Annales Mathematicae et Informaticae
***48** (2018) pp. 23–32*
http://ami.ektf.hu

A web-based programming environment for introductory programming courses in higher education

Győző Horváth

Eötvös Loránd University, Faculty of Informatics
gyozo.horvath@inf.elte.com

Submitted March 5, 2018 — Accepted September 13, 2018

Abstract

Choosing the right programming environment has a great influence on the efficiency of the educational, learning and problem solving processes. While there are many good examples for such environments for the younger generation, which involve block-based programming, gamified learning, appropriate language of the tasks and user interface design, introductory programming courses in higher education rarely take into account the role of the programming environment. In this article we have analyzed a typical problem solving process in an introductory programming course with a special focus on the programming environment. We have found that many distracting factors may make the learning process difficult.

Based on our investigation we introduce a web-based programming environment which takes into account the special needs of newcomers to the programming land. This environment tries to exclude the distracting factors and support the problem solving process in a right way. Beside our methodological considerations, the technical background of supporting traditional programming languages, such as C++, in the web browser is also presented. Finally we make methodological recommendations how this tool can be a part of the teaching and learning process through different types of tasks and learning organizing methods.

Keywords: web, teaching, programming, development environment, higher education

MSC: 97Q60, 97B40, 97U70

1. Introduction

Choosing the right programming environment has a great influence on the efficiency of the educational, learning and problem solving processes. There are many good examples for such environments mainly for the younger generation, taking into account the specific needs of users of these ages. The user interface design is appropriate: nice and colourful for the youngest ones, or comes in different thematic flavours from toys, computer games or films (e.g. [1]) for teenagers. The task descriptions are usually simple and straightforward according to age of the students. The chosen programming language is mainly block-based (like [2]) for the introductory lessons, because it hides the syntactic difficulties behind the blocks, and beginner students only have to deal with the semantic meanings of the blocks, and how to place them one after another or inside to achieve the task. And finally, task solving is often wrapped in a gamified clothes, making the learning process fun and challenging (e.g. [3]).

Introductory programming courses in higher education, however, often considers novice programmer students, as if they have a lot of experience in handling complex processes, but usually this is not the case, no matter how much this would be expected. Treating them as "mature" programmers involves: using code-based programming language from the very beginning, giving them mathematical problems to solve, and using professional or professional-like integrated development environments (IDE), while students need to cope with the more essential mental model of programming. It is hard for the students without any former programming experience to take such big steps in many areas of the programming field. Curriculum should pay attention to gradually introduce newer and newer topics, in order to evenly distribute the cognitive load and thus make the knowledge processing much more effective by students.

Programming, however, is not just coding, it is part of a more general task, problem solving. Problem solving begins with the interpretation of the *task description*, continues with abstracting out and describing the data and their relations contained therein (*specification*), then it provides a solution as a sequence of elementary steps in an abstract language (pseudo-code, *algorithm*), which is finally implemented in the given or chosen programming language (*coding*). Problem solving, however, does not end with this latter step even in a narrower scope, as we have to make sure that the program works correctly by *testing* it, and the detected errors need to be *correct*ed. For smaller programs and tasks, the problem solving may end here. Introductory courses require such programming environments that support both the basic steps and skills of problem solving and the coding phase at the same time.

In this article first we analyse a typical problem solving process in an introductory programming course with a special focus on the programming environment. After this and based on our investigation, we look for better alternatives than using the traditional IDEs, and propose a programming environment which tries to meet the required expectations.

2. Analysing traditional programming environments

In this chapter a typical problem solving process will be analysed, assuming that it takes place in an introductory programming lesson in higher education with students with different preconceptions about programming and problems solving. In order to serve this kind of heterogeneity, the course needs to introduce every concept from the basics to build a systematic knowledge common for everyone. Accordingly, the tasks are relatively small, even at the end of the course there are no tasks that need to be disassembled into several files.

Let us review the problem solving process from the aspect of the tools and development environments. The first step is to get to know the task. This can be done verbally, written on a board, or projected to the wall. The *task description* can be paper-based or digital. Digital material may be published on a general website or in a dedicated task library. Its format can be any of the well-known document formats (HTML, PDF, docx, etc.). Typically, the task description can be accessed in a different software environment from the one where implementation would take place.

The next two steps, *specification and algorithm*, are designed for planning. Planning is traditionally done on paper or on a board. As a new phenomenon, however, it is increasingly common for students to write notes on their digital devices and, on the one hand, they do not have an exercise book or pen, and on the other hand, performing the above two design steps with traditional editors (e.g. text or image editors) is a much larger task than doing it manually. Thus, students are prone to skip this planning step more and more frequently. However, this article does not want to address this issue. The message of this part of problem solving is that planning considerations are implemented in a different environment or tool than coding.

The spectacular and creative part of problem solving is the *implementation phase*, when the plan (pseudo-code) turns into code. This step often occurs in a development environment. These specific environments are chosen because they contain all the whistles and bells needed for convenient development in the chosen programming language, as opposed to a generic code editor, where setting up a basic programming session is a time-consuming task and often the user interface does not support simple usage scenarios.

Integrated Development Environments (IDEs) are prepared with tools for editing, compiling, and running certain types of programs (such as CodeBlocks, Visual Studio, Netbeans). They are convenient to use and the default settings are often sufficient. Their big disadvantage, however, is that they are designed to write much more complex programs than an initial course needs. Usually a single file is enough for solving a simpler task, but IDEs generally think in the concept of projects with many files. Their interface is often very complicated, as they provide many functionalities through menus, toolbars, panels, and settings. They give much more than is needed, and this may distract the attention.

Another thing that can make the usage of any kind of desktop environment

problematic is the installation process. Editors should be installed in classrooms, and they need to be installed at home computers by the student. Desktop applications may have other dependencies, and they need to run on different operating systems. This complexity may lead to errors.

One of the final steps after coding is testing (followed by detecting and correcting errors), consisting of two steps: syntax and semantic checking. *Syntax errors* are revealed during compilation: the error list usually appears in a separate panel, and in better environments the error is indicated in the source code as well.

Semantic testing has two important parts: preparing the test cases and the executing the tests. In worse scenarios test cases are announced only verbally, but in better cases they are written to the board or to the exercise book. Testing is initially made manually: the students enter the input data manually in the command line window and monitor the response of the program. The command line usually appears in a separate window independently of the developer environment. For longer inputs, tests are written to a file, which are redirected to the standard input of the running program. Creating these test files can be done in the development environment or in another program. Redirection is often not possible in IDEs, so testing requires the opening of a command line window, which adds complexity to this step.

Development in a traditional environment is often supplemented with online judging systems (e.g. [4, 5]), which verify the code objectively. These systems usually contain only the task descriptions and the batched, automated verification services, the development is still performed in separate IDEs.

Analysing this process, several problems can be identified in the relation of programming environments and introductory courses, beginner students:

- IDEs are too general: they are general-purpose development environments, and are not intended to support specific, methodologically-based problem solving processes, which would be better for beginner students. They are only focus on the implementation part of this process.
- IDEs can not guide the student, it is left for the teacher or the students themselves.
- Other knowledges are also needed, e.g. using the command line, redirecting, uploading and downloading files.
- Considering the available number of lessons per week, teaching the different tools and the whole toolchain proportionally takes much more time than teaching the essentials of problem solving, compared to their importance.
- If students come from a gamified environment, using professional IDEs is big gap to leap through.

Looking at the number of supporting programs, the whole workflow is too complex: in order to achieve their goals, students need to focus on eight different sources of information on seven different platforms:

1. Task description (separate window)

2. Design (board or exercise book)
3. Coding (IDE)
4. Compilation (IDE, separate panel)
5. Running (separate console)
6. Writing test files (separate window)
7. Testing with files (separate console)
8. Automated testing (separate web application)

Considering the heterogeneity of these introductory courses, mainly coming from the previous knowledges of students, and the complex workflow that traditional programming environments provide, some demands can be formulated against a programming environment:

- Support beginners: complex processes should be made easier or left out.
- Be specific for introductory courses: it is not needed to be prepared for solving complex tasks. Introductory courses has simple tasks. Support these and support them well.
- Support simple programs: programs consist of one file, they need to read from standard input and write to standard output.
- Support the steps of task solving process: programming environment should lead the students' hand during the process, and should support all of the important steps of task solving process.
- Support methodology: what is important methodologically, it should be supported by the environment, if it is possible.
- Support curriculum: be flexible enough to support different introductory curriculum.
- User-friendly interface: ignore every distracting element from the user interface.
- Monitor students' performance: support monitoring of the progression, and make room for further personalization (e.g. giving different tasks for different students based on their results).
- Stand-alone usage, practice mode: the programming environment should be used with or without the teacher's direction.
- No installation: be platform-independent avoiding errors during installation.

3. Existing alternatives as solutions

Beside the traditional development environments there are other programming platforms that can provide an alternative solution for the problematic aspects of introductory programming courses introduced above. Every alternative environment tries to operate with a simplified user interface, where all the necessary information is available in the same program. It is common in every environment that they are web-based which provides all the features that the web platform can bring: ubiquity, no-installation set-up for the user, easier maintenance. Of course, these environments are different according to their specific needs.

The first group of these programming environments consists of the *online learning platforms*, like CodeAcademy [6] or Khan Academy [7]. They are designed to be used alone without any external guidance; they proceed forward in small steps, introducing small amount of new knowledge at one time. They are using textual descriptions or video tutorials to introduce a topic, and an online editor with automatic tests for practising. It would be hard to use them in a certain curriculum or in a lesson, and they do not support some parts of the task solving procedure, like planning, manually testing and debugging.

The members of the second group are the *online code editor environments*, such as CodingGround [8], Rextester [9], jDoodle [10], or Ideone [11]. They are web-based IDEs, with single or multiple file support, and sometimes with a built-in console (CodingGround). Usually standard input can be specified, and they give back the result of the compilation and the text of the standard output. They try to be general, so they can not support many of the demands that were formulated against an introductory programming environment: there is no place to give or collect task descriptions (or at least in comments), they do not support the methodologically formed steps, they do not support testing. However, these environments are great examples, that the development of command line applications can be achieved in browsers with the help of the web-platform.

A variation of the latter group is the online programming contest platforms or *code training platforms* such as CodeChef [12] or Codewars [13]. They are more than the previous group in a sense that they provide task description, automated tests for checking the solution, and sometimes manual tests can be given also. But they are lack of flexibility, activity monitoring can not be fulfilled, and teachers can not give their own tasks to the system. From this point of view they operate as a combination of an online IDE and judge system.

The last group of alternative environments is consist of those *online editors* which try to focus *on educational problems*. One of those web-based platform is repl.it [14]. It started as an online IDE with a built-in console, but later it was extended with some very useful educational tools, like classroom management, creating assignments, automated tests, monitoring student activities, giving task descriptions. These features are great and comes handy in certain situations, but manual testing is missing among those features, and thus some methodology-based requirements are not fulfilled.

4. The proposed programming environment

Our proposed programming environment tries to solve those issues which come from the scattered nature of the traditional programming environments along with some methodological considerations to help beginner students in learning programming. The first version of our in-browser programming environment [15] eliminated the distracting elements, pulled together the different type of tasks, except for the planning phase, into one user interface, where the task description, the coding area, the input and outputs of manual tests, and automatic tests took place. With these

it kept the attention in one place, it supported the steps of the problem solving process, and code editing had all the features a beginner needed in a comfortable, user-friendly environment. One of the main features of the first version was that it worked in an isolated environment without internet connection (offline). But this latter feature was this environment main drawback: it narrowed the potential programming languages into JavaScript and TypeScript (or, strictly speaking, any language that can be compiled in the browser), it did not give any chance to monitor students activities, only one user test could be given.

Learning from the drawbacks of the first version, the new version of the proposed programming environment was rewritten from ground up around similar user interface design principles, but with very different operations in the background. The user interface of the new environment can be seen in Figure 1. It is divided into two parts. Task description, manual and automatic tests are available in a tabbed panel on the left, while an easy-to-use and feature-rich code editor fills the right side of the browser window with the necessary buttons and informations. The workflow is the following: students can choose among the available tasks in the drop-down menu beside the logo; can read the task description; can make the planning outside of the environment; can implement the solution in the code editor; can make multiple manual tests to determine the correctness of the solution by themselves; can verify the solution with the help of automatic test prepared by the teacher.

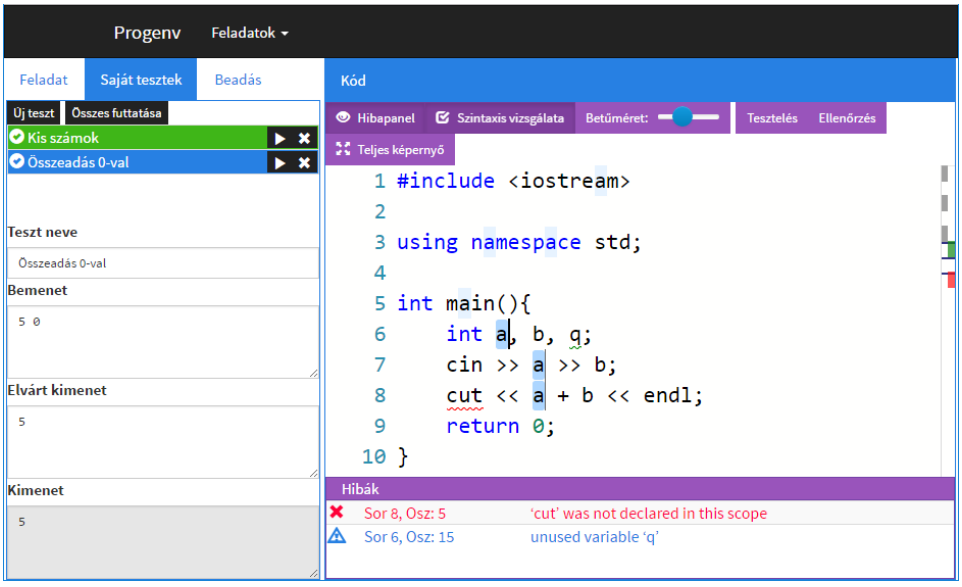

Figure 1: The user interface of the proposed programming environment

The heart of the application, the background mechanism was moved to server

side, where far more opportunities are available. The new architecture can be seen in Figure 2. Every operation that needs some language-specific feature (syntax checking, compilation, testing) sends a request to a REST API (Representational State Transfer Application Programming Interface) on the server side. Due to security reasons compilation and execution needs to be run in a sandboxed environment. The HTTP response contains the results of the requested operation, and this information is displayed on the interface.

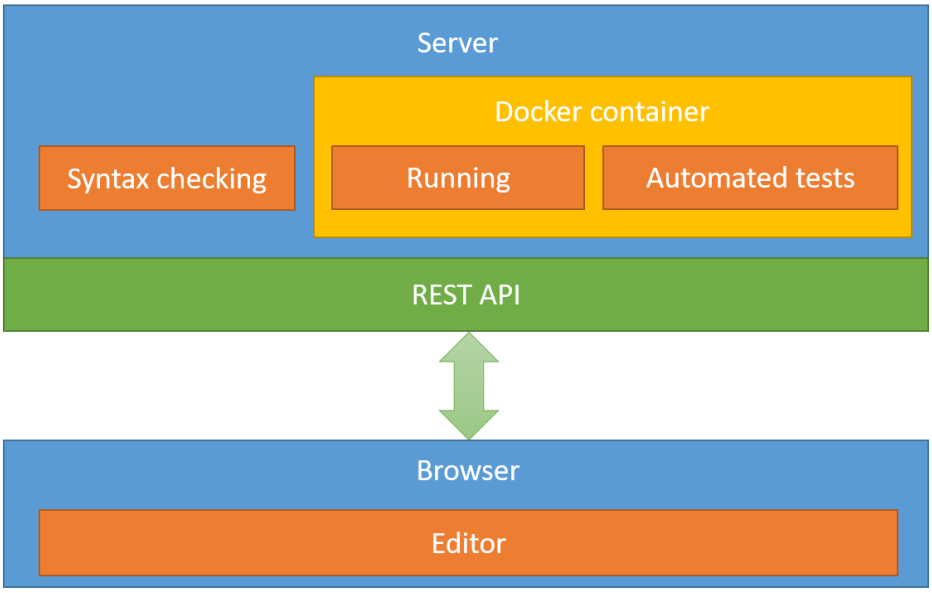

Figure 2: The schematic architecture of the proposed programming environment

This version of the environment follows the concept of single-page applications. The technologies that were used involves React, React-router, MobX, Monaco editor, Flexbox, Markdown on the client side, and node.js, express.js, Docker on the server.

5. Discussion and summary

The main advantage of the proposed programming environment that it keeps the problem solving process in focus (instead of the environment). It gains time both on the students' and the teacher's side. With different tasks and monitoring it opens up possibilities towards personalization and differentiated works. The web platform makes it possible to get rid of the installation process, to learn independently of time and place, and with sufficient automations it can offer off-class learning

opportunities [16, 17].

The following type of tasks can be used in this environment:

- implementing a solution from scratch (or with minimal initial code);
- implementing one or more specific functions, the other part of the code is written already;
- correcting errors in a pre-made, but wrong program; it develops code reading.

Summarizing, the proposed web-based programming environment can help the learning processes of beginner students in an introductory programming course. The environment is comfortable, has a non-distracting user interface, supports methodology, workflow and curriculum, and its flexible, language-agnostic architecture opens up new way towards personalisation and user monitoring. In this form it could support in-class and stand-alone usage as well.

There are many ideas for further development. User management is still missing, there should be pages where new tasks can be prepared, where task and user assignment could be achieved. User activity monitoring and personalisation would be great, and it could serve as a potential assignment platform as well. Debugging is still an issue.

References

[1] Code Studio, `https://studio.code.org/` [cited 2017 May 22]

[2] Scratch, `https://scratch.mit.edu/` [cited 2017 May 22]

[3] CodeCombat, `https://codecombat.com/` [cited 2017 May 22]

[4] "Bíró" judging system on the Faculty of Informatics, Eötvös University, `http://biro.inf.elte.hu/` [cited 2017 May 22]

[5] "Mester" judging system in Hungary, `http://mester.inf.elte.hu/` [cited 2017 May 22]

[6] Codecademy, `https://www.codecademy.com/` [cited 2017 May 22]

[7] Khan Academy, `https://www.khanacademy.org/` [cited 2017 May 22]

[8] CodingGround, `https://www.tutorialspoint.com/codingground.htm` [cited 2017 May 22]

[9] Rextester, `http://rextester.com/` [cited 2017 May 22]

[10] jDoodle, `https://www.jdoodle.com/` [cited 2017 May 22]

[11] Ideone, `https://ideone.com/` [cited 2017 May 22]

[12] CodeChef, `https://www.codechef.com/ide` [cited 2017 May 22]

[13] Codewars, `http://codewars.com/` [cited 2017 May 22]

[14] Repl.it, `https://repl.it/` [cited 2017 May 22]

[15] Horváth, Gy., Menyhárt, L. Webböngészőben futó programozási környezet megvalósíthatósági vizsgálata, *INFODIDACT 2016* Paper 3. (2016)

[16] Horváth, Gy., Menyhárt L. Oktatási környezetek vizsgálata a programozás tanításához, *INFODIDACT 2014* Paper 7. (2014)

[17] Horváth, Gy., Menyhárt, L., Zsakó, L. Egy webes játék készítésének programozás-didaktikai szempontjai, *INFODIDACT 2015* Paper 2. (2015)