WIP: Fully and Automatically Testable Tasks for Programming Novices Derived from a Modified Test-First Approach

Matthias Längrich
University of Applied Sciences
Zittau/Görlitz
Faculty of EE and CS
Brückenstr. 1, 02826 Görlitz,
Germany
m.laengrich@hszg.de

Jörg Schulze
University of Applied Sciences
Zittau/Görlitz
Faculty of EE and CS
Brückenstr. 1, 02826 Görlitz,
Germany
joerg.schulze@hszg.de

Heinz Dobler
University of Applied Sciences
Upper Austria
Faculty of Informatics,
Communication and Media
Softwarepark 11, 4232
Hagenberg, Austria
heinz.dobler@fh-hagenberg.at

Jean D. Hallewell Haslwanter
University of Applied Sciences
Upper Austria
School of Engineering
Stelzhamerstr. 23, 4600 Wels,
Austria
jean.hallewell@fh-wels.at

Abstract—This innovative practice paper presents an approach for fully and automatically testable tasks designed for programming novices, derived from a modified test-first approach. At our Universities of Applied Sciences, students from software engineering and various non-computer science degree programmes are introduced to the basics of programming using the C# programming language. The primary educational goal is to enable students to develop and test static methods for simple algorithmic problems. This is achieved through a modified test-first approach that emphasizes the practice of solving clearly defined tasks. The paper details the development of a task-oriented teaching method, the creation and organization of extensive task collections, and the implementation of a computer-assisted system for generating and verifying student tasks. The effectiveness and challenges of this approach are discussed, and potential for application in other programming languages is considered.

Keywords—programming education, automated assessment, test-driven development

I. Introduction and Challenge

A. The Challenge of Our Programming Education

At our universities of applied sciences, engineering students are taught the basics of programming. The course is offered at the beginning of their studies. Students do not all start at the same level. Some have no prior knowledge, while others may have already learned programming at another educational institution and even applied it practically. Students have achieved their learning goal when they can develop and test static methods for simple algorithmic problems using a modified test-first approach, as they can use the syntax and understand the semantics of the main parts of C# by the end of the semester (15 weeks). Therefore, we use C# terms for various aspects in this paper.

The course only covers passive object orientation, meaning up to the use of classes. However, simple tasks of active object orientation (such as programming elements of classes) are also possible using the concepts presented.

The integration of exceptions is possible (see [1]). Elements such as events (and therefore interfaces), parallel algorithms, network programming, and other concepts of advanced programming education are also not covered.

This paper focuses on a part of the tasks used for this purpose and the way they are structured. The delivery of external knowledge, such as lectures, educational videos, textbooks, or AI/ghostwriters, is not considered. Therefore, the role of the lecturer (lecture) is not examined. The tasks addressed do not deal with expressions[1].

In the computer exercises, there are two ways to solve the given tasks: The first option is to solve the tasks on paper or in a spreadsheet (notation variant). The second option is to solve the tasks in a practice environment specifically developed for this purpose (implementation variant)[2]. In the exercise, students can ask a human tutor if they have questions. AI/ghostwriters as tutors are not discussed in this paper. Ideally, the tutor would be supported by the ability to automatically check the solutions to all tasks. The tutor can also advise students on developing alternative solution approaches.

The notation variant (abbreviated N) of a sub-task indicates the representation of the solution on paper or an equivalent medium, e.g., a spreadsheet workbook template. This is called student spreadsheet template (SST). The notation in the equivalent medium and on paper should be identical. The realization variant (abbreviated R) of a sub-task indicates the representation of the solution in a carrier, here a Visual Studio project map for programming. This is called Visual Studio template (VST). R should be transformable into N and vice versa using a simple

[1] For expression tasks please see 2.
[2] Already introduced in 2006 by 1,3. Also see 4.

algorithm. The use of AI/ghostwriters to solve tasks in the exercise is prohibited, as they are not allowed during the exam (written, no tools).

The teaching approach practiced was adopted from mathematics where it has proven successful for generations: the principle of practicing with clear tasks [4]. The competencies programming novices need to learn are mentioned in [5] and are generally accepted. [6] defines the term competence in a way that is also understood in this paper: as a set of tasks that one can solve if one possesses the competence.

The final examination of the students takes place at the end of the semester through a 120-minute written exam. [7] discusses the problem of testing programming competencies through handwritten exams and instead proposes the use of exams conducted on computers. However, the legal situation for digital exams in Germany is inconsistent and vaguely formulated, which is why digital exams have not been used so far.

B. Formatting Note

Source code is formatted with Courier New and without indentation for better readability. No blank lines are inserted before or after. This saves space and avoids repetitions. Tabs are converted to two spaces. Usual blank lines are omitted. If the source code exceeds the available column width, it is displayed in multiple lines, with an additional indentation used.

C. General Definitions of Task and Task Type

A task is defined, according to [6], as the independent solution of the task using specific aids (formula collection, lecture notes, textbook, script). A task consists of the following components: given information and an operator (activity to be performed) as well as the solution presentation. Further information about tasks of the same type is summarized in a task type description, as described in [8]. The operator was deliberately chosen and addresses a competence required for a programming novice. In this teaching approach, the identified required competencies were also examined for their corresponding operators. It was checked whether an operator is "technically atomic", i.e., cannot be reasonably broken down into sub-operators. Task types and tasks were developed for these atomic operators.

[8] discusses the use of AI for simple programming tasks. However, if students no longer solve these tasks themselves but let AI do it, they only take on the role of an observer. This contradicts the competence understanding of [6], as it is no longer ensured that students can solve the tasks themselves.

Following [6] students possess a competence if they can solve a set of tasks using defined aids. A ghostwriter and AI are not among the defined aids as the principle of independence is violated. According to [6] a competence is a latent property that cannot be measured directly. Instead, indicators are identified that suggest the presence of competence. The indicator used here is called a task. All tasks in the task collection can be assigned to task types. All task types are assigned to competencies (see [2]). Task types act as a link between competencies and tasks and were discussed in [9].

The authors distinguish between internal and external unit test programming tasks (IUTP and EUTP). The term "external" describes programming tasks found in standard textbooks. These usually do not contain all the information demanded by the authors. Internal tasks are self-developed, formalized, and equipped with all necessary information. The following is an example of an external task, specifically an EUTP:

```
Specification: Compute {0} the sum of
all non negative elements of {1}.
Forbidden Types: int, char, Collections
```

Placeholders were inserted to identify the variables. Although this task was slightly modified, its didactic value was not diminished. The word "specification" describes the category of the given information.

Preliminary limitation of types:

1. Simple value types: bool, int, char, and double
2. Simple reference types: string and StringBuilder
3. Arrays: Arrays of simple value types with rank 1 and 2
4. Collections: Generic collections with simple value types, specifically List only.

In chapter V.C on generators, further restrictions for the various categories of information are discussed.

The following is a possible transformation (modeling) into an IUTP:

```
In 1: double[] x;
Out 0: double s;
Requires: x != null
Membername, Modifier, Kind: SumIf,
public static, Method
Spezification: Compute {0} the sum of
all non negative elements of {1}.
Forbidden: .Sum, return
Inputs 1, 2:
```

Inp	Sourcecode
[1]	`x = new double[] { 2, -1, 3, };`
[2]	`x = new double[] { };`
[3]	`x = new double[] { -0.001, };`
[4]	`x = new double[] { 0.001, };`

In all spreadsheet templates shown, cells with labels are displayed in light blue, cells with calculations in light gray, and cells with values in light green. The use of "Inp" instead of "Inputs" is due to space constraints. Multiple lines are possible, and it is recommended to write each variable in its own line within the cell. This also applies to the expected output.

In addition to tasks with a unit test, there are also tasks that require a test with user input. This enhances recognizability with tasks from usual textbooks. Other testing options, such as tests with input initialization and output generation or debugging, are not discussed.

The word "out" is used in place of "return." There is a simple reason for this: extracting a part of a statement list results in a method with `in`, `inout`, and `out`. The `return` is not sufficient in the general case. For expressions without side effects, any sub-expression can be extracted into a method with `return` and without `inout` and `out`. However, with side effects, the necessity arises for `return` with `inout` and `out`. Extraction tasks (see [9]) and insertion tasks would be an interesting addition to the task types.

The EUTP consists of the modeling step (MS) and the solution of the IUTP, which is the solution of the modeling step. Every IUTP can be interpreted as the result of an EUTP through the modeling step. This is helpful when checking the result of the modeling step.

In [9], the goal of the modeling step was only to generate a variable range. The generation of the input was postponed to subsequent steps. However, since the IUTP includes the input, it made sense to integrate the generation of inputs into the modeling step. In this context, we assume for arrays and collections that the first input is a normal input and the second input is a minimal input for the size of the arrays and collections that fulfill the requirements.

The given information is not necessarily limited to what is visible here but can be supplemented with additional attributes if needed. Section V.C The Generator explains the components of this internal task:

Variable range: (corresponds to the domain and co-domain in mathematics) is specified. This includes input (In, InOut), output (InOut, ReturnOut), as well as requires: the actual precondition for the static method[3]. From the input/ output, one can generate the method signature (or just signature) and conversely, from the signature, deduce the input/ output without numbers and without requires (see section III.A). The variable range, as illustrated in the example, can easily be converted into a signature, but the reverse is not possible without additional information. This provides the motivation for requiring the variable range in the tasks, rather than the signature.

Inputs: A set of inputs for testing. The expected output must be specified for the given numbers. The categories are explained in detail in the section Task Creation by the Tutor.

D. Our History

1) Up to 2006:

The 2006 publication already included a function fragment checker, which could verify student results of an internal programming task using an internal compiler [1]. For this, a solution statement list, as well as inputs and a signature, were provided. Similar to well-known products like LeetCode or Exercism, the statement list was embedded in a class with an empty body.

The information needed for students to solve a task was spread across multiple documents, which was not optimal. In addition to the task document, there was a document describing the task type. This separation historically proved advantageous during the development of the task collection. All information that tasks of the same type share can be summarized in the task type document. This way, the description of an individual task could be minimized without losing formal accuracy. As with superclasses in object-oriented programming this approach collects all common attributes in a superclass, reducing the needed information in a child class to a minimum, without losing accuracy. The two task types used at the time were the specification task (a specification to be tested) and the value task (a simply verifiable value).

2) 2007: The principle of practicing.

With the principle borrowed from mathematics of practicing by solving many tasks, the need arose for a collection of programming tasks [4]. Additionally, the necessary prerequisites had to be established for this purpose.

[3] Please see 10 for more details.

3) 2014: Presentation of a process for the systematic development of task types.

In 2014, an approach was presented by [10] on how valid task types could be developed from required competencies and then tasks derived from them using a test-first approach. Also, a theoretical fundament was laid. Clear structures were developed for task formulation and solution presentation. Due to its numerous tasks, this teaching approach was also described as "task-oriented". The task types "calculation" and "correction" were added to the existing task types. Both types are not discussed in this paper.

4) Today: The innovative practice.

The task creation process and the task types have evolved. The notation variant N is now machine-readable using a student spreadsheet template. Previously, it was exclusively written and evaluated on paper. This development was a direct consequence of the COVID-19 pandemic. The student spreadsheet templates for the notation variant are generated from the task description and the source code of the solution. Separate tasks are formulated for each section of the programming process, which are formally described in the respective task type descriptions. Each task type has its own checker, allowing all tasks to be machine-verified. The approach presented here uses a very formal task description format reminiscent of mathematics. The solution presentation is also highly formalized.

II. Related Work

A. Two of several online exercise systems: LeetCode and Exercism:

LeetCode is a learning and practice environment that can be used partially for free. It offers a limited selection of programming languages. Exercism is free to use and has an extensive selection of programming languages. The inputs and outputs are provided. The authors have not noticed any "requires" in Exercism. LeetCode uses the term "requires" in a different way than it is used here (see III.D for more details concerning our task description).

There are two different ways of executing the code: the test mode with a few test cases and the submission mode with many test cases. The failed test cases are displayed. The tasks on LeetCode were perceived as more challenging and are likely aimed more at computer scientists. Both products use a semi-intelligent editor but not a full-fledged IDE. From the authors' perspective, this has the disadvantage that assistance is provided for activities that should actually be part of the training.

B. Literature

There are many approaches to developing tasks for programming novices, as mentioned in the overview works of [9,11]. The commonality of these approaches is that no programming competencies are assumed. The technical and personnel requirements are often similar: many students are accompanied by only a few human tutors, which can lead to a problem in the quality of supervision. For further study of other exercise environments, please refer to the mentioned literature.

III. Decomposition of the internal unit test programming task into substeps/ subtasks

The goal of the R variant is to extend a given VST, which consists of three classes: "Rahmen"[4], "UTRahmen", and "UTHelp"[5].

The file Rahmen.cs contains the following source code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JS.Test
{
  public class Rahmen
  {
  }
}
```

Similarly, the following class for the unit test is used (UTRahmen.cs).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.VisualStudio. Test-
Tools.UnitTesting;

namespace JS.Test
{
  [TestClass]
  public class UTRahmen
  {
    [TestMethod]
    public void TestSignatur()
    {
    }
  }
}
```

The following approach is used: The heading is composed of R, N, or R/N, followed by the full name and finally the short name of the task in parentheses. In the following text, only the short name is used.

In the first paragraph, the information necessary for solving the subtasks is provided. For certain tasks, there are easily identifiable algorithms that yield a unique result. The variant for which the algorithm applies is specified. In the N variant, the solution is given as ObjectLinking to the student spreadsheet template, with the link being deactivated. In the R variant, the classes UTRahmen or Rahmen are specified.

A. N/R: Transformation of the variable range into a signature unit test (UTSignatur)

Required information: variable range.

This step was not yet discussed in [10]. This leads to a minimization of dependencies between the separate steps. Another effect was an improvement of student's results in the final exam.

The method in the class Rahmen is constructed with the standard initialization of the output in the order defined by the variable range. The principle of variable order is applied everywhere multiple variables occur.

The source code of the test is created in the test function. The following substeps in the N-variant are carried out:

1. Declaration of the input variables, output variables, and expected output variables. The expected output variables match the name of the input variables and are marked with a preceding "e". Each declaration of a variable from the variable range is specified on a separate line for clarity.
2. Initialization of the input with the default values of the type.
3. Initialization of the expected output with the default values of the type.
4. Execution of the method that leads to an initialization of the actual output using the default values of the type.
5. Comparison of the expected with actual output, and a class of assert methods from AssertJS to easily identify problems.

UTS-1	**Sourcecode**
[1]	#StatementList
[2]	double[] x;
[3]	double s, es;
[4]	x = null;
[5]	es = 0.0;
[6]	SumIfSignatur(x, out s);
[7]	AssertJS.AreEqual(es, s, 1E-6, "es", "s");
[8]	#ClassMemberDeclarations Rahmen
[9]	public static void SumIfSignatur(
[10]	in double[] x,
[11]	out double s)
[12]	{
[13]	s = 0.0;
[14]	}

Instead of "UTSignature-1," "UTS-1" is used for space reasons. "1" is the number of the task.

[4] The word "Rahmen" translates to "frame" in English.

[5] UTHelp will not be discussed in more detail.

1) N variant

UTS-1	Sourcecode
[1]	#ClassMemberDeclarations Rahmen
[2]	public static void SumIfSignatur(
[3]	in double[] x,
[4]	out double s)
[5]	{
[6]	s = 0.0;
[7]	}
[8]	#StatementList
[9]	double[] x;
[10]	double s, es;
[11]	x = null;
[12]	es = 0.0;
[13]	SumIfSignatur(x, out s);
[14]	AssertJS.AreEqual(es, s, 1E-6, "es", "s");

The lines beginning with # indicate the categories of the source code: #StatementList the statement list being sought, #ClassMemberDeclarations Rahmen additions are located outside the implementation method.

2) R variant

```
public class Rahmen
{
  public static void SumIfSignatur(
    in double[] x,
    out double s)
  {
    s = 0.0;
  }
}

[TestClass]
public class UTRahmen
{
  [TestMethod]
  public void SumIfTestSignatur()
  {
    double[] x;
    double s, es;
    x = null;
    es = 0.0;
    SumIfSignatur(x, out s);
    AssertJS.AreEqual(es, s,
      1E-6,"es","s");
  }
}
```

These two forms of representation make it clear that the N and R variants can be easily converted into one another.

B. N: Calculating of the expected outputs (COutput)

Required information: variable range, specification, inputs.

COut-1	Sourcecode
[1]	es = 5.0;
[2]	es = 0.0;

Only the outputs for the numbers from the list after the label "Inputs" need to be provided. No input is allowed during initialization. "COut-1“ is an abbreviation of "COutput-1".

C. R: Transformation of the signature test, the inputs, and the expected outputs into unit tests (UT)

Required information: UTSignature, inputs, outputs.

For the first input, only the first unit test method is provided. A separate test method is implemented for each required input from the inputs. Only the new ClassMemberDeclarations are specified.

```
public class Rahmen
{
  public static void SumIf(
    in double[] x,
    out double s)
  {
    s = 5.0;
  }
}

[TestClass]
public class UTRahmen
{
  [TestMethod]
  public void SumIfTest1()
  {
    double[] x;
    double s, es;
    x = new double[] { 2, -1, 3, };
    es = 5.0;
    SumIf(x, out s);
    AssertJS.AreEqual(es, s,
      1E-6,"es","s");
  }
}
```

The following approach can be applied as an algorithm:

1. The method in class Rahmen is copied, the word "signature" is removed, and the initialization of the output is done with the first expected output.
2. The method in the class UTRahmen is copied and renamed. Then, the input is initialized with the first corresponding input, and the output is initialized with the first expected output.

The approach is so straightforward that the corresponding notation form does not need to be tested. Due to the dependence on the correct solution of Steps 2 and 3, it is also not advisable to test in the notation form. In [10], this step was still performed as a notation form, but it is no longer necessary now.

D. N/R: Internal programming task (IP)

Required information: specification, UTSignature.

1) N variant

IP-1	Sourcecode
[1]	#StatementList
[2]	int n;
[3]	int i;
[4]	n = x.Length;
[5]	s = 0;
[6]	for (i = 0; i <= n - 1; i++)
[7]	{
[8]	s += x[i] >= 0? x[i]: 0;
[9]	}

2) R variant

The initialization of the output is replaced by the statement list.

```
public class Rahmen
{
  public static void SumIf (
    in double[] x,
    out double s)
  {
    int n;
    int i;
    n = x.Length;
    s = 0;
    for (i = 0; i <= n - 1; i++)
    {
      s += x[i] >= 0? x[i]: 0;
    }
  }
}
```

Students are advised to check the following aspects when implementing a statement:

1. Are all variables are declared once (D) or present in the parameter list?
2. Are all input variables in a statement initialized (I)?
3. Is the statement syntactically correct (S)?

E. Correction DIS (CDIS)

Required information: variable range, statement list with declaration, initialization or syntax (DIS) errors.

The DIS principle can be used for syntax-type correction tasks, as these are easily verifiable. However, no specification needs to be provided for the task. Correction tasks are not covered here.

F. Calculation of a Statement List (CStmt)

Required information: variable range, inputs, statement list.

This task type is also not discussed in detail, with a reference to the contribution from [10]. The notation can be structured in a way that makes the result clearly verifiable.

IV. Description of the Spreadsheet Templates (ST) for IUTP

There are two types of student templates: notation and realization. The notation template is a spreadsheet, and the realization template is a Visual Studio project. In addition to the student templates, there is a tutor template, which contains only the task description and the statement list of the solution, from which the student templates are generated.

The student spreadsheet template (SST) contains, visible to the student, the task description and the templates for the notation sub-tasks. The analysis of the input/output and the solution are read-only, among other elements. In the student template, the task is write-protected.

The generator app generates the SST after the successful analysis and verification of the data from the tutor template. The check app verifies the SST for correctness.

The student can develop his/her solution either in the spreadsheet and check it with an app or develop it in Visual Studio. In the latter case, only testing against each other is possible. Copying back and forth between the spreadsheet and Visual Studio is also an option. This is of interest because in the spreadsheet, a verification of the results is performed, while in Visual Studio, only the consistency against each other can be checked.

V. Task Creation and Application with Checker and Generator

The InOut characterization consists of In, InOut, Out, and Return. The variable range includes the InOut characterization and the Requires. The Requires corresponds to Bertrand Meyer's (ETH Zurich) "Design by Contract" principle [12].

In the InOut characterization, an initialization is not permitted, i.e., no "=". Only declaration statements without initialization are allowed.

A. General Remarks

The provided source code, which are compiled with the internal compiler, are given without the class framework, namespace framework, and "using" directives for brevity. The procedure for creating a IUTP is now explained using our example.

B. Task Creation by the Tutor

The tutor creates a copy of the workbook IUTPEmpty into a new personal folder and renames the workbook according to their needs, with "IUTP" recommended at the beginning. The file name extension "Res" for Result is mandatory. Different templates are used for EUTP, correction or calculation tasks. Alternatively, the tutor can also copy a different template from an existing task. The tutor template contains three sheets: Task-1 (Sheet with the formulation of the task, see the example task) and IP-1Res (Solution of the internal programming task – Sheet with source code of the statement list of the solution and potentially called methods). If multiple successive sub-tasks are used, the two sheets are copied and the numbering continues. Only changes to the tasks are noted, e.g.

~~**Forbidden:**~~
Forbidden: `for, while, goto, do, return`

After filling out the two sheets the tutor starts the generation app. The app first checks whether the task specifications and the solution are consistent. For this, the technology explained during the discussion of section I.D is used. Therefore, the generated source code to be checked are provided. If these conditions are met, the student template can be generated.

C. The Generator

All procedures are demonstrated using the example.

First, a copy of the tutor template is created, and the "Res" before the file extension is removed.

Explanation of Components in the order of the check:

Subject (Topics): Assignment of topics. Must not be specified and is not analyzed.

Specification: The specification contains placeholders in curly brackets characterizing the intended parameters. The numbers starts with 0 and without gaps.

InOut-Characterization: Type and names of parameters: Consists of In, InOut, Out, and Return (not yet implemented) with a number list before the colon. After the colon, for In, InOut, and Out, there are declaration statements. These must be syntactically correct. The types must be allowed types. Each declared variable can be uniquely assigned a number. After the colon, there is a type for Return. The type must be syntactically correct and an allowed type. The fact that Return has not yet been implemented is not a significant limitation because exactly one Out can be converted into a Return and vice versa. An algorithm can be formulated for this. The number lists are an average external coverage of the number list resulting from the specification. The specification number list is identical to the coverage. The names of the parameters must be pairwise different.

The verification is done in two steps. First, the syntactic correctness of the declaration statements is checked through the source code, where this time the classes in the namespace "JS.Test" are specified.

```
public class TestIn
{
  double[] x;
}
public class TestOut
{
  double s;
}
```

Upon successful compilation, the types and names of the variables can be extracted using the Reflection class and assigned to the numbers. This results in the following table:

Vars	Name	Type	Charac
[1]	s	double	Out
[2]	x	double[]	In

When creating the table, it is checked whether variable names are duplicated and whether the types meet the restrictions. This is saved with additional information on a worksheet. From this information, the unit test (UT) for the signature can be generated. See Section III for the corresponding sub-step. Unnecessary sheets will be removed by the checker after a dialog was shown.

Requires: The preconditions are a Boolean expression, may only use input parameters (In, InOut), and must be syntactically correct. The following source code is used for this purpose:

```
public class CheckRequires
{
  public static bool CheckRequires(
    in double[] x)
  {
    return x != null;
  }
}
```

Membername, Modifier, Kind: The given information: Membername is a valid C# name that matches the correct kind. Modifiers are modifiers that match also the correct kind (currently only "public static"). Kind is currently only "Method". The combination of the three must be syntactically correct.

```
public class CheckNameModifierKind
{
  public static void SumIf()
  {
  }
}
```

In the case of return we have the checked return type and can construct a return statement with the standard value of this type.

Forbidden: Forbidden names. List of substrings that must not appear in the solution of the IP. Reflections are also excluded.

Inputs: With a list of valid input numbers. For these numbers, an expected output must be specified. All inputs are used by the check app for the unit test. The table contains the initialization of all output parameters in one cell for each number of the input. This can be indicated in one line (with a line break) or in several lines in one cell. One line is recommended for space reasons. We use the following check only for the 1. Input. The other inputs are created similarly.

```
public class CheckInputs
{
  public static bool Input1(
    out double[] x)
  {
    x = new double[] { 2, -1, 3, };
  }
}
```

With this, all parts of the task description have been checked. Next is the check of the solution, assuming that the tutor provides a correct solution. The expected output is determined from this solution. The following source code is used, which can be generated with the help of the InOut characterization analysis:

```
class CheckImplementation
{
  public static void SumIf(
    in double[] x;
    out double s)
  {
    int n;
    int i;
    n = x.Length;
    s = 0;
    for (i = 0; i <= n - 1; i++)
    {
     s += x[i] >= 0? x[i]: 0;
    }
  }
}
```

Before the syntax of the source code is checked, it is verified that the forbidden strings do not appear. The results of the outputs for the inputs are displayed in the form of an initialization to allow the tutor to assess the correctness of their solution. These initializations are saved as the expected output.

This completes the verification process, and additional worksheets for the student template can then be generated.

For EUTP, calculaction and correction tasks additional worksheets are needed. Due to space constraints, these types cannot be covered in this article. After complete analysis and checking, the student template can be generated. The tutor is advised to fill out the template themselves and check it after generating the student spreadsheed template and before presenting the template to the students.

D. Task Processing and Checking by the Student

In the student workbook, the solutions are not displayed but must be worked out by the students. Possible solutions were shown in the example above. A syntax check is performed as described in section V. The signature test is a string comparison. In COutput, syntax is checked as described for the generator. The values are compared against the stored values. In IP, the student solution is treated like the tutor solution but is then verified with a unit test.

This completes the processing of the student template.

VI. Discussion of obtained results

The procedure presented here was developed during the COVID-19 pandemic when practicing on paper was temporarily not possible. In this way, students could switch to using the computer as their working environment, which was also useful for working from home.

This approach was applied to over 200 engineering students who were introduced to the fundamentals of programming in C#. After initial uncertainties with the unfamiliar tools, the students quickly adapted to the new environment and came to appreciate the advantages of working on the computer.

For tutors, the process of creating a task has changed. Although the numerous steps outlined in this paper may seem complicated, the process is ultimately clear, formalized, and includes checks. After some initial training, the time invested in learning the process pays off when creating new tasks.

No change in the final exam results has been observed so far, and none was to be expected, as the tasks remain the same, only now they are created and handled differently.

VII. Summary and Outlook

Developing a task collection for programming novices is labor-intensive. The support of a computer-assisted generator is successful if a sufficiently formal description of what is to be generated is available. The approach presented here automatically generates the student template including the required tests from a tutor template.

Due to the long experience with this technology, it can be assumed that this concept can be transferred to other programming languages since the idea is not language-dependent.

As hinted at in various sections, the work in this area is not yet complete and will continue. The new insights gained will be taken into account in future work.

Acknowledgments

We thank the anonymous reviewers for their feedback. In particular, the suggestion regarding Leetcode led to immediate new insights. The other suggestions also contributed significantly to the improvement of the paper.

The authors would also like to extend their heartfelt thanks to the following individuals for their valuable contributions to this scientific work. Their support and expertise have been instrumental in the success of this work: Prof. Dr. em. Franz Schott and PD Dr. Azizi Ghanbari for their contributions to competence orientation and tasks.

The author Jörg Schulze would like to thank his father Günter-Otte Schulze for his strict formalism and his Ph.D. father Prof. Dr. Michail Ilitsch Zelikin. Both of them were his teachers in mathematics.

Finally, we would like to thank the students who tested this approach and provided us with valuable feedback.

References

[1] Längrich, Matthias, Jörg Schulze, and Amruth N. Kumar. “Expression Tasks for Novice Programmers. Turning the Attention to Objectivity, Reliability and Validity.” 10/21/2015-10/24/2015. *Proceedings of the Frontiers in Education Conference*. Piscataway, NJ: Institute of Electrical and Electronics Engineers (IEEE), pp. 300–307.

[2] Längrich, Matthias and Jörg Schulze. “A Systematic Approach to Immediate Verifiable Exercises in Undergraduate Programming Courses.” 10/28-10/31/2006. *Proceedings of the 36th Annual Frontiers in Education Conference*. Piscataway, NJ: Institute of Electrical and Electronics Engineers (IEEE), pp. 1112–1116.

[3] IEEE-CS. ed. 2007. *Proceedings of the 37th Annual Frontiers in Education Conference.* Piscataway, NJ: Institute of Electrical and Electronics Engineers (IEEE).

[4] Schulze, Jörg, Matthias Längrich, and Antje Meyer. “The success of the Demidovich-Principle in undergraduate C# programming education.” 10/10-10/13/2007. *Proceedings of the 37th Annual Frontiers in Education Conference*. Piscataway, NJ: Institute of Electrical and Electronics Engineers (IEEE), pp. 1051–1056.

[5] Kumar, Amruth N. and Rajendra K. Raj. “Computer Science Curricula 2023 (CS2023): The Final Report.” In SIGCSE 2024: The 55th ACM Technical Symposium on Computer Science Education. 20 03 2024 23 03 2024. *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*. New York, NY, USA: ACM, pp. 1867–1868.

[6] Schott, Franz and Shahram Azizi Ghanbari. 2012. Bildungsstandards, Kompetenzdiagnostik und kompetenzorientierter Unterricht zur Qualitätssicherung des Bildungswesens. Münster: Waxmann.

[7] Bourke, Chris, Yael Erez, and Orit Hazzan. “Executable Exams.” In SIGCSE 2023: The 54th ACM Technical Symposium on Computer Science Education. 15 03 2023 18 03 2023. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. New York, NY, USA: ACM, pp. 381–387.

[8] Wermelinger, Michel. “Using GitHub Copilot to Solve Simple Programming Problems.” In SIGCSE 2023: The 54th ACM Technical Symposium on Computer Science Education. 15 03 2023 18 03 2023. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. New York, NY, USA: ACM, pp. 172–178.

[9] Längrich, Matthias and Jörg Schulze. “Rethinking Task Types for Novice Programmers.” 10/23/2014-10/25/2014. *Proceedings of the Frontiers in Education Conference*. Piscataway, NJ: Institute of Electrical and Electronics Engineers (IEEE), pp. 2597–2604.

[10] Ruf, Alexander, Marc Berges, and Peter Hubwieser. “Classification of Programming Tasks According to Required Skills and Knowledge Representation.” In A. Brodnik and J. Vahrenhold (eds). 2015, *Informatics in Schools. Curricula, Competences, and Competitions*. Vol. 9378. Cham: Springer International Publishing (Lecture Notes in Computer Science), pp. 57–68.

[11] Fincher, Sally A. and Anthony V. Robins. eds. 2019. *The Cambridge Handbook of Computing Education Research:* Cambridge University Press.

[12] Meyer, B. 1992. “Applying 'design by contract'.” *Computer*. Vol. 25 (10), pp. 40–51.