	stress test (with different large enumeration)
Counting	an empty enumeration
	an enumeration containing two items, where both satisfy the condition
	the condition is satisfied by zero, one, two or many items from the enumeration.
Selection	looking for the first item of the enumeration
	looking for the non-first item of the enumeration
Linear search	an empty enumeration
	looking for the first item of the enumeration
	looking for the last item of the enumeration
	looking for an intermediate item of the enumeration
	there is not any item, which satisfies the condition.
Maximum selection	an enumeration with two items, where the first is the greater one
	an enumeration with two items, where the last is the greater one
	an enumeration with many items, where an inner item is the greatest
	an enumeration with many items, with many greatest items
Conditional maximum search	test cases of linear search and maximum selection

Table 2. Special test cases of the programming theorems

the collection contains zero, one, two, or any number of elements which have a given property. Maximum selection must be evaluated on collections where the maximum is an intermediate item of the enumeration, and there are several maximum items. There are two important cases for the linear search theorem: an item with a given property is absent or not. It is worth trying the latter case when we search for the first or an intermediate item of the collection (note: we

have already checked the case when the last item was the one which we were seeking). For the conditional maximum search, the test cases of both the search and the maximum selection theorems should be evaluated (see Table 2).

Considering our teaching experience, grey-box test cases, which do not belong to any black-box group, can be recognized with the greatest difficulty. For example, if we calculate the union of two sets, then we must check the algebraic properties of the union operator: commutativity, associativity, and neutral element. It is interesting that these checks are unnecessary if we check the grey-box test cases of the summation theorem, which uses a special kind of combined enumerator to solve the union calculation of the two sets.

Test plan generation

In the following example, we will show how to create a test plan consisting of standard test cases for a simple program to reveal its errors. There is a matrix consisting of integer numbers. Choose the row which has the maximum sum of values. Using the analogous programming technique, we choose the maximum selection and the summation theorems to solve the task.

Specification:

Variables:	$(data\colon \mathbb{Z}^{n\times m},\ maxSumIndex\colon \mathbb{N},\ maxSum\colon \mathbb{Z})$
Pre-condition:	$(data = data' \wedge n > 0)$
Post-condition:	$(\textit{Pre-condition} \wedge maxSum = \sum_{i=1}^{n} rowSum(i) \wedge$ $maxSum = rowSum(maxSumIndex)))$
where	$rowSum : \mathbb{N} \to \mathbb{Z}$
	$rowSum(i) = \sum_{j=1}^{m} data[i,j]$

In the design phase, we must know the algorithms of the corresponding theorems, and we must apply the specialties of the actual task to them. After this, we are ready to implement the concrete program. Assume that the outer programming theorem (the maximum selection) is implemented in the *maxRowSum* procedure, which takes the matrix as an argument. It also has two integer type output parameters named *maxSumIndex* and *maxSum*, where the first will contain the index and the second one the sum of the corresponding row. The inner

summation theorem is implemented in the *rowSum* function, which sums up the values of a given row.

The declarations of the defined subroutines:

```
void maxRowSum(const std::vector<std::vector<int>> &data,
               int &maxSumIndex,
               int &maxSum);
int rowSum(const std::vector<int> &row);
```

In the next step, we would like to make a test plan. Now, it comes very handy that we have used the analogous programming technique to solve the task: we collect standardized test cases determined by the programming theorems used. We get the following test cases: boundary and enumeration size-related checks, the general case when the maximum item is in the middle of the enumeration, and the case when there are several maximum items.

First, we consider the outer theorem. We must provide an invalid test case to check the answer when the precondition is not satisfied. In this case, we do not have many options: we have created a matrix which does not have any row.

Consider the valid grey-box test cases. In this example, the lower boundary check of the enumeration means that we provide a matrix where the first row has the greatest sum among the rows, and we want to know whether the program gives the correct result or not. According to this, we create a matrix which has only two rows and one column, where e.g. the value in the first row is 10 and the value in the second row is 5. To run the upper boundary check, we only need to swap the two rows, expecting the second row as maximal.

It is worth checking the results on matrices consisting of one, two, or many rows. In the first case, the expected result is the sum of the row. For matrices with two or more rows, we provide only one column. Concrete inputs and expected results can be seen in Table 3.

In the special case of maximum selection, we must also check how the program works when the row with the maximum row sum is in the middle of the enumeration, or there are more than one row existing with the same maximum row sum value. In the latter case, we expect that the program returns the first possible correct occurrence. Similarly, to the previous test case, the provided matrices have only one column.

Testing aspect	**Test input**	**Expected result**
Invalid input	empty matrix	programmed error message
Boundary check of the enumeration	$\begin{bmatrix}10\\5\end{bmatrix}$	maxSum=10, maxSumIndex=1
	$\begin{bmatrix}5\\10\end{bmatrix}$	maxSum=10, maxSumIndex=2
Based on enumeration size	$\begin{bmatrix}10 & 32\end{bmatrix}$	maxSum=42, maxSumIndex=1
	$\begin{bmatrix}10\\42\end{bmatrix}$	maxSum=42, maxSumIndex=2
	$\begin{bmatrix}2\\1\\10\\3\end{bmatrix}$	maxSum=10, maxSumIndex=3
Maximum value is in the middle of the enumaration	$\begin{bmatrix}1\\10\\2\end{bmatrix}$	maxSum=10, maxSumIndex=2
More than one maximum value	$\begin{bmatrix}2\\1\\12\\12\end{bmatrix}$	maxSum=12, maxSumIndex=3

Table 3. Test plan, test cases based on the maximum selection theorem

Testing aspect	**Test input**	**Expected result**
Boundary check of the enumeration	[1, 2]	3
Based on enumeration size	empty vector	0
	[42]	42

Table 4. Test plan, test cases based on the summation theorem

The test plan we have made is not complete yet. It can be seen clearly in the previous examples that we assumed that the inner function is also free from implementation errors. Now, we add the corresponding test cases of the inner function to our test plan, hoping that we can find possible defects. For the summation, one test data is enough to check both boundaries of the enumeration: we choose a vector which holds only two different values, expecting that the function will return their sum. Let this vector be $[1, 2]$. We also need to check the cases related to the size of the vector. It is an important assumption that

Tool	xUnit	Fixtures	Group fixtures	Generators	Mock	Exceptions	Macros	Templates	Grouping
Bandit		●	●			●	●		○
BugEye						●			●
CATCH		●	●	●		●	●	●	●
doctest		●	●			●	●	●	●
lest		●				●	●	●	○
liblittletest	●	●	●			●	●	●	●
tpunit++	●	●					●	●	
unit.hpp		●		●		●	●		
upp11	●	●				●	●	●	●

Table 5. The abilities of test tools consisting of only one header file.

a summation called on an empty vector returns its neutral value. If the vector contains only one value, the summation should return that value. (see Table 4)

The enumeration size related checks include a vector with two items. Because this case has been already checked before, we do not need to check it again.

Test tool selection

When teaching beginner programmers, it is obvious to choose a test tool which leaves the focus on the programming instead of drawing too much attention to its usage.

In our course, we use the C++ programming language to analogous programming, so we look for a proper test tool according to this language. The list of these tools is surprisingly long, however some tools are especially programming environment-dependent, and most of them need complicated preparations. We want to choose a tool which does not have to be compiled, installed, or integrated into a programming environment after downloading. We have concentrated on the tools which are platform independent, easy to use and consisting of only one

.hpp header file without any external dependencies. Keeping this in mind, students only need to focus on choosing appropriate test cases. We have collected tools (see Table 5) which may suit our needs[1]. We have chosen the CATCH tool from this list because it is documented very well, is easy to use, and last but not least, has more functionalities.

Usage of the Catch test tool

During implementation, a main program can be created to try out functions already implemented. This can read input data, call implemented subroutines, and write out results. Because this method is not the best way to create a test environment which is easy to use and maintain, and can run automated tests, we will use a framework instead. In the last chapter, we chose the Catch tool. To use this tool, we need to include only one header file[2], where we want to create our own test environment. The framework simplifies the test environment creation by generating the main entry point: if we define an empty CATCH_CONFIG_MAIN macro, then a main function will be generated in the background, which will run all the tests written by the user. It is important that, if we make several compilation units, then we should use this macro only once in the project, practically in a separate unit. Tests can be written as test cases which will be used instead of the former main program for given inputs.

Test cases can be defined using the predefined TEST_CASE macro. We must provide each TEST_CASE a unique name which describes the purpose of the test case. Here we can also provide additional tags as a parameter to help the categorization of tests. In the body part, after the function call, we must define our requirements, which can also be made with built-in Catch macros. Among these macros, the easiest ones to use are the REQUIRE and the CHECK. They simply depend on their parameter, which is a simple logic statement (without logic and/or operator) about the result. The difference between them is that, if the checked condition is false, REQUIRE will abort the program while using CHECK, we can continue. Using this basic knowledge, the tool can be used effectively to write simple automated tests. Advanced programmers can also get many useful functions with this tool, but we will not use most of them.
Test case 1. (a trivial test case described with the Catch tool)

[1]https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B
[2]https://github.com/philsquared/Catch/blob/master/single_include/catch.hpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("One equals one", "[int_test]") {
    REQUIRE(1 == 1);
}
```

Output (successful execution):

```
All tests passed (1 assertion in 1 test case).
```

If we successfully run our first "Hello world" example, and we are sure that the test environment works well, then we can start to develop automated tests, describing our requirements as C++ codes. Now, we will see how to use Catch to realize the testing of the example shown in chapter 4, based on its test plan.

Let us begin with the test of maximum selection. The code of each test case looks similar: it initializes the input, calls the appropriate function, and verifies the result. The initialization can also be made from a file, but in these examples, to keep our example clear, the matrix will be defined in the code.

Test case 2. (maximum selection, lower bound check of the enumeration)

```
TEST_CASE("Check lower bound", "[max]") {
    std::vector<std::vector<int>> data = {{10}, {5}};
    int maxSumIndex, maxSum;
    maxRowSum(data, maxSumIndex, maxSum);
    CHECK(maxSumIndex == 0);
    CHECK(maxSum == 10);
}
```

Output (successful execution):

```
All tests passed (2 assertions in 1 test case).
```

The upper boundary check can be made similarly. Now, we write the rest of the automated tests based upon the test plan, so we do enumeration size related checks: we must describe the test cases of the matrices having one, two, or more rows.

The test cases which belong together logically can be collected into one test case, which can be defined by the SECTION macro. The run of each SECTION starts from the beginning of the test case, so we can create a common initialization part to avoid code repetition. Of course, this construction can be used well only

if there is a significant overlap between the initialization parts of the collected cases.

Although, in case of maximum selection, the empty enumeration counts as an invalid input (because it violates the precondition), we can describe this case here because it logically belongs here. An obvious method to deal with an invalid input is that it throws an exception, but a program can also handle this situation differently. Our selected testing tool is also able to check if the expected exception is thrown by the function being tested for the given input.

Test case 3. (maximum selection, enumeration size related cases)

```
TEST_CASE("Checks by enumeration size", "[max]") {
    std::vector<std::vector<int>> data;
    int maxSumIndex, maxSum;
    SECTION("Empty enumeration") {
        REQUIRE_THROWS_AS(
           maxRowSum(data, maxSumIndex, maxSum),
           std::invalid_argument);
    }
    SECTION("Enumeration with 1 element") {
        data.push_back(std::vector<int> {10, 32});
        maxRowSum(data, maxSumIndex, maxSum);
        CHECK(maxSumIndex == 0);
        CHECK(maxSum == 42);
    }
}
```

Tests for matrices containing two or more rows can be described similarly. The test plan also requires test cases to cover the situations when the maximum row sum appears in the middle of the enumeration or it appears multiple times.

Test case 4. (maximum selection, checking the interior of the enumeration)

```
TEST_CASE("Check the middle of the enumeration", "[max]") {
    std::vector<std::vector<int>> data = {{1}, {10}, {2}};
    int maxSumIndex, maxSum;
    maxRowSum(data, maxSumIndex, maxSum);
    CHECK(maxSumIndex == 1);
    CHECK(maxSum == 10);
}
```

Test case 5. (maximum selection, multiple maximum occurences)

```
TEST_CASE("Check multiple maximum values", "[max]") {
    std::vector<std::vector<int>> data = {{2}, {1}, {12}, {12}};
    int maxSumIndex, maxSum;
    maxRowSum(data, maxSumIndex, maxSum);
    CHECK(maxSumIndex == 2);
    CHECK(maxSum == 12);
}
```

Output (successful execution):

```
All tests passed (9 assertions in 4 test cases).
```

We have implemented the test cases related to the outer theorem, but the test coverage is not complete yet. We can only say that the *maxRowSum* procedure seems free from coding mistakes, but we do not have any information about the *rowSum* function yet.

We write the test cases of the inner function and we add these test cases to the automated testing. Because the inner function is based on the summation theorem, it is enough to provide only one input to do both the upper and the lower boundary checks of the enumeration. The cases related to the length of the vector are described with the SECTION macro.

Test case 6. (summation, boundary check of the enumeration)

```
TEST_CASE("Check bounds", "[sum]") {
    std::vector<int> row = {1, 2};
    int sum = rowSum(row);
    CHECK(sum == 3);
}
```

Test case 7. (summation, enumeration size related cases)

```
TEST_CASE("Checks by row size", "[sum]") {
    std::vector<int> row;
    SECTION("Empty enumeration") {
        CHECK(rowSum(row) == 0);
    }
    SECTION("Enumeration with 1 element") {
        row.push_back(42);
        CHECK(rowSum(row) == 42);
    }
}
```

Output (when all tests have been successfully passed):

```
All tests passed (16 assertions in 7 test cases).
```

Let us change one of our assumptions for a moment. This way, we can make sure that the tests are running properly, and we get the expected error messages and the exact places of the failed tests. For example, we assign false expectation to the boundary check of the summation, and we modify the expected value from 3 to 10. By running the tests again, we will see which comparison is problematic.

```
Check bounds
FAILED:
  CHECK(sum == 10)
with expansion:
  3 == 10
test cases:  7 |  6 passed | 1 failed
assertions: 16 | 15 passed | 1 failed
```

Finally, we have managed to successfully cover the developed program with standardized automated tests with which we can identify implementation errors with a good chance. In a more complex program, the code generated with our technique can get an important role because it can serve as its own documentation, showing what we expect as a result from each unit of inputs given. However, we should handle the results cautiously because they can give a false sense of safety about the correctness of the program. Because of this, complex programs should be reviewed.

Conclusion

When teaching beginner programmers, it is a great challenge to make students be able to write working programs properly. In our case, the solution of exercises is shown by using analogous programming methodology, so we solve problems using programming theorems. This method helps not only the analyzing and the planning of the solution but the testing, as well.

In this paper, we have sketched an analogous programming-related testing methodology and an automatic unit-test tool. We have shown how the knowledge of programming theorems can be built into the assembly of test cases, and how an automatic unit-test environment could be used on a C++ platform. By using grey-box testing, which is the mixture of white and black-box testing, we can create well-defined test plans instead of intuitively-created ones. Thanks to this

method, the focus is moved from quantitative tests to qualitative tests, and we get programs of higher quality.

In our courses, we have experienced that a significant number of programs made by students produce wrong results for certain test cases during verification, so we added the testing and its documentation to the assignments. After the introduction of this new requirement, the ratio of programs working correctly has increased. However, reading test reports, we have seen that, if test cases are chosen mostly by intuition, then students are not always capable of revealing the errors or deficiencies of the program.

In case of simple exercises, where the algorithm is designed with analogous programming, the traditional white-box testing can be almost completely replaced by grey-box testing. By checking only grey-box test cases, we can get a surprising confidence about the correctness of the program.

Acknowledgements

The project was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

[1] Á. Fóthi, *Bevezetés a programozáshoz*, ELTE Eötvös Kiadó, Budapest, 2005.

[2] T. Gregorics, Programming theorems on enumerator, *Teaching Mathematics and Computer Science, Debrecen* **8**, no. 1 (2010), 89–108.

[3] T. Gregorics, Abstract levels of programming theorems, *Acta Universitatis Sapientiae, Informatica* **4**, no. 2 (2012), 247–259.

[4] T. Gregorics, *Programozás 1.kötet Tervezés*, ELTE Eötvös Kiadó, Budapest, 2013.

[5] T. Gregorics, S. Sike, Generic algorithm patterns, *Proceedings of Formal Methods in Computer Science Education FORMED 2008, Satellite workshop of ETAPS* (2008), 141–150.

[6] R. Patton, *Software testing. Second Edition*, SAMS, 2005.

[7] I. Sommerwile, *Software Engineering. Eighth Edition*, Pearson Education Limited, 2007.

TIBOR GREGORICS
EÖTVÖS LORÁND UNIVERSITY, FACULTY OF INFORMATICS
HUNGARY 1117 BUDAPEST, PÁZMÁNY PÉTER SÉTÁNY I/C

E-mail: gt@inf.elte.hu

KRISZTIÁN MÓZSI
EÖTVÖS LORÁND UNIVERSITY, FACULTY OF INFORMATICS
HUNGARY 1117 BUDAPEST, PÁZMÁNY PÉTER. SÉTÁNY I/C

E-mail: mozsik@inf.elte.hu

RUDOLF SZENDREI
EÖTVÖS LORÁND UNIVERSITY, FACULTY OF INFORMATICS
HUNGARY 1117 BUDAPEST, PÁZMÁNY PÉTER SÉTÁNY I/C

E-mail: swap@inf.elte.hu

(Received April, 2018)