Café: Automatic Correction and Feedback of Programming Challenges for a CS1 Course

Simon Liénardy
Université de Liège, Montefiore Institute – Belgium
simon.lienardy@uliege.be

Laurent Leduc
Université de Liège, IFRES – Belgium
laurent.leduc@uliege.be

Dominique Verpoorten
Université de Liège, IFRES – Belgium
dverpoorten@uliege.be

Benoit Donnet
Université de Liège, Montefiore Institute – Belgium
benoit.donnet@uliege.be

ABSTRACT

This paper introduces Café ("Correction Automatique et Feedback des Étudiants"), an on-line platform designed to assess and deliver automatic feedback and feedforward information to CS1 students, both on process and products of series of programming exercises, targeting especially an informal Loop Invariant for building the code. The paper reports on the first trials of Café with a group of 80 students. Results show that Café is used, usable, and appreciated by students.

CCS CONCEPTS

• **Theory of computation → Algorithm design techniques**; • **Social and professional topic → CS1**.

KEYWORDS

Café, Graphical Loop Invariant, feedback, CS1, Assessment for Learning

1 INTRODUCTION

This paper introduces and discusses Café ("Correction Automatique et Feedback des Étudiants"[1]), an on-line platform for automatically assessing students' programming exercises. The key point of Café is that it not only focuses on the program output but also on the cognitive process inherent to the program construction. Café provides students with feedback and feedforward information (i.e., what should they do to improve their solution). Café is applied in the context of a CS1 course in which students are exposed to C programming language concepts and basic algorithmic aspects.

In Belgium, the access to the Higher Education curriculum in Computer Science is non-selective. Hence, we cannot make any kind of assumptions about first year students' background. Many of them enter the program with excitement (unfortunately due to a biased vision of the Computer Science field), but without being clearly aware of actual requirements. This contributes to a high failure rate, as well as a high withdrawal ratio throughout the year [5, 6, 39].

Typically, an algorithm requires to write a sequence of instructions that must be repeated a certain number of times. This is usually known as a *program loop*. The methodology we proposed in the CS1 course is based on an informal version of the Loop Invariant (a property of a program loop that is verified at each iteration – i.e., at each evaluation of the Loop Condition) introduced by Hoare [18, 19]. Our methodology consists in determining a strategy (i.e., the Loop Invariant) to solve a problem prior to any code writing and, next, rely on that strategy to build the code, as initially proposed by Dijkstra [15] and extended later by Back [2]. The methodology also implies to verify the loop ends. This is achieved, in our methodology, by measuring the progress made by each loop iteration through the definition of a Loop Variant (i.e., a function that measures the progress made by the loop towards the termination). As such, the Loop Invariant and the Loop Variant can be seen as the corner stones of code writing. The problem with such a methodology is that it is quite an abstract reflection phase that might confuse students who may not have the desired abstract background, specially if the Loop Invariant is expressed as a logical assertion. However, this methodology requires to be practiced on a regular basis [20, 30] with carefully scaffolded problems [34], so that students can, little by little, master it.

In the context of our CS1 course, a reflection was carried out on a teaching activity that could fulfill this need of regular exercises and that could come in addition to classic exercises sessions (12 sessions of exercises and 5 labs in front of computers in our CS1 course). Limited by multiple constraints (a single Teaching Assistant, time, room availability, etc.), those exercises have to be done at home, with an automatic correction and feedback provided to students.

The main issue of such an automatic system lies in the fact that it is very difficult to, simultaneously, assess the program output and the cognitive process inherent to the program construction. In particular, as the course puts the emphasis on the Loop Invariant and the Loop Variant, it quickly appeared as desirable to include them in the automatic correction of the student's work. Furthermore, not doing so would have resulted in a dissonance between theoretical lessons and practical sessions.

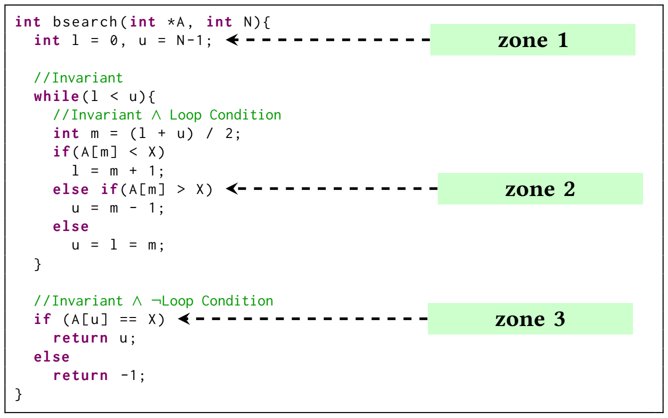
This is exactly what we propose in this paper: an on-line platform, called Café[2] ("Correction Automatique et Feedback des Étudiants") for automatically assessing students' programming exercises. In addition, this platform is inspired by *assessment for learning*[3] (AfL) [34, 40].

In a nutshell, Café proposes to students several programming *Challenges* (roughly one Challenge every two weeks) spread over the CS1 semester. It encourages students to work on a regular basis on increasing difficulties problems. In addition, those Challenges

[1] "Automatic Marking and Feedback for Students".

[2] Café source code, with examples of programming Challenges, is provided here: https://github.com/slienardy/CAFE

[3] Assessment for Learning is defined as informing learners of their progress to empower them to take the necessary action to improve their performance [20].



```
int bsearch(int *A, int N){
  int l = 0, u = N-1;

  //Invariant
  while(l < u){
    //Invariant ∧ Loop Condition
    int m = (l + u) / 2;
    if(A[m] < X)
      l = m + 1;
    else if(A[m] > X)
      u = m - 1;
    else
      u = l = m;
  }

  //Invariant ∧ ¬Loop Condition
  if (A[u] == X)
    return u;
  else
    return -1;
}
```

Excerpt 1: Binary search code with zones delimited by the Loop Invariant.

could help students to better understand the course learning outcomes [37] as well as to increase their self-efficacy [3]. Further, for each Challenge, Café allows students to submit up to three times [21] their solution, thus closing the feedback loop [8]. For each submission, Café automatically provides a feedback and feedforward information based on the literature [22], promoting self-regulated learning. Among established quality criteria for feedback, Café primarily instantiates the followings: (*i*) individualized feedback [11], (*ii*) feedback focused on the task, not on the learner [29], and, (*iii*) feedback directly made available to the student preventing her from being bogged down or frustrated [23]. Finally, Café allows the pedagogical team to collect valuable temporal data, leading to a refinement in each student profile and to assess the overall CS1 course understanding.

Café was used during Academic Year 2018–2019, with a population of 80 students. This paper discusses data we have collected and lessons learned from Café usage.

The remainder of this paper is organized as follows: Sec. 2 discusses and illustrates the programming approach we propose in our CS1 course; Sec. 3 is the heart of the paper by discussing every Café components; Sec. 4 presents a preliminary evaluation of Café impact and discusses the lessons learned from its usage; Sec. 5 positions this work with respect to the state of the art; finally, Sec. 6 concludes this paper by summarizing its main achievements.

2 PROGRAMMING METHODOLOGY

A *Loop Invariant* [18, 19] is a property of a program loop that is verified (i.e., true) at each iteration (i.e., at each evaluation of the Loop Condition). The Loop Invariant purpose is to express, in a generic and formal way through a logical assertion, what has been calculated up to now by the loop. Historically, Loop Invariant has been used for proving code correctness (see, e.g., Cormen et al. [12] and Bradley et al. [9] for automatic code verification). As such, the Loop Invariant is used “a posteriori” (i.e., after code writing).

In our CS1 course, we envision a different perspective in which the code is built upon the Loop Invariant that must be thus expressed before coding (“a priori” usage), as suggested by Dijkstra [15] and pushed further by Back [2]. This allows us to divide the code in three main zones, as illustrated in Listing 1, each zone being constructed thanks to the Loop Invariant. *Zone 1* refers to the code segment prior to the loop, typically used for initializing variables. At the end of Zone 1, the Loop Condition is evaluated, meaning that the Loop Invariant must be verified. Based on this statement, the Loop Invariant can be used to determine the required variables as well as their initial values. *Zone 2* refers to the Loop Body itself. As the Loop Condition has been verified, before executing any instruction of the Loop Body, the Loop Invariant is true and the Loop Condition is true. At the end of the Loop Body, the Loop Condition is evaluated again, meaning the Loop Invariant must be restored. Based on those two situations, one can derive the Loop Body instructions. *Zone 3* refers to the piece of code after the loop, when the Loop Condition has been invalidated. This zone contains instructions that should allow the program to finally solve the initial problem. Given that the Loop Condition has been evaluated, the Loop Invariant is true but the Loop Condition is false. Based on this situation, it is possible to derive the final instructions.

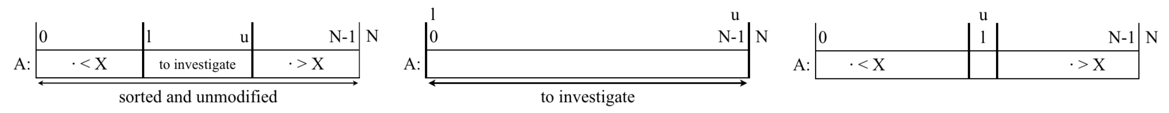
While Dijkstra expressed Loop Invariants as logical assertions, we believe this could be counter-productive in the context of a CS1 course, in which students may not have the required level of abstraction. Instead, we build our methodological approach on an informal version of Dijkstra’s process by proposing an informal *Graphical Loop Invariant*.

The Graphical Loop Invariant is supposed to contain key information that will eventually be used to actually write the code. As such, the Graphical Loop Invariant represents a strategy to solve the problem and is used to support thoughts on the code. Although being informal, this drawing must at least detail variables, constant(s), and data structures manipulated by the program; the constrains on them; the relationships they may share, and that are conserved all over the iterations. It should also express, in a general way, what has been already computed by the program after a certain number of loop iterations. With this method, we clearly shift the difficulty not anymore in writing the code itself but in the reflection phase that is prior to the code. This step requires thus training and experience. But, once mastered, it becomes possible to efficiently solve complex problem.

In the remainder of this section, we illustrate this process (from drawing the Graphical Loop Invariant to writing the code) with a well-known problem: the binary search in a sorted (in the increasing order) array `A` of length `N`, `A` being indexed from `0` to `N-1`. The function implementing the binary search will return the index of the researched value `X`, or a special value `-1` if `X` does not belong to `A`.

The basic idea of the binary search is to divide, at each step, the search zone based on the ordered property of `A` and the value of `X`. Generally speaking, one can thus divide `A` in three zones: (*i*) array elements $< X$, (*ii*) array elements $> X$, and (*iii*) array portion in which the search must be performed.

Those considerations are depicted in Fig. 1a where we delimit the $\cdot < X$ zone with a variable `l` (for **l**ower indices) and the $\cdot > X$ zone with variable `u` (for **u**pper indices). As can be seen in Fig. 1a,



(a) Graphical Loop Invariant for binary search.

(b) Initial state for binary search.

(c) Final state for binary search.

Figure 1: Graphical Loop Invariant and particular cases for binary search.

we carefully depict the array as a rectangle labeled with the array name and the indices written above it. Labeling a data structure with its name is, at first, useful for the understanding and becomes mandatory when multiple structures come into the game. It is also the first required step for making the Loop Invariant coherent with the code when it will be written. It should be noted that, in our Figure, the $\cdot < X$ zone is formally comprised between indices 0 and l − 1, which is represented by the letter l written at the right of the vertical bar determining this zone. We call such a vertical bar *dividing line* (it appears thicker in Fig. 1a). This accuracy in the drawing is of the highest importance to easily determine l initial value . The same remark applies for u. This Graphical Loop Invariant also makes easier the discovery of a Loop Variant (i.e., a function that measures the progress made by the loop towards the termination). It is, simply, the length of the "to investigate" zone, i.e., $u - l + 1$ in Fig. 1a.

As explained earlier in this section, the Graphical Loop Invariant can be used to divide the code in three zones and to write the code of those zones. In particular, Fig. 1b illustrates, graphically, the situation in the first zone. This is obtained by stretching the two dividing lines to their minimum (for l) and maximum (for u) values. At start, we cannot ensure that any values in A is less than X, neither we can ensure that any values in A is greater than X. Hence, the corresponding zones in the array A at the initialization are empty, as shown in Fig. 1b. From this drawing, we see that in this situation, l (respectively u) corresponds to index 0 (respectively $N - 1$) and we conclude that we must initialize l and u to these respective values, as done in Line 2 of Listing 1.

Similarly, we can modify Fig. 1a to depict the loop final situation (see Fig. 1c). The zone to be checked in A (i.e., the zone A[l...u]) has now the minimal size, leading to a case where l == u. In such a case, the loop must be stopped and one can derive the Loop Condition (i.e., l ≠ u). Pedagogically speaking, we can use a lighter condition, l < u that has the advantage to better represent the relationship between those indices.

Zone 2 (i.e., loop body) consists in making the loop progressing toward the final objective: finding X and returning its position in A. To do so, we must investigate the unknown zone (labeled "to investigate" in Fig. 1a) and make growing one of the two other zones, thus moving the associated dividing line. The binary search consists in looking the middle of that zone, at index $\mathtt{m} = \frac{\mathtt{l+u}}{2}$ (line 7 in Listing 1). If it appears that A[m] < X, one concludes that X does not belong in the left part of A and that the left dividing line must be translated to the right (line 9 of Listing 1) by taking the value m + 1 (+1, since we know that A[m] < X). A similar reasoning can be made for the A[m] > X case. Finally, otherwise, the loop must end (i.e., A[m] == X). This is achieved by forcing the final situation, as illustrated in Fig. 1c (see line 13 in Listing 1).

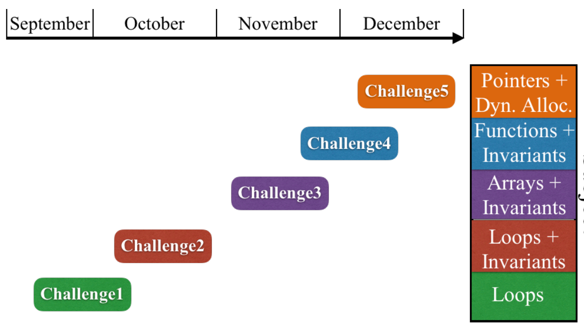


Figure 2: Challenges timeline over the semester.

Finally, zone 3 is reached once one leaves the loop, i.e., when l == u, as explained by Fig. 1c. In such a case, the zone "to investigate" contains a single element. We thus only have to check whether this element is X or not (lines 17 ⟶ 20 in Listing 1).

3 CAFÉ

This section introduces Café ("Correction Automatique et Feedback des Étudiants"), our on-line platform for automatically assessing students' programming exercises. We use Café through six assignments called *Challenges* distributed throughout the first semester[4], approximately every two weeks. Challenges account for 10% of the final grade, each Challenge having the same weight. Each Challenge consists in a programming/algorithmic task. The first Challenge (called "Challenge 0") helps students in grasping how Café works and, consequently, does not account in the final mark. Each of the five subsequent Challenges focuses on a particular subject and accounts in the final mark. It is worth to notice that we allow students to not submit one of the five Challenges (concept of *Joker*). In that case, the Challenge does not account in the final mark. This was thought to increase the students' perception of controllability that leads to higher motivation according to Viau [38] but also to get rid of students' excuses when not submitting a Challenge [10].

The Challenges are of increasing complexity (from a simple loop to write – Challenge 1 – to a modular program solving a reasonably complex problem – Challenge 4), referring thus to assessment for learning (AfL) [30, 34, 40]. The last Challenge is dedicated to pointers and dynamic allocation, as we noticed those particular

[4]In Belgium, the Academic year is divided in two semesters. The first one ranges from mid-September to Christmas, with the Final Exam taking place in January.

```c
#include <stdio.h>

int main(){
    const unsigned int N = ..., M = ..., L = ...;// large enough
    int A[N], B[M], C[L];
    // Arrays A and B are filled with values (code not provided)

    // Your code will be inserted here.
}//End of the program
```

Excerpt 2: Code template provided to the student.

topics appear to be difficult for students. The Challenges timeline is illustrated in Fig. 2 (Challenge 0 not shown).

Sec. 3.1 discusses how students interact with Café, in particular focusing on the Challenge template they have to fill in, while Sec. 3.2 explains how we implemented Café.[2]

3.1 Students Interactions with Café

A Challenge lasts three days. The first day, the subject is made available for download on the course blackboard. In addition to the subject, students must download a template to fill in with their answers. The correct way to format the answer in the template is provided in the Challenge subject as well as in the template itself.

Once ready, the student answer can be uploaded to Café via a web platform. Café immediately corrects it and produces a feedback and a feedforward that are directly made available to the student. She can then consider these feedback and feedforward to improve her answer and submit it again [17]. Students benefit up to two retries (for a total of three submissions [21] – this way, we avoid the traditional trial and error approach) over the Challenge duration. This process enables the students to learn from their errors by actually taking into account the feedback and feedforward and by submitting an improved solution, closing so the feedback loop [8]. Doing so prevents also the student from being bogged down. At the end, only the last submission accounts for the mark.

3.1.1 Challenge Instructions. The instructions describe the tasks students have to achieve and how the template must be filled to provide a valid submission (i.e., to be properly understood by Café). The Excerpt 2 presents an example of code template provided in a Challenge instructions (this is called "Solution Template" by Keuning et al. [22]). The goal of this Challenge is to compute the intersection of two sorted arrays, A and B, and to place the result in a third one, C. As it can be seen in the code Excerpt, the arrays are already declared and filled with values. The students only have to complete the code that actually computes the intersection. Note that imposing the name of the arrays eases the Challenge correction (for more details see Sec. 3.2).

Furthermore, as most of the Challenges consist in writing loops and as the course requires to write loops based on Graphical Loop Invariant (see Sec. 2), the Challenges must embed Loop Invariants so that students can train themselves. At first, it may appear difficult to combine automatic correction and graphical representation. We solve this by asking students to fill in a blank Graphical Loop Invariant. Such a blank drawing depicts only the general shape that should follow a correct and rigorous Loop Invariant. Students must then annotate properly the figure so that the drawing becomes their Loop Invariant for their solution to the particular problem to be solved. An example of blank Graphical Loop Invariant is provided in Fig. 3. The instructions state that the green boxes 1. to 15. should be replaced by variables, constants names, or left blank. The box 16. has to be replaced by a number corresponding to the multiple choice: 1: different from; 2: common to; etc. (some inconsistent possible answers are added). Finally, the boxes 17. to 19. must be replaced by a number corresponding to one of the part written in red in Fig. 3, i.e.:

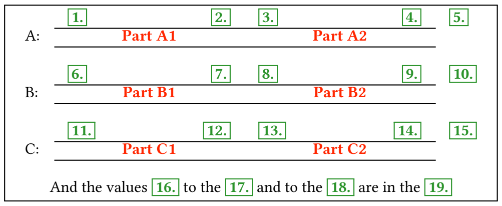


Figure 3: Example of a blank Loop Invariant given to the students in the Challenge instructions described in Sec. 3.1.1.

(1) Part A1;
(2) Part A2;
(3) Part B1;
(4) Part B2;
(5) Part C1;
(6) Part C2;

A mock example of such a replacement is always added in the instructions. The way to encode the Graphical Loop Invariant in the Challenge template is also clarified in the instructions, with an example, to be as clear as possible. The Excerpt 3 presents a template and how it could have been filled in by a student.

3.1.2 Template. The format of the template file follows very basic rules: the answers are delimited by the special symbol "#". C-style comments are allowed, everything else is considered as an answer. With such rules, the template is easily parsed. An example of template, already filled with a student's answer is presented in the Excerpt 3. The C syntax highlighting enables to distinguish the reminders of instructions (i.e., the comments in green) and the student's answers.

In particular, the lines 9 to 27 show how the Loop Invariant is encoded. As the Loop Variant is concerned, it is expressed, in the template, as a simple C expression.

3.1.3 Feedback. After each submission, a feedback is quickly provided to the student. The feedback contains the student's mark, as well as information about how Café understood her submission, feedback on her performance, and feedforward advices (i.e., what should she do to improve her mark). Each piece of feedforward advice is either informational (e.g., the instructions were not properly followed and should be re-read carefully), either theoretical (e.g., some theoretical concepts seem to not be properly understood and a reference to the course material is provided), either regulational (i.e., recommendations on actions that should be taken for improving the answer). Fig. 4 provides an example of a part of the feedback that could have been provided after the submission of the Challenge corresponding to the instructions presented in Sec. 3.1.1. As can be seen in Fig. 4, the structure of the feedback

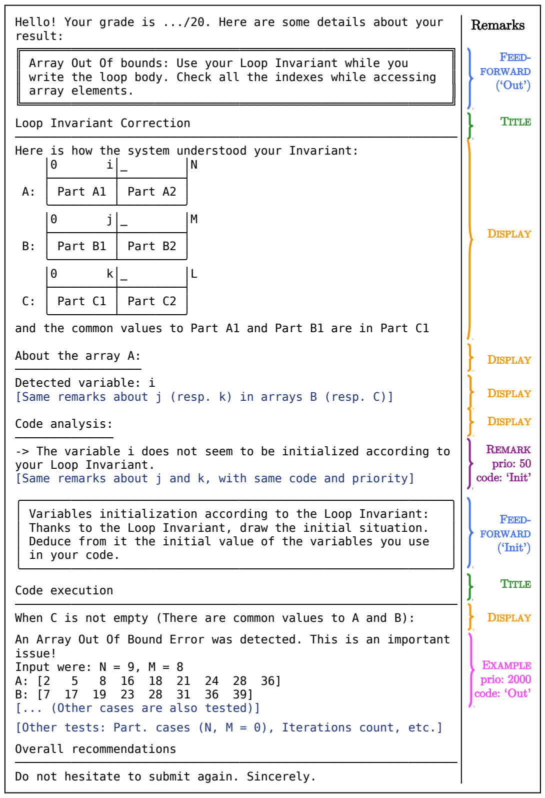
```
/* Challenge 3: Arrays and Loop Invariant
Some reminders about the submission process and the statement

Invariant
_________________________________________________________________

Encode your Loop Invariant below. For your convenience, Box
numbers are already written
*/
1. 0
2. i
3. _
4. _
5. N
6. 0
7. j
8. _
9. _
10. M
11. 0
12. k
13. _
14. _
15. L
16. 2
17. 1
18. 3
19. 5
/* Encode your Loop Invariant above */
#
/*
Variant
_________________________________________________________________

Encode your Loop Variant below, as a valid C expression */
M + N - i - j
/* Encode your Loop Variant above */
#
/*
Code
_________________________________________________________________

Type your code below, i.e. what should replace the line "Your code
will be inserted here" in the template. */
int i = 0, j = 0, k = 0;
while(i < N && j < M){
   if(A[i] < B[j]) ++i;
   if(A[i] > B[j]) ++j;
   if(A[i] == B[j]){ C[k++] = A[i]; ++i; ++j;}
}
/* Type your code above */
```

Excerpt 3: Challenge template to fill and submit to complete the Challenge.

follows submission template (See Excerpt 3). The Loop Invariant formed by the combination of the blank Loop Invariant from the instructions (See Fig. 3) and the content that should replace the 19 boxes specified by the student in her submission (See Excerpt 3) is clearly displayed. Moreover, the feedforward information is framed to draw student's attention.

3.2 Café Implementation

Café is written in Python 2.7[2] and easily extensible for new features. The Python script is run in a dedicated sandbox (for avoiding any security issue), on a submission platform, each time a student submits a Challenge. Currently, correcting and grading a Challenge is done in three main steps, as illustrated in Fig. 5: (*i*) the preprocessing (i.e., splitting the submitted Challenge into several answers – Sec. 3.2.1), (*ii*) the correction per se (i.e., each answer is corrected, graded, and commented – Sec. 3.2.2), and (*iii*) feedback generation (i.e., the various grades are combined and comments are concatenated to form the feedback to be provided to students – Sec. 3.2.3 to Sec. 3.2.4). It is worth noticing that the steps are independent from each other: one can easily modify the correction step as soon as it handles the answers from the preprocessing and generates data that can be transformed into feedback at the next step.



```
Hello! Your grade is .../20. Here are some details about your
result:

  Array Out Of bounds: Use your Loop Invariant while you
  write the loop body. Check all the indexes while accessing
  array elements.

Loop Invariant Correction
_____________________________________________________________

Here is how the system understood your Invariant:
      |0        i|_        |N
  A:  | Part A1  | Part A2 |
      |0        j|_        |M
  B:  | Part B1  | Part B2 |
      |0        k|_        |L
  C:  | Part C1  | Part C2 |
and the common values to Part A1 and Part B1 are in Part C1

About the array A:
__________________

Detected variable: i
[Same remarks about j (resp. k) in arrays B (resp. C)]

Code analysis:
______________

-> The variable i does not seem to be initialized according to
your Loop Invariant.
[Same remarks about j and k, with same code and priority]

  Variables initialization according to the Loop Invariant:
  Thanks to the Loop Invariant, draw the initial situation.
  Deduce from it the initial value of the variables you use
  in your code.

Code execution
_____________________________________________________________

When C is not empty (There are common values to A and B):

An Array Out Of Bound Error was detected. This is an important
issue!
Input were: N = 9, M = 8
A: [2   5   8  16  18  21  24  28  36]
B: [7  17  19  23  28  31  36  39]
[... (Other cases are also tested)]

[Other tests: Part. cases (N, M = 0), Iterations count, etc.]

Overall recommendations
_____________________________________________________________

Do not hesitate to submit again. Sincerely.
```

Figure 4: An example of Feedback. The right column indicates which type of remark was used to build each part of the feedback (See Sec. 3.2.3 and 3.2.4).

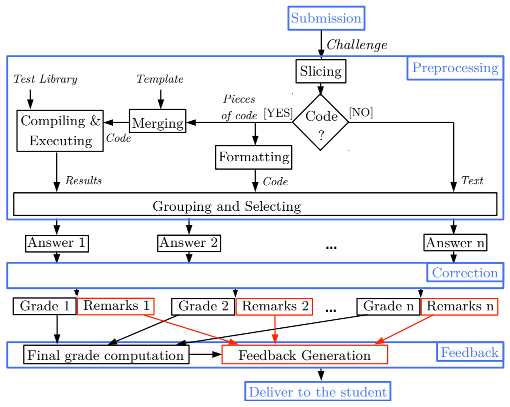


Figure 5: The three main steps of the correction: preprocessing, correction, and feedback generation.

3.2.1 *Preprocessing.*

A Challenge solution contains both pieces of code and text (e.g., the Loop Invariant that helped to write the code). The preprocessing step consists in analyzing those pieces of information and extracting the corresponding answers that will be corrected in the next step. This is illustrated in Fig. 5. One can see that the pieces of code are used in several ways. On one hand, they are merged and compiled with additional sources files and libraries into an executable to be run. The execution results are used by the correction step. On the other hand, they are gathered into a source file that is also used, by the correction step, for lexical and syntax analysis. Eventually, an actual answer regroups one or more pieces of information computed during this step. For example, the student's code and the Loop Invariant are both transmitted to a dedicated correction function.

3.2.2 Correction. As long as the code is concerned, the correction step is in charge of comparing the output of the preprocessing step with the expected result(s).

The correction step is able to check whether syntactic constraints are met by the code (e.g., using a `while` loop instead of a `for` one). In addition, by modifying student's code before the compilation, CAFÉ can also count the number of loop iterations to verify whether the code is compliant with complexity constraints and ensure that all the array accesses are within the array bounds. Regarding Challenge 5 (pointers and dynamic memory allocation), the student's code is linked to mock functions (i.e., fake `malloc`, `free`, etc.) that test whether they are called with the proper parameters and in the right order instead of actually manipulating the memory. Moreover, every call to those functions is logged and displayed in the feedback. The way errors (e.g., returning `NULL` for `malloc`) are handled by students is also assessed.

Regarding the Loop Invariant, the correction step verifies that what has been proposed as replacement of the boxes (see Sec. 3.1.1 and Fig. 3) is relevant. Most of the time, several answers are possible and are considered by CAFÉ.

Concerning the Loop Variant correction, the expression provided by the student is evaluated by giving particular values to each identifier of the expression. CAFÉ checks that the proper variable names (inferred from the Loop Invariant) appear in the expression and that the value of the expression decreases at each iteration.

Finally, there is always the risk a student will submit her Challenge with the Loop Invariant produced after the code (which clearly violates the methodology we propose). To limit this risk, CAFÉ checks if variables used in the Loop Invariant are consistent with the one in the code and if they are initialized accordingly. The matching between the Loop Condition and the Loop Invariant is checked by verifying that the Loop Condition makes used of the proper variables and leads to the correct number of iterations. The Loop Body is not checked against the Loop Invariant as it would be too time consuming to design a system that would cover all the code alternatives. If both the Loop Invariant seems correct and the code produces the expected results, we "a priori" believe the student has followed the methodology. Anyway, a student that would write the code first and later the Loop Invariant would, first, work twice and, second, just lie to herself.

During this correction step, each test, each check can give rise to a remark. Hence each correction function handling an answer will generated a list of remarks and a grade, as depicted in Fig. 5.

3.2.3 Feedback Generation. For each student's answer, the correction steps generates a mark and a list of remarks. To illustrate this, the right part of Fig. 4 shows the type of the remarks constituting the final feedback. They are four types of remark:

TITLE It is used to structure the feedback and to help the student to understand which part of her submission is commented (See Fig. 4, in green). A TITLE must always start each new question correction to ensure the feedback readability.

DISPLAY It is used to display unconditionally a message, for instance, an introductory text (See Fig. 4, in orange).

REMARK It is used to provide comments on a student's performance (See Fig. 4, in purple). In addition to a text that will be displayed, a code and a priority number must be specified for each Remark. The code is related to a feedforward message that is printed in the feedback according to rules depending on the priority number. These rules are detailed in Sec. 3.2.4.

EXAMPLE Like a Remark, it contains a message, a code, and a priority (See Fig. 4, in magenta). Unlike a Remark, the printing of their associated message is not mandatory. This feature can be helpful to shorten the feedback.

The feedback generation consists in merging and formatting all these remarks into a single text. The remarks order is preserved. Some feedforward messages can be inserted in the remarks (See Fig. 4, in blue), as detailed in the following. Finally, each type of remarks has its specific format style, e.g., the title are underlined and the feedforward messages are framed to increase the readability of the feedback, as illustrated in Fig. 4.

3.2.4 Adding feedforward. In order to provide feedforward to students, the REMARKS, and the EXAMPLES can be tagged with a code and a priority number (both depicted, in Fig. 4). Each code refers to a feedforward message and the priority number is used to decide whether this message will be eventually displayed. For each code present in the list of remarks produced by the correction step, the *total priority* (i.e., the sum of all the priority numbers with the same code) is computed. For instance, in Fig. 4, the REMARK with the code "Init" is issued three times (each time for a different variable), hence the total priority of this code will be 150. The feedforward message associated with the *n* codes having the highest total priority are actually inserted in the feedback to the student. Thus, the number of feedforward messages is limited to avoid overloading students with too long feedback. However, the priority mechanism helps to select the most relevant messages, ensuring the feedback personalization. The feedforward messages can be inserted at several places in the feedback. First, if the code with the highest priority is above a certain *warning threshold*, the feedforward message corresponding to this code will be placed at the very beginning of the feedback, as a warning, to draw the student's attention. For example, in Fig 4, this is the case of the feedforward labeled "Out", because of its highest priority. Each of the others feedforward messages is displayed as close as possible to the question it is related to, if this question is unique. If several questions are related to the same feedforward message, this one is placed at the end of the feedback, as an overall recommendation.

4 PRELIMINARY EVALUATION

Our CS1 course is organized during the first semester, starting mid-September until, roughly, mid-December. Around the end of October/beginning of November (six weeks after the beginning of

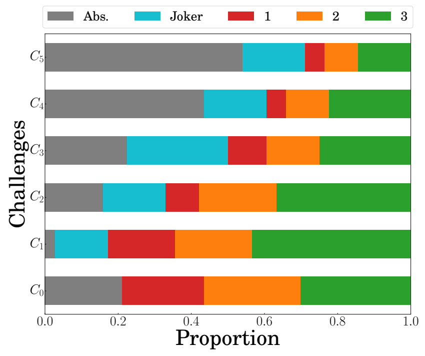


Figure 6: Distribution of students' involvement in Challenges over the semester. 1, 2 and 3 refer to the number of submissions, "Joker" to Students not submitting for the first time, and "Abs." to students not submitting at least for the second time.

the semester), courses are adjourned and Mid-Terms are organized for first year students in Computer Science. At the end of the semester, courses are again adjourned for two weeks during which students are supposed to study (this period is called *blocus*) and exams are organized during four weeks in January. Both evaluations are written tests (paper exercises, thus not in front of computers) based on a set of exercises to complete.

The platform on which we deploy Café allows us to collect various data on students involvement and performance in Challenges. In addition, we conducted a survey, during Academic Year 2018–2019, between February 2019 and March 2019, after the course and the Final Exam. The survey was anonymous, to let the students express freely their opinions. We received 22 answers over the 30 students who still continued their curriculum during the second semester. This section investigates the data collected in 2018–2019 and discusses lessons learned from Café usage.

One key point with Café is to ensure students involvement in the course and to ensure a minimum amount of regular practice over the course duration. Fig. 6 shows the distribution of students' involvement in Challenges over the semester. It is worth reminding that students can play a Joker during the semester (see Sec. 3). After this Joker has been played, any non-submission is accounted as an "Absence" and the student received a null mark for the Challenge. Finally, as Challenge 0 does not account in the final grade, not submitting it is not considered as a Joker. An Absence for Challenge 1 typically corresponds to students who lately join the cursus (registrations are open until end of October). We tag them as an "Absence" but it does not account in their final mark.

Students' involvement in Café is pretty good in the beginning of the Semester (above 60% until Challenge 2). A drop is observed starting Challenge 3 (that follows the Mid-Term Exam). It seems that the Mid-Term plays an important role regarding students' involvement. In 2018–2019, the participation rate decreases over the semester until reaching a low 28% of the students participating to Challenge 5. This is aligned with the attrition rate observed during the final Exam (61% of Participation).

The conducted survey shows that the majority of respondents (19/22, 86.4%) agrees or totally agrees that *Café and Challenges seemed to me a good motivating way to make me work regularly* (on a Likert scale [25]: Totally agree (10), Agree (9), Disagree (3), Absolutely Disagree (0)).

Café allows students to submit a solution to a given Challenge up to three times, with feedback and feedforward sent back to students after each submission. Fig. 6 also shows how students manage multiple submissions. Multiple submissions (i.e., 2 or 3) are common, suggesting so that feedback and feedforward provided by Café are useful for students for improving their solution. It is confirmed by the survey as it appears, as the Table2 shows, that, for 59.1% of the students, the feedback provided by 3 Challenges or more enabled them to better understand the course (31.8% of them if we reduce to 1 or 2 Challenges). Also, 45.5% of the students admit that the feedback helped them to realize they had a learning gap regarding the subject tackled by 3 Challenges or more (36.4% of them if we reduce to 1 or 2 Challenges). Finally, after receiving the feedback, 59.1% of the students admit they went back, for 3 Challenges or more, to the theoretical course to reread the corresponding theoretical notions (13.7% of them if we reduce to 1 or 2 Challenges).

The survey shows that, to the open question *Why did you play your Joker?*, students mainly answer they have reached an acceptable grade (and they do not want to decrease their grade by doing an additional Challenge) or because the Challenge appeared as too difficult (e.g., the Challenge 5 was qualified as such by half of the answers).

Letting the students use a Joker is a double-edged sword. On one hand, this was thought to make them responsible for their learning, to avoid excuses when not submitting, and to increase their perception of their controllability on the course. On the other hand, there is a risk that they discard an opportunity to improve their skills because a Challenge perceived difficulty is too high. According to the survey, some students (4/22) "regretted using their Joker" and recognized that "taking the Challenge would have helped them for the Final Exam" when they answer to the open question: *The Challenge 5 submission rate was low this year. However, it had been announced that one or more questions in the Final Exam would focus on the subject tackled by this Challenge. In your opinion, what is the cause of this?*. They thus recognize that they applied a poor strategy by not doing Challenge 5. The importance of doing a Challenge is confirmed by the Final Exam results, in particular for the questions related the topics tackled by Challenge 5 (pointers and dynamic memory allocation): every student that succeeded in the Challenge 5, succeeded in the Final Exam too but it worth noticing that all of them first improved their Challenge mark through multiple submissions. On the other hand, those who did not submit Challenge 5 mainly failed at those questions in the Final Exam.

We believe that the Joker system must be maintained. However, this advocates first to better present to the students the consequences their choices can lead to (i.e., applying a poor/short term strategy) and, second, to ensure to debunk any rumor on a Challenge difficulty that would prevent students to even try it.

Table 1 deepens Fig. 6. In particular, it shows how students improve their marks (and, consequently, their solution) through multiple submissions. The key point highlighted by Table 1 is that, when multiple submissions are used (2 or 3), students are mostly

Table 1: Potential mark improvement thanks to multiple submissions of challenges during Academic Year 2018–2019. Columns labeled p_x provide the proportion of students having submitted x times a Challenge. Columns labeled $\propto_{2,3}$ refer to a proportion in a Challenge submitted two or three times (i.e., a proportion in $p_{2,3}$). $E[X]$ is the average improvement or worsening, expressed as a percentage of the maximal mark (20). σ_X is the standard deviation (- means that computing standard deviation does not make sense). Stagnation means that no progress has been made in the mark even in the presence of multiple submissions.

		Number of Submissions										
		0	1	2 or 3								
					Stagnation	Improvement			Worsening			
		p_0	p_1	$p_{2,3}$	$\propto_{2,3}$	$\propto_{2,3}$	$E[X]$	σ_X	$\propto_{2,3}$	$E[X]$	σ_X
Challenge	0	0.2	0.22	0.58	0.11	0.84	45%	33%	0.05	-55%	35%
	1	0.18	0.18	0.64	0.32	0.67	79%	30%	0	0%	-
	2	0.33	0.09	0.58	0.16	0.77	42%	25%	0.07	-43%	21%
	3	0.5	0.11	0.39	0.21	0.73	47%	28%	0.03	-30%	-
	4	0.61	0.05	0.34	0	0.96	37%	24%	0.04	-15%	-
	5	0.72	0.05	0.23	0.29	0.65	47%	18%	0.06	-15%	-

Table 2: Questions and raw results of the survey. Each number corresponds to the number of respondents.

	For how many Challenge(s) ?					
	0	1	2	3	4	5
The feedback allowed me to better understand the course	2	2	5	6	5	2
The feedback made me aware of learning gaps	4	3	5	4	0	6
After I received feedback, I went back to the theoretical course	6	0	3	7	3	3

able to improve their marks, sometimes by 80% on average (Challenge 1). In some uncommon cases, multiple submissions lead to a worst solution. A single submission might be explained by the fact that it leads directly to a correct solution, with the maximum mark or very close to the maximum. The survey we conducted indicates that students, in that case, will choose to not resubmit, as the efforts required to gain a few points are not worth.

Additionally, the survey also suggests that students may be happy with their score (even if it is not really the maximum) and may fear to do worst by taking a Challenge. Consequently, those students strategically choose to not submit this Challenge. Six (over 22) of them (27.3%) mentioned this reason as an an answer to the open question *Why did you use your Joker?*. This kind of strategy might also arise between two submissions of a same Challenge.

From the surveys, some students mentioned, in open comments, that the Challenges were too easy compared to the Mid-Term and the Final Exam, highlighting so a constructive alignment issue [7]. This perceived difference is mainly due to the fact that the overall Loop Invariant structure is provided with the Challenge but not for the formal evaluations. However, a large part of students (77.3% in 2018–2019) agreed that "the Challenges enabled them to understand the Loop Invariant determination" (on a Likert scale: Totally agree 4/22, Agree 13/22, Disagree 5/22, Totally Disagree 0/22) and few of them (2/22) acknowledged, in open comments, that the concept of Loop Invariant was understood "thanks to the given blank Loop Invariant". This tends to confirm the bootstrap effect of the blank Loop Invariant and suggests to focus the classroom sessions on graphical methods to find and master the Loop Invariant since it is not fully tackled by the Challenges

5 RELATED WORK

While there is an abundant literature on Loop Invariants for code correctness and on automatic generation of Loop Invariants (e.g., [12]), their usage for building the code has attracted little attention from the research community. Tam [36] proposed incomplete and informal Loop Invariants written in natural language. Astrachan [1] is probably the closest to our approach as he proposed Graphical Loop Invariant. Finally, Back [2] proposed nested diagrams (a kind of state charts) representing, at the same time, the Loop Invariant and the code. None of these approaches come with an automatic system that is able to assess the code construction and the Loop Invariant.

More generally speaking, Graphical Loop Invariant based programming falls within the scope of *metacognition* [28], as it provides a problem-solving strategy and self-reflection on where one is in the problem-solving process. As such, Graphical Loop Invariant based programming can be related to three problem-solving stages introduced by Loksa et al. [27], i.e., *search for solutions*, *evaluate a potential solution*, and *implement a solution*. Also, writing a Graphical Loop Invariant prior to coding should help students in understanding the problem to be solved [13]. This actual impact of the Graphical Loop Invariant on problem understanding will be studied in future works.

Many automated system for providing feedback to programming exercices were already proposed (e.g., [4, 14, 16, 24, 26, 31, 32]). Most of them apply test-based feedback, i.e., student's code is corrected through unit testing (except UNLOCK [4] that tackles the problem solving skills in general, not just coding skills). WebCAT [16] even makes students write their own tests too. Kumar's Problets [24] enables step by step code execution as part of feedback. More advanced automatic feedback has been proposed by Singh et al. [35] by providing, to students, a numerical value (the number of required changes) and the suggestion(s) on how to correct the mistake(s).

With respect to metacognition, Café is an automated assessment tool increasing metacognitive awareness [33], as it relies on Graphical Loop Invariant for building the code to solve programming Challenges. However, future work should reveal to what extend Café really helps in improving students' performance.

Following Keuning et al. classification [22], Café feedback falls within

- the *knowledge of mistakes*. Café performs unit testing ("test failure"), compile students' code and, in case of compilation errors, warns the students ("compilation errors") and, finally, checks the number of iterations, as well as the proper use of memory allocation ("performance issues").
- the *knowledge about how to proceed*. Through feedforward, Café provides references to the theoretical course or hints about actions to be taken to improve the solution solution, as well as hints about improvement to the submitted Challenge.
- the *knowledge about metacognition*. Café checks that the student's code matches with Graphical Loop Invariant (allegedly) used to derive it.

6 CONCLUSION

This paper introduced Café, an online system for automatically correcting and providing feedback on programming Challenges for a CS1 course. Café is also built around the programming methodology discussed during theoretical lessons, i.e., build the code upon the Graphical Loop Invariant, ensuring so a reflection phase prior to any code writing.

Café has been used, in parallel to theoretical lectures and classic practical sessions (in front of either a computer, either a sheet of paper). All these activities are complementary. Café enables to make the students work and improve their solution according to feedback and feedforward information, three times within three days and at least five times during the semester. This would be unfeasible without an automatic assessment. In addition, the classroom sessions can be dedicated to customizing each student's learning, in particular their capability to solve a problem from scratch (including the Graphical Loop Invariant). As such, Café is not yet self-sufficient.

The preliminary evaluation provided in this paper does not allow to conclude on Café's effect on students' performance. However, a longer use of Café will enable to collect information about how it impacts students' understanding of the course and the programming methodology and how it improves their programming capabilities.

REFERENCES

[1] O. Astrachan. 1991. Pictures as Invariants. In *Proc. Technical Symposium on Computer Science Education (SGICSE)*.

[2] R-J. Back, J. Eriksson, and L. Mannila. 2007. Teaching the Construction of Correct Programs using Invariant Based Programming. In *Proc. 3rd South-East European Workshop on Formal Methods*.

[3] A. Bandura. 1993. Perceived self-efficacy in cognitive development and functioning. *Educational psychologist* 28, 2 (1993), 117–148.

[4] T. Beaubouef, R. Lucas, and J. Howatt. 2001. The UNLOCK System: Enhancing Problem Solving Skills in CS-1 Students. *ACM SIGCSE Bulletin* 33, 2 (June 2001), 43–46.

[5] T. Beaubouef and J. Mason. 2005. Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *ACM SIGCSE Bulletin* 37, 2 (June 2005), 103–106.

[6] J. Bennedse and M. E. Caspersen. 2007. Failure Rates in Introductory Programming. *ACM SIGCSE Bulletin* 39, 2 (June 2007), 32–36.

[7] J. Biggs and C. Tang. 2011. *Teaching for Quality Learning at University* (4th ed.). Open University Press.

[8] D. Boud. 2000. Sustainable Assessment: Rethinking Assessment for the Learning Society. *Studies in Continuing Education* 22, 2 (August 2000), 151–167.

[9] A. R. Bradley and Z. Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer.

[10] M. Brauer. 2011. *Enseigner à l'Université: Conseils Pratiques, Astuces, Méthodes Pédagogiques.* Armand Colin.

[11] S. M. Brookhart. 2008. *How to Give Effective Feedback to Your Students.* Association for Supervision & Curriculum Development (ASCD).

[12] T. H Cormen, C. E Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms.* MIT press.

[13] M. Craig, A. Petersen, and J. Campbell. 2019. Answering the Correct Question. In *Proc. ACM Conference on Global Computer Education (CompEd)*.

[14] G. Derval, A. Gego, P. Reinbold, B. Frantzen, and P. Van Roy. 2015. Automatic Grading of Programming Exercises in a MOOC Using the INGInious Platform. In *Proc. European MIIC Stakeholder Summit (EMOOC)*.

[15] E. W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall, Inc.

[16] S. H. Edwards and M. A. Perez-Quinones. 2008. Web-CAT: Automatically Grading Programming Assignments. In *Proc. Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.

[17] N. Falkner, R. Vivian, D. Piper, and K. Falkner. 2014. Increasing the Effectivement of Automated Assessment by Increasing Marking Granularity and Feedback Units. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*.

[18] R. W. Floyd. 1967. Assigning Meanings to Programs. In *Proc. Symposium on Applied Mathematics*.

[19] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

[20] C.A. Jones. 2005. *Assessment for learning.* Learning and Skills Development Agency.

[21] V. Karavirta, A. Korhonen, and L. Malmi. 2006. On the Use of Resubmissions in Automatic Assessment Systems. *Computer Science Education* 16, 3 (September 2006), 229–240.

[22] H. Keuning, J. Jeuring, and B. Heeren. 2019. A Systematic Literature Review of Automated Feedback Generation for Programming Exercices. *ACM Transactions on Computing Education (TOCE)* 19, 1 (January 2019).

[23] C. H. Knoblauch and L. Brannon. 1981. Teacher Commentary on Student Writing: The State of the Art. *Freshman English News* 10, 2 (Fall 1981), 1–4.

[24] A. N. Kumar. 2013. Using Problets for Problem-Solving Exercises in Introductory C++/Java/C# Courses. In *Proc. IEEE Frontiers in Educatoin Conference (FIE)*.

[25] R. Likert. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 140 (1932), 1–55.

[26] R. Lobb and J. Harlow. 2016. Coderunner: a Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (March 2016), 47–51.

[27] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*.

[28] J. Metcalfe and A. P. Shimamura. 1994. *Metacognition: Knowing about Knowing.* MIT Press.

[29] S. Narciss and K. Huth. 2002. How to Design Informative Tutoring Feedback for Multimedia Learning. In *Proc. International Workshop of SIG 6 Instructional Design of the European Association for Research on Learning and INstructions (EARLI)*.

[30] D. Nicol. 2009. *Quality Enhancement Themes: The First Year Experience: Transforming Assessment and Feedback: Enhancing Integration and Empowerment in the First Year.* The Quality Assurance Agency for Higher Education.

[31] N. Parlante. 2011. CodingBat: Code Practice. https://codingbat.com [Online; accessed: 30 March 2019].

[32] Pearson. [n.d.]. My Lab Programming. https://www.pearsonmylabandmastering.com/northamerica/myprogramminglab/ [Online; accessed: 30 March 2019].

[33] J. Prather, R. Pettit, B. A. Becker, P. Denny, D. Loksa, A. Peters, Z. Albrecht, and K. Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*.

[34] K. Sambell, L. McDowell, and C. Montgomery. 2013. *Assessment for Learning in Higher Education.* Routledge.

[35] R. Singh, S. Gulwani, and A. Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[36] W. C. Tam. 1992. Teaching Loop Invariants to Beginners by Examples. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*.

[37] V. Tinto. 1999. Taking Retention Seriously: Rethinking the First Year of College. *NACADA Journal* 19, 2 (Fall 1999), 5–9.

[38] R. Viau. 2009. *La motivation en contexte scolaire* (5th ed.). de boeck.

[39] C. Watson and F. W. Li. 2014. Failure Rates in Introductory Programming Revisited. In *Proc. Conference on Innovation & Technology in Computer Science Education (ITiCSE)*.

[40] D. Wiliam. 2011. What Is Assessment for Learning? *Studies in Educational Evaluation* 37, 1 (March 2011), 3–14.