Automatic Assessment Providing Feedback of Programs based upon Graphical Loop Invariants and its Integration in a CS1 Course

Simon Liénardy

Montefiore Institute
Department of Electrical Engineering and Computer Science
Faculty of Applied Science
University of Liège - Belgium

July 2023

Avant-Propos

Le texte qui suit est une synthèse d'un travail d'enseignement et de recherche qui a pris place à l'Université de Liège dans le cadre d'un mandat d'Assistant. Cet avant-propos expose les raisons pour lesquelles l'auteur ne poursuit pas la rédaction de ce document ni n'en soutient la thèse devant un jury.

Tout est bien qui finit

Le mandat d'Assistant dont les travaux sont exposés dans la suite a couru d'octobre 2013 à décembre 2010, soit un mandat de deux ans, deux fois renouvelé ainsi qu'une septième année et un contrat additionnel de 3 mois. Le présent document a majoritairement été écrit les six premiers mois de 2021, alors que l'auteur était demandeur d'emploi. Quelques sections ont été ajoutées en 22 et 23. Arrivé à dix années au compteur, il est temps de clôturer ce travail et de passer à autre chose.

Dans la suite, j'expliquerai pourquoi j'estime que les prétentions de ce travail sont tenues en échec. Ensuite, je conclurai en l'irréductibilité de cette impasse, en analysant ses conditions matérielles de production.

Enfin, j'ajoute un petit guide de lecture, une *légende*, littéralement « ce qui doit être lu », c'est-à-dire comment aborder le contenu de ce document.

Avertissement La situation intrinsèquement précaire des assistants est inhérente à l'enseignement supérieur universitaire : c'est ainsi que le système procède. Le tableau décrit dans la suite ne consiste pas, à proprement parler, ni un dysfonctionnement, ni – au sens propre – une aberration. Le lecteur attentif veillera donc à ne pas lire plus de mots que ceux qui sont écrits, en particuliers des attaques ou reproches adressées à qui que ce soit. Il essaiera également de ne pas inférer des passions ou du ressentiment des propos de l'auteur.

Omnia obstant

Le travail qui a été réalisé comporte plusieurs volets, chacun occupant une partie de ce document. La première partie se concentre sur une méthode de programmation utilisant un dessin, l'invariant graphique, qui est une représentation de l'invariant de boucle. Notre méthode consiste à

utiliser le plus possible ce dessin lors de la rédaction du code du programme. La version formelle de cette méthode a été introduite par Dijkstra. Par la suite, la méthode de Dijkstra a pu être illustrée par des dessins. Notre approche implique l'utilisation du dessin lui-même, à la place du formalisme logique.

La seconde partie du document présente un outil, appelé Correction Automatique et Feedback des Étudiants (CAFÉ) dans sa première version. L'idée derrière cet outil était de permettre l'exercisation des étudiants tout en promouvant la méthodologie de programmation grâce à l'invariant graphique.

La dernière partie du document détaille, comment l'introduction de CAFÉ peut être vue comme une modification des pratiques d'évaluation dans le cours qui l'utilisait. Des liens sont établis avec le paradigme de l'Assessment for Learning (AfL).

Pour bien faire, il serait souhaitable que ceux qui prétendent détenir une méthode de programmation efficace prouvent leurs dires. Je ne sais pas pour les autres mais moi, je n'en ai jamais été capable. Ma seule bonne foi ne devrait satisfaire quiconque ! Une expérience a été mise en place pour mesurer l'impact de l'invariant graphique sur les programmes des étudiants. Je ne dirais pas que c'est un échec : ça n'a pas marché. En effet, cette démarche expérimentale a été malmenée par un énième confinement dû à la Maladie à Coronavirus 2019. Mais il n'y en eut pas d'autres, faute de temps.

L'outil CAFÉ a d'abord été déployé avant que sa réception par les étudiants ne soit étudiée. Rapidement, son utilisation a donné lieu a une collecte de données, facilitée par sa nature de programme informatique. Il suffit d'enregistrer tout un tas de logs. Par la suite, nous avons analysé ces données, en étant souvent limités par les données que nous avions déjà, parce qu'il avait été aisé de les collecter. Nous ne pouvions pas non plus sacrifier l'enseignement pour la recherche et prévoir des protocoles de prise de données qui auraient impacté négativement l'apprentissage des étudiants dont nous avions la charge. Ainsi, les « questions de recherches » présentées dans ce document ont certainement des points aveugles et ou des biais. De plus, il est arrivé que le taux de réponses des étudiants aux enquêtes et sondages soit si faible qu'il aura rendu certaines données inexploitables.

L'auteur de ce document n'est plus en poste à l'Université de Liège et ne travaille plus sur CAFÉ, dont une nouvelle version a été développée par ailleurs. Il n'enseigne plus dans un cours qui met ainsi en exergue la programmation à l'aide d'un invariant graphique. Il n'est donc plus possible pour lui d'évaluer cette méthode, ni son utilisation dans un outil qui la favoriserait, ni l'impact qu'elle a dans le processus d'évaluation d'un cours de première année dans l'enseignement supérieur.

Il serait vain de s'acharner à vouloir faire dire aux données collectées ce qu'elles ne pourront jamais exprimer. Plus encore, cela irait à l'encontre de la démarche scientifique. C'est pour ces raisons qu'il vaut mieux arrêter maintenant : continuer serait une pure perte de temps.

Légende

Ce document comporte trois parties principales :

I. la présentation de notre méthode de programmation ;

II. la présentation de CAFÉ et son implémentation ;

III. la présentation de l'utilisation de CAFÉ dans le cadre d'un cours de première années (appelé *CS1* en anglais).

La partie I a fait l'objet d'une publication [34], que le chapitre 2 développe avec force détails. Ce chapitre a récemment été complété pour répondre aux objections souvent rencontrées concernant notre méthodologie par dessins. En particulier, notre méthode est tout à fait compatible à des algorithmes concernant des structures de données non linéaires et ne nécessite pas d'écrire une boucle unique. Le lecteur a l'occasion de se plonger dans la méthode à l'aide des nombreux exemples qui sont fournis. Enfin le chapitre se clôture par un lien entre notre méthodologie et la notion de *précondition la plus faible*.

La suite de cette partie balise le travail d'une évaluation de l'efficacité de la méthodologie sur la programmation en dressant une « taxonomie » des erreurs courantes et en relatant une première tentative d'évaluation qui a échoué.

La partie II n'a plus qu'une valeur historique puisqu'elle documente la première version de CAFÉ qui a évolué depuis. Certaines décisions d'implémentations des versions suivantes sont sans doute inspirées de celle qui est décrite dans ce document.

La partie III contient une approche réflexive de l'intégration de CAFÉ dans un cours de première année dans l'enseignement supérieur en Belgique. Sa lecture permet de se faire un bon aperçu des écueils rencontrés dans une telle entreprise.

Table of Contents

Page

Avant-Propos **i**

List of Tables **xi**

List of Figures **xiii**

List of Listings **xvii**

Glossary **xix**

Acronyms **xxi**

Introduction **i**
- A Challenging Context i
- INFO0946: Our CS1 Course ii
- Few Words about our Programming Methodology iii
  - Focusing on What Does not Vary iii
  - Pictures to Communicate Information iv
- Main Contributions iv
  - List of Papers and Talks vi
- Outline vii

I Graphical Loop Invariant Based Programming **1**

1 Loop Invariant Programming Background **3**
- 1.1 Programs, Programming and Loops 3
  - 1.1.1 Preliminary Definitions 3
  - 1.1.2 How to Write a Program? 4
  - 1.1.3 Loops and their Semantics 6
- 1.2 Loop Invariant and Proof 7
  - 1.2.1 Proof for the FACTORIAL Program 8

1.2.2 Loop Termination . . . 10
1.3 Code Construction and Predicate Transformers . . . 11
1.3.1 Weakest precondition . . . 11
1.3.2 Constructive approach . . . 12
1.3.3 Constructive approach for the factorial . . . 13
1.4 Related Work . . . 15
1.5 Programming Reference Books . . . 16
1.5.1 Loop Presentation and Design Method . . . 19
1.5.2 Sorting Algorithms . . . 20
1.5.3 The Binary Search Case . . . 21
1.5.4 Other Mentions of Loop Invariant . . . 21
1.5.5 Contradictions in the Literature . . . 22
1.6 Conclusion and Research Questions Introduction . . . 24

2 Graphical Loop Invariant Based Programming Principles **27**
2.1 Graphical Loop Invariant Based Programming Methodology . . . 27
2.1.1 A Graphical and Informal Version of the Loop Invariant . . . 27
2.1.2 Carefully Depicting a Graphical Loop Invariant . . . 29
2.1.3 Using the Graphical Loop Invariant to deduce the code instructions . . . 33
2.1.4 Positioning the Graphical Loop Invariant with Respect to Problem Solving . 38
2.1.5 Conclusion . . . 45
2.2 Applying Graphical Loop Invariant Based Programming to Common Data Structures 45
2.2.1 Graduated Lines . . . 46
2.2.2 Number Representations . . . 47
2.2.3 Arrays . . . 49
2.2.4 Linked Lists . . . 53
2.2.5 Queues and Stacks . . . 57
2.2.6 Graphs and Trees . . . 59
2.3 Comparison Between Graphical and Non-Graphical Loop Invariants . . . 64
2.3.1 Formal Loop Invariant . . . 64
2.3.2 Loop Invariant in Natural Language . . . 65
2.3.3 Conclusion . . . 66
2.4 GLIBP Methodology and predicate transformers . . . 66

3 Learning Tools **75**
3.1 Glide . . . 75
3.1.1 Available patterns . . . 76
3.1.2 Dividing Lines . . . 77
3.1.3 Defining zones . . . 77

3.1.4 Editing a Graphical Loop Invariant 78
3.1.5 Graphical Loop Invariant Validation 78
3.1.6 Writing the Code with GLIDE 79
3.2 Blank Graphical Loop Invariant for automatic assessment 81
3.2.1 Example of Blank Graphical Loop Invariant 81

4 Errors Taxonomy and Glibp Methodology Reception **85**
4.1 Errors Taxonomy 85
4.1.1 Graphical Loop Invariant 87
4.1.2 Loop Variant 87
4.1.3 Code 87
4.2 Methodology 88
4.2.1 Learning Analytics 88
4.2.2 Performance Tests 90
4.2.3 Surveys 90
4.3 Results 91
4.3.1 Participation Data 91
4.3.2 Performance Data 93
4.3.3 Perception Data 96
4.4 Discussion 100
4.4.1 Participation Data 100
4.4.2 Performance Data 100
4.4.3 Perception Data 102
4.5 Conclusion 102
4.5.1 Future Work 103

5 First Steps Towards Evaluation of the Blank GLI **105**
5.1 Methodology 105
5.1.1 Impossibility of Randomized Controlled Trial 105
5.1.2 Crosover RCT 106
5.1.3 Experimental Process 107
5.2 Hypotheses 112
5.3 Results 112
5.4 Conclusion and Further Work 114
5.4.1 Conducting experiments on Graphical Loop Invariant 114
5.4.2 Using GLIBP methodology 114

II Automatic Correction and Feedback to the Students 117

6 Automatic Code Assessment and Feedback Background 119
6.1 Related Work on Automatic Feedback 119
6.2 Advantages and Drawbacks of Automatic Feedback 122
6.2.1 Advantages 122
6.2.2 Drawbacks 122
6.3 Research Questions 123
6.4 Introduction to CAFÉ 123

7 CAFÉ Principles 125
7.1 Interacting with CAFÉ 125
7.1.1 Exercises Instructions 126
7.1.2 Template 127
7.1.3 CAFÉ Response 127
7.2 Overview: Processing a Student's Input in Three Main Steps 129
7.3 Preprocessing 130
7.4 Correction 132
7.4.1 Correction Principles and Grade Computation 132
7.4.2 Enforcing the exercises constrains 135
7.4.3 Assessing the Compliance with the GLIBP Methodology 136
7.4.4 Detecting Infinite Loops 136
7.5 Response Generation 143
7.5.1 Adding *feedforward* 145
7.6 Plagiarism Detection 148
7.7 Conclusion and Future Work 148

8 Café Evaluation 149
8.1 Methodology 149
8.1.1 Performance Data 149
8.1.2 Perception Data 150
8.2 Results 150
8.2.1 Performance 150
8.2.2 Perception 151
8.3 Discussion 153
8.3.1 Performance 153
8.3.2 Perception 153
8.4 Conclusion 154

III Programming Challenges Activity **155**

9 Introduction to the Programming Challenges Activity **157**
9.1 Need for a PCA . . . 157
9.2 Evolution of the CS1 Course . . . 158
9.3 Programming Challenges Activity . . . 159
9.3.1 Challenges . . . 160
9.3.2 Dealing with absences and Trump Cards . . . 161
9.4 Assessment for Learning . . . 161
9.5 Conclusion and Research Questions Introduction . . . 162

10 Programming Challenges Activity Evaluation **163**
10.1 Methodology . . . 163
10.1.1 Cohorts Presentation . . . 164
10.1.2 Data Sources . . . 164
10.2 Results . . . 166
10.2.1 Participation . . . 166
10.2.2 Performance . . . 169
10.2.3 Perception . . . 172
10.3 Discussion . . . 174
10.3.1 Participation . . . 174
10.3.2 Performance . . . 176
10.3.3 Perception . . . 177
10.4 Conclusion . . . 179

11 PCA and Assessment for Learning **181**
11.1 Methodology . . . 181
11.1.1 Data Sources . . . 182
11.2 Results . . . 182
11.2.1 Participation . . . 182
11.2.2 Performance . . . 184
11.3 Discussion . . . 186
11.3.1 Participation . . . 186
11.3.2 Performance . . . 187
11.4 Conclusion . . . 188

12 Conclusion **189**
12.1 Part I: Graphical Loop Invariant Based Programming . . . 189
12.2 Part II: Automatic Correction and Feedback to the Students . . . 191
12.3 Part III: Programming Challenges Activity . . . 191

12.4 Future Work and Research Directions 192
12.4.1 Push forward the GLIBP methodology Assessment 192
12.4.2 The Dropouts Case 192
12.4.3 GAMECODE: GLIBP methodology exercises 192
12.4.4 Integrating CAFÉ, GLIDE and even the GAMECODES 193

IV Appendices **195**

A Guarded Commands Metalanguage and Weakest Precondition **197**
A.1 weakest precondition calculation 198
A.1.1 `skip` 199
A.1.2 `abort` 199
A.1.3 Assignment 199
A.1.4 Composition 199
A.1.5 Alternative construct 199
A.1.6 Iterative construct 200

B Results of the Focus Group on CAFÉ Messages **201**

Bibliography **217**

List of Tables

Table **Page**

1.1 Algo books . . . 17
1.2 Other books . . . 17
1.3 Language books . . . 18
1.4 Non-CS books . . . 18

2.1 Evaluation of the Graphical Loop Invariant shown in Figure 2.1 for several states space points. . . . 30
2.2 Colour code for the elements in the graphical representation of data structures and Loop Invariants. . . . 46
2.3 Matching between Fig. 2.22 steps and Listing 2.14 . . . 57
2.4 Queue and Stack patterns . . . 58

3.1 Effect of the five buttons of Graphical Loop Invariant Drawing Editor (Glide) GUI . 76
3.2 Glibp guidelines and the corresponding verifications performed by Glide . . . 79
3.3 Available choices for the Blank Loop Invariant . . . 81
3.4 An example of a student's answer . . . 83

4.1 Errors Taxonomy for the Glibp methodology . . . 86
4.2 Programming activities organized during the semester . . . 89
4.3 Surveys Respondents . . . 90
4.4 Answers to the question "Does the GLIBP [...] help you afterwards when solving problems?" . . . 99

5.1 Loop Invariant for P_P: Multiple choices . . . 108
5.2 Loop Invariant for P_G: Multiple choices . . . 109
5.3 Groups for the experiment to asses Blank Graphical Loop Invariant . . . 110
5.4 Proportions of correct codes among the subgroups . . . 113
5.5 Proportions of correct codes among the students of subgroups who committed least than 4 errors in their invariant. . . . 113

6.1 Tools and concepts about automatic feedback generation . . . 120

7.1 Pieces of answer example 133

8.1 Surveys Respondents 150

9.1 Weights of the assessment in the final mark 159

10.1 Statistics about the students who took our CS1 course from Academic years 2016–2017 to 2019–2020 164
10.2 Surveys Respondents 165
10.3 Reasons given to not play one's *Trump Card* 173

11.1 Populations of the CS1 Course from Academic Years 2013–2014 to 2019–2020 182
11.2 Participation in the Mid-Term and Final Exam from Academic year 2013–2014 to 2019–2020 184
11.3 Link Between Students Participation and Performance 185
11.4 Description of the categories used in Fig. 11.3 in term of performance markers of our error taxonomy introduced in Chap. 4 185

List of Figures

Figure **Page**

1 Emergency exit direction . . . iv

1.1 A program as a black box . . . 4
1.2 Loop semantic as a flowchart . . . 6
1.3 Loop zones and logical assertions . . . 12

2.1 Graphical Loop Invariant for computing the product of integers between a and b . . . 28
2.2 Graphical Loop Invariant and particular cases for computing the product of integers between a and b. . . . 35
2.3 Determining the loop body from the Graphical Loop Invariant. . . . 37
2.4 Examples of hourglasses with different widths. . . . 38
2.5 Using a drawing to make arise simpler sub-problems . . . 39
2.6 Graphical Loop Invariant for the top triangle of the hourglass . . . 41
2.7 A different division of the hourglass . . . 42
2.8 A Graphical Loop Invariant for the HOURGLASS problem solved with a single loop. . . . 43
2.9 Relation between row and column positions and characters in an hourglass . . . 43
2.10 A line for representing ordered sets *e.g.*, Natural or Integers . . . 47
2.11 Pattern for Graphical Loop Invariant involving number representations . . . 47
2.12 Graphical Loop Invariant involving number representations . . . 48
2.13 Pattern for Graphical Loop Invariant involving an array A of size N . . . 49
2.14 Postcondition of the BINARYSEARCH problem . . . 50
2.15 Example of Graphical Loop Invariant involving an array: binary search . . . 50
2.16 Example of Graphical Loop Invariant involving an array: BINARYSEARCH . . . 51
2.17 Explaining graphically the instructions of the loop body . . . 53
2.18 Patterns for Simply and Doubly Linked List . . . 53
2.19 A Doubly Linked List Cell . . . 54
2.20 Graphical Loop Invariant of the COPYLIST problem . . . 55
2.21 Variables initialisation is eased . . . 56
2.22 Appending a cell to a linked list . . . 57
2.23 Graphs examples . . . 60

2.24 Graphical Loop Invariant for the SHORTESTPATHTOVERTEX problem 62
2.25 A one-size-fit-all Graphical Loop Invariant for iterators and for-each loops 63
2.26 Graphical Loop Invariant for BISECTIONSEARCH (already presented in Sec. 2.2.3 . . 64
2.27 Postcondition for the DUTCHFLAG problem 67
2.28 Graphical Loop Invariant for the DUTCHFLAG problem 67
2.29 wp(INIT, *Inv*) for the DUTCHFLAG problem 69
2.30 DUTCHFLAG problem: graphical weakest precondition for the White area 72
2.31 DUTCHFLAG problem: graphical weakest precondition for the Red area 72
2.32 DUTCHFLAG problem: graphical weakest precondition for the Blue area 73

3.1 GLIDE Graphical User Interface: five buttons and a canvas 75
3.2 Pattern available in GLIDE to draw Graphical Loop Invariants 76
3.3 GLIDE screenshot 78
3.4 Illustration of GLIDE capabilities when writing the code 80
3.5 Blank Graphical Loop Invariant for the problem ARRAYINTERSECTION 82
3.6 Blank Graphical Loop Invariant for the problem ARRAYINTERSECTION filled with the student answers listed in Table 3.4 83

4.1 Positioning of Programming Activity (PA)over the semester. 89
4.2 Participation rate to the long-term survey 91
4.3 Participation results. 92
4.4 Cumulative error distribution with respect to performance markers. 94
4.5 Performance marker distribution for Graphical Loop Invariant 95
4.6 Program quality over the various PA 97
4.7 Responses to the one time surveys from 2017–2018 to 2019–2020 (1/2) 98
4.8 Responses to the one time surveys from 2017–2018 to 2019–2020 (2/2) 99

5.1 Representation of a Randomized Control Trial 106
5.2 Representation of a Crossover Randomized Control Trial 107
5.3 Blank Graphical Loop Invariant and expected answer for P_P 108
5.4 Blank Graphical Loop Invariant and expected answer for P_G 109

7.1 Activity Diagram of a Student submission to Correction Automatique et Feedback des Étudiants (CAFÉ) 126
7.2 Overview of a Student's Input Processing 129
7.3 Student's Input Preprocessing 131
7.4 The array stored in the `struct fake_array` 140
7.5 An example of Message 144
7.6 Adding Feedforward during Message Generation 146
7.7 Determining the position of a *feedforward* message 147

8.1 Submissions problems using CAFÉ 151
8.2 Students' survey responses (2/2) 152

9.1 Example of a semester timeline 159
9.2 Challenges timeline 160

10.1 Distribution of students' participation in the Programming Challenges Activity (PCA) over the semester 166
10.2 Distribution of submissions per day during the PCA 167
10.3 First submission time heat map 168
10.4 Time between submissions for a Challenge 169
10.5 Box plot of the students' results for their first submission for each Challenge 170
10.6 Improvement through multiple submissions 171
10.7 Responses to the surveys (1/2) 172
10.8 Students' survey responses (2/2) 173

11.1 Participation rate in the Multiple Choice Questions (MCQs) Tests 183
11.2 Participation to the PCA, on a Per Challenge Basis, from Academic year 2016–2017 to 2019–2020 183
11.3 Final Exam Code Quality from Academic Years 2014–2015 to 2019–2020 186

List of Listings

Listing		Page
1.1	Syntax of a loop	6
1.2	Code for the factorial of n	7
1.3	Structure of a loop, zones and logical assertions	12
1.4	Factorial program deduced using constructive approach and explicit weakest pre-condition calculation.	15
2.1	Pattern of a loop	34
2.2	Zone 1	35
2.3	LOOP CONDITION	36
2.4	Zone 2	37
2.5	Final Code	37
2.6	Code for the Hourglass problem	40
2.7	Code for the SP1 – Print a “spaces/stars/newline” pattern	40
2.8	Code for the SP2 – print a trapezoid	41
2.9	Code for the SP3 – print a triangle	42
2.10	Code for the Hourglass problem with a single loop	44
2.11	Code for the ReverseNumber problem	48
2.12	Code for the BinarySearch problem	51
2.13	Structure of a Double Linked List Cell	54
2.14	Code for CopyList problem	55
2.15	Code for ShortestPathToVertex problem	62
2.16	DutchFlag: first try for INIT instructions	68
2.17	DutchFlag: final INIT instructions	68
2.18	DutchFlag program	71
5.1	Code to be filled in with the student answer for P_P	108
5.2	Code to be filled in with the student answer for P_G	109
7.1	Code template provided to the student.	127
7.2	Exercise template to fill and submit to Café	128
7.3	General Form of an class Inheriting from Groundtruth	132
7.4	General Form of a Correction Function	134
7.5	Macro Definition to Count Iterations	137

7.6 While Loop Modification to Count Iterations 137
7.7 Do While Loop Modification to Count Iterations 138
7.8 For Loop Modification to Count Iterations 138
7.9 An Example of Student Code Involving an Array 139
7.10 Header for Out-Of-Bound Accesses Checking 139
7.11 Module for Out-Of-Bound Accesses Checking 140
7.12 Student's Code after Macro Expansion 141
7.13 Out-Of-Bound Accesses Checking Use Case Example 142
7.14 Example of Mock Function Declaration: `malloc` 143
7.15 Example of Mock Function Definition: `fopen` 143
A.1 Guarded commands metalanguage "cheatsheet" 197
A.2 Guarded commands metalanguage version of our instructions 198

Glossary

CS1	This term refers to the first course in Computer Science curricula, that tackles most of the time introduction to computer programming and basic algorithms. (CS1 stands for Computer Science 1).
Dividing Line	In a Graphical Loop Invariant, this is a line that separates what has been already achieved and what should still be done. Deducing the code through graphical manipulations requires to move this line over the picture of the Loop Invariant.
feedback	Information about the student's performance..
feedforward	Information about what should be done in order to improve one's performance.
Loop Body	In the context of a loop, this is the list of instructions that are executed at each iteration.
Trump Card	Term originally designing a powerful card in card games. In the context of the PCA, it is the opportunity offered to students to skip one of the Challenges at their will. The French term actually used with the students is "*Joker*".

Acronyms

AfL Assessment for Learning.

CAFÉ Correction Automatique et Feedback des Étudiants.

CRCT Crossover Randomized Controlled Trial.

GLIBP Graphical Loop Invariant Based Programming.

GLIDE Graphical Loop Invariant Drawing Editor.

MCQ Multiple Choices Questions.

PA Programming Activity. In Chap. 4, this term designate an activity during which the students must use the Graphical Loop Invariant to deduce code instructions..

PCA Programming Challenges Activity.

RCT Randomized Controlled Trial.

Introduction

WITH the digitalisation era we are currently facing, the industry is more and more in demand for coding skills from their employees [36]. This importance of coding, first highlighted by Papert [150], is currently supported by recent researches [132]. In such a context, it is of the highest importance for Higher Education entities to enhance students training in computer programming. This document contributes to this enhancements by proposing and evaluating a programming methodology based on a graphical version of Loop Invariant as well as tools and activities that help teaching it in a first year course.

A Challenging Context

In Belgium, open access to Higher Education is the rule, with some exceptions in the Medicine and Engineering Faculties. Any student who graduates from secondary school, regardless of their curriculum, can enrol at the university. Hence, we cannot make any kind of assumptions about a first year student's background. In particular, it is usual for a first year student in Belgium to lack skills in Mathematics, leading to poor abstraction capacities as well as a lack of rigour in problem solving. However, the Joint Task force on Computing Curricula "recognize[s] that general facility with mathematics is an important requirement for all CS students" [181, p. 49].

In addition, even if we set aside the prerequisites for the CS curriculum, is is worth noticing that a freshperson must learn a lot of new concepts and tools in order to eventually compile and run their own code:

- *How a computer works?* To begin with, the data representation may be a challenge to understand. Every data manipulated by a computer is composed of bits: the integers, the floating point numbers, the characters, the images, the sound, and so forth. For example, in C language, all these expressions look like but are very different in term of values and representations: `0`, `0.f`, `0.`, `'0'`, `"0"`, `'\0'`[1].

- *Using a terminal*. In many situations, using a command-line interface is handier than a GUI to perform certain tasks *e.g.*, to call a compiler. But again, deciphering an error

[1] Respectively: 0 as an int; as a float; as a double; the character 0, which equals 48; the string composed of the character 0 and null terminating byte; the null terminating byte, which equals 0.

message printed following an aborted compilation demands some training for a neophyte: for the first time, the punctuation marks matters (or the indentation if resp. C or Python is concerned).

- *A very new language* but that reuses already known symbols, but with sometimes a quasi-opposite meaning. Let us take the example of the '='. In mathematics, writing $x = 3$ means that the value of x is fixed and will not change any more. Such an expression would typically appears in the conclusion of a mathematical reasoning. On the other hand, in a computer program, `x = 3` could appear nearly anywhere in a piece of code, especially at the beginning, to initialise the value of `x`. This value of `x` could constantly change through the program execution. Of course, an expert does know that very well[2] but it is not surprising to see beginner students struggling with such false friends.

- *New ways of thinking* Who apply Divide-and-Conquer strategy on a daily basis? Who can genuinely pretend that recursive thinking is intuitive (or any other computer-related concept)? Even if it were so, referring to such intuitions would not be of any help for novices. In French, the popular expression "*la bosse des maths*" is inherited from Phrenology [158], a pseudoscience of the 19th century that hypothesised the character of a person could be inferred from their skull shape. The expression remained in French to describe people naturally gifted in maths. Those who lack of maths skills have their *ad hoc* explanation: they do not possess "the bosse", as the Muggles cannot use magic in Harry Potter novels [161]. Those kinds of popular beliefs about maths are harmful enough to be transposed to Computer Science by invoking concept such as intuition while introducing a subject.

To sum up, here is this document challenge: to make a freshperson learn to write short programs without relying on any mathematical or CS-like background or intuitions. To do so, we introduce the students to a programming methodology based on Graphical Loop Invariant. Moreover, to help the students to strengthen their programming skills, we help them to work on a regular basis thanks to a Programming Challenges Activity (PCA) which consists for them to solve short programming problems that are automatically corrected by Correction Automatique et Feedback des Étudiants (CAFÉ), a program we developed to assess both the program and the way it was written in line with our programming methodology.

INFO0946: Our CS1 Course

In the previously introduced context, we[3] organise each year a *CS1* course at the Montefiore Institute of the University of Liège and is entitled "Introduction to Computer Programming". The

[2] The = may keep being confusing. Passing from C to Python, the = operators have slightly different semantics: C's = looks more like Python's `:=`.

[3] The author of this document was the Teaching Assistant of the course from 2014 to 2020.

course is scheduled from mid-September to mid-December. The course content addresses the following topics [58]:

- "Basic syntax and semantic of the C language;
- Simple algorithms (linear run of an array, cumulative mathematical operations, binary search, introduction to sorting problems);
- Dividing a problem into sub-problems and introduction to development methodology (problem analysis, Loop Invariant, Loop Variant);
- Algorithms analysis (complexity);
- Basic data structures (record, array, string, files);
- Program modularity (function/procedure, global variable) and documenting code with specifications (defensive programming);
- Pointers and memory dynamic allocation".

In the following of this document, the course will be referred as "our CS1" course.

Few Words about our Programming Methodology

Our CS1 course introduces a programming methodology based on what we call a "Graphical Loop Invariant". We introduce here briefly the rationale behind this concept. As the name indicates, a Graphical Loop Invariant is both a Loop Invariant and a picture.

Loop Invariant A loop is a code instructions that asks the computer to repeat a set of instructions until a condition is met. The quantities manipulated by the program vary constantly during a loop execution. However, the principle of a Loop Invariant is precisely to focus on what does not vary, hence the name: invariant.

Picture We promote graphic representation because it enables to express succinctly a large amount of information. Drawing a picture is a heuristic for solving problem mentioned by Pólya [155]. Our programming methodology also refers to diagrammatic reasoning [7] since it is based on a picture to help write the code of a program.

Focusing on What Does not Vary

Describing and understanding the changes in a system can be effectively done by focusing on what does not vary in it. The examples are numerous in Science: all the conservation equations describe what is conserved in a system and enable to predict this system evolution. Let us take a non-CS example with the chemical reaction involving methane and dioxygen (*i.e.*, methane combustion) [134]:

$$CH_4 + 2O_2 \longrightarrow 2H_2O + CO_2 \tag{1}$$

Figure 1: Emergency exit direction as it can be found in Montefiore Institute [171].

If we just consider what is changing, we notice that methane (CH_4) is burned in presence of dioxygen (O_2) and form water (H_2O) and carbon dioxide (CO_2).

Now, if we consider what is conserved, we know (from the principle of mass conservation) that atoms cannot be created nor destroyed during such a reaction, hence both sides of the equation represent 1 atom of carbon, 4 atoms of hydrogen, and 4 atoms of oxygen, enabling us to equilibrate the reaction by introducing coefficients before O_2 and H_2O in equation 1. Therefore, we can conclude that to burn 5 molecules of methane, we need 10 molecules of dioxygen and the combustion will produce 5 molecules of carbon dioxide and 10 molecules of water.

Moreover, the conservation of energy states that the total energy of an isolated system is conserved. If the methane combustion happens in a closed system, we can calculate and predict the energy that will be released during the combustion [134].

To sum up, focusing on what is conserved during a phenomenon as the methane combustion makes it possible to accurately describe and predict what is going on while we experiment even a quick and brutal change (here, a burning gas). The transposition of the observations of properties that are conserved to values manipulated by a computer program is the principle of the Loop Invariant.

Pictures to Communicate Information

Drawings are powerful to represent succinctly ideas [7, 155]. This can be illustrated *e.g.*, with Fig. 1 which is composed of three subsequent symbols: a person running, an arrow, and a rectangle. Anyone who sees this picture identifies themself as the running person, understands that the arrow represents a direction, and that the rectangle stands for a door. Additionally, the subtext associated with the pictogram is to follow the arrow direction to find a door to be used but just in case of emergency. This example illustrates that picture may communicate a lot of information with only a few symbols. Our Graphical Loop Invariant will also leverage that property.

Main Contributions

This document presents a Graphical Loop Invariant Based Programming (GLIBP) methodology and its use in a CS1 course, including is integration into a Programming Challenges Activity (PCA) that consists of small programming exercises that are automatically assessed and graded. Here are our main contributions:

1. We describe how we pushed further the use of Graphical Loop Invariant as a programming methodology. We bring rules that students can follow in order to find their own Graphical Loop Invariant for a particular problem. the rules may also be used to assess such Graphical Loop Invariants.

2. We describe how the drawing can be manipulated to deduce the code instructions. The graphical modifications take the place of calculations as equation solving.

3. We show how to represent data structures in our programming methodology framework

4. We present learning tools that may be used in to drawn Graphical Loop Invariant (the Graphical Loop Invariant Drawing Editor (GLIDE)) and to communicate Graphical Loop Invariant to an automatic correction system (the Blank Graphical Loop Invariant)

5. We present results about the student's participation in teaching activities related to our methodology to answer the research question:

   RQ 1. 1 *How the students seize the opportunity to practice the* GLIBP *methodology?*

6. We present a taxonomy of error and leverage it to answer the following research questions:

   RQ 1. 2 *What kind of error are committed using the* GLIBP *methodology?*
   RQ 1. 3 *Can we link the error committed with the* GLIBP *methodology to programming errors?*

7. We present the results of several surveys we conducted to answer the following question :

   RQ 1. 4 *How the* GLIBP *methodology is perceived by the students?*

8. We describe a Crossover Randomised Control Trial to assess the GLIBP methodology to answer the question:

   RQ 1. 5 *Does the* GLIBP *methodology enable to write better pieces of code?*

   However, the second lock-down due to the Covid-19 pandemic [190] forced us to modified this question into the following:

   RQ 1. 5 *Does the Blank Graphical Loop Invariant enable to write better pieces of code?*

9. We answer to the question:

   RQ 2. 1 *Can we get a system that automatically assess the* GLIBP *and provide relevant feedback?*

By presenting CAFÉ, a system that enables to automatically correct both students code and Graphical Loop Invariant. CAFÉ provide students with a message containing *feedback* and *feedforward*.

10. We provide an evaluation of CAFÉ's message to answer the question:

    RQ 2. 2 *How the feedback produced by* CAFÉ *is received by students?*

11. We introduce and evaluate a PCA that is a programming activity consisting for students in submitting small pieces of codes – called challenges – to CAFÉ through a submission platform several times during the semester. Doing so, we align the course on Assessment for Learning (AfL) principles. We address the following questions:

    RQ 3. 1 *Does the* PCA *and its multiple features promotes student's engagement during the semester?*

    RQ 3. 2 *Does the course transformation through Assessment for Learning (*AfL*) lead to a stronger students' engagement and a higher success rate?*

List of Papers and Talks

The work discussed in this document lead to several publications and talks that are listed hereafter:

- Simon Liénardy, Laurent Leduc, and Benoit Donnet. CAFÉ: an automatic and on-line learning system to guide freshmen towards the meeting of higher education requirements. *European First Year Experience Conference 2018*. Slides, 2018. URL `http://hdl.handle.net/2268/221117`

- Simon Liénardy, Laurent Leduc, and Benoit Donnet. CAFÉ : un système d'évaluation et de feedback automatique des étudiants en science informatique*Colloque du DIdactifen 2018 : Les disciplines enseignées : des modes de penser le monde*. Slides, 2018. URL `http://hdl.handle.net/2268/222640`

- Simon Liénardy, Lev Malcev, and Benoit Donnet. Graphical loop invariant programming in CS1. *Grascomp Doctoral Day 2019 (GDD'19)*. Long Abstract and Slides, 2019. URL `http://hdl.handle.net/2268/241671`

- Simon Liénardy, Laurent Leduc, Dominique Verpoorten, and Benoit Donnet. Café: Automatic correction and feedback of programming challenges for a CS1 course. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pages 95–104, 2020. doi: 10.1145/3373165.3373176

- Simon Liénardy. Learning computer programming around a CAFé. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 318–319. ACM, aug 2020. doi: 10.1145/3372782.3407119
- Simon Liénardy and Benoit Donnet. GameCode: Choose your Own Problem Solving Path. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, page 306. ACM, aug 2020. doi: 10.1145/3372782.3408122
- Simon Liénardy, Benoit Donnet, and Laurent Leduc. Promoting engagement in a CS1 course with assessment for learning. *Student Success*, 12(1):102–111, mar 2021. doi: 10.5204/ssj.1668
- Simon Liénardy, Laurent Leduc, Dominique Verpoorten, and Benoit Donnet. Challenges, multiple attempts, and trump cards: A practice report of student's exposure to an automated correction system for a programming challenges activity. *International Journal of Technologies in Higher Education*, 18(2):45–60, 2021. ISSN 1708-7570. doi: 10.18162/ritpu-2021-v18n2-03
- Simon Liénardy. Ça ferait presque le CAFÉ : une évaluation continue, à distance, fournissant du feedback et automatisée – Retour sur 4 ans d'évolution du cours d'Introduction à la Programmation en Sciences Informatiques à l'ULiège. *Rallye Péda.9 – Vers une évaluation 2.0 ? De la menace à l'opportunité*. Youtube Video and slides, 2020. URL `http://hdl.handle.net/2268/253206` and `https://youtu.be/NTV5al5c4PQ`
- Géraldine Brieven, Simon Liénardy, and Benoit Donnet. Lessons learned from 6 years of a remote programming challenge activity with automatic supervision. In *Annual Conference of European Distance and E-Learning Network*, pages 63–79. Springer, 2022. URL `https://hdl.handle.net/2268/292692`
- Géraldine Brieven, Simon Liénardy, Lev Malcev, and Benoit Donnet. Graphical Loop Invariant Based Programming. In Catherine Dubois and Pierluigi San Pietro, editors, *Formal Methods Teaching. FMTea 2023*, volume 13962 of *Lecture Notes in Computer Science*, pages 17–33, Cham, 2023. Springer. doi: 10.1007/978-3-031-27534-0_2

Outline

This document is divided in three part. The first part concerns our programming methodology based on Graphical Loop Invariant. Chap. 1 introduces the background about Loop Invariant from the start, *i.e.*, the definition of a computer program. Chap. 2 desribes in depth our programming methodology. Chap. 3 present tools we developed to facilitate the learning of the GLIBP methodology. Chap. 4 introduces a taxonomy of errors that can be committed using the

methodology and answers the research questions about GLIBP methodology reception. Eventually, Chap. 5 describes our attempt to assess the efficacy of the Graphical Loop Invariant to deduce correct code and explains why we had to focus on assessing the impact of the Blank Graphical Loop Invariant in a Crossover Randomized Controlled Trial (RCT) whose results turned out to not be significant.

The second part of the document introduces CAFÉ, a program to automatically assess students programs and provide them with feedback and feedforward. Chap. 6 introduces the part by presenting the background on automatic assessment and positions CAFÉ with respect to the state of the art. Chap. 7 explains how CAFÉ works and is implemented. Chap. 8 evaluates CAFÉ's message reception through the analysis of students' behaviour and perception.

The third part of the document addresses the Programming Challenges Activity (PCA). Chap. 9 presents the PCA and explain how its introduction enabled to align the course to the principles of Assessment for Learning (AfL). Chap. 10 is dedicated to the PCA evaluation while Chap. 11 focuses on assessing the overall course transformation in the context of the AfL.

Part I | Graphical Loop Invariant Based Programming

Loop Invariant Programming Background

THIS CHAPTER introduces the required background for the first part of the thesis. In particular, it focuses on Loop Invariant and its application to programming. First, Sec. 1.1 defines the main terms that are used throughout the thesis. Sec. 1.2 introduces the Loop Invariant in the context of program proofs. Sec. 1.3 shows its interest in the context of a constructive approach. Sec. 1.4 presents the literature about the Loop Invariant.

The rest of the chapter aims at introducing our own methodology. In the thesis introduction, we already announces that in the context of our *CS1* course, we promote a programming methodology based on Loop Invariant. Sec. 1.5 investigates how programming books address the introduction to loops and what kind of programming methodology they present to write them. Eventually, Sec. 1.6 draws a conclusion and introduces our research questions.

1.1 Programs, Programming and Loops

1.1.1 Preliminary Definitions

Computer programming refers to the activity consisting in writing programs code. Learning to program is typically done through a **CS1** course, usually provided to 1st year students (or students early in their curriculum).

Generally speaking, a **program** designates data manipulations performed by a computer. A computer performing the operations specified by a program is said to be **executing** the program. The **code** of the program is a text, written in a particular **programming language**, that describes non-ambiguously these data manipulations.

From the user point of view, a program can be seen as a black box, as can be seen in Fig. 1.1: they does not necessarily know how the program works but they may give data as input and

Figure 1.1: A program as a black box

observes how it is transformed and returned as output by the program. In practice, all the programs cannot accept all kinds of data and are only able to manipulate data respecting some constrains, called the **Precondition**. The description of a the result of a program execution whose precondition is fulfilled is called its **Postcondition**. Most of time, the user has access to the program manual that provides, at least its Precondition and Postcondition.

1.1.2 How to Write a Program?

We are not offering here a comprehensive description of how to write a program: several books would not be sufficient! However, the main steps to write a computer program include the following:

- Problem analysis;
- Code writing;
- Code Testing.

The order of these steps depends on the methodology followed. For instance, Test Driven Development methodology [19] asks to write tests before writing the code. The steps sequence is also not linear: one does not have to fully complete the analysis to begin the code writing. In fact, modern approaches consider cycling through the several steps as an issue needing for re-analysing the problem may arise in any other steps.

1.1.2.1 Problem Analysis

A program is written to solve a particular problem. The problem analysis consists in defining precisely the problem to solve: what are the input and the output (thus defining the Precondition and the Postcondition)? Should other data be used? This steps may also include the study of the data structures the program has to manipulate, *e.g.*, a particular data base, file(s), and so forth.

If the problem to be solved by the program is large and seems difficult to address, one must divide it in smaller sub-problems that will be eventually solved more easily. This is often referred as following a **Divide-and-Conquer** strategy. Dividing a big problem in smaller ones; specifying them by defining their Precondition and Postcondition and determining how they interact with each other in order to solve the main problem is an important part of the analysis step.

In the following of this thesis, when we introduce a particular (sub-)problem, we are going to

document it in this way:

PROBLEMNAME:

Input – Precondition

Output – Postcondition

The program names will be in small capitals and camel cases, the Precondition and the Postcondition will be stated after the keyword "**Input**" and "**Output**", respectively.

1.1.2.2 Code Testing

Testing consists in checking that a program execution has the expected behaviour (in terms of result, performance, security, and so forth).

As far as the correct result is concerned, one of the most obvious test consists in passing input data to a program and verify that the output is what was expected (*i.e.*, that if the input respects the Precondition then the Postcondition is respected at the end of the program execution). One may also assure that a particular input will not cause the program to crash or to run indefinitely. If a program consists of several sub-problems, it may be simpler to first test them independently. Then, as soon as the sub-problems passed the tests, their integration in the program must also be tested. There are a lot of techniques to perform tests on a program and this is not the subject of this part of the thesis.

However, it is worth noting that, as Dijkstra [55] wrote, "Program testing can be used to show the presence of bugs, but never to show their absence!" (p. 7). The absence of bugs, *i.e.*, the code correctness, can indeed only be assured by a mathematical proof. Due to the lack of resources to systematically prove the correctness of all the code that are written, testing is better than nothing and is sufficient in non-critical contexts.

1.1.2.3 Writing the Code

This step consists in establishing precisely how the data must be manipulated and then writing the corresponding code in a particular programming language. The precise and non-ambiguous description of a particular data manipulation is called an **algorithm**. An algorithm may be **implemented** (*i.e.*, expressed) in any programming languages. Books that present algorithms often use pseudocodes[4] to express them.

Algorithms consists of sequences of instructions that describe operations on data. There are several sort of instructions: computing a value by evaluating an expression, calling another problem to use its result, making a choice between several instructions thanks to a test and repeating instructions several times.

[4] Most of time, such pseudocodes look like C language (that is why we will not use pseudocode in the following)

Repeating instructions several times is the core of complex algorithms and can be achieved, mainly in two way:

1. By recursion, which consists, while solving a problem, in identifying this same problem among its sub-problems. This thesis does not address recursion, as the course
2. By iteration, using loops, as it is described in the following

1.1.3 Loops and their Semantics

```
1 while(loop_condition)
2    loop_body
```

Listing 1.1: Syntax of a loop `loop_condition` is an expression and `loop_body` is an instruction (most of time, it is an instructions block surrounded by curly brackets.

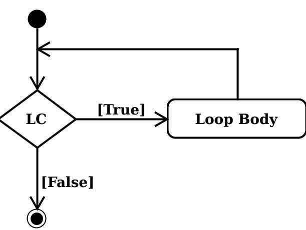


Figure 1.2: Loop semantic as a flowchart. The plain black circle at the top indicates the beginning of the execution flow and the one that is circled at the bottom indicates its end. The diamond represents a test, here called LC for Loop Condition. The outgoing arrows of the diamond are labelled with the two possible test outcomes.

The Listing 1.1 shows the syntax of a `while` loop and Fig. 1.2 illustrates its semantic: while (hence the name) the Loop Condition is evaluated at the value **True**[5], the *Loop Body* is executed. As soon as the Loop Condition is evaluated at the value **False**, the loop ends. The Loop Condition is therefore a condition upon which the iteration goes on. We refer to its logical negation as the **Stop Condition**.

Let us take an example of loop with the following problem:

> FACTORIAL:
>
> **Input** – $n \geq 0$
>
> **Output** – $f = n!$

The code corresponding to the problem is given in Listing 1.2. The variable `i` is used to

[5] In C, any value different from 0 is considered as True

enumerate all the factors from 1 to `n`. The variable `f` is used to accumulate the result of the multiplications during the iteration. At the end, the reader may convince themself that `f` contains the value $n!$. But should they do so? The next section brings a more convincing answer.

```
// Pre: 0 <= n
// Post: f= n!
i = 0;
f = 1;
while(i < n)
{
    i = i + 1;
    f = f * i;
}
```

Listing 1.2: Code for the factorial of n

1.2 Loop Invariant and Proof

Instead of reading carefully a code to convince oneself that the code is correct, it is possible to prove the code correctness. Hoare [88] proposed to write triplet of the form[6]

$$\{P\}\,Q\,\{R\}$$

where P and R are conditions (or assertions) and Q is a program. The triplet means "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" [88]. He also proposed an axiom schema and several rules of inference to write program proof. Here they are:

- "D0 Axiom of Assignement
  $\vdash \{P_0\}\, x := f\, \{P\}$
  where x is a variable identifier; f is an expression; P_0 is obtained from P by substituing f for all occurences of x." [*ibid.*]
- "D1 Rules of Consequence

  If $\vdash \{P\}\,Q\,\{R\}$ and $\vdash R \supset S$ then $\vdash \{P\}\,Q\,\{S\}$

  If $\vdash \{P\}\,Q\,\{R\}$ and $\vdash S \supset P$ then $\vdash \{S\}\,Q\,\{R\}$" [*ibid.*]
- "D2 Rule of composition

  If $\vdash \{P\}\,Q_1\,\{R_1\}$ and $\vdash \{R_1\}\,Q_2\,\{R\}$ then $\vdash \{P\}\,(Q_1;Q_2)\,\{R\}$" [*ibid.*]
- "D3 Rule of Iteration

  If $\vdash \{P \wedge B\}\,S\,\{P\}$ then $\vdash \{P\}\,\textbf{while}\ B\ \textbf{do}\ S\,\{\neg B \wedge P\}$" [*ibid.*]

[6] In Hoare's first work, the triplets are written $P\{Q\}R$. Nowadays, there are noted $\{P\}\,Q\,\{R\}$. We used the later form, even in the quotes of the original work.

1.2.1 Proof for the FACTORIAL Program

We want to prove the following triplet:

$$\{n \geq 0\}\,\text{FACTORIAL}\,\{f = n!\} \tag{1.1}$$

i.e., that if $n \geq 0$, the execution of the program FACTORIAL will give us the value $n!$ stored in the variable f. If we detail the content of FACTORIAL code (see Listing 1.2) in equation 1.1, we obtain the following:

$$\begin{aligned}&\{0 \leq n\}\\ &\; i = 0; f = 1;\\ &\; \textbf{while}(i < n)\ \textbf{do}(i = i + 1; f = f \times i)\\ &\{f = n!\}\end{aligned} \tag{1.2}$$

In order to prove the triplet 1.2, we will show that

$$\{0 \leq n\}\, i = 0; f = 1\, \{0 \leq i \leq n \wedge f = i!\} \tag{1.3}$$

$$\{n \geq i \wedge f = i!\}\,\textbf{while}\ (i < n)\ \textbf{do}\ (i = i + 1; f = f \times i)\{f = n!\} \tag{1.4}$$

are both proven triplets. By the application of the rule D2 on triplets 1.3 and 1.4, we will conclude that triplet 1.2 is proven.

1.2.1.1 Proof of $\{n \geq 0\}\, i = 0; f = 1\, \{n \geq i \wedge f = i!\}$

First, let us take the definition of the factorial as proven:

$$\vdash 0! = 1 \tag{1.5}$$

$$\vdash i! = (i-1)! \times i\ (\text{if } i > 0) \tag{1.6}$$

By applying the axiom schema D0 we can write

$$\vdash \{0 \leq 0 \leq n \wedge 0! = 1\}\, i = 0\, \{0 \leq i \leq n \wedge i! = 1\} \tag{1.7}$$

$$\vdash \{0 \leq i \leq n \wedge i! = 1\}\, f = 1\, \{0 \leq i \leq n \wedge i! = f\} \tag{1.8}$$

By the rule D2 on triplets 1.7 and 1.8 we can infer

$$\vdash \{0 \leq 0 \leq n \wedge 0! = 1\}\, i = 0; f = 1\, \{0 \leq i \leq n \wedge i! = f\} \tag{1.9}$$

We can also observe the following:

$$\vdash 0 \leq n \supset 0 \leq n \wedge true \tag{1.10}$$

$$\vdash 0 \leq n \supset 0 \leq 0 \leq n \tag{1.11}$$

$$\vdash 0 \leq n \wedge true \supset 0 \leq n \wedge 0! = 1 \tag{1.12}$$

$$\vdash 0 \leq n \supset 0 \leq 0 \leq n \wedge 0! = 1 \tag{1.13}$$

Those (trivial) results make appear the base case of factorial definition (equation 1.5). The last equation allows us to apply the second form of the rule D1 on triplet 1.9 to get the proof of triplet 1.3:

$$\vdash \{0 \le n\}\, i = 0; f = 1\, \{0 \le i \le n \wedge f = i!\} \tag{1.14}$$

1.2.1.2 Proof of $\{n \ge i \wedge f = i!\}$ while $(i < n)$ do $(i = i + 1; f = f \times i)\{f = n!\}$

From the axiom D0, we can write the two triplets:

$$\vdash \{0 \le i < n \wedge f = i!\}\, i = i + 1\, \{0 \le (i-1) < n \wedge f = (i-1)!\} \tag{1.15}$$

$$\vdash \{0 \le (i-1) < n \wedge f \times i = (i-1)! \times i\}\, f = f \times i\, \{0 \le (i-1) < n \wedge f = (i-1)! \times i\} \tag{1.16}$$

As, $(0 \le (i-1) < n \wedge f = (i-1)!) \equiv (0 \le (i-1) < n \wedge f \times i = (i-1)! \times i)$, we can use the rule D2 and the two last triplets to write:

$$\vdash \{0 \le i < n \wedge f = i!\}\, i = i + 1; f = f \times i\, \{0 \le (i-1) < n \wedge f = (i-1)! \times i\} \tag{1.17}$$

By definition of the factorial (equation 1.6) and rule D1, we then write:

$$\vdash \{0 \le i < n \wedge f = i!\}\, i = i + 1; f = f \times i\, \{0 \le (i-1) < n \wedge f = i!\} \tag{1.18}$$

We can note that, on one hand $(i < n) \wedge (0 \le i \le n \wedge f = i!) \supset 0 \le i < n \wedge f = i!$ and on the other hand $(i-1) < n \equiv i \le n$ to deduce the following triplet from 1.18 using rule D1 twice:

$$\vdash \{(i < n) \wedge (0 \le i \le n \wedge f = i!)\}\, i = i + 1; f = f \times i\, \{0 \le i \le n \wedge f = i!\} \tag{1.19}$$

Therefore, using rule D3 and triplet 1.19, we can conclude:

$$\vdash \{0 \le i \le n \wedge f = i!\}\, \textbf{while}\ (i < n)\ \textbf{do}\ (i = i + 1; f = f \times i)\{0 \le i \le n \wedge f = i! \wedge i \ge n\} \tag{1.20}$$

Since $(i \le n \wedge i \ge n) \supset (i = n)$, we use rule D1 and triplet 1.20 to obtain

$$\vdash \{0 \le i \le n \wedge f = i!\}\, \textbf{while}\ (i < n)\ \textbf{do}\ (i = i + 1; f = f \times i)\{f = n!\} \tag{1.21}$$

1.2.1.3 Conclusion

By the application of rule D2 and triplets 1.14 and 1.21, we can write (which is what we wanted to prove):

$$\begin{aligned} \vdash &\{0 \le n\} \\ &\quad i = 0; f = 1; \\ &\quad \textbf{while}(i < n)\ \textbf{do}(i = i + 1; f = f \times i) \\ &\{f = n!\} \end{aligned} \tag{1.22}$$

□

In the proof, the logical assertion $0 \leq i \leq n \land f = i!$ is not altered by the loop execution (see equation 1.16) and is therefore called **Loop Invariant**. Moreover, this Loop Invariant expresses how the loop works: the variable `i` takes a value between 0 and `n` while the variable `f` is used to accumulate the product $1 \times 2 \times \cdots \times i$, which is the multiplication of all the values taken by `i` up to a certain iteration.

1.2.2 Loop Termination

We have shown that the loop is correct, but we did not prove that its evaluation will end. This is referred as **Partial Correctness**. In order to get **Total Correctness**, we have to prove the loop termination. To do so, Floyd [67] proposed to use a property of well-ordered sets that is there is no infinite sequence of decreasing elements of a well-ordered set. He introduces a "W-function" (W for Well-ordered) that transforms the state of a program at a certain iteration into a value of a well-ordered set. The values corresponding to two consecutive iterations must be decreasing (according to the set ordering relation). If such a function can be provided, the sequence of the iterations of a certain loop is transformed into a sequence of elements of a well-ordered set that cannot be infinite. Therefore, providing the W-function ensures that the number of iterations will be finite *i.e.*, the loop execution has an end. Later this "W-function" was called *function t* by Dijkstra [56] and is also referred as **Loop Variant**.

Let us take the example of the FACTORIAL problem. The most straightforward well-ordered set is the set of the positive integers (it was indeed already noted by Floyd [67]). The state of each iteration is defined by the value of the two variables `i` and `f`. As Loop Variant, we propose the function $n - i$ (Here, n is considered as a constant).

While the loop iterates (*i.e.*, the Loop Condition is true), this function has a positive integer value:

$$i < n$$
$$0 < n - i$$

Since the variable `i` is increased at each iteration, the values of consecutive iterations are decreasing:

$$n - i > n - (i + 1)$$

Therefore, by providing the function $n - i$, we prove the loop termination.

In the following, we will use Loop Variant that will have a positive integer value (although we have just seen that this is not mandatory) and we can sum up the requirements for such a function:

1. The function combines the values of the program variables (*i.e.*, its domain is the states space);

2. The value of the function must be an integer and positive if the loop condition is true;

3. Let us note t_1 and t_2 the values corresponding to two consecutive iterations, one must have $t_1 > t_2$

Providing a function that respects those three requirements is sufficient to prove a loop termination and the invocation of well-ordered sets properties can be omitted.

1.3 Code Construction and Predicate Transformers

In the previous section, we presented the proof of the correction of the FACTORIAL program. Even for such a small program, the text of the proof is quite long. Moreover, we have to confess to the reader that we assured ourselves to be able to prove the program before writing the code. We followed the methodology presented by Dijkstra [56] and [57] that consists in determining the Loop Invariant first and using it to write the code, that is referred as a **constructive approach**.

1.3.1 Weakest precondition

In his work, Dijkstra introduces the **the weakest Precondition corresponding to a Postcondition**, which is defined as:

> “The condition that characterizes the set of all initial states such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given post-condition” [56, p. 16]

and that is noted[7]

$$\mathrm{wp}(\mathtt{S},\ R)$$

where `S` is the system that is activated and R the Postcondition. Being able to compute wp(`S`, R) from any R means that one can describe the effect of the activation of a system `S`, or in other words, its semantics. Moreover, for a given system `S`, the rules that allow to compute wp(`S`, R) from R describe how to obtain a predicate (the weakest precondition) from another predicate (the Postcondition R) and is thus referred as a **predicate transformer**. To sum up, the semantics of `S` is described by its predicate transformer.

The details of the computation of the weakest precondition for all the commands we use in our pseudocode are given in the appendices (see App. A).

[7] Later work [57] uses another notation for weakest precondition: $wp.S.R$. We still keep wp(`S`, R) as we are used to it from reading [48] fifteen years ago. The command `S` is written with a mono-spaced font to recall programming language and the predicate R is written in italic.

1.3.2 Constructive approach

Since the Loop Invariant was properly chosen, the program proof was easy to demonstrate! In fact, the constructive approach allows to write correct programs without needing for proving them.

```
// Precondition
Zone 1
// Invariant
while(LC)
{
    // Invariant ∧ LC
    Zone 2
    // Invariant
}
// Invariant ∧ ¬LC
Zone 3
// Postcondition
```

Listing 1.3: Structure of a loop, zones and logical assertions

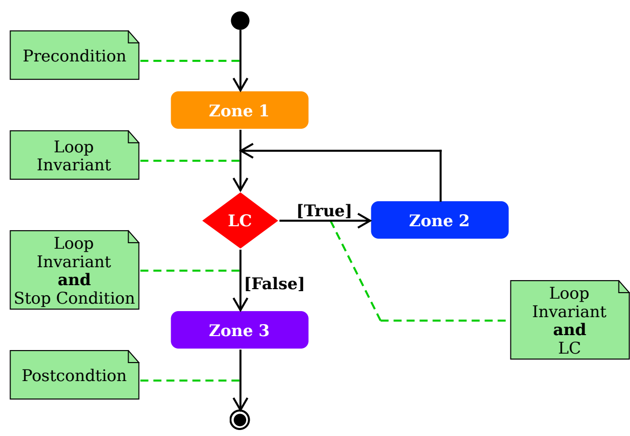


Figure 1.3: Loop zones and logical assertions

The general structure of a loop is presented in Listing 1.3 and its corresponding flowchart is shown in Fig. 1.3. The comments (in blue in Listing 1.3 and green in Fig. 1.3) represent logical assertion that correspond to Hoare's triplet previously presented. For example, one can see that the all program is surrounded by the Precondition and Postcondition forming the triplet

$$\{Precondition\}\ Program\ \{Postcondition\}$$

As far as the loop is concerned (*i.e.*, the LC plus the Loop Body/ZONE 2), it is surrounded by two assertions forming the triplet:

$$\{Loop\ Invariant\}\ Loop\ \{Loop\ Invariant \land \neg LC\}$$

We can see in Listing 1.3 or Fig. 1.3 that the code is divided in three main zones, each zone being constructed thanks to the Loop Invariant.

ZONE 1 refers to the code segment prior to the loop, typically used for declaring and initializing variables. Before ZONE 1, the Precondition is verified. At the end of ZONE 1, the Loop Condition is evaluated for the first time, meaning that the Loop Invariant must be verified. Based on this statement, the Precondition and Loop Invariant can be used to determine the required variables as well as their initial values.

Formally, we must ensure[8] that:

$$Pre \Rightarrow \mathrm{wp}(\text{ZONE1}, Inv) \tag{1.23}$$

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.

Formally, this means showing that:

$$(Inv \wedge B) \Rightarrow \mathrm{wp}(\texttt{if (B) } \text{ZONE2}, Inv) \tag{1.24}$$

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. In some cases, those instructions should lead to the Postcondition of the program.

In other words, ensuring that:

$$(Inv \wedge \neg B) \Rightarrow \mathrm{wp}(\text{ZONE3}, Post) \tag{1.25}$$

To sum up in a constructive approach, the Loop Invariant is the cornerstone of the programming methodology as it is involved in the writing of the whole loop zones. As such, the complexity is not any more in code writing but in finding a strategy (the Loop Invariant) to solve the problem.

1.3.3 Constructive approach for the factorial

The problem of the factorial consists in establishing the postcondition

[8] Dijkstra himself does not systematically computes weakest precondition in his book. Instead, he occasionally uses it to prove his reasoning.

$$Post \equiv f = n! \tag{1.26}$$

As the definition of the factorial is a repetition of products, one could use a loop. Dijkstra indicates that a good way of finding an invariant is by weakening the postcondition[9] . A common way to do that is to replace, in the postcondition, a constant by a variable.

From the postcondition, one could try this Loop Invariant:

$$Inv \equiv f = i! \wedge 0 \leq i \leq n \tag{1.27}$$

This Loop Invariant indicates that two variables must be used : `f` and `i`. Let us denotes the initial values of these two variables by a and b, respectively. One can solve for a and b :

$$\mathrm{wp}(\mathtt{f} = \mathtt{a}; \mathtt{i} = \mathtt{b},\ P) = \mathrm{wp}(\mathtt{f} = \mathtt{a},\ f = b! \wedge 0 \leq b \leq n) \tag{1.28}$$

$$= (a = b! \wedge 0 \leq b \leq n) \tag{1.29}$$

Since n is an integer and can be 0, we conclude that $b = 0$ and $a = b! = 0! = 1$.

One can also note that:

$$Inv \wedge i = n \Rightarrow f = n! = Post \tag{1.30}$$

That suggest to form a loop of the form:

```
f = 1;
i = 0;
while(i != n){
   Body
}
```

The variable i should increase from 0 to n. It suggests to increments the value of `i` at each iteration. Let us compute $\mathrm{wp}(\mathtt{i} = \mathtt{i} + 1,\ Inv)$:

$$\mathrm{wp}(\mathtt{i} = \mathtt{i} + 1,\ f = i! \wedge 0 \leq i \leq n) = (f = (i+1)! \wedge 0 \leq i + 1 \leq n) \tag{1.31}$$

$$= (f = (i+1)! \wedge 0 \leq i < n) \tag{1.32}$$

Compared to $P \wedge B = f = i! \wedge 0 \leq i < n$, we find that `f` should be multiplied by `i + 1` before increasing `i`. The code becomes

```
f = 1;
i = 0;
while(i != n){
   f = f * (i + 1);
   i = i + 1;
}
```

[9] He also mentions his own experience(!) in [56], while [48] argues in favour of intuition.

The order of the two instructions can be inverted (*i.e.*, `i = i + 1;f = f * i`) because:

$$\begin{aligned}
\mathrm{wp}(\mathtt{i = i + 1;f = f * i},\ f = i! \wedge 0 \le i \le n) &= \mathrm{wp}(\mathtt{i = i + 1},\ \mathrm{wp}(\mathtt{f = f * i},\ f = i! \wedge 0 \le i \le n)) && (1.33)\\
&= \mathrm{wp}(\mathtt{i = i + 1},\ f * i = i! \wedge 0 \le i \le n) && (1.34)\\
&= (f * (i+1) = (i+1)! \wedge 0 \le i+1 \le n) && (1.35)\\
&= (f = i! \wedge 0 \le i < n) && (1.36)\\
&= \mathit{Inv} \wedge B && (1.37)
\end{aligned}$$

The penultimate line is obtained thanks to the recursive case of the definition of the factorial. The final program is given in Listing 1.4

```
1 f = 1;
2 i = 0;
3 while(i != n){
4    i = i + 1
5    f = f * i;
6 }
```

Listing 1.4: Factorial program deduced using constructive approach and explicit weakest precondition calculation.

1.4 Related Work

Historically speaking, ensuring the correctness of a program before running it was a necessity: the computing resources were scarcer than nowadays and a program had to be correct before being executed on a computer [86].

In 1967, Floyd [67] proposed to assign meanings to programs and used flowcharts and logical assertions to do so. His paper also introduced the principle of Loop Variant based upon well-ordered sets. Hoare [88] continued the work and set an axiomatic basis for computer programming that is nowadays known as Floyd-Hoare logic. Morris and Jones [144] mentions an article on program correctness from Alan Turing nearly twenty years earlier but acknowledges having "no evidence that the paper influenced the later contributors to the ideas of program proofs".

While there is an abundant literature on Loop Invariants for code correctness and on automatic generation of Loop Invariants (*e.g.*, [30, 42, 69, 108, 109, 160, 165, 167]), their usage for building the code has attracted little attention from the research community.

With respect to Loop Invariant based programming (*i.e.*, the Loop Invariant applied in a constructive approach), the seminal work has been proposed by Dijkstra [56], followed by Meyer [142], Gries [81], and Morgan [143]. As such, the program construction becomes a form of problem-solving, and the various control structures are problem-solving techniques. Those works proposed Loop Invariant as logical assertion.

Tam [179] suggests to introduce students to Loop Invariant as early as possible in their curriculum. Tam describes several examples of code construction based on informal Loop Invariants expressed in natural language.

Astrachan [10] is probably the closest to our work (see Chap. 2) as he suggests the use of Graphical Loop Invariants in the context of CS1/CS2 courses. However, his approach is incomplete as the suggested drawing lack of completeness (*e.g.*, objects manipulated, such as arrays, are not named according to code variables), might lead to confusion (*e.g.*, variables positions around the dividing line are somewhat unclear), and the drawing is not explicitly manipulated to derive particular situations (*e.g.*, code prior and after the loop).

Finally, Back et al. [11–13] proposed nested diagrams (a kind of state charts) representing, at the same time, the Loop Invariant and the code. However, in such a situation, Loop Invariants are expressed as logical assertions, possibly leading to difficulties to students with a low mathematical and abstraction background.

To the best of our knowledge, none of these works evaluate the reception, by students, of a programming methodology based on Loop Invariant, as we do in Chap. 4.

Systems using predicate transformers (such as the weakest precondition) theory to prove code correctness have been developed, *e.g.*, [14–16, 20, 26, 40, 41, 64, 66, 103, 113, 114]. They are beyond the scope of this document.

1.5 Programming Reference Books

A large variety of books addressing programming have been proposed since the 1970s. The following review does not pretend to be exhaustive. However, we selected references that introduced Loop Invariant, on one hand, and the books published after 2000 available in the Liege university's library, on the other hand. These last are a good sample of what was published both in French and in English during the last two decades. Although they were displayed on the same bookshelf, we omitted books about Unified Model Language [27] and those about design patterns programming as they are off-topic. All the considered books are presented in Tables 1.1, 1.3 and 1.4.

We classified the books in three main categories and a subcategory. The first one, that we refer as LANGUAGE books consists of those who present a particular programming language : C, C++, C#, Fortran, Java, Lua, Python or even MATLAB (see Table 1.3). Some of them aim at a particular public like physicists, mathematicians, data scientists, or engineers (see Table 1.4). We denote this sub-category as NON-CS books.

The second category, that we refer as ALGO books, consists of the books addressing algorithmics or program design (see Table 1.1). They usually introduce their own pseudocode which often looks like C language very much. Sometimes, their proposes a translation of the pseudocode in another language(s).

Table 1.1: ALGO books. The “Lang” column refers to the consulted document language: ENglish or FRench. FR* means that the book is a French translation.

Title	Ref.	Lang
The B-Book. Assigning programs to meanings	[3]	EN
Concepts fondamentaux de l’informatique	[4]	FR*
The design of well-structured and correct programs	[6]	EN
Algorithmique : construction, preuve et evaluation des programmes	[25]	FR
Introduction to algorithms	[42]	EN
Introduction à l’Algorithmique I	[48]	FR
A Discipline of Programming	[56]	EN
Starting out with programming logic and design	[71]	EN
Data Structures and Algorithms in Java	[77]	EN
Mini manuel d’algorithmique et de programmation : cours + exos corrigés	[79]	FR
Informatique : algorithmes en Pascal et en langage C : rappels de cours, questions de réflexions, exercices d’entraînement	[80]	FR
Algorithmes fondamentaux et langage C : programmation : codage, alternatives, boucles, tableaux, modularité	[93]	FR
Cours et exercices corrigés d’algorithmique vérifier, tester et concevoir des programmes en les modélisant	[95]	FR
The Art of Computer Programming: Volume 3: Sorting and Searching	[106]	EN
Introduction to Programming Concepts with Case Studies in Python	[184]	EN

Table 1.2: OTHER books. The “Lang” column refers to the consulted document language: ENglish or FRench.

Title	Ref.	Lang
Structure and interpretation of computer programs	[2]	EN
The software optimization cookbook : high-performance recipes for IA-32 platforms	[74]	EN
A Guide to Experimental Algorithmics	[138]	EN
Programming language pragmatics	[169]	EN

Table 1.3: LANGUAGE books. The language is specified in the title. The “Lang” column refers to the consulted document language: ENglish or FRench. FR* means that the book is a French translation of an English book.

Title	Ref.	Lang
Le langage C	[5]	FR*
The Java programming language	[9]	EN
Méthodologie de la programmation en C : norme C 99 - API POSIX	[31]	FR
C++ pour les nuls	[47]	FR*
Programmer en langage C : cours et exercices corrigés	[50]	FR
Thinking in Java	[60]	EN
Java : l’essentiel du code et des commandes	[65]	FR*
Starting Out with Visual C#	[72]	EN
Starting Out with C++	[73]	EN
Java 6 : les fondamentaux du langage Java	[82]	FR
Programming in Lua	[91]	EN
Programming with C	[96]	EN
Le langage C norme ANSI	[100]	FR*
Programming in C	[107]	EN
Le langage C	[116]	FR
Initiation à l’algorithmique et à la programmation en C cours avec 129 exercices corrigés	[133]	FR
Modern Fortran explained	[140]	EN

Table 1.4: NON-CS books. The “Lang” column refers to the consulted document language: ENglish or FRench.

Title	Ref.	Lang
Intro to Python for Computer Science and Data Science: Learning to Program with Ai, Big Data and the Cloud	[49]	EN
Math Adventures with Python: An illustrated guide to exploring math with code	[63]	EN
Essential MATLAB for engineers and scientists	[83]	EN
Python pour le data scientist : des bases du langage au machine learning	[94]	FR
A student’s guide to Python for physical modeling	[102]	EN
Classical FORTRAN : programming for engineering and scientific applications	[111]	EN
A first course in computational physics and object-oriented programming with C	[192]	EN

Our analysis focuses on the way the books introduce loops and iterations; whether they give any advice or method to design a loop in general (see Sec. 1.5.1) and how they present algorithms. as introductory books were very likely to address array sorting (Sec. 1.5.2) and searching (Sec. 1.5.3), we focused on these problems.

We also encountered some books mentioning invariants that do not focus on programming. They form our third category, which we refer as OTHER books. They are listed in Table 1.2 and briefly discussed in Sec. 1.5.4.

Finally, Sec. 1.5.5 lists the contradictions with the rest of the literature and/or – what we believe being – mistakes we encountered during our analysis.

1.5.1 Loop Presentation and Design Method

All the books introducing the different loops do so by giving their syntax followed by an example that is then explained. The books limiting their explanations to this are always those from LANGUAGE NON-CS categories (see Tables 1.3 and 1.4).

Some of them add a flowchart that illustrates the semantic [49, 72, 73, 96].

Few propose a general methodology to write a loop. By methodology, we mean advice to follow or steps to take or whatever that could help someone that just discovered the concept of a loop to write their own :

- Deitel and Deitel [49] present a three steps method (pp. 90–91):

  1. State the requirements;
  2. Write the pseudocode for the algorithm;
  3. Code the algorithm in Python.

  In order to write the pseudocode, a "top-down step-wise refinement" (p. 93) approach is detailed: one should write "*top*", which is a single plain text statement that describes what should be done (*i.e.*, the Postcondition). Then "*top*" must be detailed through refinement steps in several simpler tasks, thus referring to a divide-and-conquer strategy, until "there is enough detail for you to convert the pseudocode to Python" (p. 95).

- Léry [116] proposes a three steps method to write a loop:

  1. Write the Loop Body by finding what must be made to go from step N to $N+1$ (p. 100);
  2. Determine the variables initial values by observing the effect of the first iteration (p. 101), and adjusting them by making the loop run by hand (p. 99);
  3. Determine the Loop Condition by posing one a priori and verify it by making the loop run (p. 101) (*e.g.*, to decide which of <= and < should be used).

- Kupferschmid [111] mentions a program design in which a step is "find appropriate data

structures and algorithms" (p. 228) and points to other books addressing these subjects. In fact, the whole chapter about "Design, Documentation and Coding Style" (chap. 12, pp. 219–264) is rather a long list of pieces of advise than a structured method.

As far as the ALGO books are concerned (see Table 1.1), they do not necessarily remind the loop structures to their readers (*e.g.*, [4, 6, 42, 95, 106]). In contrast, they often offer a programming methodology.

Dijkstra [56] introduces a methodology based on formal Loop Invariant used to deduce code instructions and on Loop Variant to ensure loop termination . Berlioux and Bizard [25], de Marneffe [48], Julliand [95] present the same method. de Marneffe [48] provides sometimes an illustration of the Loop Invariants. Julliand [95] mentions the method being "lighter than an a posteriori proof" (p. 126) and a "fundamental algorithmics concept in order to master programs design" (p. 127).

Abrial [3] introduces the B method based upon mathematical and logical reasoning, which would be harsh to follow by a CS1 student.

Cormen et al. [42], Goodrich and Tamassia [77] both presents the Divide and Conquer method as well as Greedy algorithms and Dynamic Programming. Cormen et al. [42] uses Loop Invariant to prove code correctness.

Granet [79] mentions the Loop Invariant as representing "the semantic of an iterative statement" (p.48, own trad.) but does not offer any indication on how to find one.

Imbert [93] illustrates algorithms thanks to operational semantics, that is a succession of graphical representations of the memory through the program execution. The author mentions a design method whose one of the steps is the "strategy" (p.40) to solve the problem but there is no clue to find one.

Üçoluk and Kalkan [184] introduce "Tips for creating Iterative Solutions" (p. 141) that suggest to work with a "case example where the iteration has gone through several cycles and partially built the solution" [*ibid.*] to determine "what must have been changed" and "what will remain the same" [*ibid.*] in order to deduce the code. The expression of "what will remain the same" can be identified to the Loop Invariant but the book does not refer to it. On the other hand, the tips advise to deduce the Loop Body, Loop Condition and the variables initialisations independently, as in [25, 48, 56, 95]. Finally, the tips also mentions to "Make sure that the looping will terminate for all possible task cases" [*ibid.*] but does not explain how to do so nor refer to the notion of Loop Variant that is usually used for that purpose.

Gaddis [71], Granjon [80], Knuth [106] do not offer, properly speaking, any kind of general method to write loops.

1.5.2 Sorting Algorithms

Most of the books we consulted presented at least one sorting method among the *Bubble Sort*, the *Selection Sort*, and the *Insertion Sort*. The most exhaustive ones (*e.g.*, [4, 42, 48, 77]) also

introduce more efficient algorithms such as the *Heap Sort*, the *Quick Sort*, or the *Merge Sort*. Knuth [106] provides a systematic review and comparison of sorting algorithms and is therefore often cited by newer references.

Books presenting the sorting more deeply often display a graphical representation of an array being sorted before or after having stated the idea of the algorithm, followed by the code, which is not necessarily commented nor discussed.

It is worth noting that LANGUAGE books often only describe how to invoke properly the standard functions or methods available in a particular language's library to sort arrays [5, 9, 31, 60, 82]. This is also the case for NON-CS books [49, 83, 94, 192].

1.5.3 The Binary Search Case

The binary search is said to be difficult to implement by Knuth [106] and Scott [169, p.302]. However, the idea of this algorithm is considered by Knuth [106] to be straightforward. It is worth noting that a large majority of books that present the code of this algorithm begin by establishing that the search area must be divided by two at each iteration. To our knowledge, Abrial [3] and de Marneffe [48], both using formal methods, deduce the division while writing the code.

McGeoch [138] presents the cache aware version of the binary search presented by Ladner et al. [112].

As for the sorting algorithms, the books presenting a particular language often only describe how to invoke properly the functions to search in arrays. Among them, few mention that using binary search in an array always requires to have this array sorted but do not necessarily offer a practical example in which this operation is actually efficient (sorting an array for a unique search is not efficient as a linear search does the trick more efficiently).

1.5.4 Other Mentions of Loop Invariant

Abelson and Sussman [2] addresse algorithmics illustrated with Scheme but mentions iterative methods in an exercise specifying "In general, the technique of an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms" (p.46). However, the way to find or isolate this invariant quantity is not mentioned.

Gerber et al. [74] address code optimisation and, in articular, loops in a dedicated chapter (pp. 143–157). Several techniques are presented: loop distribution, loops fusion, loop peeling (*i.e.*, moving one or more iterations outside a loop), loop unrolling (*i.e.*, combining several iterations together), and loops interchanging (*i.e.*, exchanging inside and outside loops). The books explains in which cases loops may be modified while keeping their semantic. The transformations justifications are not linked to the notion of Loop Invariant while it would have fitted to prove that a modification did not alter the result of the original loop. However, the book introduces *loop invariant computations* (p.155) (*i.e.*, results of calculation not depending on the loop), *loop*

invariant branches (p.156) (*i.e.*, tests not depending on the loop that can be put outside of it) and *loop invariant results* (p.157) (*i.e.*, loops used to initialise values that can therefore be hard-coded). Although similar in name, these three notions are different from what we mean by Loop Invariant.

McGeoch [138] addresses computational experiments on algorithms and introduces Loop Invariants to verify a loop (there may be several Loop Invariants per loop) in three questions: (i) "Are the invariants true when the loop is entered the first time?" (p. 157) (ii) "Assuming the invariants are true at the top of the loop, do they hold at the bottom of the loop" [*ibid.*] (iii) "Do the invariants imply correctness after the loops ends?" [*ibid.*] In fact, these three questions describe informally a proof based on Hoare logic (See Sec. 1.2). The book mentions that writing Loop Invariants is a "powerful insurance against common coding mistakes like off-by-1 errors, omitting initialisations, mishandling loop control variables, running of the end of arrays and so forth. It also helps to write a procedure or macro to check preconditions at runtime" [*ibid.*]. The book also presents code tuning for efficiency, especially loop modifications (fusion, unrolling, loop memory access order) but does not link that to Loop Invariants.

Scott [169] addresses programming languages design and implementation and introduces Loop Invariant as a way to prove code correctness in the context of an exercise consisting in finding it for the binary search (p. 302). The author indicates that "programmers who identify (and write down !) the invariants for their loops are more likely to write correct code" [*ibid.*] and point to [56, 81] to get insights on how to do so (p. 306).

1.5.5 Contradictions in the Literature

When reading some references, we were surprised to notice that some books stand out strongly from others in the way they exhibit content that is either in contradiction with the rest of the literature or that we believe is incorrect. This section reviews them briefly.

1.5.5.1 About Programming Methodology

Léry [116]'s loop design method is not aligned with the rest of the literature (*e.g.*, with [48]). The sooner determines the initial values and the Loop Condition thanks to the loop and thus requires the Loop Body to be determined. On the contrary, the method of the later, based upon Loop Invariant, clearly states that the initial values, the Loop Body, and the Loop Condition can be determined independently. Moreover, Léry asserts that making the loop run by hand is the only efficient way to understand its behaviour and correct it (p. 100), which is clearly in contradiction with the state of the art on Loop Invariant programming.

Davis [47, p.277] asserts that executing step-by-step a method is invaluable to understand why its behaviour is odd and that "nothing is more revealing than the step-by-step execution" [*ibid.*] of a program to understand how it works.

Farrell [63] presents an algorithm that lists the positive divisors of a number `num` exhibiting a complexity of $O(\mathtt{num})$ (pp. 39–41). The algorithm is implemented to illustrate a `for` loop. It would have been better to use the same example to present the interest of a `while` loop and to notice that if $x \mid y$ then $(y/x) \mid y$ since $x \mid y$ means there exists k such as $y = k \times x$. An algorithm using this mathematical property can list the positive divisors of a number `num` in a time that is in $O\left(\sqrt{\mathtt{num}}\right)$.

We believe that the last example delivers a counter-productive message to their readers in the sense that they do not insist on a programming methodology that begins with an accurate analysis of the problem that would prevent them to write inefficient code.

1.5.5.2 About Programming Language

Some references [63, 71, 107, 140] introduce the loops as a mean to avoid repeating the Loop Body several times. This statement is an over-simplification. Even if the number of iterations is fixed, as with a for loop, this number is often a parameter whose value is not known at the compilation time. As far as the while loop is concerned, it is particularly useful precisely to write a loop without knowing precisely the number of iterations (*e.g.*, this is the case of the binary search).

Aitken and Jones [5] assert that loops can be infinitely nested in C (p. 139), which is not in accordance with the standard that establishes minimum translation limits [180].

Braquelaire [31] presents wrongly the comma operator in C by explaining it serves to concatenate variable increments (p. 175) without delivering its correct semantics nor pointing to a reference that deepens its presentation.

Kochan [107] suggests that `++i` and `i++` are equivalent in C: "Some programmers prefer to put the `++` or `--` after the variable name, as in `n++` or `bean_counter--`. This is acceptable, and is a matter of personal preference." (p. 50). This is in contradiction with the language standard [180]: the post- and pre-increment operators do not have the same priority and the two expressions `++i` and `i++` do not have the same value. This is an example of over-simplification. If the values of the expressions is not needed and thus only the side-effects of these operators matter, `i++`, `++i`, and `i += 1` may be mistaken from each other. Semantically speaking, the closest expressions to `++i` is `(i += 1)` or `(i = i + 1)` (the parentheses are mandatory).

Granjon [80] presents the C `for` loop (*i.e.*, `for(i = 1; i < y; i++)`) and states that `i` is an iteration variable that cannot be modified in the Loop Body (p. 27). First, semantically speaking, `i++` is in the Loop Body as the last instruction prior to re-evaluating the Loop Condition. Second, the C standard [180] does not establish such a constrain. We believe that the author meant that the value should not be modified elsewhere in the Loop Body to count accurately the iterations.

Imbert [93] defines a pointer as a variable that designates an object in a dynamic collection (p. 152) but acknowledges that this definition is more general than the ones met in programming languages (p. 153). He states that an array index thus falls in the definition of a pointer while the

C language [180] makes clearly the difference: a pointer is a variable that contains an address while an array index is an integer indicating a relative position in an array. We do not know what the author meant but we believe that as far as learning C language is concerned, mistaking the two notions for each other is harmful and would lead to confusion for the reader.

1.6 Conclusion and Research Questions Introduction

This chapter presented the interest of Loop Invariant in the contexts of program proofs and of a constructive approach to deduce the program code. Most of time, the Loop Invariant is expressed formally and this can discourage teaching it in the context of a CS1 course to students who may lack of mathematical or logical skills.

We believe this is why the programming books introducing a particular language and those intended for non computer science audience rarely mention it. Yet paradoxically, one could expect from those references to provide a methodology for programming or at least to point to other work doing so. This is not what we saw in our review of the available books at our university's library.

We wonder what could someone learn from reading a book that just discusses the syntactical aspects of a language and lists some interesting functions available in its standard library (and that, aside the over-simplifications and mistakes we noticed).

To overcome the (potential) lack of formal background while ensuring students follow a strict programming methodology, we propose propose our own programming methodology, based on a graphical representation of the Loop Invariant. We call it the Graphical Loop Invariant Based Programming (GLIBP) methodology that we introduce in details in Chap. 2.

Although informal, this Graphical Loop Invariant describes, at least, variables, constant(s), and data structures manipulated by the program; the constraints on them; the relationships they may share, and that are preserved all over the iterations. It also expresses, in a general way, what has been already computed by the program after a certain number of loop iterations. Following Furia et al. [70] classification, the Graphical Loop Invariant falls within the scope of essential (*i.e.*, a Loop Invariant defining what has already been achieved so far) and bounding Loop Invariant (*i.e.*, variables are bounded by, *e.g.*, an array limits).

In addition to natural advantages of drawings [75, 149, 155], the Graphical Loop Invariant allows the programmer to visually deduce instructions before, during, and after the loop.

In addition, in terms of learning, the Graphical Loop Invariant relies on Cognitive Load Theory as engaging multiple modalities (in this case, visual and textual) can increase cognitive ability and learning [178]. Further, the Graphical Loop Invariant may help developing Spatial Skills for students. It has been demonstrated that Spatial Skills are important for success in certain Computer Sciences activities and for success in Introductory Computer Sciences courses in particular [152, 174].

Finally, programming with the help of the Graphical Loop Invariant falls within the scope of

metacognition [141], as it provides a problem-solving strategy and self-reflection on where one is in the problem-solving process. As such, the Graphical Loop Invariant based programming can be related to three problem-solving stages introduced by Loksa et al. [130], i.e., search for solution, evaluate a potential solution, and implement a solution. Also, determining a Graphical Loop Invariant prior to coding should help students in understanding the problem to be solved [44] as well as determining a strategy to solve it [162].

The use of the GLIBP in the context of teaching in CS11 leaded us to formulate the following research questions:

RQ 1.1: *How the students seize the opportunity to practice the* GLIBP *methodology?*

RQ 1.2: *What kind of error are committed using the* GLIBP *methodology?*

RQ 1.3: *Can we link the error committed with the* GLIBP *methodology to programming errors?*

RQ 1.4: *How the* GLIBP *methodo is perceived by the students?*

RQ 1.5: *Does the* GLIBP *methodology enable to write better pieces of code?*

The rest of this document part is the following: Chap. 2 presents in details the programming methodology we propose that is based on a graphical version of the Loop Invariant; Chap. 3 introduce learning tools supporting our methodology teaching; Chap. 5 evaluates the Blank Graphical Loop Invariant, which is one of our learning tools and Chap. 4 presents a taxonomy of errors committed while using the methodology and relates them to code quality.

CHAPTER 2

GRAPHICAL LOOP INVARIANT BASED PROGRAMMING PRINCIPLES

THIS CHAPTER introduces the Graphical Loop Invariant Based Programming methodology whose purpose is to write correct pieces of code based on a drawing. Sec. 2.1 first presents the methodology with an example on a simple problem; second it deepens the presentation with guidelines that formalise the method and eventually show how to derive the code from a Graphical Loop Invariant. Section 2.2 discusses how to represent common data structures in our methodology framework. Finally, Sec. 2.3 compares our Graphical Loop Invariant with Formal ones and those written in natural language.

2.1 Graphical Loop Invariant Based Programming Methodology

2.1.1 A Graphical and Informal Version of the Loop Invariant

Dijkstra [56] proposed to first determine the Loop Invariant and then use it to deduce the code instructions. The Loop Invariant is therefore used "a priori". The methodology we propose differs from Dijkstra's in the way we express the Loop Invariant: we propose to represent it as a picture. This picture must represent the variables, constants and data structures that will appear in the code, as well as the constraints on them; the relationships they may share, and that are conserved all over the iterations. Here is an example with the simple problem defined hereafter:

> INTEGERSPRODUCT:
>
> **Input** – Two integers $\mathtt{a}$ and $\mathtt{b}$ such as $\mathtt{a} < \mathtt{b}$
>
> **Output** – the product of all the integers in $[\mathtt{a}, \mathtt{b}]$, (*i.e.*, $\prod_{i=\mathtt{a}}^{\mathtt{b}} i$)

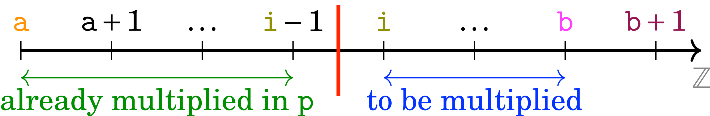


Figure 2.1: Graphical Loop Invariant for computing the product of integers between a and b. In the following, every drawing will follow the same color code: the name of the structure in **grey**; the *Dividing Line* in **red**: the Dividing Line label in **olive**; the minimal value for that label in **orange**; its maximal value in **magenta** and the value following the maximum in **wine**. Both areas will always be in **green** for what has been achieved and **blue** for what should still be done.

Fig. 2.1 shows the corresponding Graphical Loop Invariant. We first represent the integers between the boundaries of the problem (a and b) thanks to a number line (each tick mark denotes an integer) representing the integer line, labelled with the integers symbol ($\mathbb{Z}$). Then, since all the integers between a and b are going to be considered by the program, we represent the situation after a certain number of iterations. Thus, a vertical red bar is drawn in the middle of the integer line, dividing it into two areas (such a line is called Dividing Line). The left area, in green, represents the integers that were already multiplied in a variable p (p is thus the accumulator storing intermediate results, iteration after iteration). The right area, in blue, represents the integers that have yet to be multiplied. We decide to label the nearest integer at the right of the dividing bar with the variable i that will play the role of the iterator variable in the range [a, b]. Of course, the variables i and p must be used in the code.

Drawing such a Graphical Loop Invariant amounts to determining a strategy to solve the problem. We clearly shift the difficulty not anymore in writing the code itself but in the reflection phase prior to the code. This step requires thus training and experience. But, once mastered, it becomes possible to efficiently solve complex problem. Moreover, in the following, we also provide a methodology for easing the building of a correct Graphical Loop Invariant that relies on seven guidelines (see Sec. 2.1.2). As to how to deduct the code instructions from a Graphical Loop Invariant, the section 2.1.3 details it with the example of the Graphical Loop Invariant depicted in Fig. 2.1.

2.1.1.1 Graphical Loop Invariant Interpretation

In Chap. 1, we defined a Loop Invariant as a logical properties being True before the Loop Condition evaluation. In fact, the Graphical Loop Invariant shares this property with its non-graphical counterpart. Hence, we must have a way of deciding whether a Graphical Loop Invariant is True or False for a given combination of variable values (*i.e.*, a certain point in the states space). This can be done by examining if the general shape of the Graphical Loop Invariant is conserved as particular values are assigned to the variables, *i.e.*, the relationships between the variables described by the drawing still hold in the particular case.

The Table 2.1 presents several examples based on the Graphical Loop Invariant shown in

Fig. 2.1. Each line in the Table represents a particular case. The first column gives the values of the variables for each state. The second column depicts the corresponding situation, with the same colours code as in Fig. 2.1 to ease the comparison. Finally, the third column indicates whether the Graphical Loop Invariant is True or False and provides additional explanations. To sum up, deciding if a Graphical Loop Invariant is True or False in a particular situation consists in redrawing this situation and playing a kind of "*Spot the difference*" game that requires more observation than mathematical skills.

2.1.2 Carefully Depicting a Graphical Loop Invariant

Since the Graphical Loop Invariant is intended for deducing the code instructions, one should have a mean to assess the general quality of such an Loop Invariant prior to code writing. The problem that arises is the following:

> Given a particular problem,
>
> - What kind of information must be represented and how?
> - When can one stop drawing to write the code? *i.e.*, Is a particular drawing *sufficient* to rely on it to deduce code instructions?

Obviously, as the Graphical Loop Invariant Based Programming methodology relies on informal drawing, one cannot claim to bring a formal decision procedure to answer these questions. Instead, we propose seven guidelines (or rules) that should be followed by any drawings to be used as Graphical Loop Invariant to deduce code instructions. There are listed hereafter:

Guidelines for Depicting a Graphical Loop Invariant

Rule 1: The drawing shall correspond to the problem and be labelled;

Rule 2: The boundaries of the problem shall be provided;

Rule 3: One (or more) dividing line(s) shall be provided;

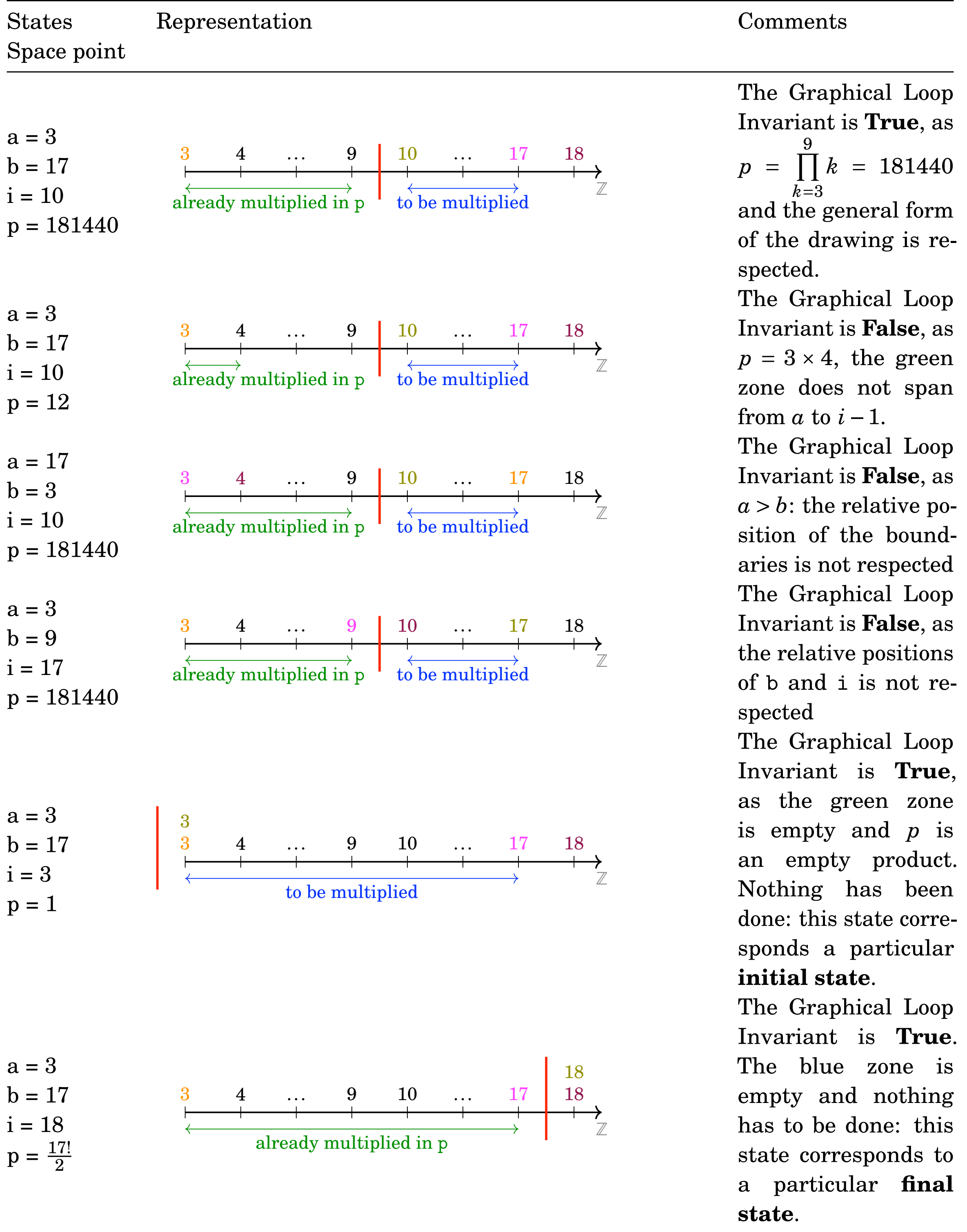
Rule 4: Each dividing line shall be properly labelled;

Rule 5: The drawing shall be labelled for explaining what has been achieved so far;

Rule 6: The drawing shall be labelled to indicate what should still be done;

Rule 7: All the named structures and variables shall be present in the code.

These drawing guidelines can be classified into two categories: the syntax rules and the semantic ones. Rules 1 to 4 are syntax rules,*i.e.*, they provide a framework for the drawing: they determine how to represent the graphical elements that form the Graphical Loop Invariant.



States Space point	Representation	Comments
a = 3 b = 17 i = 10 p = 181440	3 4 … 9 | 10 … 17 18 ℤ already multiplied in p · to be multiplied	The Graphical Loop Invariant is **True**, as $p = \prod_{k=3}^{9} k = 181440$ and the general form of the drawing is respected.
a = 3 b = 17 i = 10 p = 12	3 4 … 9 | 10 … 17 18 ℤ already multiplied in p · to be multiplied	The Graphical Loop Invariant is **False**, as $p = 3 \times 4$, the green zone does not span from a to $i-1$.
a = 17 b = 3 i = 10 p = 181440	3 4 … 9 | 10 … 17 18 ℤ already multiplied in p · to be multiplied	The Graphical Loop Invariant is **False**, as $a > b$: the relative position of the boundaries is not respected
a = 3 b = 9 i = 17 p = 181440	3 4 … 9 | 10 … 17 18 ℤ already multiplied in p · to be multiplied	The Graphical Loop Invariant is **False**, as the relative positions of b and i is not respected
a = 3 b = 17 i = 3 p = 1	| 3 3 4 … 9 10 … 17 18 ℤ to be multiplied	The Graphical Loop Invariant is **True**, as the green zone is empty and p is an empty product. Nothing has been done: this state corresponds a particular **initial state**.
a = 3 b = 17 i = 18 p = $\frac{17!}{2}$	18 3 4 … 9 10 … 17 | 18 ℤ already multiplied in p	The Graphical Loop Invariant is **True**. The blue zone is empty and nothing has to be done: this state corresponds to a particular **final state**.

Table 2.1: Evaluation of the Graphical Loop Invariant shown in Figure 2.1 for several states space points. The color code of the Graphical Loop Invariant is respected : a is in orange, b in magenta, b + 1 in wine and i in olive.

On the other hand, Rules 5 and 6 are related to the semantic, *i.e.*, the meaning of the drawing: they determine what is represented, aligned with the particular problem to solve.

However, one should note that some of the syntax rules also contain semantic-related guidelines. For example, adding the boundaries actually increases the information contained in the picture. In fact, the graphical elements of the Loop Invariant, that are required by the syntax rules, provide meaning just by their presence in the drawing.

Of course, the seventh rule is not a drawing guideline but ensure a strong link between the Graphical Loop Invariant and the code that is the core and purpose of our methodology.

In addition to using these guidelines as a reminder to help to find a Graphical Loop Invariant, they enable to assess a particular drawing. For this purpose, we have also proposed an error taxonomy that is presented in Chap. 4. In the following, we detail each rule and provide explanations on its raison d'être in line with our methodology.

2.1.2.1 Rule 1 – The drawing shall correspond to the problem and be labelled

The first part of this guideline may appear as obvious. It recommends to draw an accurate representation of the data or the data structures that are concerned by the particular problem to be solved. It also reminds to carefully think about the data that must be processed by the program. Hence this guideline emphases on the correct definition of the problem inputs and their types, as the way to represent them will depend on this last piece of information.

The second part of the guideline – to label the drawing – consists in writing the name of any depicted data structure next to it (most of time, it is a variable or constant name). One can see why it is essential if several data structures are handled by the program, as they could be mistaken during the code writing. However, it is always useful to accurately label data structures, even the single ones. In order to use the Graphical Loop Invariant to deduce the code instructions, one must think on the basis of a drawing that may contain a lot of information. Any annotation in the Graphical Loop Invariant could refer to a particular data structure that must therefore be clearly identifiable to insure the consistency of the whole drawing.

This corresponds to the integer line, labelled $\mathbb{Z}$ in Fig. 2.1.

2.1.2.2 Rule 2 – The boundaries of the problem shall be provided

The reason of the existence of this guideline is straightforward. First, it helps representing the limit of the problem to be solved and reminds to determines these limit before code writing. This requires to have previously clearly stated the goal of the program.

Second, when the Graphical Loop Invariant is used to deduce code instructions, it prevents some common mistakes such as array out of bound errors or overflow. These errors should indeed be more unlikely if the length of the data structures are properly mentioned in the Graphical Loop Invariant.

This corresponds to a and b in Fig. 2.1. We also drew $\mathsf{b}+1$ that must not be considered in the

problem to avoid coding mistakes.

2.1.2.3 Rule 3 – One (or more) Dividing Line(s) shall be provided

The *Dividing Lines* are the core of the Graphical Loop Invariant Based Programming methodology. They symbolize the division between what was already computed by the program and what should still be done to reach the program objective. They enable to graphically manipulate the drawing in order to deduce the code instructions, as it is further illustrated in Sec. 2.1.3.

This corresponds to the red line in Fig. 2.1.

2.1.2.4 Rule 4 – Each Dividing Line shall be properly labelled

Since a Dividing Line separates what has been done and what is going to be, that means that if we depicted such a line on the data representation as the program is executed, this line would move from a position to another: the first position would correspond to the initial state while the last position would correspond to the final one. As the position of the line during the code execution is moving, the Dividing Line must be labelled with a variable name (or an expression involving at least one variable). This guideline thus can help to declare new variables that are needed in the code, such as, for example, array indexes. These new variables are often iteration variables: they are used to span the range of interest delimited by the problem boundaries (See Rule 2).

This corresponds to the variable i in Fig. 2.1.

2.1.2.5 Rule 5 – The drawing shall be labelled for explaining what has been achieved so far

Once again, this guideline holds several purposes. On one hand, it helps thinking about the behaviour of the program. In order to determine "what has been achieved so far", one should ask the question:

> In order to reach the program goal, what should have been computed until now?
> Which variable properties must be ensured?

Most of the time, this reflection phase highlights either the need for additional variables that contain partial results or relationships between variables that must be conserved throughout the code execution. On the other hand, the information about what has been achieved so far is crucial during the code writing as it helps to decide what are the instructions to be performed during an iteration, *i.e.*, to deduce the *Loop Body*.

This corresponds to the green arrow in Fig. 2.1.

2.1.2.6 Rule 6 – The drawing shall be labelled to indicate what should still be done

This may perhaps appear as the less important guideline as it does not bring additional information in the Graphical Loop Invariant. In fact, if we expressed a Graphical Loop Invariant as a formal one, there would be no logical notation to describe "what should still be done!" Nevertheless, drawing an area indicating what should still be done (from now on, let us call it the "*to do area*") is a good way to ease the representation of the initial and final states of the program. In the initial state, this "to do area" should span over all the data that are concerned by the program. On the contrary, In the final state, this area should have disappeared while the only remaining area represents what has been achieved by the program. It is then easy to check if the purpose of the program is met in such a state.

Moreover, when deducing the code instructions, this "to do area" helps to deduce the updates of the variables labelling the Dividing Lines, since the lines have to be moved in order to shrink the area.

At last, one should note that this rule is of high importance when proposing a Loop Variant to show loop termination as the size of the "to do area" is often a good candidate for the Loop Variant (or a good hint to find a proper one).

This corresponds to the blue arrow in Fig. 2.1.

2.1.2.7 Rule 7 – All the named structures and variables shall be present in the code

One one hand, the most straightforward meaning of this guidelines is to actually use the Graphical Loop Invariant to deduce the code instructions, as shown in Sec. 2.1.3. On the other hand, this is also a reminder to check if all the variables identified during the reflection phase which is the drawing of the Graphical Loop Invariant were actually included in the code instructions.

2.1.3 Using the Graphical Loop Invariant to deduce the code instructions

2.1.3.1 Introduction

Once a Graphical Loop Invariant meeting the seven guidelines previously introduced is drawn, it can be used to write the loop instructions[10]. The general pattern of such a loop and the code locations where the Loop Invariant must be True is reminded in the Listing 2.1. Since a Loop Invariant must be True just before the evaluation of the Loop Condition, it is obviously the case at line 3. The evaluation of the Loop Condition is not supposed to modify the truth value of the Loop Invariant[11] hence the Loop Invariant is also True at lines 6 and 10. Finally, it is up to the programmer to make sure that the Loop Invariant is True at line 8, at the end of the iteration,

[10] In this chapter, we focus of this particular usage of the Graphical Loop Invariant. Of course, it can be used to prove the partial correctness of the code, or to explain how a loop works but it is important to note that, in our approach, the code is not written yet! And the Graphical Loop Invariant is needed to do so.

[11] To make it simple, we do not consider here side effect expressions, *e.g.*, pre- or post-increment.

just before the Loop Condition is evaluated, before the potential next iteration.

```
// Pre
ZONE 1
// Inv
while(LOOP CONDITION)
{
   // Inv ∧ loop_condition
   ZONE 2
   // Inv
}
// Inv ∧ ¬loop_condition
ZONE 3
// Post
```

Listing 2.1: Pattern of a loop

One can see in the pattern four parts that must be completed: ZONE 1, ZONE 2, ZONE 3, and `LOOP CONDITION`. By replacing each part by the proper expression or instruction(s), we will form the code. It is worth noting that deducing the replacement of each part can be done independently, and this, with the help of the Graphical Loop Invariant.

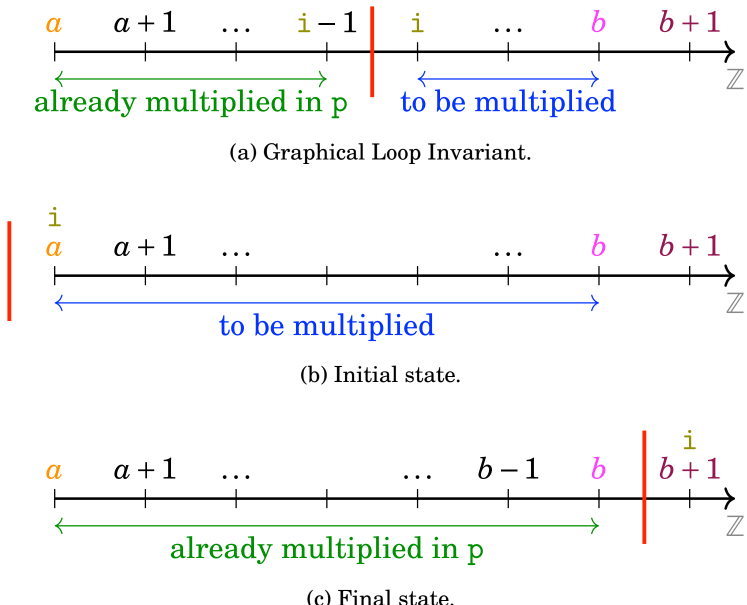
To be precise, each part is surrounded, in the pattern, by two commentaries that represent conditions that must be satisfied, *i.e.*, be True (*e.g.*, in Listing 2.1, ZONE 1 (line 2) is surrounded by *Pre* (line 1) and *Inv* (line 3)). While writing the code of a particular part, we must take for granted the information contained in the condition that precedes it and find instructions that will ensure that the condition that follows it is True. The following details theses four steps :

1. Deducing variables initialisation (ZONE 1) from the drawing of the initial state;
2. Deducing the Loop Condition from the final state ;
3. Deducing the Loop Body (ZONE 2) from the Graphical Loop Invariant;
4. Deducing the instructions coming after the loop (ZONE 3) from the final state ;

These four steps can be followed in any order, except that determining the Loop Body may require to know the Loop Condition. Both initial and final states are obtained from the Graphical Loop Invariant through graphical modifications that are detailed below.

2.1.3.2 Deducing variables initialisations from the drawing of the initial state

The initial values of the variables can be read in the drawing of the initial state (See Fig. 2.2b). This picture is obtained from the Graphical Loop Invariant (See Fig. 2.2a) by shifting the Dividing Line (in red) to the left in order to make the green zone disappear. Of course, the variable labelling the Dividing Line (`i`) is shifted to the left accordingly and stay at the right of the Dividing Line



(a) Graphical Loop Invariant.

(b) Initial state.

(c) Final state.

Figure 2.2: Graphical Loop Invariant and particular cases for computing the product of integers between a and b.

(otherwise the depicted situation would not respect the Graphical Loop Invariant anymore! – See Sec. 2.1.1.1). Doing so, we can see in Fig. 2.2b that the initial value of i must be a. As far as the variable p is concerned, we know from the Graphical Loop Invariant (Fig. 2.2a) that its value corresponds to the product of the integers present between a and the left-side of the Dividing Line. As this zone is empty, we deduce the initial value of p is the empty product, *i.e.*, 1. The Listing 2.2 sums up the deduced instructions.

```
int i = a;
int p = 1;
```

Listing 2.2: ZONE 1

[12] By writing this, we do not agree with Dijkstra [56] who recommended to "choose [the] guards as weak as possible" (p.57). In this case, that would mean to use $i \neq b+1$ instead of $i \leq b$. Dijkstra mention two reasons: 1) using $i \neq b+1$ enables to conclude that $i = b+1$ "upon termination without an appeal to the [Loop Invariant]" (p.56); 2) it "makes termination dependent upon (part of) the [Loop Invariant *i.e.*, here, $i \leq b+1$] and is therefore to be preferred for reasons of robustness." (ibid.). On the contrary, a stronger guard enables 1) to represent the relationship between the iteration variable and the boundaries, helping to reduce errors such s overflow; 2) it is safer against infinite loop, regardless how the iteration variable is modified in the Loop Body; 3) it gives an hint on how the iteration variable should be modified to meet the Stop Condition. We think these reasons advocate for using strong guards, in particular in *CS1* course. From a computer security point of view, using "robust loop termination conditions" is the MSC21-C recommendation of the SEI CERT C Coding Standard [177] .

2.1.3.3 Deducing the Loop Condition from the final state

Determining the Loop Condition requires to draw the final state of the loop. *I.e.*, a state in which the goal of the loop is reached. Since the purpose of our problem is to compute the product of the integers between a and b, we can obtain such a representation from the Graphical Loop Invariant (Fig. 2.2a) by shifting the Dividing Line (in red) to the right. As in the previous point, the labelling variable i at the same time as the Dividing Line. This graphical manipulation leads to Fig. 2.2c. From this Figure, one can read that the goal of the loop is reached when $i = b+1$ and the iterations must thus be stopped. The Stop Condition of the loop is therefore $i = b+1$. The Loop Condition, being the logical negation of the Stop Condition, is then $i \neq b+1$. We recommand[12], to properly illustrate the relation between i and b, to use a stronger condition that is $i < b+1$ or $i \leq b$ that is, of course, equivalent. The listing 2.3 shows the corresponding piece of code.

```
1 while(i <= b)
```

Listing 2.3: LOOP CONDITION

2.1.3.4 Deducing the instructions coming after the loop from the final state

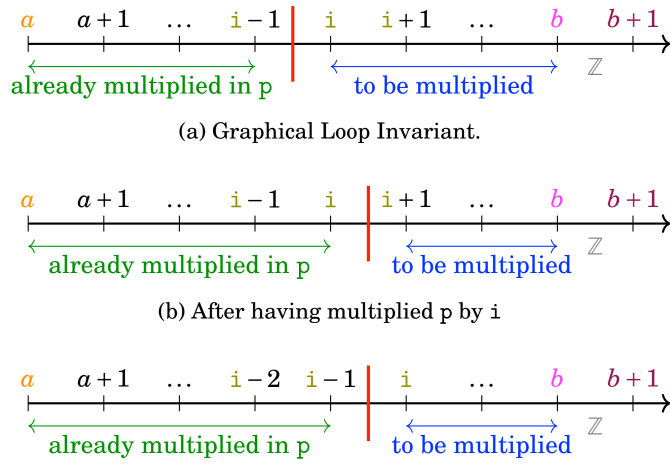
As we just represented the final state (See Fig. 2.2c), we can observe that the variable p contains the product of the integers a and b that is the goal of the program. Thus, there is nothing to do after the loop.

Please note that ZONE 3 is not necessarily empty: *e.g.*, think about a program that compute an average of a certain numbers of values. In this case, the loop would sum and count the values; ZONE 3 would be the division of the sum by the number of counted values.

2.1.3.5 Deducing the Loop Bodyfrom the Graphical Loop Invariant

Determining the Loop Body is often the most difficult step. We start from what we know: both the Loop Condition and the Graphical Loop Invariant are True (See the general loop pattern in Listing 2.1). To be more explicit, we use a new representation of the Graphical Loop Invariant (See Fig. 2.3a). We must find instructions that will make progress the situations towards the goal of the program. In other words, make the green zone increase and make the blue zone decrease. As the green zone represents the integers that are multiplied in p (thus from a to i-1), we can make grow this zone by multiplying the next integer to p. This next integer is read in the Graphical Loop Invariant at the right of the Dividing Line: this is i.

After having multiplied p by i, the situation in Fig. 2.3b is obtained. It must be noted that is not the Graphical Loop Invariant any more since the variable i is now at the left of the Dividing Line. In this particular situation, the Graphical Loop Invariant is False (see Sec. 2.1.1.1), whatever the particular values of a,b,i, or p! According to the loop pattern (See listing 2.1), we



(a) Graphical Loop Invariant.

(b) After having multiplied p by i

(c) Restored Graphical Loop Invariant.

Figure 2.3: Determining the loop body from the Graphical Loop Invariant.

must *restore* the Graphical Loop Invariant, *i.e.*, make it true again, before the end of the Loop Body. How can this be done? By comparing the Figures 2.3a and 2.3b, we can see that in the Graphical Loop Invariant, the value labelling the right side of the Dividing Line is `i` and in the current situation, this is `i+1`. Therefore, by assigning the value `i+1` to `i` (*i.e.*, increasing `i`), the Graphical Loop Invariant is restored (See Fig. 2.3c, that is strictly equivalent to Fig. 2.3a). Finally, Listing 2.4 shows the loop body instructions.

```
{
   p *= i;
   ++i;
}
```

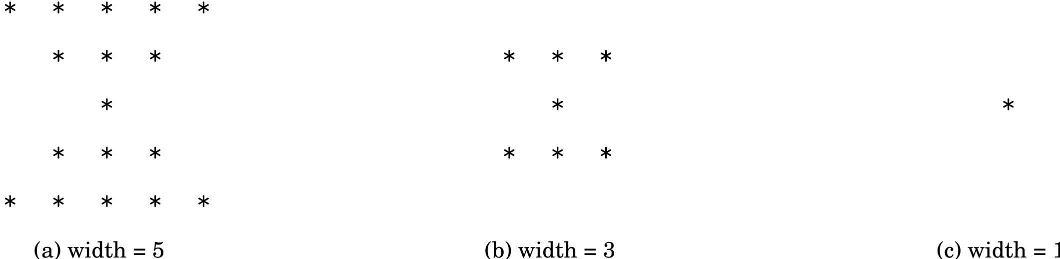
Listing 2.4: ZONE 2

2.1.3.6 Final code and Loop Variant

```
int i = a;
int p = 1;
while(i <= b)
{
   p *= i;
   ++i;
}
// ZONE 3: Nothing to do
```

Listing 2.5: Final Code

The final code is given in Listing 2.5. As far as the Loop Variant is concerned, it can also be obtained from the Graphical Loop Invariant: here, this is the expression of the length of the



(a) width = 5 (b) width = 3 (c) width = 1

Figure 2.4: Examples of hourglasses with different widths.

blue arrow, *i.e.*, $b-i+1$. It is obvious that this quantity is an integer, strictly positive if the loop condition is true and that decreases from an iteration to another.

2.1.4 Positioning the Graphical Loop Invariant with Respect to Problem Solving

[Nouveau]

[to be moved elsewhere] One of the limit of the Graphical Loop Invariant Based Programmingmethodology is the capacity of its user to represent graphically problems. As long as they figures how to illustrate a problem with a drawing,**[(insert this)]**

In the previous sections, we introduced the Graphical Loop Invariant Based Programming methodology and illustrated it with an example involving a simple problem. As for now, we have not yet discussed important questions :

- How can we know that loops are needed to solve a problem?
- And if so, how many of them?

These questions do not belong, strictly speaking, to the Graphical Loop Invariant Based Programming methodology, which focuses on writing the loops. However, they will be answered thanks to a preliminary analysis of the problem to solve. As for the GLIBP methodology, we promote to make a drawing of the problem and to leverage this picture to make arise possible sub-problems, that are easier to solve (which is referred as applying a divide and conquer strategy), and to identify, for each sub-problem, if a loop is required to solve it.

Let us take the example of a problem consisting in drawing an hourglass shape:

> HOURGLASS:
>
> **Input** – An odd positive integer `width`.
>
> **Output** – A hourglass-shaped text composed of asterisks ($*$) whose base length is `width`.

Fig. 2.4 shows several examples of hourglasses.

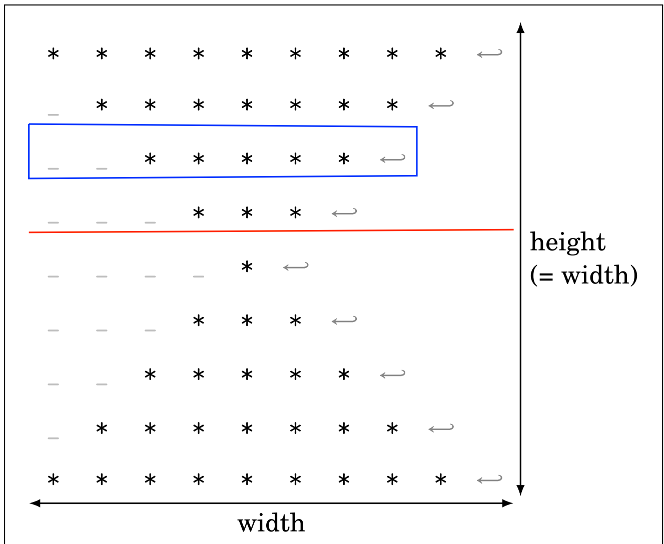


Figure 2.5: Using a drawing to make arise simpler sub-problems

2.1.4.1 Dividing the Problem into simpler Sub-problems

In order to analyse the HOURGLASS problem, we begin with an accurate drawing of an hourglass (here of width equal to 9) that can be seen in Fig. 2.5. We call this figure "accurate" because it shows all the details, including the white-space characters (spaces and newlines, in gray in Fig. 2.5) required to obtain an hourglass when printing on standard output.

Then, by looking at the figure, we can recognise repeating patterns. As can be seen in Fig. 2.5, a pattern consisting of spaces followed by stars and a newline (squared in blue in Fig. 2.5) repeat itself several times. How many times? Let us call this quantity the **height** of the hourglass. The height of the hourglass is equal to its width but the following will show that we do not need to know that to solve the problem.

From Fig. 2.5, we also note that the hourglass shape can be divided into two simpler parts: a top trapezoid and a bottom triangle, as it is depicted in Fig. 2.5 thanks to an horizontal red line. In the top trapezoid, the "spaces/stars/newline" patterns evolve from top to bottom by increasing the number of spaces and decreasing the numbers of stars. On the other hand, in the triangle, the evolution of the "spaces/stars/newline" patterns is mirrored: the spaces decrease while the stars increase.

To sum up, we identified three sub-problems (SP):

SP1: Print a “spaces/stars/newline” pattern;

SP2: Print a trapezoid;

SP3: Print a triangle.

We can even divide the SP1 into smaller sub-problem calls, since printing a "spaces/stars/new-

line" pattern consists in printing a certain number of spaces, followed by a certain number of stars, followed by a newline. We denote this sub-problem as SP4:

SP4: Print a character a certain number of times.

As a result, SP1 will use SP4 several times; SP2 and SP3 will both use SP1 and the HOURGLASS problem will consist in applying SP2 followed by SP3, as shown in Listing 2.6.

```
void hourglass(int width){
   width = (width % 2)? width : width + 1; // Handle even width
   printTrapezoid(width); // SP2
   printTriangle(width);  // SP3
}
```

Listing 2.6: Code for the HOURGLASS problem

For the following, let us assume that we have already solved the SP4 (which is quite simple and require a loop to print several times the same character) and that we have a function PRINTC that enable us to print a character several times:

> PRINTC:
>
> **Input** – A character c and a positive integer nb
>
> **Output** – Print nb times the character c.

Thanks to PRINTC, implementing SP1 is straightforward since it consists of three calls to the function, as it can be seen in Listing 2.7.

```
void printSpacesStarsNL(unsigned int nbSpaces, unsigned int nbStars){
   printc(nbSpaces, ' '); // SP4
   printc(nbStars , '*'); // SP4
   printc(1       , '\n');// SP4
}
```

Listing 2.7: Code for the SP1 – Print a “spaces/stars/newline” pattern

2.1.4.2 Solving SP2: print a trapezoid

Let us focus on the top part of the Fig. 2.5, that forms the trapezoid. The first line of the trapezoid consists of width stars while the last one consists of 3 stars. Printing the lines of the trapezoid require to repeat the SP1 several time. To do so, a loop is needed. From the top of the Fig. 2.5, let us draw a Dividing Line that separate the lines already printed and the remaining ones. By doing so, we get the Graphical Loop Invariant of the SP2, shown in Fig. 2.6.

The Graphical Loop Invariant helps us to identify variables: the number of stars nbStars, which will be the iteration variable. One also could have used the number of spaces nbSpaces as

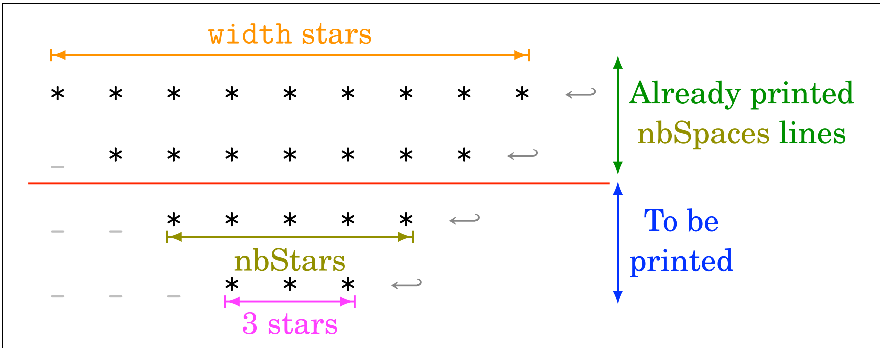


Figure 2.6: Graphical Loop Invariant for the top triangle of the hourglass

```
1  void printTrapezoid(unsigned int width){
2     // Parameter verification is omitted.
3
4     unsigned int nbStars = width;
5     unsigned int nbSpaces = 0;
6
7     while(nbStars >= 3){
8        printSpacesStarsNL(nbStars, nbSpaces); // SP1
9        nbStars -= 2;
10       nbSpaces += 1;
11    }
12 }
```

Listing 2.8: Code for the SP2 – print a trapezoid

iteration variable. The number of spaces in the next line to be printed is equal to the number of lines already printed.

The initial values of the variables is easily calculated: `nbSpaces` is the number of printed lines, hence 0; `nbStars` is the number of stars in the first line to be printed, hence `width`.

The loop has to be stopped when the number of stars is less than 3 and the Loop Condition is therefore `nbStars >= 3`. It is worth noting that if the hourglass `width` is 1 (see Fig. 2.4c), the Loop Condition is false before the first iteration and the loop is not executed.

After having printed the line (thanks to SP1), we restore the Graphical Loop Invariant by subtracting 2 to the numbers of stars and by incrementing the number of printed lines (*i.e.*, `nbSpaces`).

The code corresponding to the SP2 is shown in Listing 2.8. To solve the SP3: printing a triangle, one could draw a Graphical Loop Invariant with a triangular shape and follow a similar approach. The code of the SP3 is given in Listing 2.9 without further development.

2.1.4.3 Another possible problem division

Another cut in the hourglass One could have cut the hourglass in such a way that it consisted of a top triangle and a bottom trapezoid, as shown in Fig. 2.7. The program would look like to

```
void printTriangle(unsigned int width){
   // Parameter verification is omitted.

   nbStars = 1;
   nbSpaces = width / 2;
   while(nbStars <= width){
      printSpacesStarsNL(nbStars, nbSpaces); // SP1
      nbStars += 2;
      nbSpaces -=1;
   }
}
```

Listing 2.9: Code for the SP3 – print a triangle

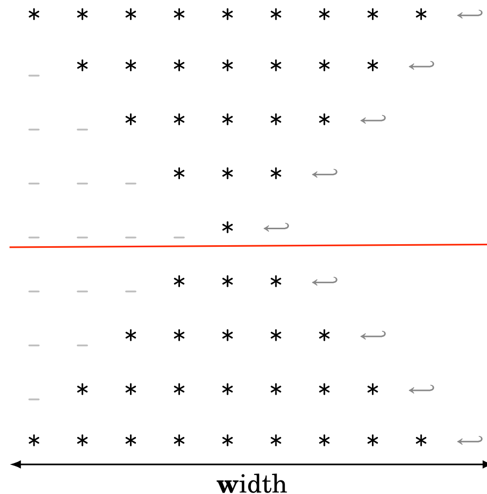


Figure 2.7: A different division of the hourglass

the one already presented in Listing 2.8. However, since the numbers of stars of the triangle would range from `width` to 1 by decreasing their number by 2 at each iteration, there is a risk of infinite loop due to an underflow if the Loop Condition is not carefully chosen (*e.g.*, if one blindly replace the loop condition by `nbStars >= 1`).

A one-size-fit-all loop This solution would consist in using a single loop to solve the problem. The following illustrates why we do not recommand doing so. A Graphical Loop Invariant for such a single loop would look like Fig. 2.8. Two iterations variables were introduced: `i` and `j` that represent the current row and column, respectively. At each iteration, a character is printed depending on the values of `i` and `j`.

Finding the conditions upon which a certain character must be printed can once again be eased by a drawing. Fig. 2.9 shows an hourglass annotated with four diagonal segments and their respective equations. Those segments and the horizontal line divide the drawing in six areas: three at the top and three at the bottom. Each of the three correspond to a character among the

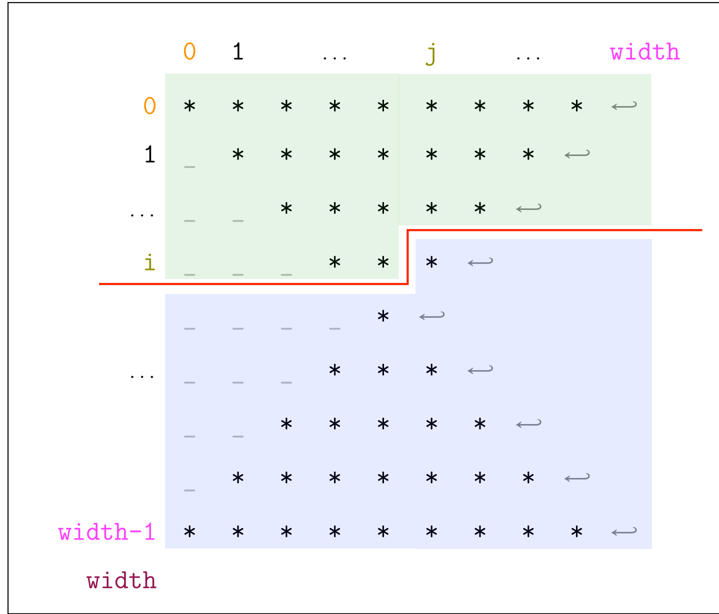


Figure 2.8: A Graphical Loop Invariant for the HOURGLASS problem solved with a single loop.

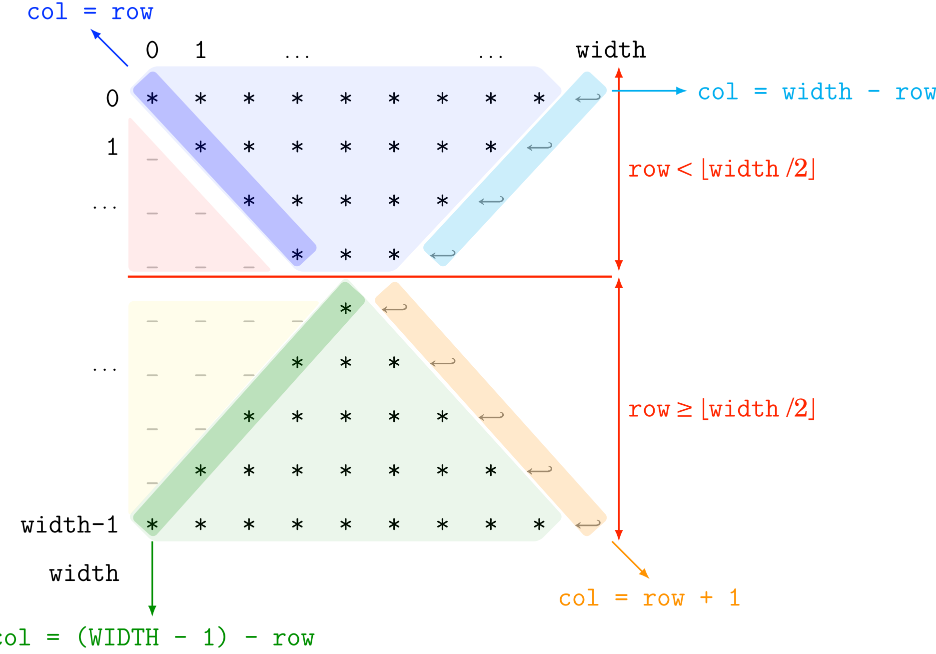


Figure 2.9: Relation between row and column positions and characters in an hourglass

```
void hourglass(unsigned int width){
   // Parameter verification is omitted.

   unsigned int i = 0;
   unsigned int j = 0;

   while(j < width){

      if (j <= width / 2)
         // Top part
         if (i < j)
            printc(1, ' '); // Red area
         else
            printc(1, '*'); // Blue area
      else
         // Bottom part
         if (i < (width - 1) - j)
            printc(1, ' '); // Yellow area
         else
            printc(1, '*'); // Green area

      ++i;
      if (((j < width / 2) && (i == width - j)) // Cyan line
          || ((j >= width / 2) && (i-1 == j)))  // Orange line
      {
         printc(1, '\n');
         i = 0;
         j++;
      }
   }
}
```

Listing 2.10: Code for the HOURGLASS problem with a single loop. The comments in the loop body refer to Fig. 2.9.

space, the star and the newline. The segments equations enable to write alternative statements in the code (see Listing 2.10).

2.1.4.4 Which division is preferable?

The previous section shows how Graphical Loop Invariant Based Programming methodology fits in the broader context of problem solving. In order to divide a complex problem into simpler sub-problems, a graphical approach has several advantages. First, it allows to identify repeating patterns that will require loops. Second, the drawings may be adapted into Graphical Loop Invariants, as in the example of the SP2 of the HOURGLASS problem.

As we show in the last example, trying to solve a problem (which is not that difficult) in a unique loop leads to a code that is longer, requires more calculation during the writing of the code (see Fig. 2.9), is less reusable and less maintainable hence error prone.

As the division in sub-problems is not unique, when should we stop dividing? A rule of thumb

is to keep sub-problems easy to solve. We recommand to try to limit a sub-problem to one short loop, properly documented by its Graphical Loop Invariant. This is in line with coding best practice of keeping short the length of functions [136].

2.1.5 Conclusion

In the above presentation of our methodology, we focused on graphical modifications and observation of the Graphical Loop Invariant to deduce codes instructions. We proposed a lot of drawings that explicit our approach. It goes without saying that most of the drawings are, in practice, not necessary. This is, for example, the case of the Fig. 2.3. In addition, when the methodology is mastered, even the Figures 2.2 may be just imagined on the basis of the sole Graphical Loop Invariant that is actually drawn. However, we do not recommend to just think about the Graphical Loop Invariant without sketching it: applying graphical transformations on an image that is just thought is rather complex and may lead to programming errors. We insist on the code deduction phase: the algorithm is deduced from the Graphical Loop Invariant that is a particular representation of the actual data the programmer must deal with and nothing else, a remark at the end of Sec. 2.2.3.1 illustrates accurately our point.

2.2 Applying GLIBP to Common Data Structures

The following section presents several common data structures and describes how they may be graphically represented and the information contained in their drawings. Of course, using a particular shape to represent a data structure is not mandatory but in the context of a lecture, it would be better to keep using the same pattern for a particular data and make sure that the picture meaning is shared by both the teaching team and the students. One must emphasis on these points :

- The shape of the design and why it is relevant;
- The part of the drawing that should be labelled;
- The position of particular pieces of information (*e.g.*, boundaries);
- The meaning of the relative position of the several elements in the picture;
- The graphical elements that represent specific implementation details (*e.g.*, pointers).

In the following, we keep using our colour code previously introduced. Table 2.2 shows for each element present in a Graphical Loop Invariant, the colour that we used and reminds the guideline that is associated with the represented element (see Sec. 2.1.2).

The rest of this section is organised in this way: Sec. 2.2.1 reminds the way of representing a graduated line; Sec. 2.2.2 shows how to represent the digits of a number and illustrates how it

Elements in the picture	Colours	Rule #
Names of the structures		Rule 1
Minimal boundaries		Rule 2
Maximal boundaries		Rule 2
Sizes of the data structures		Rule 2
Dividing Lines		Rule 3
Dividing Lines labels		Rule 4
What has been achieved so far		Rule 5
"To do" zones		Rule 6
Properties that are conserved		Rule 5 Rule 6

Table 2.2: Colour code for the elements in the graphical representation of data structures and Loop Invariants. Rule numbers refer to Sec. 2.1.2.

may save us from complex calculation; Sec. 2.2.3 introduces the way of representing arrays, shows how to find a Graphical Loop Invariant from the Postcondition and details how the Graphical Loop Invariant can be used to illustrate how a loop works; Sec. 2.2.4 shows how to draw lists and elaborates on pointers manipulation illustration rendered handy in our framework. Sec. 2.2.5 shows to draw queues and stacks in several way and discusses when to use a particular pattern rather than another. Finally Sec. 2.2.6 addresses graphs, Graphical Loop Invariants for for-each loops and GLIBP methodology in the context of oriented-object programming.

2.2.1 Graduated Lines

One of the most basic pattern is the graduated line, allowing us to represent ordered sets such as subsets of Natural or Integers. Fig. 2.10 shows the graphical representation of such a line. The line is labelled with the set name (*e.g.*, $\mathbb{N}$ or $\mathbb{Z}$, as in the Graphical Loop Invariant used in the previous section (See Fig. 2.1)). Each tick on the line corresponds to a value and all the represented values are offset by the same step (x in Fig. 2.10). The arrow at the far-right of the line indicates the increasing order of values. A value located at the right of another will, consequently, be greater than this last. Fig. 2.10 also shows how the boundaries should be placed at both ends of the picture. One can directly see at a glance that the figure represents the relation $a \leq b$. Sec. 2.1.3 illustrates how to deduce code instructions from a Graphical Loop Invariant drawn from this pattern.

† This representation is naturally inherited from numbering systems (*i.e.*, $N = a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + \ldots + a_1 \times b^1 + a_0 \times b^0$, or $\sum_{i=0}^{n-1} a_i \times b^i$), where $a_i \in \{0, 1, \ldots, b-1\}$

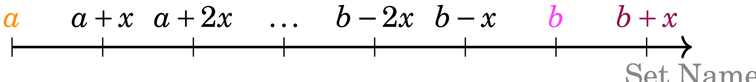


Figure 2.10: A line for representing ordered sets *e.g.*, Natural or Integers

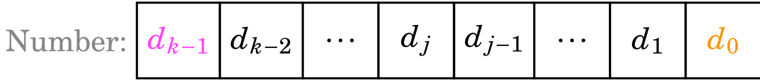


Figure 2.11: Pattern for Graphical Loop Invariant involving number representations

2.2.2 Number Representations

For problems concerning number representations (whether they are binary, decimal, hexadecimal, ...), one can represent this number as a sequence of digits named d_j[†], like in Fig. 2.11. The most significant digits are at the right and the least significant ones at the left.

Often, the d_j are not variables used in the code but rather figure the information contained in an actual variable, as it can been seen in the example (See Fig. 2.12). If a program must investigate the values of the digits in a certain order, it is possible to mention in the picture which is the first and last digits to be handled, as it is, for example, done in Fig. 2.11 where the least signifiant digit (in orange) will be used first and the most significant one (in magenta) will be used last.

2.2.2.1 Example: Reversing the Digits of a Number

To illustrate the pattern for representing numbers, here is an example of problem:

> REVERSENUMBER:
>
> **Input** – A positive integer s
>
> **Output** – A number r whose decimal representation is the decimal digits of s taken in reverse order.

The corresponding Graphical Loop Invariant is provided in Figure 2.12. First of all, there are two numbers in this problem: the source s and the result r. Hence the Graphical Loop Invariant must represent both numbers and here we see why correct labels are essential: we will not confuse the two numbers using the Graphical Loop Invariant. We first represent the initial value of s, that we label s_0 (See in the top of Fig. 2.12) and we represent the number r that contains several (but not all) digits d from s_0 (See in the bottom of Fig. 2.12). A Dividing Line is drawn to represent which part of s_0 was already reversed in r and the green zones are labelled accordingly. The remaining digits of s_0 constitute the "to do zone". These remaining digits will be in fact contained in the current value of s that will help us to enumerate all the digits of s_0

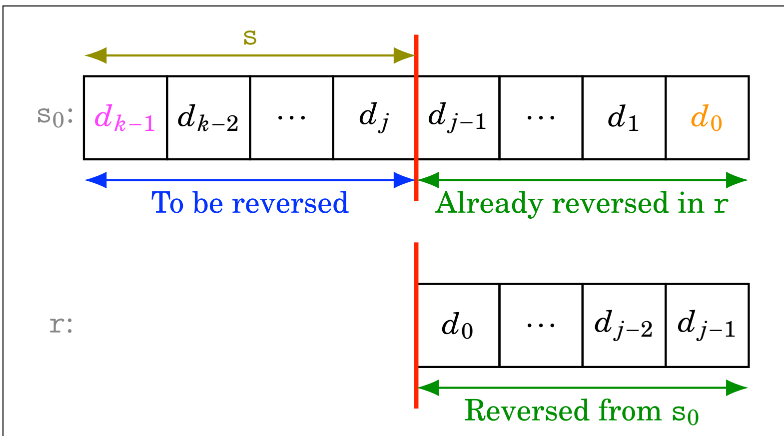


Figure 2.12: Graphical Loop Invariant involving number representations: the REVERSENUMBER problem

through the iterations. We coloured s in Fig. 2.12 in the same color as the Dividing Line labels because it is its role. To be precise, s will be our iteration variable, as can be seen in Listing 2.11.

```
const int B = 10;
int r = 0;
while(s)
{
   r = r * B;
   r = r + s % B; // % is the modulo operator
   s = s / B;
}
```

Listing 2.11: Code for the REVERSENUMBER problem, whose Graphical Loop Invariant is depicted in Fig. 2.12. The code does not depend on the base B.

2.2.2.2 The Graphical Loop Invariant May Save us from Complex Calculation

Using the Graphical Loop Invariant (See Fig. 2.12) save[s] us from rather complex computation. Let us illustrate that by expressing the values of s and r with mathematical notations. If we develop the initial decomposition of s (denoted by s_0) in decimal digits, we get the equation 2.1:

$$s_0 = \sum_{j=0}^{\lfloor \log s_0 \rfloor} 10^j \left(\left\lfloor \frac{s_0}{10^j} \right\rfloor \bmod 10 \right) \tag{2.1}$$

In this equation, the terms $\left(\left\lfloor \frac{s_0}{10^j} \right\rfloor \bmod 10 \right)$ correspond to the digits d_j and $\lfloor \log s_0 \rfloor$ corresponds to the value $k-1$, all depicted in Fig. 2.12. We can then express the final value of r (denoted by r_f) that is computed by the program and get the equation 2.2:

$$r_f = \sum_{j=0}^{\lfloor \log s_0 \rfloor} 10^{k-1-j} \left(\left\lfloor \frac{s_0}{10^j} \right\rfloor \bmod 10 \right) \tag{2.2}$$

As far as the values of `s` and `r` after a certain number of iterations i are concerned, they can be calculated thanks to resp. equations 2.3 and 2.4.

$$s = \sum_{j=i}^{\lfloor \log s_0 \rfloor} 10^{j-i} \left(\left\lfloor \frac{s_0}{10^{j-i}} \right\rfloor \bmod 10 \right) \tag{2.3}$$

$$r = \sum_{j=0}^{i-1} 10^{k-1-j} \left(\left\lfloor \frac{s_0}{10^{j}} \right\rfloor \bmod 10 \right) \tag{2.4}$$

One can see that if $i = 0$, *i.e.*, before the loop, $s = s_0$. At the end of the loop, all the digits of s_0 have been reversed in `r` and $i = k = \lfloor \log s_0 \rfloor + 1$ and thus $r = r_f$ while $s = 0$ (it is indeed the Stop Condition). Increasing the number of iterations decreases the number of terms in the sum 2.3 (*i.e.*, the number of digits of `s`) and increases those in the sum 2.4 (*i.e.*, the number of digits of `r`). This is exactly what is done by the program (See Listing 2.11).

Writing all those equations are not necessary in the Graphical Loop Invariant Based Programming framework. The careful representations of $\mathtt{s}_0$, `s`, and `r` in Fig. 2.12 are enough to lead to a correct code. Thanks to the drawing, we do not need to know the exact number of digits in `s` and `r` and the picture expresses well how the digits of `r` are related to those of $\mathtt{s}_0$ without the need for complex mathematical notations. This demonstrates clearly the advantage of a graphical approach, especially in the context of a CS1 course taken by students that are not particularly skilled in maths (*e.g.*, here, the Capital-sigma notation "$\sum$" or the modulo operation "mod" could be confusing since they are not used to them).

2.2.3 Arrays

An **array** is a ordered collection of elements of the same type. Each element can be directly accessed thanks to its relative position called **index**. The number of elements stored in a array is its **size**. The indices range from 0 to $size - 1$. Arrays are basic data structures: they can be used to build more complex ones. The simplest implementations of the arrays store the elements contiguously in memory (this is the case for C arrays [180]) making accesses to their elements very efficient.

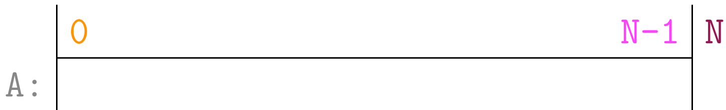


Figure 2.13: Pattern for Graphical Loop Invariant involving an array `A` of size `N`

Fig. 2.13 shows the representation of an array `A` containing `N` elements. The pattern follows a rectangular shape to depict the contiguous storage of the elements. Above this rectangle, we indicate indices of interest: at least the first (*i.e.*, 0) and the size `N`. It is important to see that `N` is written at the right of the array's border to mean that `N` is not a valid index as it is out of the array's bound that are in $[0..N-1]$. The name of the array is written at its left.

2.2.3.1 Finding the Loop Invariant from the Postcondition: the Example of the Binary Search

The problem illustrating the array pattern is the binary search that consists in locating a particular value in a strictly[13] sorted array. The problem specifications are the following:

BINARYSEARCH:

Input – A, a strictly sorted array of N integers and X, an integer

Output – A integer i such that $0 \leq$ i $<$ N and A[i] = X if X is in the array tab, -1 otherwise. tab is not modified.

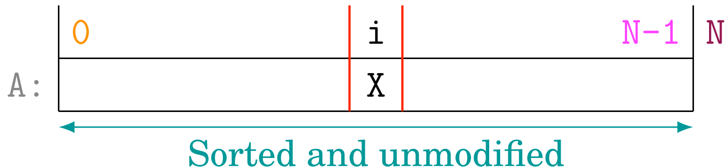


Figure 2.14: Postcondition of the BINARYSEARCH problem

A good starting point to find a Graphical Loop Invariant is to begin with the representation of the problem Postcondition. In Fig. 2.14, we represent such a state in the case where X was from the beginning in A. Since the searched value is present, we write X somewhere in the rectangular shape. The Postcondition uses the letter i to denote the index of X, thus we write i just above the X. By doing so, we create 3 parts in the array A: before X, X itself, and after X. In Fig. 2.14, we also add the array bounds and the properties that were kept True by the program: A is sorted and is left unmodified. In order to obtain a Graphical Loop Invariant, we must modify the Postcondition to represent the general situation of the loop, in which X was not yet found. We must then replace the particular position i by a zone in A where X could be found. We can do that easily by taking the two bars that surround X in Fig. 2.14 and shift both of them towards the array bounds.

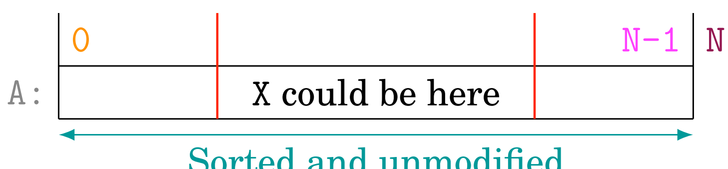


Figure 2.15: Example of Graphical Loop Invariant involving an array: binary search

We obtain the Fig. 2.15 that just has to be properly labelled in order to be a Graphical Loop Invariant. In the left part of A, we conclude that all the elements are lesser than X since the array is sorted and X is the middle part of A. We can make the same reasoning about the right part of

[13] To handle the case of a non-strict sorting, the Graphical Loop Invariant (and thus the program) must be modified accordingly.

A in which all the elements are greater than X. We notice that there are two Dividing Lines in the array and we label them with two variables l and u (for resp. **l**ower and **u**pper index). The middle part of the array, where X could be present must be investigated by the program and is labelled accordingly.

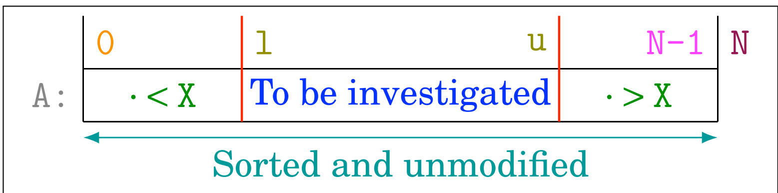


Figure 2.16: Example of Graphical Loop Invariant involving an array: BINARYSEARCH

Finally, we get the Fig. 2.16 that is the Graphical Loop Invariant for the BINARYSEARCH problem. It is worth noting that the color of the property "Sorted and unmodified" was not chosen randomly: it is a mix of blue and green that are used for respectively what should be done and what has already been achieved. In fact, it mentions a property that is already established (green) and this property must be kept True as the iterations go (blue). Hence mixing both blue and green is the best way to respect our own colours usage rules. The code of BINARYSEARCH problem is given in Listing 2.12.

```
1  int binarySearch(int *A, int N){
2    int l = 0, u = N-1;
3
4    //Invariant
5    while(l < u){
6      //Invariant ∧ Loop Condition
7      int m = (l + u) / 2;
8      if(A[m] < X)
9        l = m + 1;
10     else if(A[m] > X)
11       u = m - 1;
12     else
13       u = l = m;
14   }
15
16   //Invariant ∧ ¬Loop Condition
17   if (A[u] == X)
18     return u;
19   else
20     return -1;
21 }
```

Listing 2.12: Code for the BINARYSEARCH problem

To sum up, it may be easy to draw a Graphical Loop Invariant from the representation of

the problem Postcondition: one has to relax the Postcondition drawing to make appear a more general situation in which a "to be investigated" zone is visible (this always require to move one or more Dividing Line(s)) and then finish to label the new picture to respect the seven guidelines of the Graphical Loop Invariant (see Sec. 2.1.2).

Important note When deducing the code of the BINARYSEARCH problem, the fact that the zone to be investigated is divided by two at each iteration is a *consequence* of the particular properties exhibited by the array `A`. Some programming books (see Sec. 1.5.3) straightforwardly presents this algorithm by stating that the search zone is divided in half at each iteration, doing so, they start by cutting off the reflection phase that make appear the interesting and efficient part of the algorithm! On the other hand, our framework allows to make the learners discover by themselves the efficient algorithm from the properties that are observed in the data that must be processed by the program to be written.

2.2.3.2 Illustrate how an Algorithm Works thanks to the Graphical Loop Invariant

Since the beginning of this chapter, we focus on the Graphical Loop Invariant as a tool to deduce code instructions. In the following, we are going to detail another use of this drawing: to illustrate how an algorithm works. Let us imagine that we just discovered the code provided in Listing 2.12 and that we do not know its purpose. One of the best documentation for this piece of code is, in fact, its Graphical Loop Invariant!

Indeed, just by looking at it (see Fig. 2.16), we can understand that the algorithm investigates the content of an array `A` of size `N` thanks to index variables `l` and `u` and that another variable `X` is involved. `l` (resp. `u`) delimits a zone of elements of `A` less (resp. greater) than `X`. This information is available at a glance!

Of course, we need the Postcondition to understand the goal of the program but the Graphical Loop Invariant reveals us nearly all the implementation details.

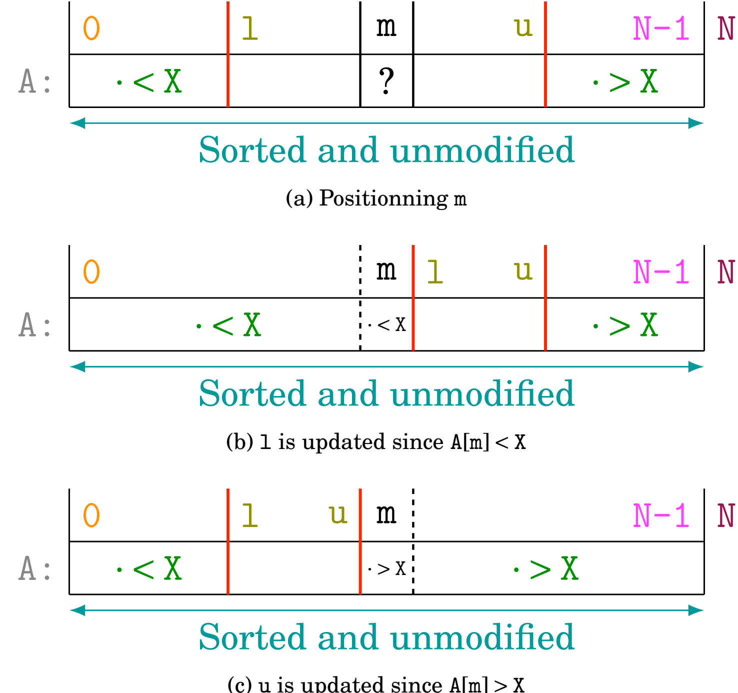
Moreover, if we wish to precisely explain the instructions of Listing 2.12 from lines 7 to 11, we will use the Graphical Loop Invariant, again. Here is how it can be done.

First, we place in the picture of `A` the variable `m` introduced at the line 7: `m` is in the middle of `l` and `u`. This situation is shown in Fig. 2.17a.

Then, if the test `A[m] < X` is True, we can draw the Fig. 2.17b. As it can be seen in the Figure, we can conclude that all the elements in `A[0..m]` are less than `X` since `A[0]` and `A[m]` are both less than `X` and the array is sorted. In conclusion, we can read the new value of `l` in Fig. 2.17b: it must be $m+1$ as `l` delimits a zone of elements less than `X` (see Listing 2.12, line 9).

On the other hand, if it is the test `A[m] > X` that is True, the Fig. 2.17c can be drawn. From the Figure, we conclude that all the elements in $A[m..N-1]$ are less than `X` and the new value of `u` must be $m-1$ (see Listing 2.12, line 11).

To be complete, we should also develop graphically the case in which `A[m] == X` but we will

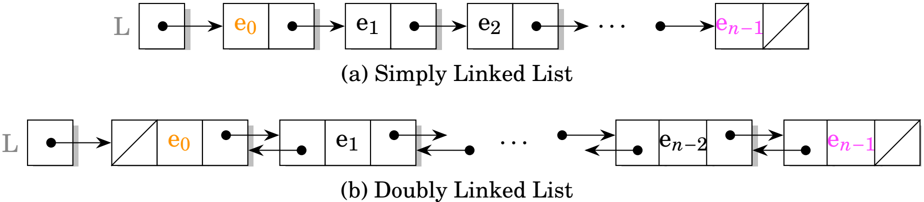


(a) Positionning m

(b) l is updated since A[m] < X

(c) u is updated since A[m] > X

Figure 2.17: Explaining graphically the instructions of the loop body



(a) Simply Linked List

(b) Doubly Linked List

Figure 2.18: Patterns for Simply and Doubly Linked List

not because it is very simple: this situation corresponds to the Postcondition (see Fig. 2.14).

In conclusion, we have shown in this section how the Graphical Loop Invariant is a good way to document a loop and how, on the basis of it, we can illustrate every single loop instructions.

2.2.4 Linked Lists

A **List** is a collection of elements that can be accessed in a sequential ordered. A **Linked List** is a sequence of **cells**. Each cell contains both data and a link to another cell. The link may be a reference[14] or a pointer (a variable that stores an address), as it is the case in the following examples.

Fig. 2.18a shows a **Singly Linked List** whose cells are linked by a unique pointer to the next cell. A cell can contain multiple pointer, as it is shown in Fig. 2.18b that depicts a **Doubly**

[14] *E.g.*, in Java [77]

Linked List: each cell is linked to the previous and next cells in the list. The last cell of the linked lists and the first cell of the doubly linked list contain a special pointer, represented by an oblique bar, that means that they are not linked to another cell. In both Fig. 2.18a and Fig. 2.18b, the pointers to the lists, that are named L, are drawn at the left and point to the first cell of their respective list. In the pictures, the elements in the lists are noted e_i with i ranging from 0 to $n-1$. n is the **length** of the list and the position i of an element is often called its **rank**.

2.2.4.1 Illustrating Data Structure Internals

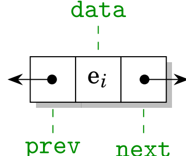


Figure 2.19: A Doubly Linked List Cell

The way we represent the lists cells is directly derived from their implementation. Fig. 2.19 allows to take a closer look at a doubly linked list cell. The pointers to a previous cell or a next cell are named `prev` and `next` respectively and are depicted as arrows. The field used to store the cell information is called `data`. The C implementation of such a cell is provided by the Listing 2.13. Of course, this representation of a cell is not new (*e.g.*, [77] uses it too) but it fit particularly well in our programming framework since it keeps the drawing close to the code (see Sec. 2.1.2, rule 7).

```
1 typedef struct cell{
2    T data;
3    struct cell *prev;
4    struct cell *next;
5 } DLinkedList
```

Listing 2.13: Structure of a Double Linked List Cell

2.2.4.2 Dealing with Pointers: the Example of the Copy of a Double Linked List

In order to illustrate the Graphical Loop Invariant Based Programming methodology with the lists, let us introduce the problem consisting in copying a doubly linked list:

> COPYLIST:
>
> **Input** – A doubly linked list L
>
> **Output** – Cc, a copy of L, which is left unmodified

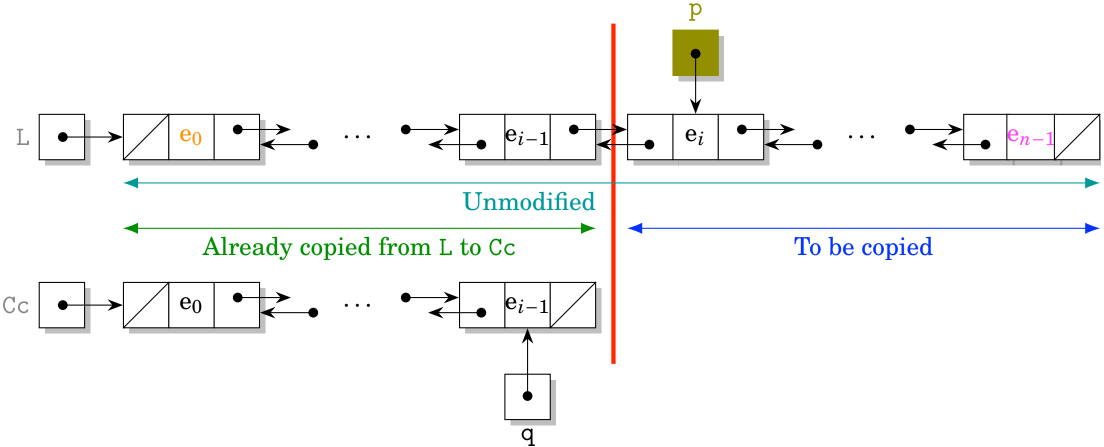


Figure 2.20: Graphical Loop Invariant of the CopyList problem

The corresponding Graphical Loop Invariant is shown in Fig. 2.20. The list L to be copied is entirely drawn and the fact that the list will not be modified is written below it. To represent a general situation in which only a part of the list L has been copied, a Dividing Line is placed in the middle of it. The cell in L at the right of the Dividing Line is the next cell to be copied and must be accessible with a pointer. We then draw such a pointer and name it p. We then label the drawing with what should still be done: to copy the rest of L, starting by the cell pointed by p. Finally, we indicate that the cells of L from the first to the one before p were copied in the list Cc. Of course, we represent that list just below L with the same data content in the cells. In order to make the list Cc grow, one must have access to its last cell: this is why the pointer q is introduced. The code for the CopyList problem is shown in Listing 2.14.

```
1  DLinkedList *copyList(const L *DLinkedList)
2  {
3     DLinkedList *p = L->next;
4
5     DLinkedList *Cc = malloc(sizeof(DLinkedList));
6     Cc->data = L->data;
7     Cc->prev = NULL;
8     Cc->next = NULL;
9
10    DLinkedList *q = Cc;
11
12    // Invariant
13    while(p != NULL)
14    {
15       q->next = malloc(sizeof(DLinkedList));
16       q->next->data = p->data;
17       q->next->prev = q;
18       q->next->next = NULL;
19
```

```
20         q = q->next;
21         p = p->next;
22     }
23
24     return Cc;
25 }
```

Listing 2.14: Code for COPYLIST problem

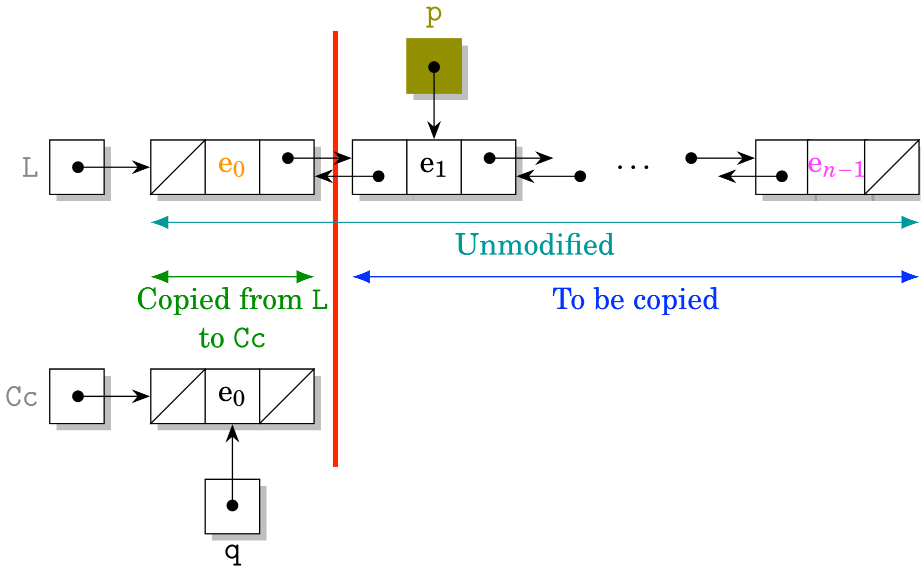


Figure 2.21: Variables initialisation is eased

Once again, the Graphical Loop Invariant enabled us to initialise the variables (*i.e.*, the ZONE 1, see Listing 2.14, lines 3 → 10). This initial situation is shown in Fig. 2.21: p must point to the second cell of L; Cc must be a list of a single cell, that is the copy of the first cell of L and q must point to the single cell of the list Cc. The Fig. 2.21 was obtained from Fig. 2.20 by shifting the Dividing Line to the left until the list Cc contains only one cell. Shifting the Dividing Line further to the left would have given a situation in which p would have pointed to the first cell of L and Cc would have been an empty list but q would not have been initialised and thus the Graphical Loop Invariant would not have been respected! In conclusion, a proper manipulation of the Graphical Loop Invariant allows to ease even quite complex variables initialisation.

Finally, grasping how pointers work may be difficult for first-year students [43], especially when dealing with a data structures such as a linked list which contains a large number of them. In the example of the COPYLIST problem, a new cell has to be created and linked at the end of the list Cc. To ease students understanding, it is common to decompose the pointers operations and to explain them with an illustration. This is of course possible in the GLIBP framework. More, using a picture from the beginning makes those graphical approaches straightforward and natural. In Fig. 2.22, we represent the appending of a new cell to the list Cc. One can see

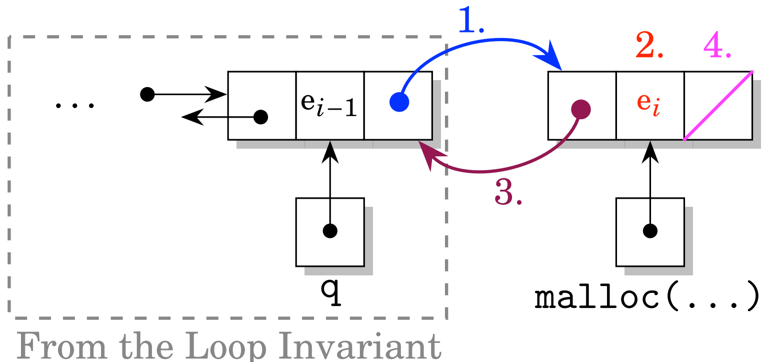


Figure 2.22: Appending a cell to a linked list

Step in Fig. 2.22	Operation	Line in Listing 2.14
1	Updating the `next` field of the last cell of `Cc`	15
2	Updating the `data` field of the new cell	16
3	Updating the `prev` field of the new cell	17
4	Initialize the `next` field of the new cell to `NULL`	18

Table 2.3: Matching between Fig. 2.22 steps and Listing 2.14

in the Figure that the respective situations of `Cc` and `q` were directly taken from the Graphical Loop Invariant (see Fig. 2.20). After having allocated a new cell, there are four steps that are listed in the Table 2.3. This table also references the line of Listing 2.14 corresponding to each instruction.

2.2.5 Queues and Stacks

The previous sections have presented the application of the Graphical Loop Invariant Based Programming methodology applied to some data structures: number lines, number representations, arrays, and lists. This is not a comprehensive list of the capabilities of this programming method: as long as a data structure can be drawn, all the loops that manipulate it can be written with Graphical Loop Invariants. It is up to the programmer to propose a relevant drawing that exhibit the main properties of the data structure he wishes to use. Let us take two more examples with the queue and the stack.

A **Queue** is a collection of elements that follows a **First-In First-Out** (FIFO) access policy: the elements are stored in the order of their entry and the first element to come out is the first that entered the queue. There are thus two allowed operations:

- **enqueue**: adding an element at the end of the queue (which is often referred as its **tail**)
- **dequeue**: removing an element from the beginning of the queue (which is often referred as

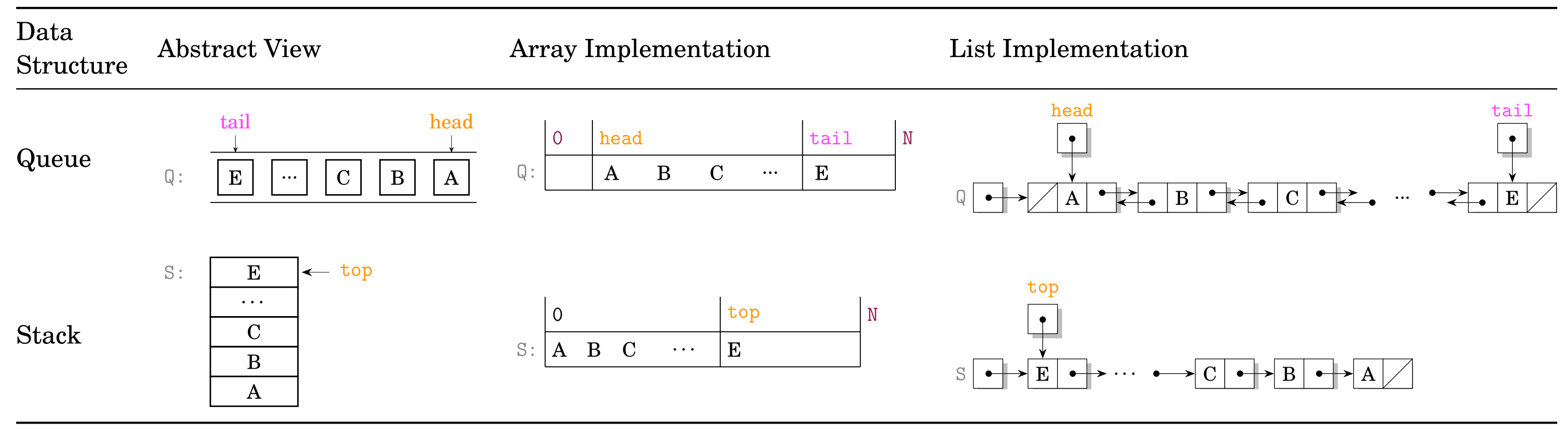


Table 2.4: Queue and Stack patterns

its **head**).

On the other hand, a **Stack** is a collection of elements that follows a **Last-In First-Out** (LIFO) access policy: the elements are stored in the order of their entry and the first element to come out is the last that entered the stack. There are two basic operations :

- **push**: adding an element on the top of the stack.
- **pop**: removing an element from the top of the stack.

Table 2.4 proposes several way of representing queues and stacks. Each drawing represent either of the two data structures in which the elements from A to E were added in the lexicographic order. The first column shows abstract views of these data structures: One can see the head, tail[,] and the FIFO policy of the queue and the top and the LIFO policy of the stack.

The second column of the Table shows the same structures but implemented with an array. For the queue, two array indices `head` and `tail` are used and the elements of the queue is stored between them. For the stack, an index represents the `top` position. For both structures, the array may be larger than the actual number of elements and the array size is of course drawn as well.

The third and last column show the queue and stack implemented as lists. The queue is implemented with a doubly linked list with a pointer `head` pointing to the first element and a pointer `tail` pointing to the last one. On the other hand, the stack can be implemented with a singly linked list with a pointer `top` pointing to the element at the top of the stack stored in the first cell of the list.

Which drawing to choose to make a Graphical Loop Invariant? It depends on the program to be written. If it consists in using an already implemented data structure through an interface, the abstract drawing is probably the best solution. If it consists in implementing a new function in the source code of a data structure library, the methodology recommand to draw a Graphical Loop Invariant that represents as best as possible the real data implementation and therefore to go for the array, the list or whatever is used in this particular case.

2.2.6 Graphs and Trees

The previous sections have introduced basic data structures exhibiting a linear common pattern: numbers in an enumeration, digits of a number, elements of an array, cells of a list,... all of them form sequences that may be graphically represented in a one-dimensional way.

The Graphical Loop Invariants do not need to be restricted to one-dimensional shapes. The section Sec. 2.1.4 already gave a flavour of that matter with a hourglass-shaped Graphical Loop Invariant that was not, in that case, the most appropriate representation to easily solve the problem. In this section, we describe non-linear shaped Graphical Loop Invariants, taking graphs

[15] In this section, we focus on connected and undirected graphs to illustrate our programming methodology. Graphs implementation, properties and algorithms are discussed in dedicated books, *e.g.*, [42, 77].

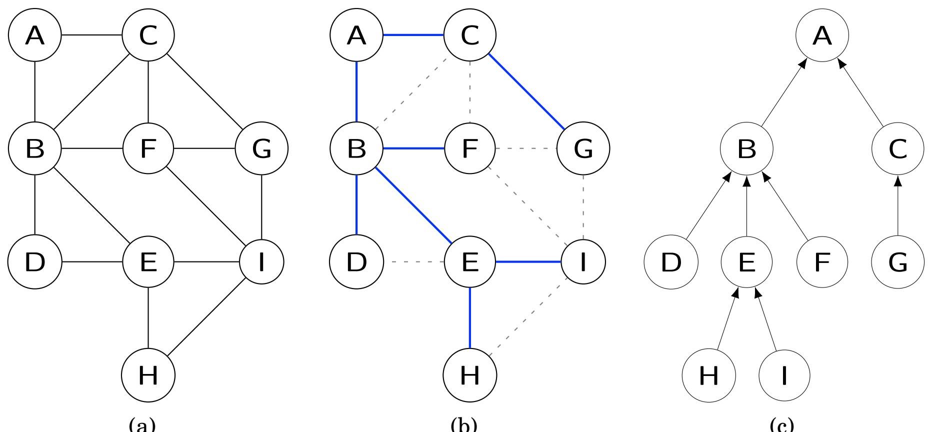


Figure 2.23: A graph consisting of 9 vertices labelled from A to I and 15 edges, the shortest paths from every vertices to A and the corresponding spanning tree.

as examples.

Let be $G = (V, E)$, an undirected and connected[15] **graph** that consists of a set of vertices V connected by edges e such that all $e \in E$. Such a graph is shown in Fig. 2.23a.

Let us define a **path** in a graph as a sequence of alternating vertices and edges such that for all successive vertices in the path, there exists an edge in the graph that links them. For example, in Fig. 2.23a, there are paths between all the nodes since the graph is connected. $A-C-F-G-I-E$ is a path between vertices A and E. $C-A-F-E$ is not a path since edges $A-F$ and $F-E$ do not exist. The length of a path is the number of edges in the path. The length of $A-C-F-G-I-E$ is 5. A shortest path is a path of minimum length. The shortest path between A and E is $A-B-E$, with a length of 2. There may be several shortest paths between two vertices, *e.g.*, $A-B-F$ and $A-C-F$.

We are going to apply GLIBP methodology in the context of graphs with this problem:

> SHORTESTPATHTOVERTEX:
>
> **Input** – An undirected and connected graph Graph g and `start`, a vertex belonging to g.
>
> **Output** – Annotate each vertex of g with its distance to `start` and the next vertex on a shortest path to `start`.

If the graph shown in Fig. 2.23a and vertex A are given as inputs to SHORTESTPATHTOVERTEX, the result could[16] be the graph shown in Fig. 2.23b, whose shortest paths to vertex A are highlighted in blue, forming thus a spanning tree that is represented in Fig. 2.23c.

[16] Since the shortest paths are not unique, those that are actually kept as a result depends on the particular implementation of the graph.

2.2.6.1 Graph Abstract Data Type

There are several ways to implement a graph : one can use adjacency lists, an adjacency matrix, an adjacency map,... but here, we do not focus on a particular graph implementation: we assume that we have one that complies with the graph abstract data type and possesses at least the following operations [77] :

`vertices()`	Get an iterator of all the vertices.
`outGoingEdges(v)`	Get a collection of the edges going out of the vertex `v`.
`opposite(v, e)`	Get the other vertex that is on the same edge `e` as the vertex `v`.
`neighbours(v)`	Return the neighbours of the vertex `v`, *i.e.*, the subset $N \subset V$ containing the vertices that are connected to the vertex `v` by a single edge.

`neighbours` operation is the application of `opposite(v, e)` to all edges returned by `outGoingEdges(v)`

2.2.6.2 Graphical Loop Invariant and Oriented Object Programming

In order to illustrate how Graphical Loop Invariant fits in Oriented-Object (OO) programming paradigm and to simplify the reading of the code, we amend our pseudocode with the following OO features:

- Instances are created with `new`;
- Objects are passed by references;
- Attributes and methods are accessed with the dot (`.`) operator.
- Objects attributes are accessible, regardless of the proper visibilities they should have if OO principles were correctly applied.

E.g., `g.neighbours(v)` will denote a call to the method `neighbours` on an instance of a `Graph` g with an instance of `Vertex v` given as actual parameter.

2.2.6.3 Graphical Loop Invariant depicting graphs

As usual, we start by drawing the result of the problem (see Fig. 2.23b). In such a final situation, all the vertices of the graphs were discovered (*i.e.*, added in a shortest path to the `start` vertex).

To get an Graphical Loop Invariant, we represent a situation in which some (*but not all*) vertices at a certain distance from the start vertex were discovered. This is what is shown in Fig. 2.24: D, E and F are at a a distance 2 of A, as well as G that has not been discovered yet.

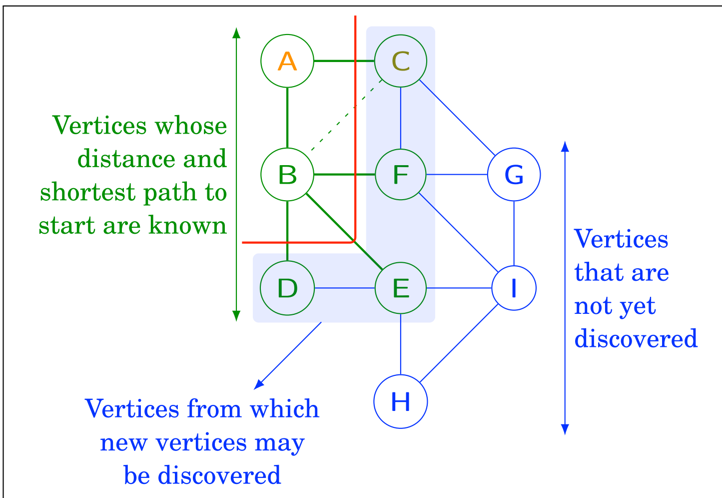


Figure 2.24: Graphical Loop Invariant for the SHORTESTPATHTOVERTEX problem

C which is at a distance 1 from A must have been discovered before D, E and F (the closer the vertex the sooner they must be discovered to ensure shortest paths by construction).

Doing so, we established three kinds of vertices:

1. Those that were discovered and whose neighbours were discovered too;
2. Those that were discovered but not their neighbours;
3. The undiscovered ones.

In Fig. 2.24, discovered vertices are drawn in green to follow our colour code of what was previously achieved. The undiscovered vertices are thus coloured in blue. We must also memorise in a data structure the newly discovered vertices to continue the discover of their neighbours in the next iterations. What should be the access policy of such a data structure? Since it may contain vertices with different distances from the start and that vertices closer to start are discovered first, a FIFO structure is required to handle the vertices in the proper order. In our example, C must be retrieved before D, E or F to discover G at a distance 2 of A.

The code corresponding to the Graphical Loop Invariant shown in Fig. 2.24 is presented in Listing 2.15. The careful reader will have recognised a breadth-first search algorithm.

```
1 Graph shortestPathToVertex(Graph g, Vertex start) {
2    // Check that start is in g.vertices() (omitted)
3
4    foreach(Vertex v in g.vertices()) {
5       v.dist = UNKNOWN; // Special value for not yet computed distances
6       v.prev = NULL;
7    }
8
9    Queue q = new Queue();
```

```
10     start.dist = 0;
11     q.enqueue(start);
12 
13     while(!q.isEmpty()) {
14        Vertex v = q.dequeue();
15        foreach(Vertex n in g.neighbours(v)) {
16           if (n.dist == UNKNOWN) {
17              n.dist = v.dist + 1;
18              n.prev = v;
19              q.enqueue(n);
20           }
21        }
22     }
23     return g;
24  }
```

Listing 2.15: Code for SHORTESTPATHTOVERTEX problem

2.2.6.4 For-each loops, iterators and Graphical Loop Invariant

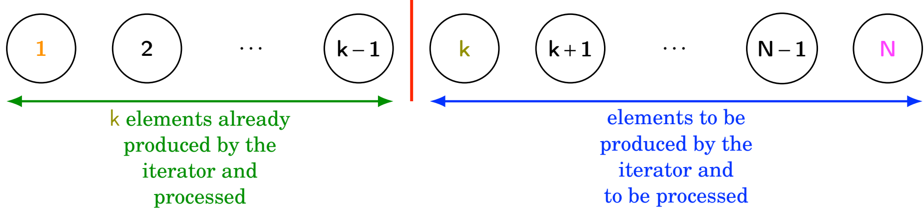


Figure 2.25: A one-size-fit-all Graphical Loop Invariant for iterators and for-each loops

In Listing 2.15, we have written three loops : a `while` loop, based on the Graphical Loop Invariant shown in Fig. 2.24, and two `for-each` loops. The reader may ask themself where are their respective Graphical Loop Invariant? In fact, such simple for-each loops performing the very same operation on every elements of a collection (often enumerated by an iterator) have the same type of Graphical Loop Invariant.

One can create a "Meta" Graphical Loop Invariant that covers all these loops, once for all. The drawing is simplistic and writing the corresponding code is rather trivial. Such an Graphical Loop Invariant is shown in Fig. 2.25.

Generally speaking, we do not recommend to stop drawing Graphical Loop Invariant for `for-each` loops, especially if the operation performed in the Loop Body has complex side effects. However, we acknowledge that some simple loops may be written without the help of a Graphical Loop Invariant. Moreover, in the context of presenting the interest of GLIBP, choosing such trivial loops as use cases would be counterproductive to make one's point.

2.3 Comparison Between Graphical and Non-Graphical Loop Invariants

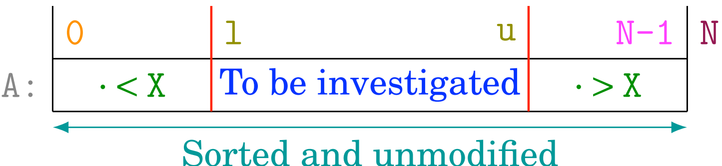


Figure 2.26: Graphical Loop Invariant for BISECTIONSEARCH (already presented in Sec. 2.2.3

This section examines other ways of representing an Loop Invariant, namely the formal Loop Invariant and its natural language translation. We will illustrate both of them with the Loop Invariant for the BISECTIONSEARCH problem (see Sec. 2.2.3). In order to ease the comparison between the different representations, the Graphical Loop Invariant for the BISECTIONSEARCH is reminded in Fig. 2.26 and the following will use the same colour code as in the Figure, although this last one may help to render the Non-Graphical Loop Invariants more readable than they actually are!

2.3.1 Formal Loop Invariant

A formal Loop Invariant is exactly what proposed Dijkstra [56]. The formal Loop Invariant for the bissection search is the following:

$$Inv \equiv \mathtt{A} = \mathtt{A}_0 \tag{2.5}$$

$$\wedge\ 0 \leq \mathtt{l} \leq \mathtt{u} < \mathtt{N} \tag{2.6}$$

$$\wedge\ \forall i, 0 \leq i < \mathtt{N}-1, \mathtt{A}[i] < \mathtt{A}[i+1] \tag{2.7}$$

$$\wedge\ \forall i, 0 \leq i < \mathtt{l}, \mathtt{A}[i] < \mathtt{X} \tag{2.8}$$

$$\wedge\ \forall i, \mathtt{u} < i < \mathtt{N}, \mathtt{A}[i] > \mathtt{X} \tag{2.9}$$

The equation (2.5) means that the array A is not modified. The equation (2.6) indicates the relative positions of the indices l and u with respect to the boundaries 0 and N. The equation (2.7) states that the array is sorted (since for all couples of contiguous elements, the one that follows is always bigger than the one it precedes). The equations (2.8) and (2.9) describe the zone were the elements are less than X and greater than X, respectively.

At first glance, a neophyte cannot see that the array is divided in three zones but a more trained eye will identify the quantifications as zones in the array. Still, it is worth noting that there is no way to express formally the existence of the "to do zone" with formal notations. Since we explained in Sec. 2.2.3.1 that this zone comes into play when finding the Loop Invariant from the Postcondition, this operation is therefore a bit more difficult with formal notations[17].

[17] When justifying an example of Postcondition transformation into an Loop Invariant, Dijkstra invokes his experience [56, p. 53]. We cannot assume that a neophyte benefits from the same experience!

The manipulation of notations like the equations (2.5) to (2.9) to write code may quickly become tedious. It is possible to simplify it with predicate definitions:

$$\text{Sorted}(T,a,b) \equiv \forall i, a \leq i < b, T[i] < T[i+1] \tag{2.10}$$

$$\text{Less}(T,a,b,x) \equiv \forall i, a \leq i < b, T[i] < x \tag{2.11}$$

$$\text{Greater}(T,a,b,x) \equiv \forall i, a \leq i < b, T[i] > x \tag{2.12}$$

And the Formal Loop Invariant becomes:

$$Inv \equiv \mathtt{A} = \mathtt{A}_0 \tag{2.13}$$

$$\wedge\, 0 \leq \mathtt{l} \leq \mathtt{u} < \mathtt{N} \tag{2.14}$$

$$\wedge\, \text{Sorted}(\mathtt{A}, 0, \mathtt{N}-1) \tag{2.15}$$

$$\wedge\, \text{Less}(\mathtt{A}, 0, \mathtt{l}, \mathtt{X}) \tag{2.16}$$

$$\wedge\, \text{Greater}(\mathtt{A}, \mathtt{u}+1, \mathtt{N}, \mathtt{X}) \tag{2.17}$$

Introducing the three predicates enhances the clarity when reading the Loop Invariant but barely helps when it comes to write the code! For example, to initialise the variables `l` and `u`, we have to find two values that make the Formal Loop Invariant True, *i.e.*, the terms (2.14), (2.16) and (2.17) must be all True with this particular choice.

Considering the definition of the predicates *Less* and *Greater*, we can take particular values of `l` and `u` that make the universal quantifications trivially True. This is, of course, the case if we take $\mathtt{l} = 0$ and $\mathtt{u} = \mathtt{N} - 1$ (one could also argue that these particular values are suggested by the term (2.14). However, we still have to manipulate logic formulae to find the initialisation of the variables while we just read them in a drawing when we used the Graphical Loop Invariant! Similar arguments may be advanced about deriving other parts of the loop.

2.3.2 Loop Invariant in Natural Language

A Loop Invariant in Natural Language would look like this:

- "The array `A` is sorted and unmodified.
- `A` is divided in three zones:
  1. The elements in `A[0 .. l-1]` are lesser than `X`;
  2. The elements in `A[u+1 .. N-1]` are greater than `X`;
  3. The remaining part (`A[l .. u]`) has to be investigated for `X`."

This Loop Invariant contains the same information as in the Graphical Loop Invariant and the Formal Loop Invariant. Admittedly, one could argue that the notation `A[i..j]` is yet too

formal but that just would help us to make our point: this kind of Loop Invariant is practically useless to deduce the code.

First of all, it requires mathematical and logical skills to evaluate a sentence as logically True! If we take the example of the initialisation of `l`, concluding that “The elements in A[0..0-1] are lower than X” is trivially True asks for recognizing an universal quantification over an empty set. We might as well directly write a Formal Loop Invariant!

Moreover, this Loop Invariant lacks of accuracy as the relative positioning of the three zones is unclear to figure.

2.3.3 Conclusion

Both the Formal and the Natural Language Loop Invariants require mathematical and logical skills to be used to deduce the code. Moreover, there is a kind formality / human readability trade-off between these two. On one hand, the Loop Invariant in Natural Language may be useful to describe how the loop works while the formal one requires to dive into logic formulae. On the other hand, the Formal Loop Invariant is more handy to deduce the code, provided the programmer is used to it.

Anyway, the Graphical Loop Invariant brings the best of both worlds: it is clear enough to illustrate how the loop works, even better than the Loop Invariant in Natural Language and it enables to deduce the code with much less skills requirement than the Formal Loop Invariant!

Finally, as proving the code correctness is concerned, we recommand, of course, to go for the Formal Loop Invariant that is the only version that allows to write formal proofs.

2.4 GLIBP Methodology and predicate transformers

Previous sections mentioned that the GLIBP methodology required to draw a Graphical Loop Invariant and use it to deduce code instructions thanks to drawing manipulations. We explain here that these are in line with the predicate transformers introduced by Dijkstra, namely the weakest precondition.

To illustrate this, we introduce the Dutch national flag problem. This problem was proposed by W.H.J. Feijen. In the following, the design and colours of our flag may appear as a French one: it is a mere coincidence.

DUTCHFLAG:

Input – A flag (*i.e.*, an array) containing elements of three colours: blue, white and red, in any order.

Output – The flag with the colours in the proper order: blue first, then white and red. The result must be a permutation of the initial flag.

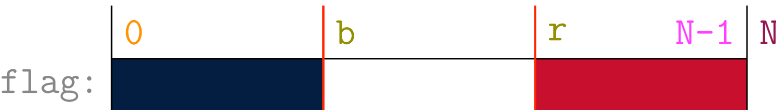


Figure 2.27: Postcondition for the DUTCHFLAG problem

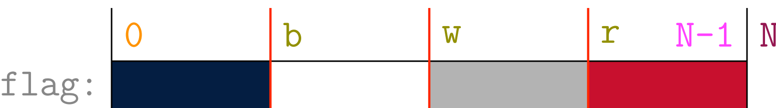


Figure 2.28: Graphical Loop Invariant for the DUTCHFLAG problem. the $flag \operatorname{perm} flag_0$ part is omitted (one could add a double arrow spanning the entire flag to mention it.

Calling it the “French flag problem” would allow to insist easily that the algorithm should work on several corner cases such as all-white (French royal flag), all-blue (royal again, without the lilies) or all-red (Paris Commune flag) flags. Usually, it is also requested that the colours positions are are just swapped, leading to a linear algorithm (or any other constraint that lead to this conclusion).

The Postcondition is given in Fig. 2.27. As we are going to compare drawings manipulations and weakest precondition computation, we must introduce a formal version of the Postcondition. In order to do so, we first introduce the predicate *Colour*:

$$Colour(f,i,j,c) \equiv \forall k, 0 \leq i < j, f[k] = c \tag{2.18}$$

which is true when the portion of the flag $f[i..j-1]$ is coloured in plain colour c. We can the express the Postcondition:

$$\begin{aligned} Post \equiv\ & 0 \leq b \leq r \leq N \\ & \wedge Color(flag, 0, b, Blue) \\ & \wedge Color(flag, b, r, White) \\ & \wedge Color(flag, r, N, Red) \\ & \wedge flag \operatorname{perm} flag_0 \end{aligned} \tag{2.19}$$

Where $X \operatorname{perm} Y$ means that X is a permutation of Y.

The Loop Invariant should denote that a part of the flag is still unordered (see Fig. 2.28 and equation 2.20). Using this Loop Invariant suggests to introduce a variable `w`.

$$\begin{aligned} Inv \equiv\ & 0 \le b \le w \le r \le N \\ & \wedge Color(flag, 0, b, Blue) \\ & \wedge Color(flag, b, w, White) \\ & \wedge Color(flag, r, N, Red) \\ & \wedge flag \, \mathrm{perm} \, flag_0 \end{aligned} \tag{2.20}$$

The unordered area is bounded by `w` and `r`. If this unordered area span the entire flag, this suggests to have as initialisation the following commands:

```
int w = 0;
int r = N;
```

Listing 2.16: DUTCHFLAG: first try for INIT instructions

One can compute wp(`w = 0;r = N;`, Inv) to figure the initial value of `b` and ensure this is compatible with the problem's Precondition :

$$\mathrm{wp}(\texttt{w = 0;r = N;}, Inv) = 0 \le b \le 0 \wedge \le r = N \tag{2.21}$$
$$\wedge Color(flag, 0, b, Blue) \tag{2.22}$$
$$\wedge Color(flag, b, 0, White) \tag{2.23}$$
$$\wedge Color(flag, r, r, Red) \tag{2.24}$$
$$\wedge flag \, \mathrm{perm} \, flag_0 \tag{2.25}$$
$$\tag{2.26}$$

The line 2.22 imposes $b = 0$. Therefore, lines 2.23 to 2.25 simplify to *true* since $Colour(f, x, x, c)$ is *true* for any x. At the beginning of the program, by definition, $flag = flag_0$, hence, we can conclude that

$$Pre \Rightarrow \mathrm{wp}(\texttt{INIT}, Inv) \tag{2.27}$$

where `INIT` is the following instructions (in any order):

```
// a.k.a 'INIT'
int b = 0;
int w = 0;
int r = N;
```

Listing 2.17: DUTCHFLAG: final INIT instructions

In the GLIBP methodology, we would have derived the very same instructions by gliding the two Dividing Lines to expand the grey zone, as can be seen in Fig. 2.29. As such, manipulating the drawing consists in graphically computing wp(`INIT`, Inv).

Figure 2.29: wp(INIT, *Inv*) for the DUTCHFLAG problem. One can check that this situation can be deduced from the Precondition

.

To be precise, here is our point:

> The Graphical Loop Invariant Based Programming methodology asks for representing conditions on state and thus predicate (namely, the Loop Invariant). In this framework, the graphical transformations are predicate transformers.

Until now, we have shown that it worked for an example of composition of variable assignments. It also work for alternative constructs. In the DUTCHFLAG problem, we have three variables : `b`, `w` and `r`. The two former should increase and the later should decrease as the loops run. Here again, the weakest precondition can be used to determine upon which condition the commands `b = b + 1`, `w = w + 1` and `r = r - 1` may be executed:

$$\begin{aligned} \mathrm{wp}(\mathtt{w}=\mathtt{w}+1,\ Inv) = 0 \leq b \leq (w+1) \leq r \leq N \\ \wedge\, Color(flag, 0, b, Blue) \\ \wedge\, Color(flag, b, w+1, White) \\ \wedge\, Color(flag, r, N, Red) \\ \wedge\, flag\ \mathrm{perm}\ flag_0 \\ = Inv \wedge w < r \wedge \mathtt{flag[w]} = White \end{aligned} \tag{2.28}$$

$$\begin{aligned} \mathrm{wp}(\mathtt{r}=\mathtt{r}-1,\ Inv) = 0 \leq b \leq w \leq (r-1) \leq N \\ \wedge\, Color(flag, 0, b, Blue) \\ \wedge\, Color(flag, b, w, White) \\ \wedge\, Color(flag, r-1, N, Red) \\ \wedge\, flag\ \mathrm{perm}\ flag_0 \\ = Inv \wedge w < r \wedge \mathtt{flag[r-1]} = Red \end{aligned} \tag{2.29}$$

From 2.28 and 2.29, one can conclude that $w < r$ should be ensured in the loop. In fact, it is the Loop Condition, as $(Inv \wedge w \geq r) = Post$. Both equations give us the condition upon which their

commands must be activated: `w` should be increased if $\texttt{flag[w]} = \mathit{White}$ and `r` should be decreased if $\texttt{flag[r} - \texttt{1]} = \mathit{Red}$. `b`'s situation is a bit more complex, as

$$\begin{aligned} \mathrm{wp}(\texttt{b} = \texttt{b} + \texttt{1},\ \mathit{Inv}) = 0 \le (b+1) \le w \le r \le N \\ \wedge\, \mathit{Color}(\mathit{flag}, 0, b+1, \mathit{Blue}) \\ \wedge\, \mathit{Color}(\mathit{flag}, b+1, w, \mathit{White}) \\ \wedge\, \mathit{Color}(\mathit{flag}, r, N, \mathit{Red}) \\ \wedge\, \mathit{flag}\ \mathrm{perm}\ \mathit{flag}_0 \end{aligned} \tag{2.30}$$

requires that $\texttt{flag[b]} = \mathit{Blue}$. That is not ensured by the invariant because `flag[b]` is supposed to be *White*! What if we exchange `b` and `w` colours before increasing `b` ? We introduce the notation `swap(b,w)` that is self-documenting. A real function would require a reference to a flag, which is omitted here for the sake of brevity.

$$\begin{aligned} &\mathrm{wp}(\texttt{swap(b,w)}; \texttt{b} = \texttt{b} + \texttt{1},\ \mathit{Inv}) = \\ &\mathrm{wp}(\texttt{swap(b,w)};,\ \mathrm{wp}(\texttt{b} = \texttt{b} + \texttt{1},\ \mathit{Inv})) = 0 \le (b+1) \le w \le r \le N \\ &\wedge\, \mathit{Color}(\mathit{flag}, 0, b, \mathit{Blue}) \\ &\wedge\, \mathit{Color}(\mathit{flag}, b, w, \mathit{White}) \\ &\wedge\, \mathit{Color}(\mathit{flag}, r, N, \mathit{Red}) \\ &\wedge\, \mathit{flag}[w] = \mathit{Blue} \\ &\wedge\, \mathit{flag}\ \mathrm{perm}\ \mathit{flag}_0 \end{aligned} \tag{2.31}$$

This cannot be deduced from the Loop Invariant since $(b+1) \le w$ violates the first part. However, one could try to have $(b+1) \le (w+1)$, suggesting to also increase `w`. This leads to:

$$\begin{aligned} &\mathrm{wp}(\texttt{swap(b,w)}; \texttt{w} = \texttt{w} + \texttt{1}; \texttt{b} = \texttt{b} + \texttt{1},\ \mathit{Inv}) = \\ &\mathrm{wp}(\texttt{swap(b,w)};,\ \mathrm{wp}(\texttt{w} = \texttt{w} + \texttt{1}; \texttt{b} = \texttt{b} + \texttt{1},\ \mathit{Inv})) = 0 \le (b+1) \le (w+1) \le r \le N \\ &\wedge\, \mathit{Color}(\mathit{flag}, 0, b, \mathit{Blue}) \\ &\wedge\, \mathit{Color}(\mathit{flag}, b+1, w, \mathit{White}) \\ &\wedge\, \mathit{Color}(\mathit{flag}, r, N, \mathit{Red}) \\ &\wedge\, \mathit{flag}[w] = \mathit{Blue} \\ &\wedge\, \mathit{flag}[b] = \mathit{White} \\ &\wedge\, \mathit{flag}\ \mathrm{perm}\ \mathit{flag}_0 \end{aligned} \tag{2.32}$$

That simplifies into

$$\begin{aligned}
\text{wp}(\texttt{swap(b,w);w = w + 1;b = b + 1}, \textit{Inv}) = \\
&= 0 \leq (b+1) \leq (w+1) <= r \leq N \\
&\quad \wedge \textit{Color}(\textit{flag}, 0, b, \textit{Blue}) \\
&\quad \wedge \textit{Color}(\textit{flag}, b, w, \textit{White}) \\
&\quad \wedge \textit{Color}(\textit{flag}, r, N, \textit{Red}) \\
&\quad \wedge \textit{flag}[w] = \textit{Blue} \\
&\quad \wedge \textit{flag}\ \text{perm}\ \textit{flag}_0
\end{aligned} \tag{2.33}$$

This requires that $w < r$, that is ensured by the Loop Condition. The sequence of instructions can be activated if `flag[w]` $= \textit{Blue}$.

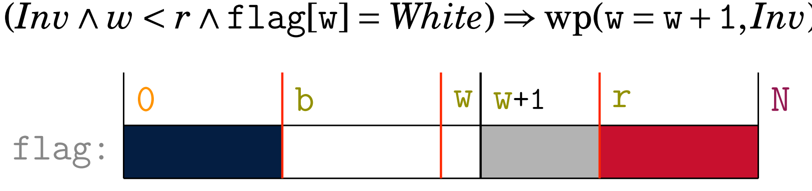
What if `flag[w]` $= \textit{Red}$? We can swap positions `w` and `r - 1` and then decrease `r` (see above). At the end, we only have to branch on the value of `flag[w]`,a s can be seen in Listing 2.18.

```
1  void dutchFlag(flag, N) {
2     int b = 0, w = 0, r = N;
3     while(w < r) {
4        switch(flag[w]) {
5           case white:
6              w += 1;
7              break;
8           case red:
9              swap(flag, w, r-1);
10             r = r - 1;
11             break;
12          case blue:
13             swap(flag, w, b);
14             b = b + 1;
15             w = w + 1;
16             break;
17       }
18    }
19 }
```

Listing 2.18: DUTCHFLAG program

All these computations could be easily avoided using the GLIBP methodology. Equation 2.28 is depicted in Fig. 2.30, equation 2.29 in Fig. 2.31 and equation 2.33 in Fig. 2.32. In practise, it is even not required to be aware of the concept of weakest precondition to use the Graphical Loop Invariant to deduce code instructions! One can be satisfied with a well-executed drawing. This example shows that our methodology is both theoretically well grounded and within a first year student reach.

$(Inv \wedge w < r \wedge \texttt{flag[w]} = White) \Rightarrow \mathrm{wp}(\texttt{w} = \texttt{w} + 1, Inv)$



Inv again:

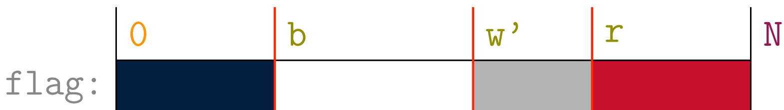


Figure 2.30: DUTCHFLAG problem: graphical weakest precondition for the White area. In order to restore the Loop Invariant when `w` is increased, `flag[w]` must be *White*.

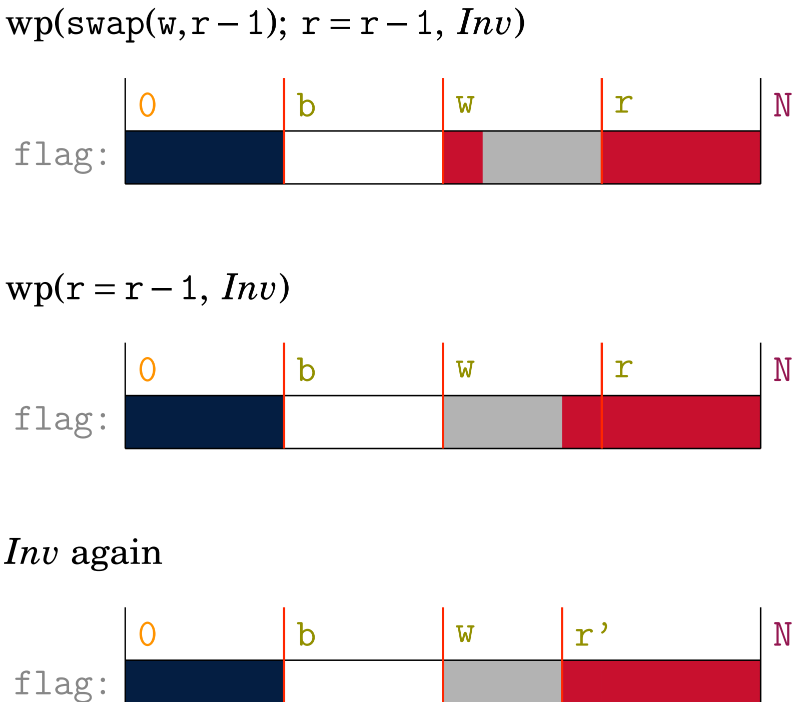


Figure 2.31: DUTCHFLAG problem: graphical weakest precondition for the Red area. In order to restore the Loop Invariant, if `r` is decreased, one should ensure that `flag[r-1]` is *Red*. That is the case if `flag[w]` is *Red* before swapping `w` and `r-1` positions.

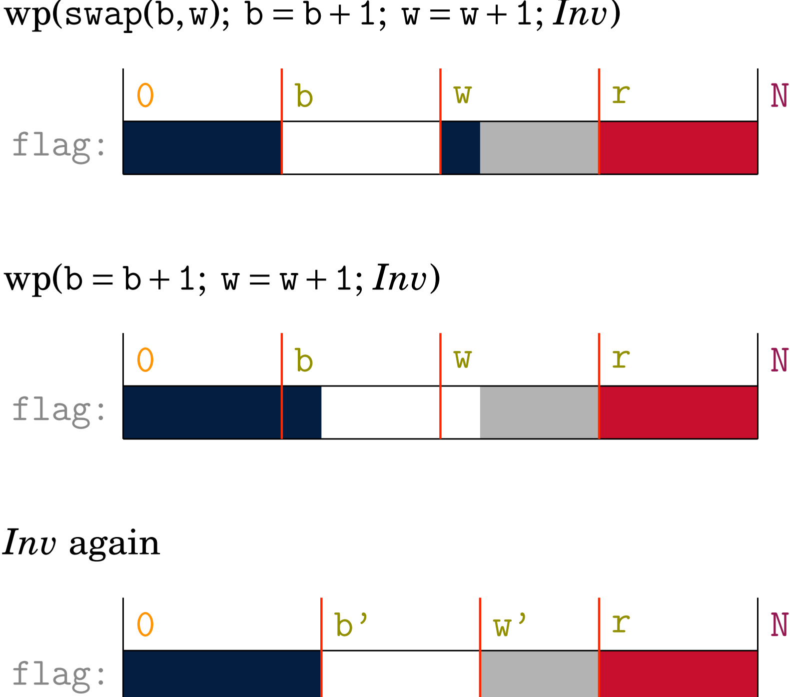


Figure 2.32: DUTCHFLAG problem: graphical weakest precondition for the Blue area. In order to restore the Loop Invariant if `b` is increased then `w` must be increased too. One should ensure first that `flag[w]` is *White* and `flag[b]` is *Blue*. This will be the case if `flag[w]` is *Blue* before swapping `w` and `b` positions.

Chapter 3

Learning Tools

This chapter introduces tools we developed to help students grasping how to draw Graphical Loop Invariants. Sec. 3.1 firstly present an application to draw and to manipulate Graphical Loop Invariant in a web browser. This application is called "Graphical Loop Invariant Drawing Editor (GLIDE)". Second, Sec. 3.2 presents the Blank Graphical Loop Invariant, that enables to use a graphical approach in the context of automatic assessment.

3.1 GLIDE

GLIDE is an application written in javascript using the FabricJS HTML5 canvas library [193]. It enables students to easily draw Graphical Loop Invariants that respect the guidelines mentioned in the Sec. 2.1.2. The application is able to provide students with feedback about their drawing. Finally, a validated drawing can be graphically manipulated to deduce the code as described as in Sec. 2.1.3.

The application GUI is fairly simple, as illustrated in Fig. 3.1. There are five buttons whose

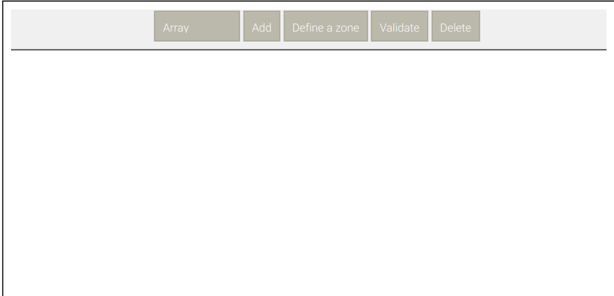


Figure 3.1: GLIDE Graphical User Interface: five buttons and a canvas

Table 3.1: Effect of the five buttons of GLIDE GUI

Button	Effect
PIECE OF INVARIANT	Enables to select a Loop Invariant pattern. Sec. 3.1.1 elaborates on the available patterns inspired by Sec. 2.2.
ADD	Actually draws the selected pattern in the canvas.
DEFINE A ZONE	Enables to draw a zone between two bars. Sec. 3.1.2 explains what we mean by *bars* and Sec. 3.1.3 details how zones can be added in the drawing.
VALIDATE	Launches several tests checking whether the Loop Invariant respects the guidelines (see Sec. 2.1.2). More details are provided in Sec. 3.1.5. After having passed the validation, a Graphical Loop Invariant can be manipulated to derive the initial and final situations to deduce code instructions (see Sec. 3.1.6).
DELETE	Enables to delete an item from the canvas (see Sec. 3.1.4.1).

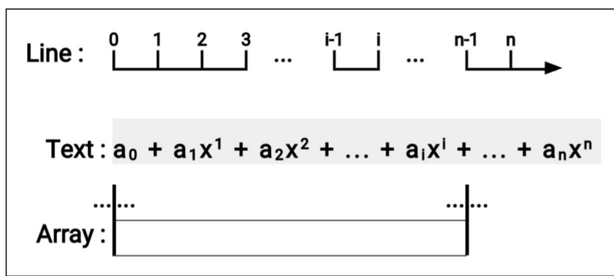


Figure 3.2: Pattern available in GLIDE to draw Graphical Loop Invariants. The text zone is here used to depict a polynomial thanks to subscripts and superscripts

utility is shown in Table 3.1. The rest of the GUI consists of a canvas where the elements that form the Loop Invariant can be drawn.

The remainder of the section details GLIDE capabilities: the available patterns (see Sec. 3.1.1), how to draw *Dividing Lines* (see Sec. 3.1.2), how to add zones (see Sec. 3.1.3), how to edit a drawing (see Sec. 3.1.4), the validation process (see Sec. 3.1.5), and how the drawing can be manipulated to expose particular situations and to derive the associated code (see Sec. 3.1.6).

3.1.1 Available patterns

In the current GLIDE implementation, three patterns are available (this is sufficient for our *CS1* course), they are illustrated in Fig. 3.2. All these three patterns follows the guidelines provided in Sec. 2.2 about representing data structures:

- The graduated line (see Sec. 2.2.1). The ticks of the line that is going to be drawn are prompted to the user who may type specific ones as a string of values separated by semi-

colons. Inserting dots between two semicolons creates a gap in the line. Fig. 3.2 shows such a line for which the user would have entered the string “`0;1;2;3;...;i-1;i;...;n-1;n`” (which is the default value pre-written in the prompt window);

- The text that allows to represent a large number of problem (any text content can be typed in the light grey zone)and, of course, the best way to depict strings. Fig. 3.2 shows how to use the text to represent a polynomial thanks to subscripts and superscripts. A sequence of the form “`a_b`” will be drawn in the canvas as “a_b” and a sequence of the form “`a^b`” will be drawn in the canvas as “a^b”. For now, subscripts and superscripts cannot be mixed nor nested.
- The 1-dimension array. The pattern consists of the typical rectangular shape of the array (see Sec. 2.2.3), surrounded by two vertical bars used to precise the array first and last indices (or the array size).

A given Graphical Loop Invariant may include multiple patterns (*e.g.*, several arrays or an array and a text). One just have to click several times on the ADD button to draw as many times the selected pattern.

3.1.1.1 Future Work

The list of available patterns can be extended. The first candidates are the other data structures presented in Sec. 2.2: the lists, queues, stacks, etc.

3.1.2 Dividing Lines

The user can click on a pattern drawn in the canvas to add a movable Dividing Line and label it (the cursor shape changes into ✛). The array pattern exhibits also vertical lines but they are not Dividing Lines and so they cannot move. All bars sharing a common variable name in their label are coloured in red when hovered, regardless of the pattern they belong to. This allows us to document several patterns with the same variable (*e.g.*, for browsing two arrays with the same index position).

3.1.3 Defining zones

Two bars[18] can be linked with an horizontal coloured bar, defining so a zone. The legend of the zone must be provided in the box of the same colour. When defining a zone, its colour can be chosen. By default, the colours of consecutive defined zones are selected froma circular queue of 20 distinct hues [183], easily distinguishable, even by colour-blind people.

An example of fully labelled Graphical Loop Invariant is provided in the Fig. 3.4a.

[18] Here, by “bar”, we mean either a Dividing Line or a fixed vertical bar that forms the boundary of an array. Both ends of the line and text patterns may also be selected to create a zone

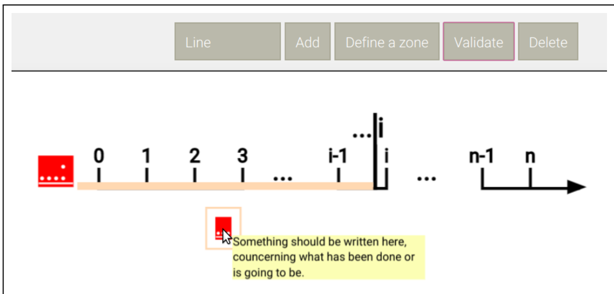


Figure 3.3: GLIDE screenshot. Missing items or errors are mentioned in red. More details about the problem and how it can be solved is provided by a tooltip.

3.1.4 Editing a Graphical Loop Invariant

3.1.4.1 Deleting elements

Dividing Lines and zones can be deleted by clicking on the DELETE button when selected or by pressing the `delete` key of the keyboard.

It is worth noting that the patterns cannot be deleted in this way. In order to suppress them, one have to restart from a blank canvas. Rule 1 of our methodology states to draw a shape that is relevant for the problem. Using the wrong pattern thus means that the problem is either not clearly defined or not well understood by the programmer. Restarting from a blank sheet and correct one's problem definition is always a good solution if a problem arises at this first programming step.

3.1.4.2 Modifying elements

The graduated lines may be redrawn with new ticks after double clicking on the triangle that forms the arrow of the line (the cursor shape changes into the cell shape (*i.e.*, ✚ when hovering the arrow).

All the text contents can be edited (the cursor shape changes into the text editing symbol, *i.e.*, I). The zones labels can be dragged to any position in the canvas. The horizontal position of a zone colour bar can be adjusted at will (the cursor shape changes into ✋).

3.1.5 Graphical Loop Invariant Validation

The application can perform several checks ensuring the methodological guidelines (see Sec. 2.1.2) are respected. Table 3.2 resumes the tests that are performed to validate a drawing and provides, for each test, the guideline that is enforced. If one of the tests fails, an error is reported in red in the canvas and a tooltip appears when the mouse hovers the red zone, as depicted in Fig. 3.3.

It is worth noting that these tests can only challenge the syntactical part of the guidelines (*e.g.*, the presence of a particular item) and not their semantics (*e.g.*, the soundness of a particular drawing with respect to the problem actually being tackled). Nevertheless, the tool was designed

Table 3.2: GLIBP guidelines and the corresponding verifications performed by GLIDE

Guideline	Verification performed
Rule 1	All the patterns must be named thanks to the text label at their left.
Rule 2	The boundaries of the array patterns must be provided (the boundaries of the graduated lines and texts are always present by design).
Rule 3	The presence of at least one Dividing Line is ensured by the need of at least two different zones.
Rule 4	All the DLS must be labelled at their left or at their right. If both labels are used, a symbolic computation verifies that they represent consecutive integers (*e.g.*, $N-i-1$ and $-i+N$ are supposed by the program to be consecutive integers).
Rule 5	There should be at least one zone that indicates what was previously computed. GLIDE does not understand the zone legend but consider all string that do not start with "To be" or composed of question marks as compliant.
Rule 6	One zone that designates what should be done. Its legend must start with "To be" or be fully composed of question marks to be considered by GLIDE as compliant. In order to be user-friendly, the french regular expression that handles the verification of this rule is even more complex and tolerates the common grammar mistake consisting in confusing infinitive and past participle of first group verbs.

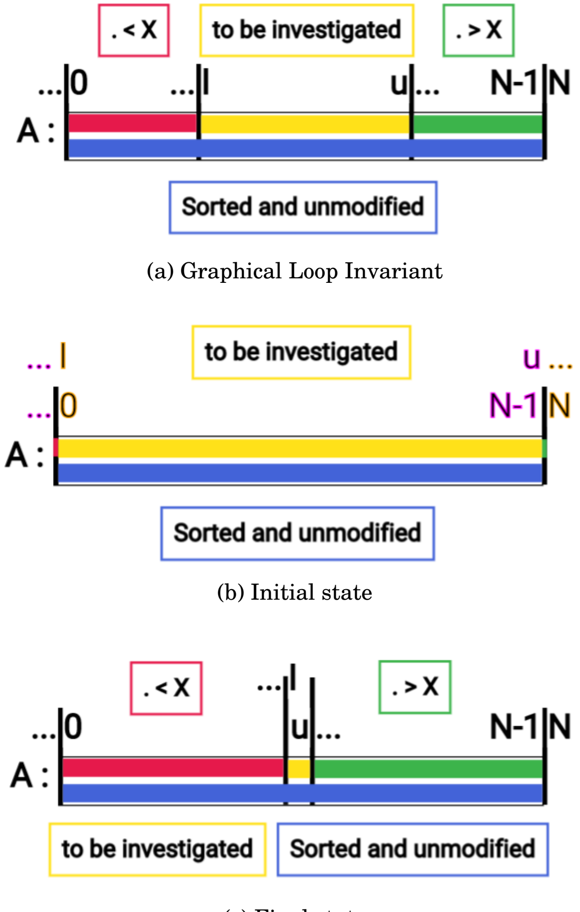
to give students quick *feedback* on the form of their Graphical Loop Invariant and serve as a reminder in case of missing items.

3.1.6 Writing the Code with GLIDE

Once the validation step is passed without error, GLIDE can be used to write the associated piece of code, as explained in the Sec. 2.1. The DLS can be actually moved to transform the Loop Invariant into the initial step (ZONE 1) and the final step (ZONE 3). To move a Dividing Line, one just has to point to it with the mouse and maintain the left click during the displacement (the cursor shape is changed into ✊).

From the initial steps, the initial values of the variables can be deduced, GLIDE can even highlights the matching of two overlapping bars, as illustrated in Fig. 3.4b. The final state illustrates the Loop Condition, as shown in Fig. 3.4c.

Is is also possible, thanks to symbolic computation, to ask GLIDE to compute the length of a selected zone by pressing on the "T" key (T is for *Termination*). Doing so for the "to do" zone of a Graphical Loop Invariant often gives the Loop Variant of the corresponding code. However, this functionality was never unveiled to the students to not make them rely too much on a software (we took the decision after having watched all the seasons of *Mayday: Air Disaster* TV show [39] that relates too much air crashes due to over-confidence in the autopilot).



(a) Graphical Loop Invariant

(b) Initial state

(c) Final state

Figure 3.4: Illustration of GLIDE capabilities when writing the code

3.2 Blank Graphical Loop Invariant for automatic assessment

In our CS1 course, small programming exercises that are automatically corrected have been introduced (see Chap. 9). As most of these exercises consist in writing loops and as the course requires to write loops based on Graphical Loop Invariants (see Sec. 2.1), these exercises 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. The following details an example of Blank Graphical Loop Invariant. The chapter 5 addresses an evaluation of the impact of the Blank Graphical Loop Invariant on student's learning.

3.2.1 Example of Blank Graphical Loop Invariant

Let us take as an example the problem consisting in finding the intersection of two sets (stored in sorted arrays):

ARRAYINTERSECTION:

Input – Two sorted arrays of integers `A`, of size `N`, and `B`, of size `M`, and a empty array `C` of size `L` large enough (*i.e.*, `L` = min(`M`,`N`))

Output – `C` contains the integers common to `A` and `B` and the function returns the number of elements in `C`

An example of Blank Graphical Loop Invariant is provided in Fig. 3.5. The instructions state that the numbered boxes 1. to 15. should be replaced by variables, constants names, or left blank.

Table 3.3: Available choices for the boxes **16** to **28**. The † refers to the sentence at the bottom of Fig. 3.5.

For the boxes 16 to 24	**For the box 25**	**For the boxes 26 to 28**
1. sorted	1. different from;	1. Zone A1
2. sorted and unmodified	2. found in;	2. Zone A2
3. unmodified	3. browsed in;	3. Zone B1
4. to be investigated	4. analysed in;	4. Zone B2
5. available space	5. singular to;	5. Zone C1
6. †	6. alien to;	6. Zone C2
7. (nothing)	7. common to.	7. Zone 51

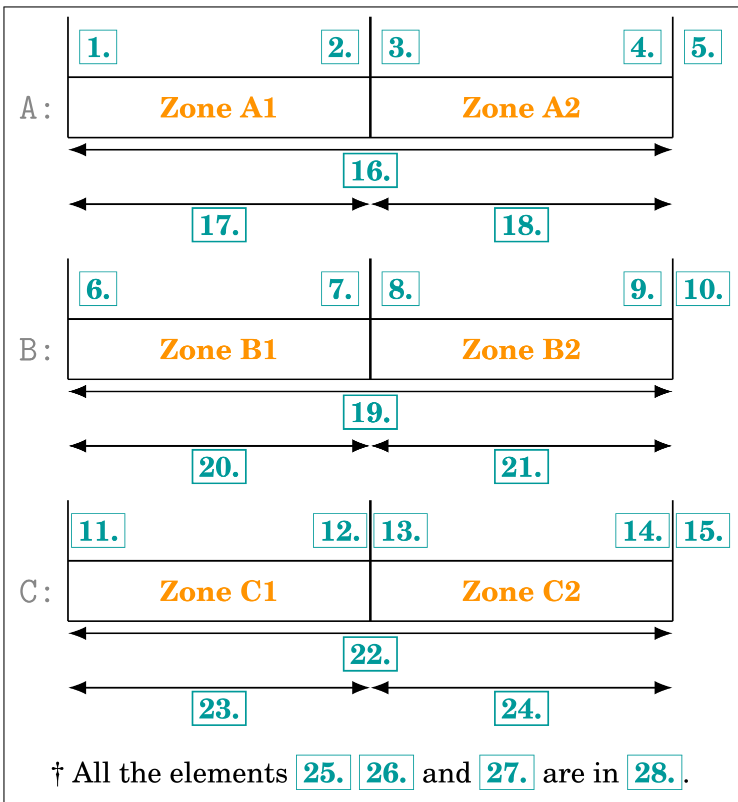


Figure 3.5: Blank Graphical Loop Invariant for the problem ArrayIntersection

The boxes 16. to 28 have to be replaced by a number in [1..7] corresponding to the multiple choices presented in Table 3.3, or left blank. The multiple choices contain some inconsistent possible answers. The bottom line of Fig. 3.5 contain a sentence that should eventually be used to express what was already achieved by the previous iterations.To ease understanding, we recommand to always add a mock example of such a replacement in the exercises instructions.

Designing a Blank Graphical Loop Invariant is a difficult task: one must find an equilibrium between, on one hand, not giving too much information in the blank drawing that would render the exercise too easy or limit students' creativity and, on the other hand, giving too much possibilities (e.g. by increasing the box numbers) that would make the exercise either over-complicated, either much more difficult to automatically correct, since all the correct possibilities of box replacements must be considered during the correction. For example, in Fig. 3.5, we fixed the positions of the array `A`, `B` and `C`.

By doing so, we made the drawing respect the Rule 1 of our methodology's guidelines (see Sec. 2.1.2). The Rule 3 is also implicitly enforced since the Fig. 3.5 already exhibits the Dividing Lines.

The Blank Graphical Loop Invariant can be related to instructional scaffolds, definded in [68] as "temporary support structures faculty put in place to assist students in accomplishing new tasks and concepts they could not typically achieve on their own", *i.e.*, here, we propose exercises

Table 3.4: An example of a student's answer. The drawing corresponding to the 28 substitutions listed in this Table is provided in Fig. 3.6

1. 0	8. _	15. L	22. 7
2. i	9. _	16. 2	23. 6
3. _	10. M	17. 7	24. 5
4. _	11. 0	18. _	25. 7
5. N	12. k	19. 2	26. 1
6. 0	13. _	20. 7	27. 3
7. j	14. _	21. 7	28. 5

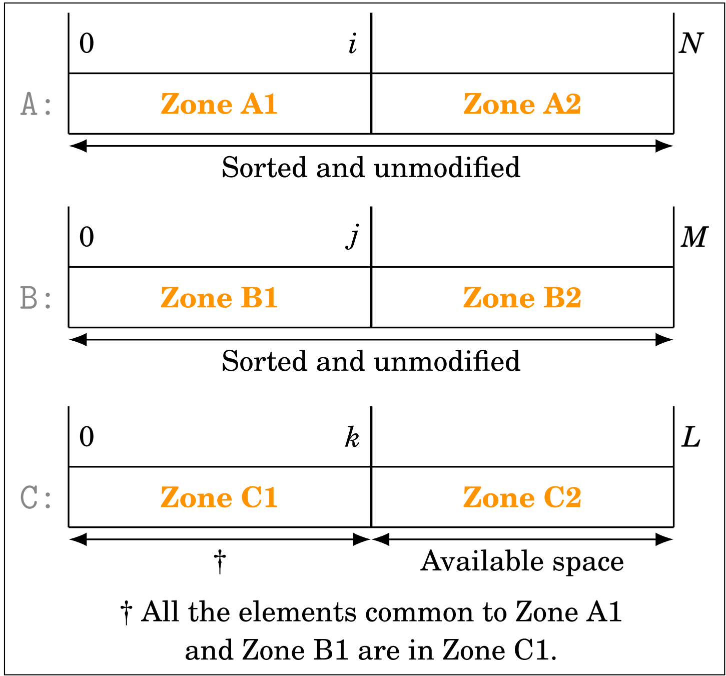


Figure 3.6: Blank Graphical Loop Invariant for the problem ARRAYINTERSECTION filled with the student answers listed in Table 3.4

that do not ask students to draw an Graphical Loop Invariant from scratch but offering them a correct template they just have to fill in.

Table 3.4 shows what a student's answer looks like: each number corresponds to a coloured box in Fig. 3.5. The Graphical Loop Invariant thus specified is shown in Fig. 3.6. Some simple expressions (*e.g.*, $k+1$ and $N-j$) may be specified as indices. The box content left blank (*e.g.*, box 4 or 18) means that nothing must be drawn. How such a filled in Graphical Loop Invariant can be submitted to an automatically assessed is the subject of Chap. 7.

CHAPTER 4

ERRORS TAXONOMY AND GLIBP METHODOLOGY RECEPTION

THIS chapter tackles the students reception of the GLIBP methodology by answering to the following research questions:

RQ 1. 1 *How the students seize the opportunity to practice the* GLIBP *methodology?*
RQ 1. 2 *What kind of error are committed using the* GLIBP *methodology?*
RQ 1. 3 *Can we link the error committed with the* GLIBP *methodology to programming errors?*
RQ 1. 4 *How the* GLIBP *methodo is perceived by the students?*

The chapter is organized as follows: we first propose a taxonomy of the errors committed by students when using Graphical Loop Invariant in Sec. 4.1. Then, Sec. 4.2 describes the methodology we used to answer the research questions. Sec. 4.2 presents our results and Sec. 4.4 them. Finally, Sec. 4.5 concludes this chapter.

4.1 Errors Taxonomy

In order to analyse the errors committed while using the GLIBP methodology, we built a list of performance markers that indicate whether a particular Graphical Loop Invariant, Loop Variant, or code are in line with our methodology or not. This list is shown in Table 4.1 that is divided in three main parts corresponding of what is assessed: the Graphical Loop Invariant, the Loop Variant, or the code. Each performance marker is labelled with an identifier (see column "ID" in Table 4.1) to easily refer to each marker in the text and in plots. The following details the rationale of the taxonomy we propose.

Table 4.1: Errors Taxonomy for the GLIBP methodology

Point of Interest		ID	Description	Rule(s)
Graphical Loop Invariant	Syntax	GLI-SY_1	The Graphical Loop Invariant is syntactically correct (Rule 1 → Rule 4 met)	
		GLI-SY_2	The drawing is not labelled (with variable)	Rule 1
		GLI-SY_3	The drawing has no boundaries	Rule 2
		GLI-SY_4	The drawing has no *Dividing Line*	Rule 3
		GLI-SY_5	The Dividing Line(s) is (are) not documented with variable(s)	Rule 4
	Semantic	GLI-SE_1	The Graphical Loop Invariant is semantically correct (Rule 1 and Rule 5 → Rule 7 met)	
		GLI-SE_2	The Graphical Loop Invariant is unrelated to the problem to be solved	Rule 1
		GLI-SE_3	The drawing has no sense w.r.t. the problem to be solved	Rule 1
		GLI-SE_4	The Graphical Loop Invariant does not contain information on what has been achieved so far by previous loop iterations	Rule 5
		GLI-SE_5	The Graphical Loop Invariant does not contain information on what remains to compute	Rule 6
		GLI-SE_6	There is no relationship between variables in the Graphical Loop Invariant and in the code	Rule 7
Loop Variant		LV_1	Correct	
		LV_2	Not provided	
		LV_3	Incorrect	
		LV_4	The Loop Variant is not an integer function	
		LV_5	The Loop Variant is not decreasing at each iteration	
Code		CODE_1	Correct and built upon the Graphical Loop Invariant	
		CODE_2	Not provided	
		CODE_3	The piece of code does not solve the problem	
		CODE_4	Buffer overflow	
		CODE_5	Incorrect ZONE 1 (variable unrelated with Graphical Loop Invariant)	
		CODE_6	Incorrect ZONE 1 (variable not initialized according to Graphical Loop Invariant)	
		CODE_7	Incorrect Loop Condition	
		CODE_8	Infinite Loop	
		CODE_9	Incorrect ZONE 2	
		CODE_{10}	Incorrect ZONE 3	

4.1.1 Graphical Loop Invariant

We started from the guidelines we presented in Sec. 2.1.2. For six rules over seven, breaking the rule corresponds to a specific error (see Table 4.1, last column). As can be seen in the Table, we distinguished two categories of possible errors, *i.e.*, syntactic and semantic ones that come from the rules categorisation we introduced in Sec. 2.1.2.

Concerning the Rule 1 (*i.e.*, the drawing shall correspond to the problem and be labelled), it may be broken in three different situations: either the label of the drawing is lacking or incorrect (GLI-SY_2), or the shape of the drawing does not correspond to the problem to be solved (GLI-SE_2 and GLI-SE_3). The first way is rather a syntax error while the second and third ones are rather semantic issues. Note that the difference between GLI-SE_2 and GLI-SE_3 is subtle: GLI-SE_2 refers to a Graphical Loop Invariant that is correct but unrelated to the problem to be solved (*e.g.*, the problem is about to sort an array while the Graphical Loop Invariant is about a binary search). GLI-SE_3 focuses on the drawing that is unrelated to the problem (*e.g.*, a problem that ask to draw a square on the standard output, while the Graphical Loop Invariant illustrates an array).

In addition to these eight markers, we added two more to label syntactically (GLI-SY_1) and semantically (GLI-SE_1) correct Graphical Loop Invariants.

4.1.2 Loop Variant

The performance markers associated with the Loop Variant directly come from its definition. A Loop Variant-candidate is considered incorrect if it is not an integer function (LV_4 – *e.g.*, if the Stop Condition of the loop, which has a Boolean value, is proposed) or if its value is not decreasing as the iterations go (LV_5). LV_3 refer to the other incorrect cases, *e.g.*, the Loop Variant value is negative while the Loop Condition is true or a function containing variables that are not present in the Graphical Loop Invariant or in the code. We also added a marker to use when the Loop Variant is not provided (LV_2) and a last to label correct Loop Variants (LV_1).

4.1.3 Code

In Sec. 2.1.3, we divided the code of a loop into three zones and our taxonomy reflects this scheme. In ZONE 1, the variables may be not related to the Graphical Loop Invariant or to the problem to be solved (CODE_5). They also may not be initialised according to the Graphical Loop Invariant or in a way that solves the problem (CODE_6). CODE_7 focus on the Loop Condition correctness. CODE_9 (resp. CODE_{10}) mentions an error in ZONE 2 (resp. ZONE 3). We added a marker for when the code is absent (CODE_2) and one for codes that do not solve the problem (CODE_3). We also added markers for common errors such as buffer overflow (CODE_4) and infinite loops (CODE_8).

4.2 Methodology

In order to evaluate the students reception of the GLIBP methodology, we follow the "3 P's framework" of Verpoorten et al. [185] which recommends consistent analysis of any pedagogical innovation by gathering and meshing three types of data that reflect aspects of the students' learning experience: Participation, Performance, and Perception. All chapters presenting results in the remainder of this document will also be organised in the same way.

In the next sections, we present results and discuss them to answer the following questions:

RQ 1. 1 *How the students seize the opportunity to practice the* GLIBP *methodology?*

RQ 1. 2 *What kind of error are committed using the* GLIBP *methodology?*

RQ 1. 3 *Can we link the error committed with the* GLIBP *methodology to programming errors?*

RQ 1. 4 *How the* GLIBP *methodo is perceived by the students?*

The answer to the RQ 1. 1 requires to analyse participation data (see Sec. 4.2.1). The answers to the RQ 1. 2 and RQ 1. 3 use mainly performance data (see Sec. 4.2.2). Finally, the answer RQ 1. 4 is based on perception data (see Sec. 4.2.3).

Both participation and performance data were collected during the academic year 2019–2020. During the year 2019-2020, 82 students were enrolled for our *CS1* course, 71.2% of them were in their first year at the university (the other either repeated the year, either reoriented from another curriculum).

As far as the perception data is concerned, we collected it from academic years 2017–2018 to 2019–2020.

4.2.1 Learning Analytics

During the semester, various PA are provided to students in addition to classic practical sessions (exercises and labs). Two types of PA are proposed: those that are automatically graded with a tool we designed (see Chap. 7) and those that are manually graded by the educational team.

On the total of five PA graded automatically over the semester, three of them focus on Graphical Loop Invariant based programming (For more information about these PAs, see Chap. 9). For each PA graded automatically, each student works in isolation, on their own computer, and can submit their solution up to three times (a message containing feedback and feedforward are received for each submission). In addition, students are helped in the Graphical Loop Invariant construction through the Blank Graphical Loop Invariant (see Sec. 3.2).

The manually graded PA refers to the Mid-Term exam, typically organised around the end of October or the beginning of November and the Final Exam, organised in January. Those manually graded PA correspond to exercises on paper, without any kind of help (no computer, no Blank Graphical Loop Invariant).

Table 4.2: Programming activities (Programming Activities (PA)) organized during the semester and for which we can assess the usage of the Graphical Loop Invariant. The column "Grading" provides information on the way the PA was evaluated. Automatic refers to automatic grading with blank Graphical Loop Invariant provided. In that case, students can submit their code up to three times, each submission providing *feedback* and *feedforward*. Manual means that students do the PA on paper, without any help, and is manually graded by the educational team. The column "ID" refers to identifier used for each PA in text and plots. Some PAs requires multiple loops, this information being provided by column "Details".

#	Grading	Topic	ID	Details
1	Automatic	Drawing geometrical figure on standard output based on particular characters	PA_1 PA_2	Outer loop Inner loop
2	Manual	Counting the number of '1' in binary representation of numbers $\in [1; n]$	PA_3 PA_4	Outer loop Inner loop
3	Automatic	Compressing an integer array into an other	PA_5	A single loop
4	Automatic	Twins prime numbers (n and p, prime and $n > p$, are said **twins** if $n - p = 2$)	PA_6 PA_7	`is_prime(x)` function `display_twins(x)` procedure
5	Manual	Luhn algorithm with an array	PA_8	A single loop

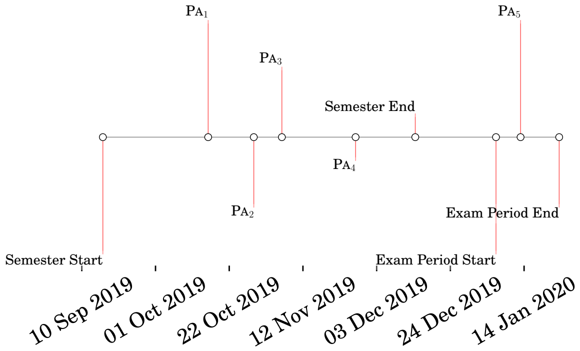


Figure 4.1: Positioning of PAover the semester.

Table 4.2 summarizes the various PA, while Fig. 4.1 positions the PAover the semester.

Our Graphical Loop Invariant Drawing Editor (GLIDE) (see Sec. 3.1) requires a registration based on students' University ID. We monitored students' GLIDE usage over the semester by logging connection attempts, connection duration, and usage of the tool (i.e., Graphical Loop Invariant drawings and Graphical Loop Invariant validation).

The PAsubmission (both automatic and manually graded) with connection to GLIDE correspond to the **Participation data** of the 3 P's framework [185].

Table 4.3: Surveys Respondents. N is the total number of students. # is the number of respondents. Two percentages are provided: %(Total) is computed on the total number of enrolled students and %(Part.) on the number of Participants in the Final Exam

		Respondents		
Year	**N**	**#**	**% (Total)**	**% (Part.)**
2017–2018	72	28	38.9	50.0
2018–2019	76	22	28.9	47.8
2019–2020	82	16	19.5	26.7

4.2.2 Performance Tests

Each PA(see Table 4.2) comes with three pieces of information: the Graphical Loop Invariant, the Loop Variant, and the code written in C. This allows us to assess students' performance according to the error taxonomy presented in Sec. 4.1.

For PAgraded automatically, the output of the program we use (see Chap. 7) comes with information allowing us to map their performance with markers provided in Table 4.1. For PA manually graded, the educational team manually maps students production with the markers.

Also, each time the students make use of GLIDE, in particular by clicking on the "Validate" button, several tests are launched in order to check whether the Graphical Loop Invariant respects the rules of tha GLIBP methodology (See Sec. 2.1), mainly those related to the syntactical aspect of the Graphical Loop Invariant (GLI-$\text{SY}_{1\to5}$ in Table 4.1) as well as the presence of a zone describing what has already been achieved so far (GLI-SE_4 in Table 4.1) and what remains to compute GLI-SE_5 in Table 4.1). The output of the validation process on the GLIDE has been logged.

All of these correspond to the **Performance data** of the 3 P's framework [185].

4.2.3 Surveys

We conducted two sort of surveys that we call the **one-time** surveys and the **long-term** survey. Both surveys were anonymous, to let the student express themselves freely. This dataset corresponds to the **Perception data** of the 3 P's framework [185].

4.2.3.1 One-time Surveys

First, from academic years 2017–2018 to 2019–2020 we surveyed students having followed our CS1 course, after the Final Exam, during the second semester. Table 4.3 presents the numbers of respondents each year and compares them to other course statistics. The number of answers we received may appear as few if they are compared to the total number of students. However, the last column of Table 4.3 provides the proportion of respondents computed on the number of participants in the final Exam. In 2017–2018 and 2018–2019, one can see in the Table that

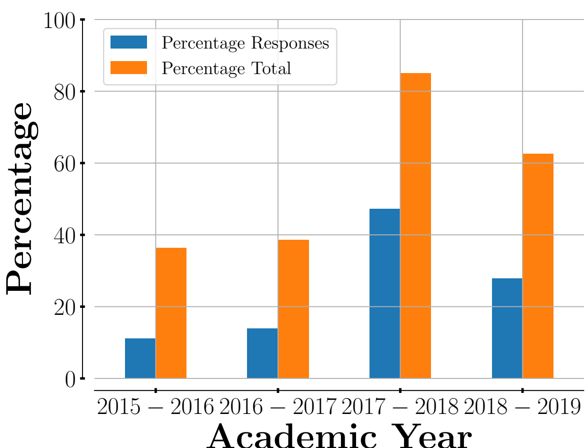


Figure 4.2: Participation rate to the long-term survey (36 answers over 60 surveyed students). The orange bar corresponds to the percentage of respondents in the corresponding Academic Year, while the blue bar refers to the percentage of respondents in the overall 36 answers.

nearly half of them answered the surveys. As far as 2019–2020 is concerned, we have received 16 answers (to be compared with 82 enrolled students). But, when looking at the number of students who still follow their curriculum during the second semester (roughly 30 students[19]), 16 answers represents half of them.

4.2.3.2 Long-term Surveys

We surveyed students who have followed our CS1 course during the past years. Over the 60 surveyed students, we received 36 answers (60% of respondents). Those answers are distributed as shown in Fig. 4.2.

4.3 Results

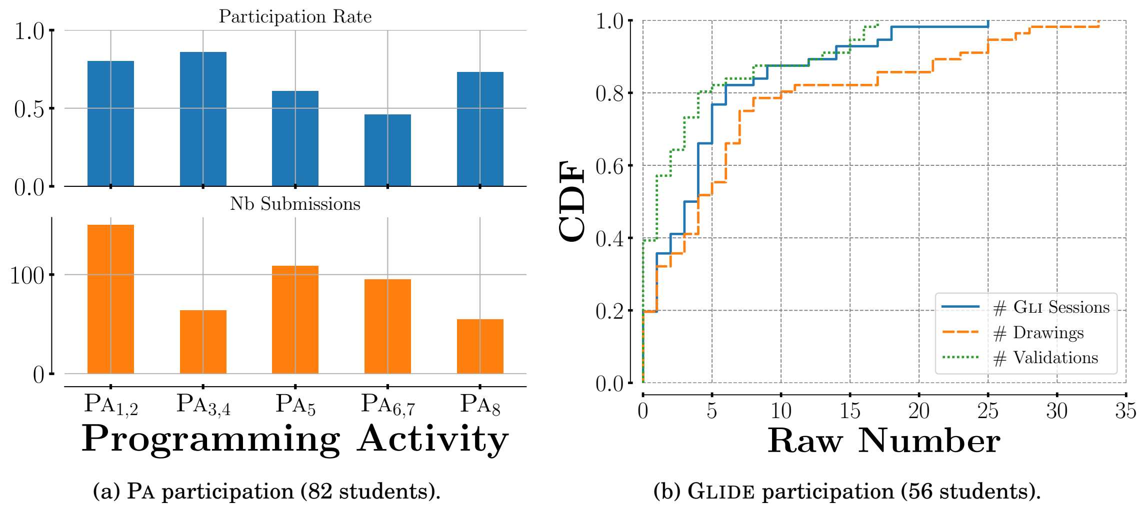
4.3.1 Participation Data

Fig. 4.3 shows participation data. In particular, Fig. 4.3a focuses on PA participation (participation rate – top plot – and number of submission – bottom plot – per PA), while Fig. 4.3b focuses on the GLIDE tool.

The top plot of Fig. 4.3a presents the participation rate to the PA (PA_i). The translation of the code associated with each PA is available in Table 4.2. 80% of the students took part in $PA_{1,2}$, 86% in $PA_{3,4}$, 61% in PA_5, 46% in $PA_{6,7}$, and 73% in PA_8.

The bottom plot of Fig. 4.3a presents the number of submissions per PA. As the automatically graded PA's are concerned, the students are allowed to submit up to three times. One can see

[19] Unfortunately, it is usual in our country to have a high attrition rate (see Chap. 11)



(a) PA participation (82 students).

(b) GLIDE participation (56 students).

Figure 4.3: Participation results.

that they were 150 submissions for $PA_{1,2}$, 64 for $PA_{3,4}$, 109 for PA_5, 95 for $PA_{6,7}$, and 55 for PA_8.[20]

Fig. 4.3b describes students' usage of GLIDE. It is worth noticing that only 56 students (68.3% of the total) have used GLIDE (its usage has not been made mandatory).

If we look first at the distribution of the drawings (orange curve), we see that 20% of the students have never used the tool, i.e., they only created an account. In addition, half of the users submitted less than five drawings over the semester, 20% submitted ten drawings or more while, finally, the remaining 10% submitted more than 25 drawings.

The blue curve in Fig. 4.3b provides the distributions of **sessions**, i.e., a time frame during which a student is connected on the GLIDE, draws Graphical Loop Invariants, and possibly validates them. The average session lasts 46min (minimum being 0min, 1st quartile (P25) being 1min, 2nd quartile (median) begin 3min, 3rd quartile (P75) being 13min, max being 27h). As the distribution is concerned, one can see that a large majority (70% of the users) used the GLIDE for less than five sessions.

Finally, the green curve in Fig. 4.3b illustrates how students use the GLIDE for validating their Graphical Loop Invariant, i.e., how often they click on the "Validate" button (see Sec. 3.1). This button must be used in order to validate a drawing and, then, to use the resulting picture as a Loop Invariant to write the related piece of code. One can see that 40% of the users never requested for a validation of their drawings (they just left the website and the GLIDE recorded their performance). Another 40% of the users tried to validate five drawings or less. The rest of the users (20% of them) validated five drawings or more.

[20] The number of submissions does not match with participation rate for $PA_{3,4}$ and PA_8. This is due to the fact that the (raw) number of "submission" may be lower than the students having participated to the PA (i.e., number of participants > number of responses for the dedicated questions).

4.3.2 Performance Data

Fig. 4.4a shows the distributions of the errors made by the students over all PA with respect to performance markers (see Table 4.1). Looking at the orange curve (GLI-SE$_{2\to6}$), one can see that 60% of the Graphical Loop Invariant over all PA does not contain any semantic errors. 20% of them contain at least one semantic error and the remaining 20% contains two semantic errors or more.

The blue curve (GLI-SY$_{2\to5}$) shows that nearly 40% of the Graphical Loop Invariant does not contain any syntactic error. Another 40% of the Graphical Loop Invariant contains one error and the remaining 20% contains two errors ore more.

The green curve (LV$_{2\to5}$) shows that more than half of the submissions have a correct Loop Variant.

The red line (CODE$_{2\to10}$) follows roughly the same trend as the blue one. 45% of the pieces of code submitted are correct, nearly 40% of the code contains one error, and the remaining 15% contains two errors ore more.

Fig. 4.4b provides the distributions of errors, with respect to performance markers, for the GLIDE tool. The errors detected by the GLIDE are mostly the syntactic ones.

The blue and orange lines overlap. They both show that 70% of the drawings are free from GLI-SY$_2$ and GLI-SY$_3$ errors. 25% of the drawings contains one error and the remaining 5% contains more than two errors.

The green line shows that most of the dividing lines were labelled: more than 80% of the drawings does not contain any GLI-SY$_5$ error. 10% of the drawings contains one error. The remaining 10% contains two errors or more, with the maximum value in a same drawing being 79 (The student actually drew 79 dividing lines without labeling them).

The red line indicates that 65% of the drawings does not contain any GLI-SE$_4$ error. 20% of them contains one error, 10% of them contains two error, and the remaining 5% contains three errors or more.

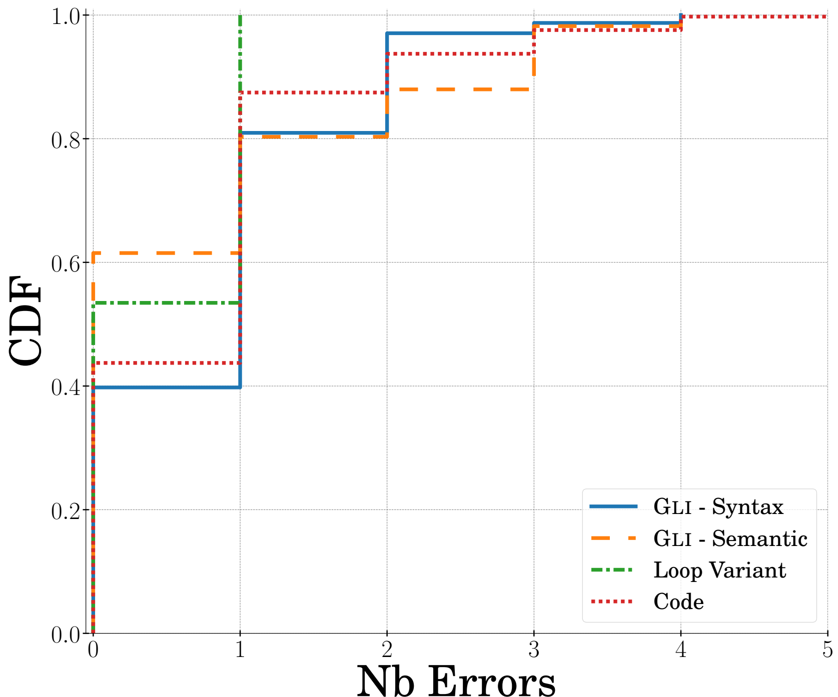
As purple line presents that nearly 50% of the drawings are free from GLI-SY$_5$ errors while 25% of them contains one error, 20% of them contains two errors and the 5% remaining contains three errors or more.

4.3.2.1 Markers Distribution by PA

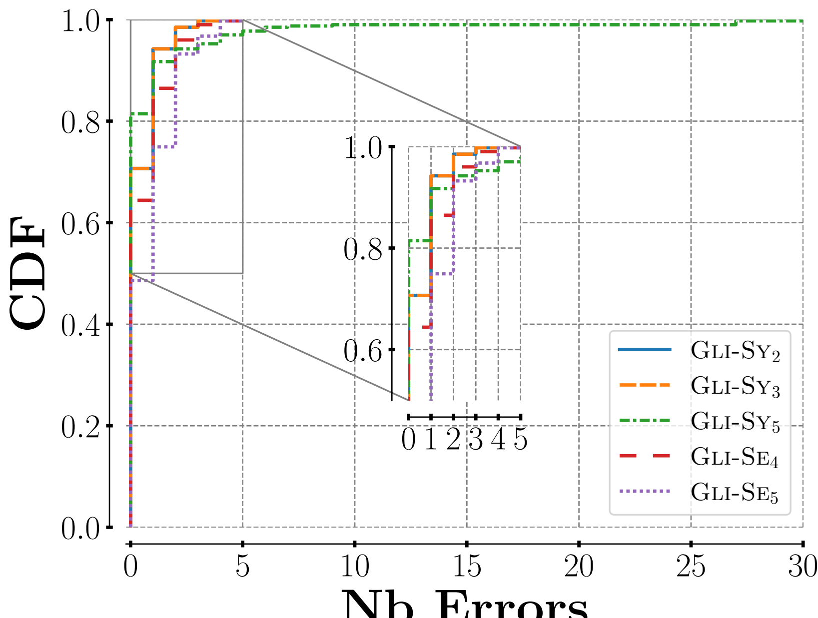
The results fromFig. 4.4a can be deepen by looking at the markers distributions for each individual PA. This can be achieved with Fig. 4.5

Fig. 4.5a focuses on the Graphical Loop Invariant syntax markers. The GLI-SY$_1$ (Syntax is correct) is the most frequent marker for PA$_7$ (71%) and for nearly half of the PA$_1$ submissions (46%). For other PA (PA$_{2\to6}$), the GLI-SY$_1$ is low.

GLI-SY$_2$ (Unlabelled drawing) is the most frequent marker for both PA$_3$ and PA$_4$ (resp. 79%

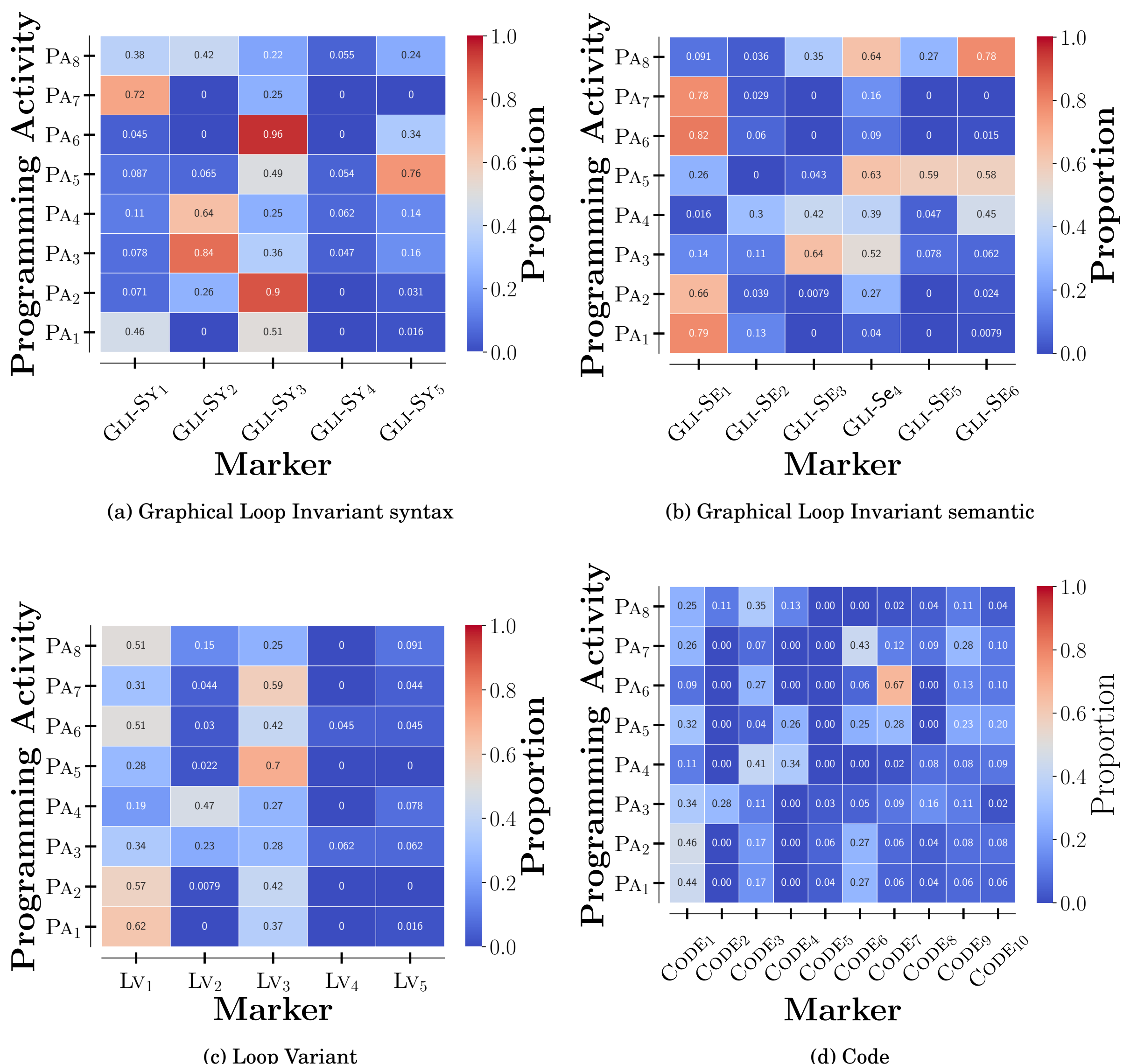


(a) Distribution over all PA. It concerns GLI-SY$_{2\to5}$, GLI-SE$_{2\to6}$, LV$_{2\to5}$, and CODE$_{2\to10}$.



(b) Distribution for the GLIDE tool. The X-Axis is trimmed at 30 but the maximum value for GLI-SY$_5$ is 79.

Figure 4.4: Cumulative error distribution with respect to performance markers.



(a) Graphical Loop Invariant syntax

(b) Graphical Loop Invariant semantic

(c) Loop Variant

(d) Code

Figure 4.5: Performance marker distribution for Graphical Loop Invariant (syntax and semantic), Loop Variant, and code produced by students during each PA.

and 83%) and quite prevalent for the PA$_8$ (43%).

GLI-SY$_3$ (Absence of boundaries) is the most prevalent marker for PA$_1$ (50%), PA$_2$ (88%), and PA$_6$ (96%). This error is also present in more than 20% of the submissions of the others PA, especially nearly half (49%) of those for PA$_5$.

The marker GLI-SY$_4$ (No dividing line) is very uncommon for all PA. But, while dividing lines are present in most of the drawings in each PA, they are sometimes not properly labeled, as the marker GLI-SY$_5$ (Unlabelled dividing line) is the most prevalent for PA$_5$ and quite common to PA$_6$ (34%) and PA$_8$ (25%).

Fig. 4.5b displays the distribution of the Graphical Loop Invariant semantic markers. One can see that GLI-SE$_1$ (Semantic is correct) is the most prevalent marker for PA$_1$, PA$_2$, PA$_6$, and PA$_7$.

The marker GLI-SE$_2$ (No relation to the problem) is quite uncommon for all PA, except PA$_4$ (37%).

A trend can be seen for PA$_{3,4,8}$ (the Mid-Term and the Final Exam). The marker GLI-SE$_3$ (Drawing has no sense) is very common (resp. 63%, 52%, and 36%), as well as the marker GLI-SE$_4$ (No info about what has been achieved) (resp. 51%, 48%, and 66%). As GLI-SE$_6$ (Relationship with the code) is concerned, it is the most prevalent marker for both PA$_4$ and PA$_8$.

Regarding PA$_5$, that tackles Loop Invariant describing algorithm on arrays, the most frequent markers are GLI-SE$_4$ (62%), GLI-SE$_5$ (No info about what remains to compute – 59%), and GLI-SE$_6$ (No relationship with the code – 58%).

Fig. 4.5c shows that the Loop Variant is either correct (LV$_1$) or incorrect (LV$_3$) for most of PA. This marker distribution goes from resp. 62% and 36% for PA$_1$ to resp. 27% and 70% for PA$_5$. For PA$_{3,4}$ (Mid-term Exam), a large number of Loop Variants were not provided (resp. 22% and 29%).

Fig. 4.5d presents the markers related to the code. Over all the PA, one of the most prevalent marker is CODE$_1$ (Code is correct), especially for PA$_1$ (44%), PA$_2$ (46%), PA$_4$ (34%), and PA$_5$ (30%). CODE$_3$ (Code does not solve the problem) is the most frequent marker for PA$_4$ (40%) and PA$_8$ (36%) that are both related to Exams (resp. Mid-term and Final one). CODE$_6$ (Incorrect zone 1) is the most prevalent marker for PA$_7$ (43%) and the marker CODE$_7$ (Incorrect Loop Condition) is the first marker for PA$_6$ (67%). Regarding PA$_5$, several markers are above 20%: CODE$_4$, CODE$_6$, CODE$_7$, CODE$_9$, and CODE$_{10}$.

4.3.2.2 Correlation between Program Quality and Graphical Loop Invariant

One of the most interesting question concerning performance data is: does the Graphical Loop Invariant based programming lead to better quality programs? Fig. 4.6 shows several probabilities computed for each PA. The variable A refers to a correct piece code while B refers to a correct Graphical Loop Invariant. Thus $P(A|B)$ is the probability to have a correct piece of code if the Graphical Loop Invariant is correct and $P(\overline{A}|\overline{B})$ is the probability to have an incorrect piece of code if the Graphical Loop Invariant is incorrect. It is worth noticing that the Graphical Loop Invariant is likely to be noted as incorrect since one semantic or syntactic error is enough to consider it as incorrect. The same rule is applied regarding the code. This explains the small $P(A|B)$ values.

4.3.3 Perception Data

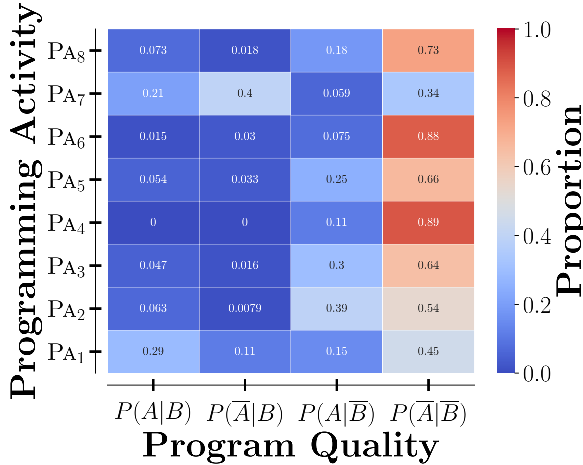


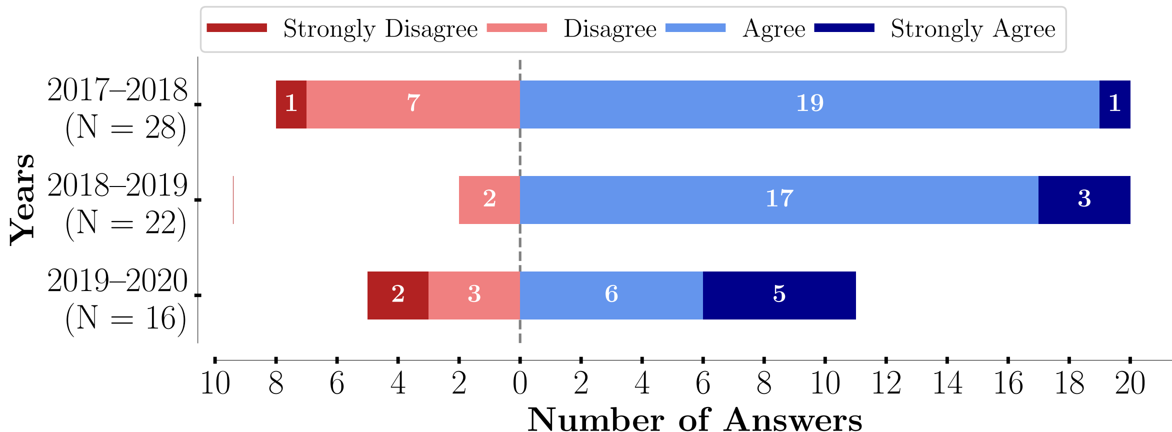
Figure 4.6: Program quality (X-Axis) over the various PA (Y-Axis). A refers to a correct code (CODE$_1$) produced ($\overline{A}$ being therefore an incorrect code – i.e., at least one of the CODE$_{2\to10}$ has been marked). B refers to a correct Graphical Loop Invariant (GLI-SY$_1$ and GLI-SE$_1$) ($\overline{B}$ being therefore an incorrect Graphical Loop Invariant – i.e., at least one of the GLI-SY$_{2\to5}$ or the GLI-SE$_{2\to6}$ has been marked). $P(\ldots|\ldots)$ refers to the conditional probability. For instance, $P(A|B)$ is the probably of producing a correct piece of code given that the associated Graphical Loop Invariant was correct.

4.3.3.1 Opinions about the GLIBP Methodology

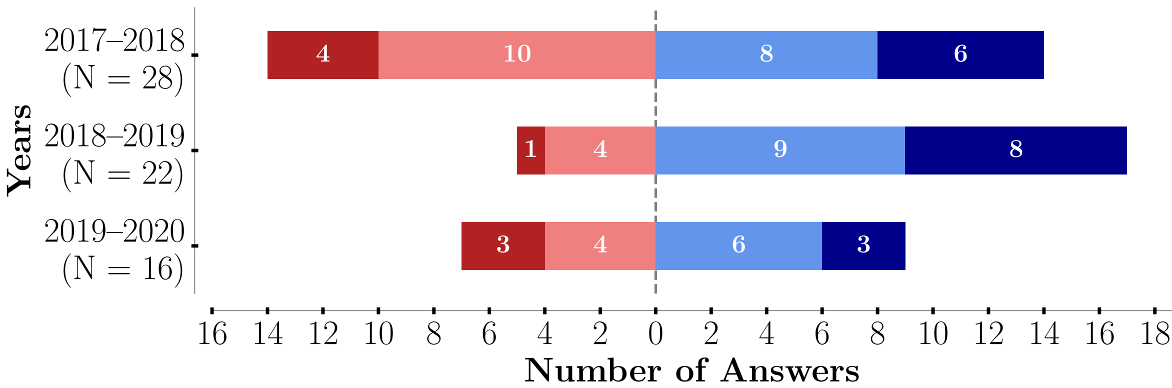
According to the *one time* surveys results, A majority of respondents declares each year that “they feel they can make a good use of the Loop Invariant to write a loop code” (see Fig. 4.7a – 20/28 in 2017–2018, 20/22 in 2017–2018 and 11/16 in 2017–2018).

Moreover, half of them or more acknowledges that “the Loop Invariant enabled them to understand how loops work” (see Fig. 4.7b – 14/28 in 2017–2018, 17/22 in 2017–2018 and 9/16 in 2017–2018).

In 2019–2020 survey, we deepened our investigation about the reception of the GLIBP methodology. We asked students what they thought about being forced to draw a Graphical Loop Invariant before writing any piece of code. 75% of the answers were positive: 25% (4/16) acknowledged it as a reflection phase; 18.8% (3/16) mentioned the rigor needed to solve a problem; 18.8% (3/16) declared that they would draw a Graphical Loop Invariant in any case and 12.5% (2/16) recognized it was a useful visualization process. On the other hand, 25% of the answers were negative: two respondents stated this was part of the course, hence mandatory, one answer mentioned this was useless since the loops studied in the course were trivial and another one declared that they preferred using formal Loop Invariant (*i.e.*, written as formal logic formulae).



(a) "I feel I can make a good use of the Loop Invariant to write a loop code"



(b) "The Loop Invariant enabled me to understand how loops work"

Figure 4.7: Responses to the one time surveys from 2017–2018 to 2019–2020. All the plots use a Likert [126] scale.

4.3.3.2 Compliance with The GLIBP Methodology

We also asked students for how many automatically graded PAs they initialised their variables thanks to the Graphical Loop Invariant (see Fig. 4.8, bottom lines). The majority of respondents declares having done that for 2 or 3 PA in 2018–2019 and 2019–2020.

With respect to establishing the Loop Condition, the majority of students recognizes relying on the Loop Invariant every years (see Fig. 4.8, middle lines).

Regarding the *Loop Body*, a large majority of the respondents declares having used the Loop Invariant for 1 PA or less in 2017–2018 (20/28) and 2019–2020 (12/16). In 2018–2019, half of them used the Loop Invariant for 1 PA or less (see Fig. 4.8, top lines).

Finally, the long term survey results are depicted in Table 4.4. These results show that the Loop Invariant based methodology is still used by a majority of the students (72.1%) after the

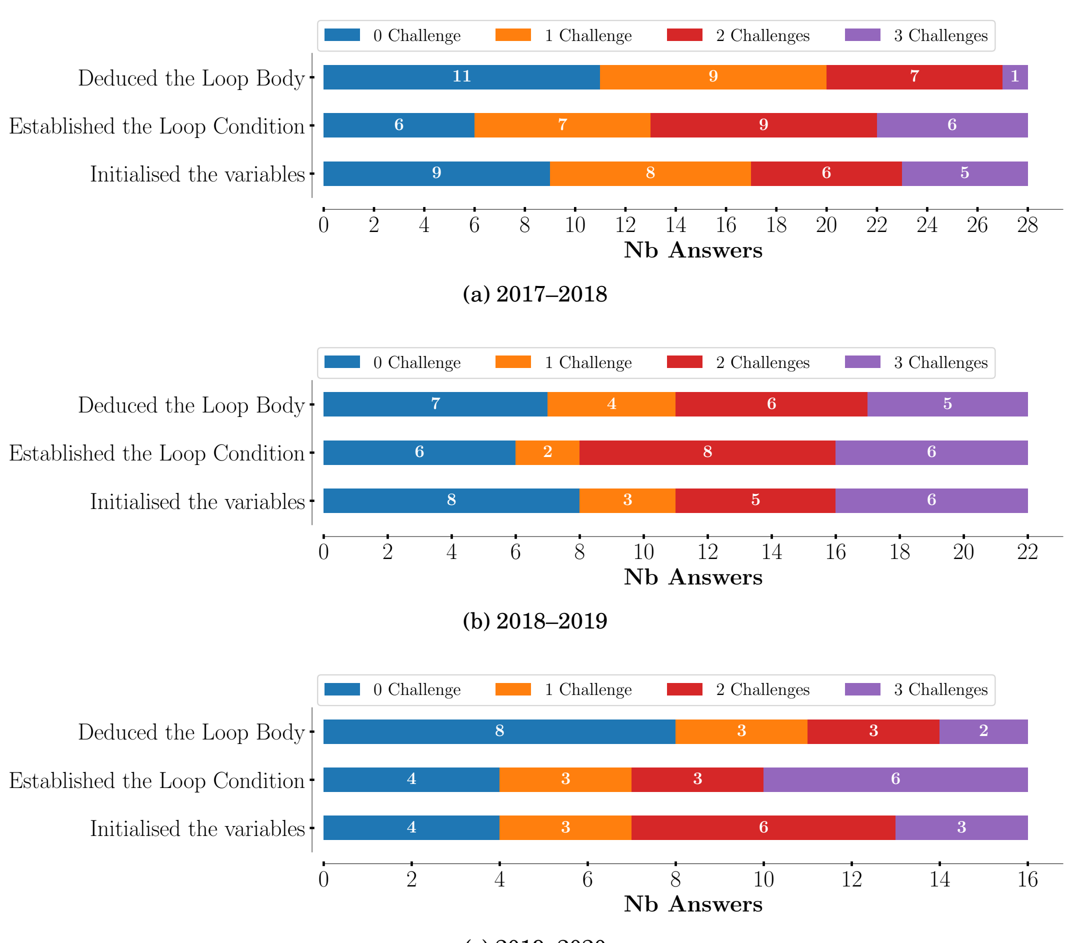


Figure 4.8: Responses to the one time surveys from 2017–2018 to 2019–2020. We asked students for how many PAs they use their Graphical Loop Invariant to perform the actions written on the Y-axis. It should be noted that here, PA_1 and PA_2 as well as PA_6 and PA_7 account for a single PA. The maximum is therefore 3 PAs.

Table 4.4: Answers to the question “Does the Graphical Loop Invariant based programming, introduced in CS1, help you afterwards when solving problems?” ($N = 36$). We excluded from the answers the CS2 course that also relies on Loop Invariant and is given by the same educational team.

Answers	#	%
Have regularly used Loop Invariant	17	47.2
Have used Loop Invariant in subsequent courses	4	11.1
Have used informally Loop Invariant	5	13.8
Never have used again Loop Invariant	10	27.7

CS1 course. 47.2% regularly uses it, 13.8% mentions using an informal Loop Invariant, and 11.1% declares using it in subsequent courses (different from the CS2 course taught by the same educational team). 27.7% has never used it again.

4.4 Discussion

4.4.1 Participation Data

The data has shown that the participation decreases as the semester goes on (see Fig. 4.3a, top). Results show that students tend to abandon the curriculum during the semester, as the participation rate to PA decreases over time. One should note that the participation to the Final Exam (PA_8) is mandatory to continue the curriculum during the second semester and to keep the education grant (if any), possibly explaining its (relatively) high participation rate.

Concerning the GLIDE tool, Fig. 4.3b shows that a majority of students used it for few than 5 sessions. One may note that these sessions took place in October, when students were introduced to the tool. We notice a minority of students used the tool during more than 15 sessions (10% of the users), those sessions being quite spread over the semester, indicating so a regular usage.

The validations figures corroborate this as a majority of them were triggered by a minority of users. One should note that only 22.5% (90/399) of the drawings passed the validation process and could be used to write pieces of code.

The students can hence be split in two main categories: (i) a majority of them used the GLIDE once or a bit more as a try and submitted a small number of drawings and, (ii), a minority of users submitted drawings throughout the semester.

This suggest to bring some improvement to GLIDE such as easier drawing, easier handling, more consistency with the course, possible export of the drawing to PDF (for assignment reports),...

4.4.2 Performance Data

From the distribution of the errors over all the PA (see Fig. 4.4a), we can conclude that all the curves follow the same trend: between 40% and 60% of the programs does not contain errors, 80% of them contains one error and 20% contain two errors or more.

One should note that the errors concerning the Loop Variant are not cumulative. This explains that the number of error is either 0 or 1.

As far the code is concerned, this means that less than half of the programs are correct.

From the errors distribution for the GLIDE tool (see Fig. 4.4b), we can observe that the curves corresponding to semantic markers (SE_4 and SE_5) are under the curves relative to syntactical markers ($\text{SY}_{2\rightarrow 5}$), even though these semantic errors are easily avoidable: the drawing has just to be labelled properly. These mistakes reveal a certain lack of students' involvement in the use

of the GLIDE tool, as it was also suggested by the few number of validations discussed earlier.

4.4.2.1 Markers Distribution by PA

Several markers are more prevalent for the $\text{PA}_{3,4}$ and PA_8 which correspond to the Mid-Term Exam and the Final Exam, respectively. This is the case of GLI-SY_2 (see Fig. 4.5a), GLI-SE_3 and GLI-SE_6 (see Fig. 4.5b). These errors can be partially explained by the absence of blank Graphical Loop Invariant during the Exams: students have to produce a Graphical Loop Invariant from scratch and label it properly. Our data shows students have issues doing that.

On the other hand, the large prevalence of the GLI-SE_1 PA_1, PA_2, PA_6 and PA_7 (see Fig. 4.5b) suggest a positive bootstrap effect from the Blank Graphical Loop Invariant provided in the automatically corrected PA (see Sec. 3.2).

Among the automatically corrected PA, PA_5 shows more semantic error. The nature of the problem to be solved in the PA (array manipulations) is certainly at the root of those issues. One can also see that this PA shows a high prevalence of GLI-SE_3 and GLI-SE_5 (resp. No boundaries and no Dividing Line label) that may explain CODE_4, CODE_6 and CODE_7 errors (resp. Buffer Overflow, Variable not initialised according to the Graphical Loop Invariant and Incorrect Loop Condition).

From these markers distributions over the PAs, one can also conclude that the students do not seem to progress over the semester. Most of the errors indeed remain, especially those that can be easily avoided such as drawings lacking of labels. This suggests again a lack of students involvement that was also observed in GLIDE usage.

Regarding the bootstrap effect of the blank Graphical Loop Invariant, it surely helps students in finding a Graphical Loop Invariant (and thus a solution) for complex PA (see PA_5) but, unfortunately, it does not enable them to develop skills that could be reused for other PAs, such as the Mid-Term and Final Exam.

All of this argues in favour of a careful teaching of the programming methodology by constantly referring to the rules when building a Graphical Loop Invariant and deducing the code.

4.4.2.2 Correlation between Program Quality and Graphical Loop Invariant

At first glance, one can see in Fig. 4.6 that $P(\overline{A}|\overline{B})$ has a high value for almost every PA. One can see in the Table that, $P(\overline{A}|\overline{B}) \geq P(A|\overline{B})$ for all the PAs. This is in line of our expectations about the programming methodology: an erroneous Graphical Loop Invariant is likely to lead to an incorrect program.

In the same way, $P(A|B) \geq P(\overline{A}|B)$ for $\text{PA}_{1\to5}$ and PA_8: if the Graphical Loop Invariant is correct, it is more likely to get a correct code. For PA_6, the proportion of correct Graphical Loop Invariant is very small and for PA_7, the CODE_6 (variables not initialized according to the Graphical Loop Invariant) is the most prevalent marker concerning the code (see Fig. 4.5d) and is a code vs. Graphical Loop Invariant matching error.

To sum up, these performance data tends to suggest that when applied correctly, the Graphical Loop Invariant based programming methodology does help in writing correct pieces of code but the high number of incorrect codes and Graphical Loop Invariant advocates deepening our investigations.

4.4.3 Perception Data

The surveys being anonymous and conducted after the course end (during the second semester), we had no mean to collect the opinion of the students who left the CS curriculum.

Concerning the answers we collected, On one hand, the students seem to acknowledge the importance of the GLIBP to help them to understand how loop work (see Fig. 4.7b) and declare feeling able to use it for programming (see Fig. 4.7a).

On the other hand, they do not fully apply it when they could (see Fig. 4.8), as it was previously shown in Fig. 4.5.

As time goes by, a majority of the students declares keeping using the GLIBP methodology and not only for subsequent courses, therefore recognising it as usefull (see Table 4.4.

4.5 Conclusion

This chapter addressed the student reception of the GLIBP methodology. We presented and discussed participation, performance and perception data that enable us to answer the following research questions :

RQ 1. 1 *How the students seize the opportunity to practice the* GLIBP *methodology?*

One can first conclude that our programming methodology does not help in keeping the students in the class, although this is not its goal.

Regarding the tools, the participation in automatically graded programming activities follows students' involvement, while for the GLIDE tool, two students profiles arise: a majority of students used it close to its introduction in early October (as a kind of curiosity) while a minority of them kept using it as a companion tool during the semester. We can also add that the students do not fully practice the methodology since they are few to validate their Graphical Loop Invariants with the GLIDE tool.

RQ 1. 2 *What kind of error are committed using the* GLIBP *methodology?*

We proposed a taxonomy of errors based on the guidelines to carefully depict a Graphical Loop Invariant. The frequency of the errors depend on the programming tasks but regarding the Graphical Loop Invariant, drawings lacking of boundaries, unlabelled Dividing Line(s) and lack of relationship with the code are the most prevalent errors. This is unfortunate since these elements are required to use the Graphical Loop Invariant at its full potential to deduce the code instructions.

Data show that some errors can be mitigated thank to Blank Graphical Loop Invariant that

seem to have a bootstrapping effect on the students performance during the semester. However, this effect does not seem to have an impact on students performance at the Mid-Term and Final Exam.

RQ 1. 3 *Can we link the error committed with the* GLIBP *methodology to programming errors?*

The data presented in the chapter tend to show that an incorrect Graphical Loop Invariant is more likely to be correlated to an incorrect code while a correct Graphical Loop Invariant is more likely to be correlated to a correct code. As the methodology requires to draw a Graphical Loop Invariant to deduce the code, this suggest that the Graphical Loop Invariant does help in writing piece of code. The relationship between Graphical Loop Invariant and code correctness must be further investigated and is discussed in Chap. 5.

RQ 1. 4 *How the* GLIBP *methodology is perceived by the students?*

The students who are still enrolled during the second semester acknowledges the GLIBP methodology as useful to understand how a loop works and declare they feel able to properly use it to write a code.

However, when they are in a situation to apply the methodology, it appears from the survey results that they not necessarily use it at its full potential: while they often deduce the variables initialisation and the Loop Condition, they use it much less as far as the Loop Body is concerned.

On the other hand, order students who took formerly our CS1 course reported using regularly the Graphical Loop Invariant, sometimes in a more informal form.

4.5.1 Future Work

The taxonomy of errors could be amended in the future to represent more accurately common students errors. For example, we proposed the marker CODE_4 to represent buffer overflow but we could think of other errors like dereferencing Null-pointer, memory leaks, etc. depending on the context in which the taxonomy is used.

Such a taxonomy would also enable to design a system capable of generating custom exercises, based on the most frequent errors committed by a student to allow them to overcome their learning gaps or practice in the subjects they is less proficient in.

First Steps Towards Evaluation of the Blank Graphical Loop Invariant

This chapter was originally intended to answer the RQ 1. 5 *Does the Graphical Loop Invariant Based Programming methodology enable to write better pieces of code?* Unfortunatelly, due to the COVID-19 pandemic [190], there was a second lock-down at the time we planned to conduct our experiment, during the first semester of the academic year 2020–2021. We hence were forced to collect our data remotely, especially regarding the Graphical Loop Invariant. In order to mitigate this limitation, we decided to use the Blank Graphical Loop Invariant (see Sec. 3.2) in our data collection. Our Research Question has therefore been slightly modified and become:

RQ 1. 5 *Does the Blank Graphical Loop Invariant enable to write better pieces of code?*

The rest of the chapter is organised as follows: Sec. 5.1 details the methodology to asses the Blank Graphical Loop Invariant; Sec. 5.3 discusses the experiments results. Finally, Sec. 5.4 proposes some experimental parameters regulation, the Blank Graphical Loop Invariant usage, and concludes on further work.

5.1 Methodology

5.1.1 Impossibility of Randomized Controlled Trial

The Randomized Controlled Trial (RCT) is a well-know method to determine whether a particular protocol show evidence of efficacy and is broadly used, *e.g.*, in medicine [188]. It consists mainly in dividing at **random** the population on which the protocol is going to be tested in two subgroups:

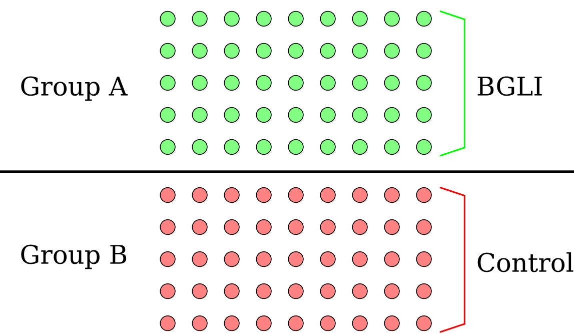


Figure 5.1: Representation of a Randomized Control Trial

a group that follows the protocol and a **control** group (see Fig. 5.1 for an application to GLIBP evaluation). Most of time, the control group receives another already known protocol to allow the comparison with the one that is assessed or a fake protocol (which is known, in medicine, as a placebo). The randomisation of the population division reduces biases. If the experimenter does not know if they delivers the assessed protocol or the placebo to a person who has either no idea the subgroups they belongs to, the study is said to be double-blinded.

As far as the assessment of the Blank Graphical Loop Invariant (or even the Graphical Loop Invariant) is concerned, a RCT is not feasible and, we believe, unethical. In fact, this is not feasible because there is no mean to divide the students cohort into two subgroups with the first programming with the help of Graphical Loop Invariant and the second with a kind of placebo Loop Invariant. We also did not previously identify another standard programming methodology in the literature that could be used by the control group (see Sec. 1.5).

For ethical reasons founded on equity between students, we could not afford, in first year, to divide the the students cohort into two groups, one of which would have learned the Graphical Loop Invariant Based Programming methodology and the other, as control group, would not.

5.1.2 Crosover RCT

To overcome the limitation of the RCT, we set up a Crossover RCT (see Fig. 5.2): the student cohort is randomly divided in two groups of the same size. The students are asked to solve two problems. Group A students are asked to solve the first problem using the GLIBP methodology and the second problem without it. On the contrary, group B students are asked to solve the second problem using GLIBP methodology and the first problem without it. Each subgroup acts as a control group for one of the two problems: group A for the problem 2 and group B for the problem 1.

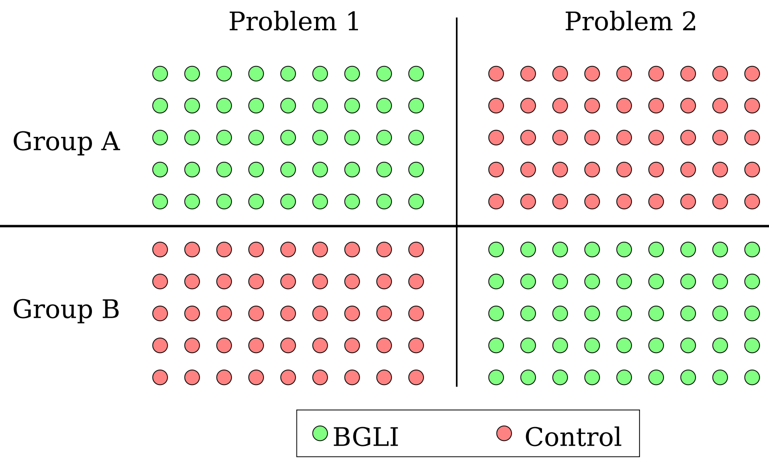


Figure 5.2: Representation of a Crossover Randomized Control Trial

5.1.3 Experimental Process

Our experiment took place on November 25th, 2020. Students did not know the Blank Graphical Loop Invariant was assessed but were told it was a mean to practise before the exam. After the experiments, all the students received a semi-automatically generated *feedback*. The duration of the experiment was set to two hours in order to fit in the course schedule to not impair with the students normal workload.

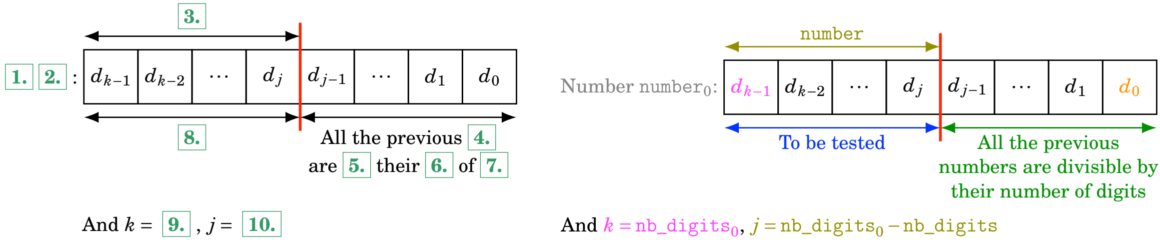
5.1.3.1 Problems

The first problem we consider is to calculate if an integer is a polydivisible number that we denote as P_P. The second problem is to compute the Gini Index from two arrays containing the cumulated proportions of the population and the corresponding cumulated proportions of their incomes using the Brown [35] formula, that we denote as P_G. Both problems definitions are summed up hereafter:

POLYDIVISIBLE NUMBER (P_P):

Input – An integer `number` and its number of digits `nb_digits`

Output – Display on the standard output whether `number` is a polydivisible number (*i.e.*, all the truncations of its decimal representation are divisible by their numbers of digits) or not.



(a) Blank Invariant for P_P. Multiple choices for boxes 1 and 4 to 8 are shown in Table 5.1

(b) Expected Invariant (among others) for P_P

Figure 5.3: Blank Graphical Loop Invariant and expected answer for P_P

Table 5.1: Loop Invariant for P_P: Multiple choices for boxes **1.** and **4.** to **8.**. Expected answers are italicised.

Box 1.	**Boxes 4. and 7.**	**Box 5.**	**Box 6.**	**Box 8.**
1. Word	1. *digits*	1. divided by	1. digit	1. To be displayed
2. Digit	2. factors	2. summed to	2. factor	2. To do
3. *Number*	3. terms	3. multiplied by	3. term	3. to be decypher
4. Array	4. polydivisible	4. subtracted from	4. polydivisible	4. *To be divided*
5. Drawing	5. bits	5. *divisible by*	5. bit	5. *To be tested*
6. Sentence	6. *numbers*	6. isolated by	6. *number*	6. To be browsed

```
1  #include <stdio.h>
2
3  int main(void){
4     unsigned int number = ...;
5     unsigned int nb_digits = ...;
6     const unsigned int number_0 = number;
7     const unsigned int nb_digits_0 = nb_digits;
8
9     // Your code will be pasted here
10
11 }// end main
```

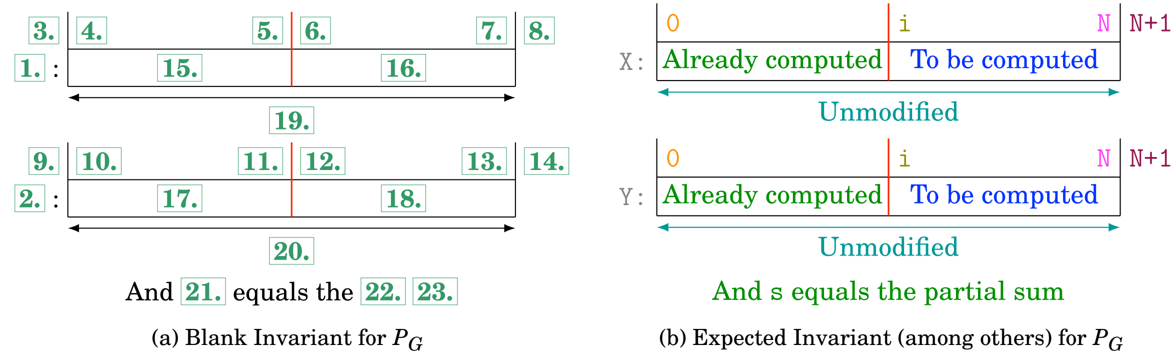
Listing 5.1: Code to be filled in with the student answer for P_P

GINI COEFFICIENT (P_G):

Input – Two sorted arrays X and Y of size N+1 containing real numbers between 0.0 and 1.0

Output – Display on the standard output the corresponding Gini coefficient when X and Y respectively contain the cumulated proportions of the population and the corresponding cumulated proportions of their incomes, *i.e.*, $G = 1 - \sum_{k=1}^{N-1}(X_{k+1} - X_k)(Y_{k+1} + Y_k)$

Concerning the P_P problem, the Blank Graphical Loop Invariant provided to Group A students is shown in Fig. 5.3a while an example of expected solution is provided in Fig. 5.3b. The multiple



(a) Blank Invariant for P_G

(b) Expected Invariant (among others) for P_G

Figure 5.4: Blank Graphical Loop Invariant and expected answer for P_G

Table 5.2: Loop Invariant for P_G: Multiple choices for boxes **15.** to **20.** and **22.** and **23**. Expected answers are italicised.

Boxes 15. to 18.	Boxes 19. and 20.	Boxes 22. and 23.
1. *To be computed*	Modified	division
2. *Already computed*	*Unmodified*	total
...	...	...
9. To be filled in	Displayed	*sum*
10. Already filled in	Gini	*partial*

```
1  #include <stdio.h>
2
3  int main(void){
4     const unsigned int N = ...;
5     float X[N+1];
6     float Y[N+1];
7     // Code to populate arrays X and Y with values
8     // this code is hidden from students.
9
10    // Your code will be pasted here
11
12 }// end main
```

Listing 5.2: Code to be filled in with the student answer for P_G

choices that are available for boxes 15 to 20, 22 and 23 the Blank Graphical Loop Invariant are shown in Table 5.1. The other boxes must be replaced by variable or constant names or simple expression including variables and constants. We also provide both groups students with a code template shown in Listing 5.1.

As far as the P_G is concerned, the Blank Graphical Loop Invariant provided to Group B students is shown in Fig. 5.4a while an example of expected solution is provided in Fig. 5.4b. This last figure is coloured with the colour code introduced in Chap. 2. The multiple choices that are available for boxes 1 and 4 to 8 of the Blank Graphical Loop Invariant are shown in Table 5.2. The other boxes must be replaced by variable or constant names or simple expression including variables and constants. We also provide both groups students with a code template shown in

Table 5.3: Groups for the experiment to asses Blank Graphical Loop Invariant

Subgroup	Size	GLIBP	Control
Group A	24	P_P	P_G
Group B	26	P_G	P_P

Listing 5.2.

The students are asked to return an answer containing:

- The code for the P_P problem;
- The code for the P_G problem;
- Depending on the group they belong to, what should replace the boxes in the Blank Graphical Loop Invariant they are asked to used.

To do so, we provide them a template they have just to fill in to complete their answer.

At the time of the experiment, students have already been asked to use Blank Graphical Loop Invariant at least three times: twice in the context of what we call the Programming Challenges Activity (PCA) (see Chap. 9 and following) and in the context of a mid-term exam. We therefore assumed students were familiar with both the Blank Graphical Loop Invariant and the answer submission process.

The Table 5.3 sum up the parameters of the experiment.

5.1.3.2 Collecting Students Answers

In order to collect students answers, we used a submission platform that is able to run scripts as soon as a file is submitted. The program called Correction Automatique et Feedback des Étudiants (CAFÉ) (see Chap. 7) analyses the student file and delivers them a report that contains information on their submission, namely the student's codes for both P_P and P_G problems and how their Graphical Loop Invariant is understood: the student submitted what should replace the boxes and CAFÉ displays the drawing obtained after the replacement. The student then can check their answer. During the experiment, the students answer are simply collected and the CAFÉ's report does not contain any information about the correctness of the answer.

5.1.3.3 Correction

The correction was semi-automatic. After the end of the experiment, another instance of CAFÉ was run to preprocess students files. For each student it produced a preprocessed file containing:

- The student's code for P_P problem;
- The student's code for P_G problem;

- The translation of the student's Graphical Loop Invariant into a human-readable figure;
- The results of the execution of programs obtained after the compiling of both problems codes. Each program was assessed with few test cases.

These preprocessed files were saved into separate directories for both groups. At this time, each student's file was named according to the student's ID. Before the correction of the Graphical Loop Invariant (performed by a human), the students files were renamed with a number (from 0 to $N-1$, where N is the group size). These numbers were randomly attributed to the files.

Both the codes and the Graphical Loop Invariants were corrected by a member of the pedagogical team according to the error taxonomy introduced in Chap. 4. It is worth noting that the codes and Graphical Loop Invariants for P_P (resp. P_G) in Group A (resp. Group B) were corrected separately: the corrector did not know what was the student's performance regarding the Graphical Loop Invariant while correcting the code.

5.1.3.4 Students Feedback

As we presented this experiment as a practise exercise before the exam, we sent to all the participants a feedback about their performance. It was semi-automatically generated: since the codes and Graphical Loop Invariants corresponded to performance markers of our error taxonomy, we converted the performance markers corresponding to a student's answer into remarks that we concatenated to form a feedback message. We also added an overall comment the corrector wrote during the correction step.

5.1.3.5 Possible Biases

This study has been foreseen since August 2020 when we did not know there would be a second lock-down. As the students would benefit from it as a rehearsal exercise, we decided to still conduct the experiment.

Albeit we tried to reduce the biases as best as possible, *e.g.*, by reordering and randomly renaming the students file and by correcting the codes and the Graphical Loop Invariants separately; we were not able to control several parameters that are listed in the following:

- The workplace: each student works from home and their environment may have an influence on its work;
- The understanding of the instructions: the teaching team was available to answer students questions but could not do so as well as in a classroom;
- The time students allocated to a particular problem. We asked them to allocate it evenly on both problems but we did not have the mean to check;

- The order in which the problems were solved by the students: they may be less successful in the second one (*e.g.*, due to tiredness), which is referred as *carry-over effect* in the literature;
- Students collaboration: we could not check if the students collaborated while they were supposed to work alone. Even for a practical exercise, any student could copy another student's answer for the sake of receiving a positive feedback.

5.2 Hypotheses

Since we believe our methodology does help to write correct pieces of code, we formulate the following hypotheses:

Hyp 1 Group using Graphical Loop Invariant has better performance than the other group.

Hyp 2 Among Graphical Loop Invariant users, a better Graphical Loop Invariant leads to better piece of code.

To define the performance associated with the Graphical Loop Invariant and the code, we refer to our own error taxonomy (see Chap. 4) and we provide the following definitions:

Correct Graphical Loop Invariant refers to a Graphical Loop Invariant labelled with the markers GLI-SY_1 and GLI-SE_1 *i.e.*, a Graphical Loop Invariant whose syntax and semantic are both correct ;

Correct code refers to code that once compiled and executed, shows the expected behaviour. Unlike the marker CODE_1, the assessed code does not require to have been build upon a correct Graphical Loop Invariant ;

Better Graphical Loop Invariant A Loop Invariant is said better than another if it is labelled with fewer GLI-SY and GLI-SE markers (excluding markers GLI-SY_1 and GLI-SE_1);

Better code A code is better than another if it is not a "very incorrect code" and if is labelled with fewer CODE markers (except $\text{CODE}_{1)}$

Very incorrect code A code labelled with at least one of the following markers: CODE_2, CODE_3, CODE_4, and CODE_8. As a result, all the code that are not "very incorrect" are better than the very incorrect ones and comparing two very incorrect codes is not relevant.

5.3 Results

In order to verify our first hypothesis, we counted the correct codes, for each problem in each subgroup. According to our hypothesis, students using the GLIBP methodology should perform

Table 5.4: Proportions of correct codes among the subgroups. # column refers to the number of correct codes. (/N) column recalls the group size. The last two columns refer to χ^2 independence test statistics (including Yates' correction) and its associated p-value.

	GLIBP		Control			
	# (/N)	%	# (/N)	%	χ^2	p-value
P_P	5 (/24)	20.8	14 (/26)	53.8	4.457	0.034
P_G	11 (/26)	42.3	8 (/24)	33.3	0.131	0.718
$P_P + P_G$	16 (/50)	32.0	22 (/50)	44.0	1.061	0.303

Table 5.5: Proportions of correct codes among the students of subgroups who committed least than 4 errors in their Graphical Loop Invariant. # column refers to the number of correct codes. (/N) column recalls the group size. The last two columns refer to χ^2 independence test statistics (including Yates' correction) and its associated p-value.

	GLIBP		Control			
	# (/N)	%	# (/N)	%	χ^2	p-value
P_P	4 (/12)	33.3	8 (/16)	50.0	0.246	0.620
P_G	6 (/16)	37.5	6 (/12)	50.0	0.076	0.783
$P_P + P_G$	10 (/28)	35.7	14 (/28)	50.0	0.656	0.418

better. Results are shown in Table 5.4 and Table 5.5. Table 5.5 focuses on students whose Graphical Loop Invariant are either correct or labelled with less than 4 markers. The rationale of this filter is that our methodology consists in deducing the code from a correct Graphical Loop Invariant. Thus we compare students who make the fewest mistake with it.

For each lines in the tables, we performed a chi-square independence test with Yates's correction (as the degree of freedom is 1). The H_0 null hypothesis is "There is no relationship between using (or not) the Graphical Loop Invariant and the code correctness" and the H_A alternative hypothesis is "There is a relationship between using (or not) the Graphical Loop Invariant and the code correctness". The test statistics and its p-value are provided in the tables. As can be seen in the table, except for the first line of Table 5.4, the p-values are above the 0.05 significance level and the null hypothesis cannot be rejected.

As far as the first line of Table 5.4 is concerned, the p-value 0.034 is less than the significance level and the null-hypothesis can be rejected in favour of the alternative one: there is a relationship between using (or not) the Graphical Loop Invariant and the code correctness. The table shows that students from the control group (who does not use the Graphical Loop Invariant) perform better than the students who were asked to draw a Graphical Loop Invariant.

Given these first results in regards of our hypotheses, we do not deepen the analysis of the experiment data.

5.4 Conclusion and Further Work

The experiment was not able to show a positive effect of the Graphical Loop Invariant on the students' coding performance (see Table 5.4 and Table 5.5).

Even worse, the first line of Table 5.4 suggests that for problem P_P, asking students to draw a Graphical Loop Invariant was harmful! This is a very unexpected and counter-intuitive result because in this case, the figures do not take account of the student's actual aptitude to draw a Graphical Loop Invariant. As if just asking to use the Blank Graphical Loop Invariant had a negative impact on students' performance. We cannot bring a better explanation than the two groups being not even in term of performance from the start *i.e.*, Group B performed better than Group A, regardless of students' mastery of the Graphical Loop Invariant. Either because we failed at constituting random even groups or because students collaborated in Group B, what we could not ensure due to the experiment conditions.

For the very same reason, we could not ensure the students actually used their Graphical Loop Invariant to deduce the code instructions.

Although this experiment is not a success, we propose regulations for both new experiment on the GLIBP methodology and the teaching of the methodology itself.

5.4.1 Conducting experiments on Graphical Loop Invariant

In order to conduct new trials on Graphical Loop Invariant, we suggest to not perform them through a remote system but in a classroom. Hence, the students cannot collaborate and the teaching team can be present to answer all the questions that arise.

We suggest to try to have a larger test population to rise the accuracy of the results and ensure their statistical significances. It would be interesting to carefully assess all the data structure templates (see Sec. 2.2) to see if some of them lead to better codes and why.

We recommend to write the instructions for the Graphical Loop Invariant so that the students are asked to explain how they use it to derive the code, forcing them to actually (at least) try to use it.

Assuming new studies confirm that the Graphical Loop Invariant is harmful, one must then study why a graphical methodology confuse so much the students.

5.4.2 Using GLIBP methodology

Beyond the lack of statistical significance or the results of the study, there is a risk that the GLIBP methodology could be harmful to students. Here are some recommendations on its usage to prevent such bad impacts:

- Make sure all the students understand well the drawing pattern presented in Sec. 2.2;

- Train students to use Blank Graphical Loop Invariant;

- While presenting the Graphical Loop Invariant, use consistently a colour code for the elements drawn in the Loop Invariant;
- Until evidences of (Blank) Graphical Loop Invariant benefit are brought forward, do not use either the Graphical Loop Invariant or the Blank one in a summative remote assessment.

Part II Automatic Correction and Feedback to the Students

CHAPTER 6

Automatic Code Assessment and Feedback Background

THIS CHAPTER introduces the second part of the document, dedicated to a tool we developed, Correction Automatique et Feedback des Étudiants (CAFÉ), that is able to automatically assess student programs. Sec. 6.1 presents the related work about automatic feedback generation and program assessment. Sec. 6.2 sums up the advantages and drawbacks of such systems. Sec. 6.3 presents our research questions which link automatic assessment to the GLIBP methodology introduced in the previous part of the document. Eventually, Sec. 6.4 introduces our tool, CAFÉ, and positions it with regard to the state of the art.

6.1 Related Work on Automatic Feedback

A lot of systems for automatic assessing students programs have been already proposed. Ihantola et al. [92] and later Keuning et al. [101] review most of them. We do not pretend here to present a deeper analysis. However, Table 6.1 shows a few of systems and solutions that we consulted to align our solution to the state of the art. The column "Grade?" indicates whether the tool embed a feature to grade students.

Most of the tools produce *feedback i.e.*, information provided to the student about their performance. Most of time, the student's program is assessed through unit testing and the feedback consists in listing the cases for which the tests failed and the ones for which they succeeded (which is referred as *simple* feedback in [101]). As far as the tools in Table 6.1 are concerned, the column "Comments" indicates that we provide, in the following, further information of the tool particular features, especially in term of more informative feedback (which is referred as *elaborated* feedback in [101]).

Marwan et al. [137] propose what they call an Adaptative Immediate Feedback (AIF) that works in the context of a block-based programming environment. The system continuously

Table 6.1: Tools and concepts about automatic feedback generation. Column “Grade?” is marked with ✓ if the corresponding reference (see “Ref.” column) mentions the tool is able to compute a grade. A ✓ in the “Comments” column means that the corresponding tool or concept feedback is further discussed.

Name or concept	Ref.	Grade?	Comments
Adaptative Immediate Feedback	[137]		✓
AutoGrader	[85]	✓	✓
AutomataTutor	[46]	✓	✓
CodeOcean	[170]	✓	✓
Coderunner	[129]	✓	
CodeRunner plug-in to assess OpenGL	[191]	✓	✓
CodingBat	[153]		
Coursemarker	[87]	✓	✓
Feedback from Error Model	[172]		✓
Flexible Program Alignment	[135]		✓
GradeIT	[151]	✓	✓
INGInious	[51]	✓	
MobileParsons	[99]		✓
MUMUKI	[24]		
My Lab Programming	[154]	✓	
Problets	[110]	✓	✓
Solutions spaces analysis	[159]		✓
UNLOCK	[18]		✓
Verificator	[157]		✓
Web-CAT	[61]	✓	✓
CAFÉ	Chap. 7	✓	
GLIDE	Sec. 3.1		

analyses the students’ work and detect if an objective is fulfilled. As soon as it is the case, a panel listing all the objectives is directly updated and a positive message containing emojis (selected according the state of the student work) is displayed in a pop-up to congratulate, cheer or encourage them. The system is well received by students, improve the retention in CS, the students’ engagment, performance and learning.

AutoGrader [85] leverages Object Oriented interfaces to launch students’ tests.

Automata [46] is a tool to assess and provide feedback on Deterministic Finite Automata that accept strings matching a certain pattern. The tool leverages DFA properties to propose students either counterexample of incorrect processed strings or hints that indicate how to modify their answers.

CodeOcean [170] is “an educational, web-based execution and development environment for practical programming exercises designed for the use in Massive Open Online Courses”. It enables MOOC tutors to write their own unit tests to deliver custom feedback to students. The

tool offers the possibility for a student to ask other students questions about its own code.

Wünsche et al. [191] present a solution based on CodeRunner [129] to assess OpenGL computer Graphics assignments. The paper presents several solution to grade such graphical problems: compare pixels colors; ask for students to determine correct parameters of a function; analyse parametric equations instead of their 3D rendering and analysing intermediate OpenGL state rather than drawing the corresponding object.

Coursemarker [87] performs unit tests, as well as typographic test (*i.e.*, testing the indent, the presence and length of comments, the identifier name choice and length, etc.) and feature test (eg testing whether a switch or an if-then-else was use). It enables the teacher to provide feedback but it is not clear in what extent.

Singh et al. [172] propose more advanced automatic feedback by providing, to students, a numerical value (the number of required changes) and the suggestion(s) on how to correct the mistake(s).

Marin et al. [135] propose a method using program dependence graphs (PDG) and semantic information extracted from the programs that computes repair suggestions to correct a program. The method constructs the PDGs for both the correct and incorrect programs. Then, it computes an alignment between the graphs *i.e.*, it determines which vertex in the smaller graph is similar to which vertex in the larger one. Then, based on which program is the correct one from the smaller or the larger, it can suggest to add or remove code instructions.

GradeIT [151] is a grading tool that assess programs by performing unit tests comparing the program result with an expected output. The tool embeds a feature that is able to repair non-compiling code before the tests. The tool delivers an improved feedback that consists in providing richer information with the compiler message, such as an explanation of the compiler message, an example of correct statement and a counter-example of an incorrect statetement.

MobileParsons [99] is a tool built on js-parsons [97] for solving Parsons problem on mobile. The app can deliver richer feedback by detecting the code chunks that do not belong to the longest common subsequence shared with a model. The tool also recognises answers previously tried by students preventing them to be bogged in loops when finding a solution. The tool also limit feedback queries to force students to think about what they're doing instead of using a trial-and-error strategy.

Kumar's Problets [110] enables step by step code execution as part of feedback.

Rivers and Koedinger [159] propose to use solution spaces that is a graph representing the path taken by a student from a problem to the correct answer. The solutions spaces is built by analysing previous students data. By determining the position in the solution spaces of a new program, authors suggest it would be possible to provide accurate feedback of this new program.

UNLOCK [18] tackles the problem solving skills in general, not just coding skills.

Verificator [157] is a DevC++ plug-in that enable student to write, test and debug their programs. To prevent students from copying code and writing a lot of instructions without

checking the syntax, it forces them to type the instructions and to compile their code after a certain number of lines typed. The output of the compiler is enriched to made it more understandable. Verificator also enables to enforce good programming habits such as coding style.

Web-CAT [61] makes students write their own unit tests and assesses their code coverage.

6.2 Advantages and Drawbacks of Automatic Feedback

This section sums up the advantage and drawbacks on students learning presented in the papers listed in Table 6.1. Some of them do not provide any evaluation of the effect on students (*e.g.*, [85, 99, 135, 159, 170, 172]).

The majority of the papers that reports an effect of students perception and learning actually reports broadly positive effects or positive effects with few caveats. We wonder to what extent this might be due to publication biases (*i.e.*, tools that do not show evidence of positive effects would not be published, as it is well studied in health research [173]).

6.2.1 Advantages

The first and foremost advantage of automated systems is that they enable to assess a very large students cohort [51, 87, 129, 170], some of them are therefore used in the context of MOOCs.

Regarding the feedback that is provided, the positive effects reported are positive students reactions [137, 157]; an increase in student retention [24, 137]; better student understanding of their mistakes [46] or of the subject [191]; engagement [137] or motivation [87] improvement; better student performance [87, 137] and better learning [137, 157].

As far as the gradation is concerned, Higgins et al. [87] reports that Coursemarker grades "at least as well as human do", saving so hours. GradeIT [151] was compared to manual grading and the results show that both human and automatic gradings were close, GradeIT being more consistent than humans.

6.2.2 Drawbacks

Parihar et al. [151] assessed the quality of the feedback provided by GradeIT with (only) 8 students. In majority, the feedback is considered as substantially better than the compiler message but, in some cases (*e.g.*, multiple errors on the same line), the enriched feedback can be incorrect and thus confusing.

Radošević et al. [157] report that students using Verificator declared that frequent compilations took them time and that the limitation before compiling seemed them too strict.

In addition to the papers previously presented in Table 6.1, Hsu et al. [89] present the student reception of tool for grading short answer that is based on Artificial Intelligence techniques. They report that some students developed folk theories about how the grader worked and that "misalignment between folk theories about how the autograder worked and how it actually worked

could lead to suboptimal answer construction strategies". They thus recommend to implement a "robust appeal process" for students considering the tool graded them badly.

Leite and Blanco [115] compared the student reception of human and automatic feedback in the context of an AI introductory course with programming assignments. The students were divided in two groups. Both received detailed automatic feedback and one group "received human-written feedback describing how their programs' syntax relates to issues with their logic, and qualitative (style) recommendations for improving their code". The study suggest also in a complementary analysis of their data that ' students in the middle two quartiles of the human feedback group performed much better overall than those that received computer feedback.". The study concludes that "disambiguation is one main purpose of human feedback".

6.3 Research Questions

Systems to automatically assess student programs and provide them with feedback seem to be promising and beneficial for the students. But since our *CS1* course introduces the Graphical Loop Invariant Based Programming methodology (see Sec. 2.1) that relies on Graphical Loop Invariant to write pieces of code, the following questions quickly arise:

RQ 2.1: *Can we get a system that automatically assess the* GLIBP *and provide relevant feedback?*

RQ 2.2: *How the feedback produced by such a system is received by students?*

The Chap. 7 answers positively to the first question by presenting the tool we developed, CAFÉ. The next section positions CAFÉ with the state of the art, especially Keuning et al. [101] classification.

The Chap. 8 addresses the evaluation of the feedback provided to the students by CAFÉ.

6.4 Introduction to CAFÉ

We propose CAFÉ [119, 123], a program that assesses and grades students exercises and deliver them a message containing feedback and *feedforward* (*i.e.*, hints on how to improve one's performance) information. As far as the GLIBP methodology is concerned, CAFÉ can assess both a code and the Graphical Loop Invariant that was used to deduce it. To correct automatically Graphical Loop Invariants, CAFÉ relies on the Blank Graphical Loop Invariant introduced in Sec. 3.2.

The feedback and feedforward produced by CAFÉ were carefully based on the literature promoting self-regulated learning. Among the established quality criteria for feedback (Keuning et al., 2019), CAFÉ instantiates the following procedural items: (i) individualized feedback [104], (ii) feedback focused on the task, not the learner [146], and, (iii) feedback made directly available to the student to prevent them from becoming bogged down or frustrated [105].

Content-wise, following Keuning et al. classification [101], which extends Narciss' [145], CAFÉ feedback falls within

- the **Knowledge About Task Constrains** CAFÉ is able to verify if a particular library function was used (whether it is mandatory or prohibited by the task), as well as for particular language instructions (`if/then/else` vs `switch`, loops types, etc.), operators or additional variables.
- the **knowledge About Mistakes**. CAFÉ performs unit testing ("test failure"), compiles students' code and, in case of compilation errors, warns the students ("compilation errors"), it can detect out-of-bound accesses and infinite loops ("solutions errors") and, finally, checks the number of iterations, as well as the proper use of memory allocation ("performance issues"). For all of these points of interest, CAFÉ provides a detailed description of the mistake.
- 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, as well as hints about improvement to the submitted exercise.
- the **Knowledge About Meta-cognition**. CAFÉ checks that the student's code matches with Graphical Loop Invariant (allegedly) used to derive it and thus increases metacognitive awareness [156].

The next chapter details in depth how CAFÉ works and is implemented.

CHAPTER 7

CAFÉ PRINCIPLES

THIS CHAPTER introduces our tool for automatically assessing students' programming exercises which is named after the French acronym Correction Automatique et Feedback des Étudiants (CAFÉ) [119, 123]. We use CAFÉ as part of the Programming Challenges Activity (PCA), that is a programming activity spread over the semester that allows students to submit pieces of code (called Challenges) on an on-line platform (developed for assignment submissions) that are then automatically corrected and graded. Each student also receives *feedback* and *feedforward* for each piece of code submitted.

CAFÉ is precisely the program that is run on the on-line platform and has two main roles: (i) to correct and grade exercises (including the correction and grading of Graphical Loop Invariant and programs written in the context of the GLIBP methodology) and (ii) provide students with feedback and feedforward, enabling them to correct their exercises and to submit again improved versions.

This chapter strictly focuses on how CAFÉ works and is implemented[21]. Sec. 7.1 discusses how students submit their pieces of code to CAFÉ, in particular focusing on how to fill a template that will be properly understood by CAFÉ. Sec. 7.2 presents an overview of the correction operated by CAFÉ in three main steps further detailed below: the preprocessing (see Sec. 7.3), the correction (see Sec. 7.4) and the generation of a message containing feedback and feedforward (see Sec. 7.5).

7.1 Interacting with CAFÉ

As its French name indicates, CAFÉ allows to automatically correct students exercises and to provide them with feedback (feedforward has been added later and does not fit in the anagram).

[21] The PCA is presented and evaluated in the chapters 9 and 10 respectively.

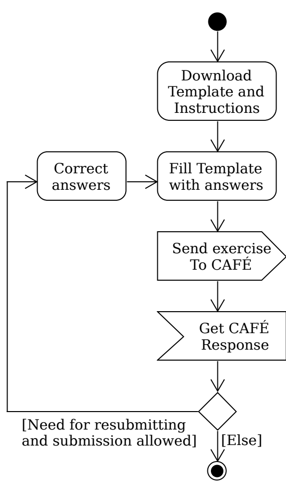


Figure 7.1: Activity Diagram of a Student submission to CAFÉ

Fig. 7.1 shows a UML® Activity Diagram [27] illustrating how a student can use CAFÉ: they first downloads the exercises instructions and a template. Once the template is filled with the student's answers, forming their submission file, they sends this file to CAFÉ. CAFÉ corrects and grades the submission and responds with a message that contains a grade, feedback and feedforward. If the student want to submit again and has the opportunity to do so, they may correct their submission file and submit again. The submissions modalities (subject, time, number, etc.) are discussed in the chapters 9 and 10.

7.1.1 Exercises 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 Listing 7.1 presents an example of code template provided in a exercise instructions (this is called "Solution Template" by Keuning et al. [101]). The goal of this exercise 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 Listing, the arrays are already declared and filled with values although this part is hidden to students. The students only have to complete the code that actually computes the intersection. Note that imposing the name of the arrays eases the exercise correction (for more details see Sec. 7.4).

As far the GLIBP methodology is concerned, CAFÉ was designed to assess both the Graphical Loop Invariant and the Loop Variant used by a student to write their code. The way a student communicates their Graphical Loop Invariant – the Blank Graphical Loop Invariant – has been discussed in Sec. 3.2.

7.1.2 Template

The template is a simple text file whose format 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 Listing 7.2. The C syntax highlighting enables to distinguish the reminders of instructions (i.e., the comments in blue) and the student's answers.

In particular, the lines 10 to 23 show how the Graphical Loop Invariant is encoded (see Sec. 3.2). As the Loop Variant is concerned, it is expressed, in the template, as a simple C expression.

7.1.3 CAFÉ Response

After each submission, a message is quickly provided to the student. This message contains the student's mark, as well as information about how CAFÉ understood their submission, feedback on their performance, and feedforward advices (i.e., what should they do to improve their 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. 7.5 provides an example of a part of the message that could have been provided after the submission of the exercise corresponding to the instructions presented in Sec. 7.1.1. As can be seen in Fig. 7.5, the message structure follows submission template (See Listing 7.2). The Graphical Loop Invariant formed by the combination of the Blank Graphical Loop Invariant from the instructions (See Sec. 3.2) and the content that should replace the boxes specified by the student in their submission (See Listing 7.2) is clearly displayed. Moreover, the feedforward

```c
1  #include <stdio.h>
2
3  int main(){
4      const unsigned int N = ...;
5      const unsigned int M = ...;
6      const unsigned int L = ...; // large enough
7      int A[N];
8      int B[M];
9      int C[L];
10
11     // Arrays A and B are filled with values (code not provided)
12
13     // Your code will be inserted here.
14
15 }//End of the program
```

Listing 7.1: Code template provided to the student.

```
/* Exercise: 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
[...]
23. 6
24. 5
25. 7
26. 1
27. 3
28. 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 */
```

Listing 7.2: Exercise template to fill and submit to CAFÉ

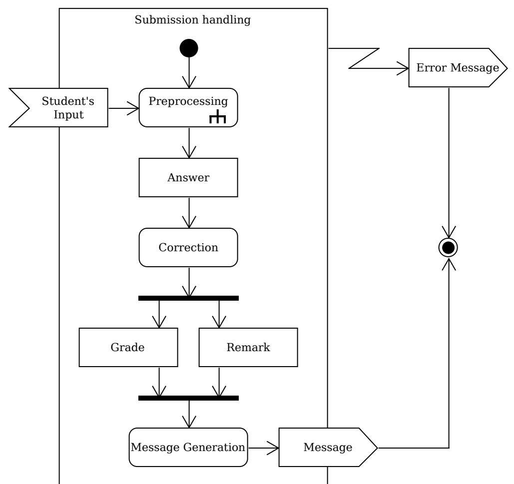


Figure 7.2: Overview of a Student's Input Processing

information is framed to draw student's attention.

7.2 Overview: Processing a Student's Input in Three Main Steps

CAFÉ [119] is written in Python 2.7 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 an exercise. Currently, correcting and grading an exercise is done in three main steps, as illustrated in Fig. 7.2: (i) the preprocessing (i.e., splitting the submitted exercise into several answers – see Sec. 7.3), (ii) the correction per se (i.e., each answer is corrected, graded, and commented – see Sec. 7.4), and (iii) message generation (i.e., the various grades are combined and comments are concatenated to form the message to be provided to students – see Sec. 7.5).

It is worth noting 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 a message at the next step. Moreover, the following illustrates the three main steps with correction of exercises consisting in writing pieces of code in C but CAFÉ itself does not depend on a particular language. In particular, handling code written in Python would be straightforwardly easy.

If an unexpected runtime error occurs and an exception is thrown, it is caught and transformed in an error message that is delivered to the student (See Fig. 7.2, top right). The message states an error occurred and that the corresponding Challenge grade is 0/20 (by default, we consider the error is due to the student's file content). It also prompts the student to quickly contact the Teaching Assistant.

7.3 Preprocessing: from a Student's Submission to a List of Answers

A submitted exercise contains both pieces of code and text (e.g., the Graphical Loop Invariant that helped writing the code – see Chap. 2 and Sec. 3.2). The preprocessing step consists in extracting the text and using the code to generate the student's answers that will be corrected in the next step.

This is illustrated in Fig. 7.3: the first preprocessing operation aims at splitting the student file into several slices and at isolating the text on one hand and the code on the other hand (CAFÉ must be configured to know which slice contains text or code). Both pieces of text and preprocessed code fragments form what we call "pieces of answers" that may be grouped and selected to form the actual answers that are considered by the correction step. E.g., the Loop Invariant and the code it helped to write are the combination of (resp.) a text and a code that eventually form an answer.

The main step in the preprocessing consists in compiling and executing the student's program. Before compiling, the student's pieces of code are injected into a larger file. At the beginning of the semester, when some C language features (*e.g.*, functions) were not yet introduced, it enables to put each piece of code in its own test function and call it at will in a test suite. The file containing student's pieces of code may be compiled with as many other source files as desired: they just have to be put in the right directory with a `makefile`.

One can then see in Fig. 7.3 that the pieces of code are used in several ways to generate three sort of pieces of answers:

Formatted code It is just the texts of all the student's pieces of code that are formatted into a single text string in order to be lexically and syntactically analysed during the correction step (the syntax parser take a C source file content as input and the student's pieces of code, taken separately, might not pass the parsing, hence this formatting).

Compiler output It is the result of the compilation. A compilation failure means that the student did not submit a valid C code. If the compilation failed, the compiler message is modified to be more readable: the student code fragments are added and the line numbers in the compiler output are modified so that they only reference displayed student code. This modification is mandatory since the student code was merged in a bigger file and the original compiler output is related to this file, whose content does not need to (and sometimes must not) be revealed to the student.

Program Result After a first compilation, the student code may be modified to ease correction (see Sec. 7.4) and is thus compiled again[22] into an executable program that is run. This

[22] The `makefile` must accept two targets, "`check`" and "`main`" that correspond respectively to student code syntax check and actual compilation with the teacher test suite. This feature also allows to write conditional preprocessor directives in the source files that include or not code depending on the target.

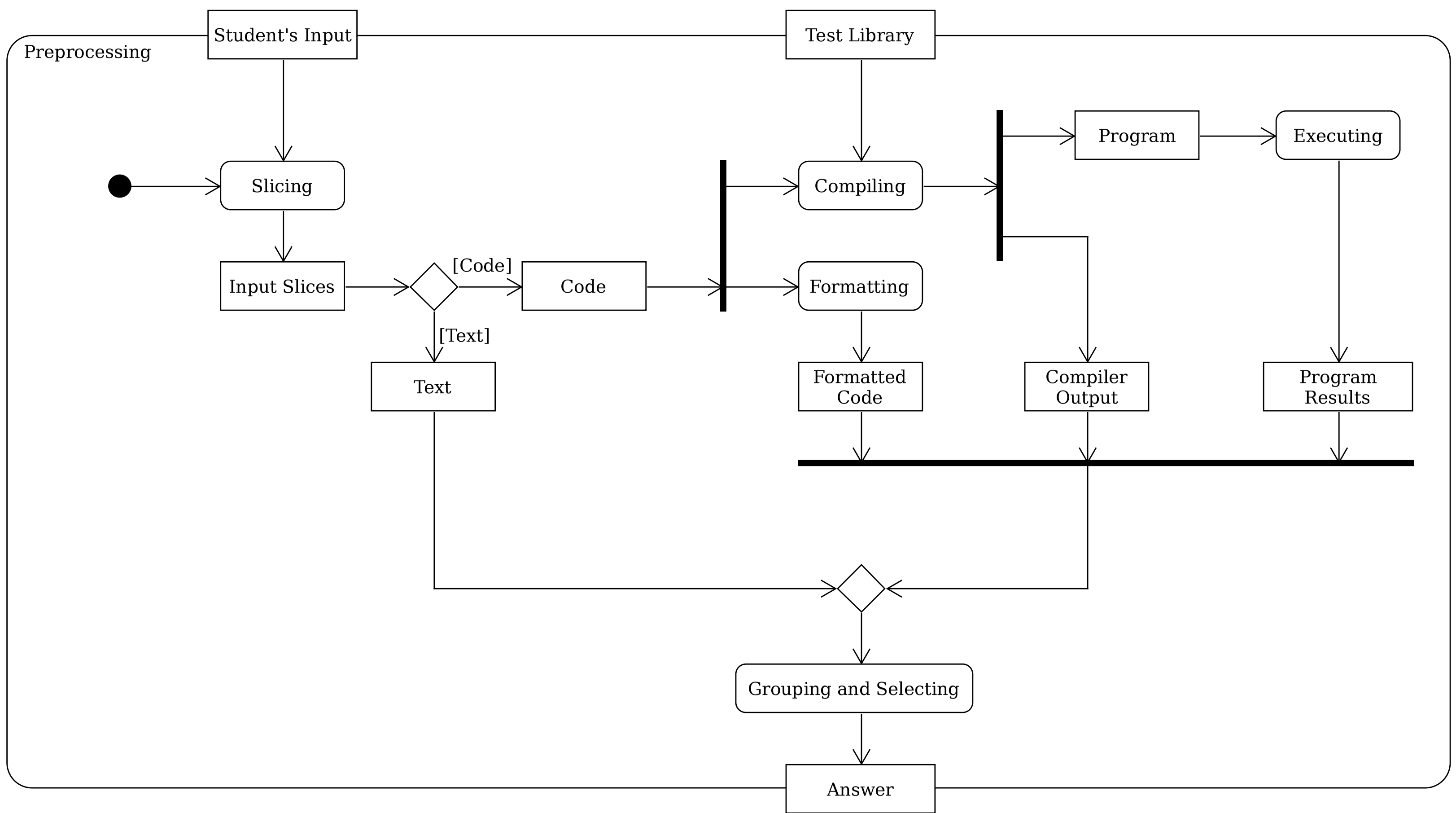


Figure 7.3: Student's Input Preprocessing

executable writes information to the standard output (either because the student was asked to print something in stdout, either because the tests of their program did so – e.g., for displaying the particular parameters values with which their code was tested) and terminates with a certain status number. Both the content of the standard output and the status form what we call the program result.

7.4 Correction: from a Lists of Answers to a List of Marks and Remarks

This steps consists in applying, to each answer produced by the preprocessing step, a correcting function that is in charge of comparing the answer with the expected result(s). CAFÉ was designed not only to be able to perform unit tests (see Sec. 7.4.1) but can also verify if constrains established in the exercise instructions are respected (see Sec. 7.4.2) and assess if the Graphical Loop Invariant Based Programming methodology was followed while programming (see Sec. 7.4.3).

As far as the C language is concerned, the correction step is able to detect infinite loops (see Sec. 7.4.4); to count iterations to enforce eventual complexity constrains (see Sec. 7.4.4.1); to detect out-of-bounds accesses (see Sec. 7.4.4.2) and to mock standard library functions to offer better feedback on their usage (see Sec. 7.4.4.3).

During the correction step, each test, each check can give rise to a remark. Hence each correction function handling an answer will generate a list of remarks and a grade, as depicted in Fig. 7.2. Both of them are eventually handled by the message generation step (see Sec. 7.5).

7.4.1 Correction Principles and Grade Computation

```
1  # CAFE's Python implementation
2  class ExerciseGT(GroundTruth):
3
4     def __init__(self):
5        self.structure = [
6           {'action': self.correct_invariant,
7            'params': {},
8            'weight': 8.,},
9           {'action': self.correct_output,
10           'params': {},
11           'weight': 12.,},]
12
13       # Numbers refer to
14       self.requirements = ((0,2), (4,5,6))
```

Listing 7.3: General Form of an class Inheriting from Groundtruth.

Writing the correction for an exercise in CAFÉ requires to write Python functions that will correct the answers generated from the student file by the preprocessing step. These correction

Table 7.1: Pieces of answer example. They are related to the exercise shown in Listing 7.2

#	Piece of answer
0	Encoded Loop Invariant
1	Loop Variant
2	Text of the code
3	Compilation result
4	Test result 1
5	Test result 2
6	Test result 3

functions must be provided via an object inheriting from the `GroundTruth` class, which contains pre-defined methods (*e.g.*, a method for computing students final mark).

Listing 7.3 shows an example of such a class that specifies how to combine the pieces of answers into answers and which correction method must be called to handle each answer.

The class defines two fields: `structure` and `requirements`. `structure` is a list of dictionaries that contain three informations:

`action` A reference to a method to correct an answer;

`params` A dictionary of named parameters to be passed to the `action` method;

`weight` The weight of the correction function result in the final mark

The `requirements` are used to inform CAFÉ which pieces of answers must be combined to form an answer. Each inner tuple form an answer combination and the tuples order correspond to the actions order in the `structure` list.

Let us take the example of Listing 7.2. During the preprocessing, it will be divided in three slices: Graphical Loop Invariant, Loop Variant and code. The preprocessing step will also add other pieces of answer after the code compilation and the program execution[23]. They are all listed in Table 7.1.

The numbers in the `requirements` value in Listing 7.3 follow the numbering of the pieces of answer from Table 7.1. It means that the method `correct_invariant` must be given the encoding of the Loop Invariant (0) and the code (2) as parameters, whereas the method `correct_output` will received the three tests results (4,5,6).

The Listing 7.4 shows and explains the principal part of a correction function. It is this kind of function references that are passed as values in the `structure` field of the `GroundTruth` (see Listing 7.3, *e.g.*, `correct_invariant`).

The correction function performs unit test, *i.e.*, comparison between the student answer and the expected behaviour.

[23] The number of test results (here: 3) depends on the number of tests launched on student's program.

```
# CAFE's Python implementation
# To be put in an object inheriting of GroundTruth class
def correct(self, student_answer, **kvargs):
    """
       Return: a list of remarks and a grade
    """

    # 1. Compute the expected values or use the ones passed as params
    expected_result = f(student_answer, **kvargs)

    # 2. Analyse the student answer passed
    answer_part = g(student_answer)

    # 3. Create an empty commentaries list and put the grade at maximum value
    coms = []
    grade = TOTAL

    # 4. For one test :

    # 4.1 Add Title and display info
    coms += TITLE("Question X: a title")
    coms += DISP("Some Information")

    # 4.2 Compare answer and expected values:
    test = h(expected_result, answer_part)

    # 4.3 Compare answer and expected values:
    if (test):
       # Success: display a positive message
       coms += DISP(Positive Message)
    else:
       # Failure: add a remark, a code and a priority and decreases grade
       coms += REMARK("This is incorrect because... ", code, priority)
       grade -= penalty

    # Perform other tests if needed
    ...

    # At the end, return the grade (/1) and the commentaries
    return grade/TOTAL, coms
```

Listing 7.4: General Form of a Correction Function. `f`, `g` and `h` are teacher defined operations and can be as complex as necessary.

As far as assessing the program is concerned, the comparison between the program result and the expected result may be processed either by the C code itself that embeds the student code as a subroutine. In this case, the Python correction function just consists in parsing a report that is printed on stdout by the C code.

On the other hand, the C code may just invoke the student code and make sure that its result is printed on the standard output. The Python correction function then parses this result and compute the comparison with the expected behaviour.

In both case, we recommand to make the program that launches the student code print

the parameters with which the student code was invoked on stdout in order to later use this information while generating the response message.

One can see in Listing 7.4 that the function generates a list of remarks (`TITLE`, `DISP` and `REMARK`). They refer to ways of providing student with feedback and feedforward (See Sec. 7.5). In addition to a list of remarks, the correction function must also return a grade that is a real number in [0..1].

7.4.2 Enforcing the exercises constrains

Some exercise instructions may contain constrains such as using a particular loop type or limit the number of selection statements (i.e. `if` and `switch` [180]), etc. In order to enforce such constrains, CAFÉ is able to perform a lexical and syntactical analysis of C source code thanks to a module based on `pycparser` [22].

By default, the analysis of the code computes all this information that may be used in the following of the correction if it is relevant :

1. Each type of loops is counted: `while`, `for`, and `do-while`.

2. All the jump statements are counted. Most of time **[Always?]**, jumping from the *Loop Body* does not respect the GLIBP methodology: `goto` – always to penalise its usage [54] – , `continue`, `break` (used outside switches), and `return`.

3. All Identifiers, arrays, and initial values are registered: the identifiers themselves as well as their initial values are stored in a dictionary; in particular, the identifiers used as indices are isolated and the arrays names are gathered too.

   The list of the arrays names allows to limit their usage to those defined by the exercise instructions. The list of indices and the initial values are helpful to confront a code and its Loop Invariant (see below).

4. All the selection statements (i.e., `if` and `switch`) are counted so that their use may be constrained. Two counts are carried out: inside and outside the loops.

5. All the called functions are remembered in a set. It allows either to restrict the use of particular libraries (e.g., `math.h`) or to check if a particular function is actually properly called by a code (e.g., in an exercise tackling code modularity). In addition, it is better to make sure that a student code compiles even if it means including a large number of needless (or even forbidden) libraries: the called functions set allows eventually to tackle the instructions enforcement and to deliver a precise and more understandable feedback message in case of problem. Otherwise, trying to compile a code that does not include a particular library leads to a linking error that is often hard to decipher by a neophyte.

6. CAFÉ can be asked to list the operands of a certain operator. It will produce two separate lists:

    - a list containing both operands of all the matching binary operator occurrences;
    - a list of the R-values of all the assignment-expressions if the considered operator has an assignment-operator version.

    An example of use-case for this feature is an exercise that would ask to compute a value with only multiplication and division by 2. CAFÉ would therefore make 4 lists of operands for the operators `*`, `/`, `*=` and `/=`.

7. For each loop, a list of information is gathered: its type, its nesting level, its guard, and the list of identifiers in its guard. The two last pieces of information are used to confront a code with its Loop Variant (see below).

7.4.3 Assessing the Compliance with the GLIBP Methodology

Regarding the Loop Invariant, the correction step verifies that what has been proposed as replacement of the boxes (see Sec. 3.2) 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 both in the expression and in the code and that the value of the expression decreases at each iteration.

There is always the risk a student will submit their exercise with the Loop Invariant produced after the code, which clearly violates the methodology we propose (see Chap. 2). 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 initialised accordingly (*i.e.*, this corresponds (partially) to ZONE 1 in the GLIBP methodology). 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 themself.

7.4.4 Detecting Infinite Loops

During the preprocessing step, the student's code is always run with a timer. If the student program is still running at the timeout, it is terminated (a TERM signal is triggered) and the

status of the student's program is equal to 124 rather than 0. This execution status is part of the answer delivered to the correction step.

A timeout terminated program can then be considered as an infinite loop in the correction step. Of course, the time allowed for the program execution must be chosen to be sufficiently long, depending on the particular programming task asked as exercise to avoid false-positives.

7.4.4.1 Counting Iterations

As mentioned in Sec. 7.3, CAFÉ can modify C code in order to count the number of iterations. This may be useful to enforce some complexity constrains. Here is how it is done. First, a special C header is included in the student code. This header contains, among others things, two macro definitions: `while(...)` and `for(...)` and global variables declaration (`for_loopcounter` and `while_loopcounter` (see Listing 7.5)). Both macros are variadic (the number of their arguments is variable) just to allow the use of the comma (i.e., `,`) C operator. We remind the reader that the comma operator takes to operands `op1` and `op2`. Evaluating the expression `op1 , op2` consists in evaluating `op1` then `op2` (the evaluation order is guaranteed). The value of the expression `op1 , op2` has the same value as `op2` and the evaluation of `op1` is thus only useful if it has side effects (e.g., in this use case, incrementing a counter at each iteration).

```
1 #define while(...) while(++while_loopcounter, __VA_ARGS__)
2 #define for(...) for(__VA_ARGS__ , ++for_loopcounter)
3 
4 extern int for_loopcounter;
5 extern int while_loopcounter;
```

Listing 7.5: Macro Definition to Count Iterations

The results of these macro expansions is presented in the Listings 7.6, 7.7, and 7.8. One can see in these Listings that `while_loopcounter` will contain the number of guard evaluations (i.e., the number of iterations +1) after the execution of a `while` loop and the number of iterations after a `do...while` loop execution.

As far as `for_loopcounter` is concerned, it will contain the number of iterations of a `for` loop after its execution. If several loops are nested, the interpretation of the value of `while_loopcounter` and `for_loopcounter` depends on the loops types and their respective number of iterations. For example, if two `while` loops are nested and if the inner loop takes a iterations while the outer one takes b, `while_loopcounter` will contain the value $(a+1)\times b+1 = ab+b+1$.

```
1 // Before macro expansion:
2 while(i < N)
3 {
4    // Loop Body
5 }
6 
7 // After macro expansion:
```

```
8  while(++while_loopcounter, i < N)
9  {
10     // Loop Body
11 }
```

Listing 7.6: While Loop Modification to Count Iterations

```
1  // Before macro expansion:
2  do
3  {
4      // Loop Body
5  }
6  while(i < N);
7
8  // After macro expansion:
9  do
10 {
11     // Loop Body
12 }
13 while(++while_loopcounter, i < N);
```

Listing 7.7: Do While Loop Modification to Count Iterations

```
1  // Before macro expansion:
2  for(i = 0; i < N; i++)
3  {
4      // Loop Body
5  }
6
7  // After macro expansion:
8  for(i = 0; i < N; i++, ++for_loopcounter)
9  {
10     // Loop Body
11 }
```

Listing 7.8: For Loop Modification to Count Iterations

7.4.4.2 Detecting Out-Of-Bound Accesses

C language does not offer any mechanism to verify that an array is accessed within its boundaries. The Gnu C Compiler (`gcc`) offers a way of warning about “subscripts to arrays that are always out of bounds” [176] thanks to the option `-Warray-bounds` but that does not prevent the user from out-of-bounds accesses at the runtime. The solution we propose to detect these illegal accesses is based on the fact that the C standard guarantees that theses two expressions, in which both `E1` and `E2` are expressions, are identical [180, p. 58]:

- `E1[E2]`

- `(*((E1) + (E2)))`

In our solution, instead of manipulating a pointer to an array, we use a pointer to a more complex structure, called `fake_array` that contains a bigger array than needed. This bigger array embeds the array actually used by the student but allow them to write a code that overflows without triggering a segmentation fault.

The `fake_array` pointer is always handled by methods ensuring the data consistency. All of this is hidden from to the student whose code is assessed for obvious pedagogical reasons. Listing 7.9 shows a function implemented by a student who used `A[i]` to access the (i+1)th element of the array `A`. Between the two compilations[24], the left and right square brackets were respectively replaced by left and right parentheses thanks to a script launched by the `makefile`. Now, the code contains `A(i)` that is the syntax of a function call or a macro with parameters.

```
1 void student_function(int *A, unsigned int N)
2 {
3    for(int i = 0; i <= N; ++i)
4       printf("%d\n", A(i)); // modified on purpose from
5     //printf("%d\n", A[i]); thanks to, e.g., a script
6 }
```

Listing 7.9: An Example of Student Code Involving an Array

By using the proper macro definition (see Listing 7.10), we can replace `A(i)` by the dereferencing of the result of a call to a a function, called `get`, that returns a pointer.

The idea behind this operation is that `get(A, i)` returns the value `A + i` that once dereferenced is equal to `(*(A + i))` that is equivalent to `A[i]`, as we previously mentioned.

In the Listing 7.10, one can see that the type of the array behind which we hide a more complex structure (i.e., `FAKE_TYPE`) is also a macro and is hence configurable at compilation time. The macro defined at the line 1 works only with the array `A` and must be repeated if there are other arrays. As a consequence, the names of the arrays manipulated in such corrected exercises must be known and fixed in the instructions.

```
1  #define A(...) (*get(A, (__VA_ARGS__)))
2
3  #define FAKE_TYPE int
4
5  // Allocate space for the structure that replace the array and return a pointer
6  // to it.
7  // @param the size of the array the user thinks they is using.
8  FAKE_TYPE *new_tab(int);
9
10 // Get the element at index 'index' in the fake array and check for bad
```

[24] We remind the reader that the code is compiled twice with two different `make` targets, enabling us to also run scripts between the two compilations

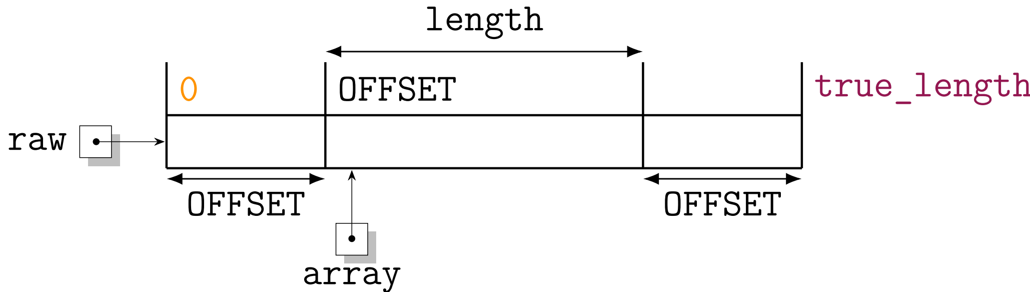


Figure 7.4: The array stored in the struct `fake_array`. The value of `true_length` is equal to $2 \times$ `OFFSET` + `length`. The two sub arrays `raw[0..OFFSET-1]` and `raw[true_length-OFFSET..true_length-1]` are supposed to be left empty during a normal use and any modification correspond to an out-of-bounds access.

```
11 // accesses.
12 // @param fake_array a pointer to a more complex structure
13 // @param index the index in the array the user thinks they is using.
14 // @return The expected value at the index in the array if the index is in the
15 //         array bounds. Undefined otherwise
16 FAKE_TYPE *get(FAKE_TYPE *fake_array, int index);
17
18 // Get the number of out-of-bounds accesses intended using a fake_array since
       its
19 // creation
20 // @param fake_array a pointer to a more complex structure
21 // @return the number of out-of-bounds accesses involving the fake array.
22 int get_bad(FAKE_TYPE *fake_array);
23
24 // Other methods to ease the fake array manipulation not detailed here.
```

Listing 7.10: Header for Out-Of-Bound Accesses Checking

The Listing 7.11 shows the details of the structure `fake_array` that contains an array, its length and the number of bad accesses to this array. When allocating a new `fake_array` to be used with a student code, we allocate a larger array than needed to prevent student's bad accesses to cause segmentation faults. The structure `fake_array` stores thus both the real array and its length (resp. `raw` and `true_length`) and the array intended to be actually used by the student and its length (resp. `array` and `length`). `array` is thus a sub array of the array `raw`, as it can be seen in Fig. 7.4.

In Listing 7.11, one can also see that every call to `get` function register bad accesses to the structure `fake_array`. All the functions for manipulating the `fake_array` structure receive a pointer to the `FAKE_TYPE` (the type the user thinks they is using) and must thus first cast it to the proper type before accessing its fields.

```
1 #define OFFSET 4
```

```
2
3 struct fake_array{
4     int bad_accesses;
5     int true_length;
6     int length;
7     FAKE_TYPE *raw;
8     FAKE_TYPE *array; // Points to a sub array of 'raw'
9 }
10
11 FAKE_TYPE *new_tab(int N)
12 {
13     struct tab *T = malloc(sizeof(struct tab));
14
15     T->raw = (FAKE_TYPE *) calloc(N + 2 * OFFSET, sizeof(FAKE_TYPE));
16     T->array = T->raw + OFFSET;
17     T->length = N;
18     T->true_length = N + 2 * OFFSET;
19     T->bad_accesses = 0;
20
21     return (FAKE_TYPE*) T;
22 }
23
24 FAKE_TYPE *get(FAKE_TYPE *fake, int i)
25 {
26     struct fake_array *obj = (struct fake_array *) fake;
27
28     if(i < 0 || i >= obj->length)
29         ++(obj->bad_accesses);
30
31     return obj->array + i;
32 }
33
34 int get_bad(FAKE_TYPE *fake)
35 {
36     struct fake_array *obj = (struct fake_array *) fake;
37     return obj->bad_accesses;
38 }
```

Listing 7.11: Module for Out-Of-Bound Accesses Checking

The Listing 7.12 shows a student's code after the macro expansion. The Listing 7.13 presents a use case for checking out-of-bound accesses: first the `fake_array` is created, then it must be initialised. After that, the student's code may be called. Eventually, the number of bad accesses can be retrieved from the `fake_array` structure to be, e.g., printed to be forwarded to the correction step as part of the student's answer.

```
1 void student_function(int *A, unsigned int N)
2 {
```

```
3     for(int i = 0; i <= N; ++i)
4         printf("%d\n", (*get(A, (i))));
5 }
```

Listing 7.12: Student's Code after Macro Expansion

```
1  void assistant_function(void)
2  {
3      int N = ...;
4      // 1. Create a fake_array
5      int *fake_array = new_tab(N);
6
7      // 2. Initialise the fake array
8      Functions not presented
9
10     // 3. Call student's code
11     student_function(fake_array, N);
12
13     // 4. Using the number of illegal accesses, e.g. :
14     printf("OOB accesses: %d\n", get_bad(fake_array));
15 }
```

Listing 7.13: Out-Of-Bound Accesses Checking Use Case Example

7.4.4.3 Leveraging Macros to Mock Standard Functions

In order to assess the correct utilisation of C standard library functions such as `malloc`, `free`, or even `fopen`, `fprintf`, etc., one can again use macros that expand into corresponding mock functions. There are several advantages to do so:

- It avoids to make student code actually manipulate dynamic memory or files ;
- The execution of the student code does not fail on the first incorrect behaviour and all their code may be assessed throughout a complete use case scenario ;
- All the corner cases of the standard functions can be assessed. For example, it is easier to mock a `malloc` function returning a `NULL` pointer in case of error [76] rather than try to reproduce this behaviour during a test ;
- It allows to make better feedback since a macro expansion can add information, e.g., the calling function name and line number as it can be seen in Listing 7.14. The remark delivered to the student is therefore more precise (see Listing 7.15).

Mock functions thus defined do not need to exactly mimic the behaviours of the mocked functions. Instead, they just have to offer the same interface and their definitions often just consist in performing tests on their parameters and on other functions possibly called before

them. For example, a `mock_free` function replacing a call to `free` would check that `malloc` (i.e. `mock_malloc`) was previously called to allocate the memory pointed by the pointer passed as its parameter.

```
// __LINE__ expands into the line number where malloc is called
// __func__ expands into the function name string in which malloc is called
#define malloc(...) mock_malloc((__VA_ARGS__), __LINE__, __func__)

// Mock malloc behaviour and perfom verifications.
// @param size same parameter as malloc
// @param line a line number where the function is called
// @param function the function name where the
// @return a pointer mocking the result of a call to malloc(size)
void *mock_malloc(size_t size, int line, const char *function);
```

Listing 7.14: Example of Mock Function Declaration: `malloc`

```
FILE *mock_fopen(const char *path, const char *mode, int line,
                 const char *function)
{
   // Decide if the call should success depending on the scenario, in a
   // dedicated function
   if (fopen_failure())
      return NULL;

   // Can be used later in feedback
   printf("Function %s, line %d: opening of file %s in <%s> mode",
          function, line, path, mode);

   // Redirect (here, writing) to standard output.
   return stdout;
}
```

Listing 7.15: Example of Mock Function Definition: `fopen`

7.5 Message Generation: from a List of Remarks to a Message Containing Information, feedback and feedforward

For each student's answer, the correction steps generates a mark and a list of remarks. To illustrate this, the right part of Fig. 7.5 shows the type of the remarks constituting the final message that is delivered to the student. They are four types of remark generated by the correction step:

TITLE It is used to structure the feedback and to help the student to understand which part of their submission is commented (See Fig. 7.5, in green). A TITLE must always start each

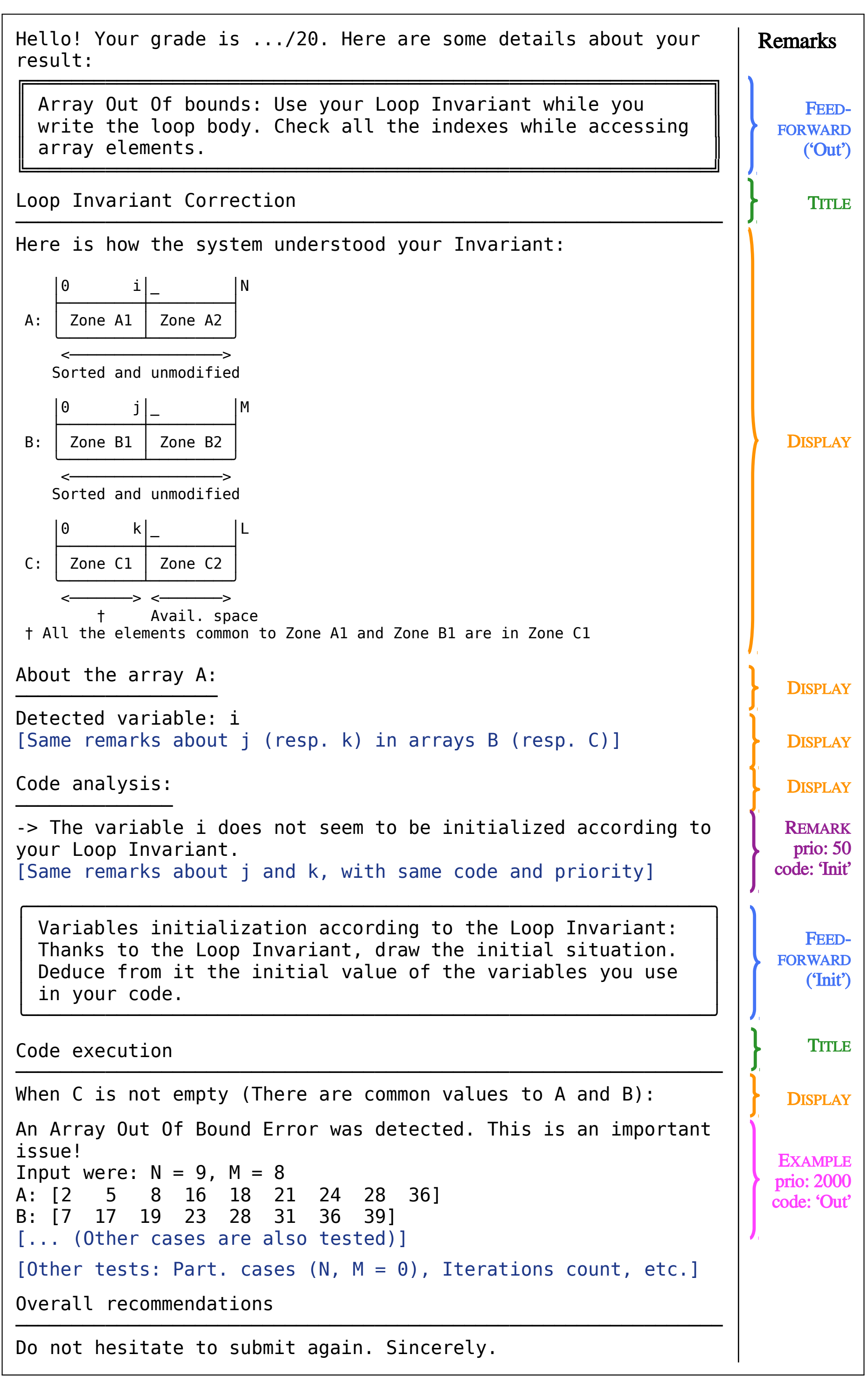


Figure 7.5: An example of Message. The right column indicates which type of remark was used to build each part of the message (See Sec. 7.5 and 7.5.1).

new question correction to ensure the feedback readability. Each TITLE defines thus a new **section**.

DISPLAY It is used to display unconditionally a message, for instance, an introductory text (See Fig. 7.5, in orange).

REMARK It is used to provide comments on a student's performance (See Fig. 7.5, 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. 7.5.1.

EXAMPLE Like a Remark, it contains a message, a code, and a priority (See Fig. 7.5, in magenta). Unlike a Remark, the printing of their associated message is not mandatory. This feature can be helpful to shorten the final message.

The message generation consists in merging and formatting all these remarks into a single text. The remarks order is preserved. Some feedforward remarks can be inserted in the remarks (See Fig. 7.5, 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 remarks are framed to increase the readability of the message, as illustrated in Fig. 7.5.

7.5.1 Adding feedforward

The process of adding feedforward remarks is shown in Fig. 7.6 that consists of two columns. The left one represents the remarks as they were generated by the correction step while the right one shows the remarks list after the feedforward information was added. For more readability, the colour code used in Fig. 7.6 is the same as in Fig. 7.5.

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 the left of Fig. 7.6). Each code refers to a feedforward message that suggests an action to be taken to correct the code: it can be a short remainder of the GLIBP methodology, a link to a theoretical lesson to be read again or more generally a piece of advice.

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. 7.6, the REMARK with the code "Var" is issued two times, hence the total priority of this code will be $200 + 200 = 400$. The feedforward message associated with the n codes having the highest total priority are actually inserted in the message to the student. Thus, the number of feedforward messages is limited to avoid overloading students with a too long message. However, the priority mechanism helps to select the most relevant messages, ensuring the message personalisation. The number of feedforward messages that are printed (n) was fixed to five after a discussion with

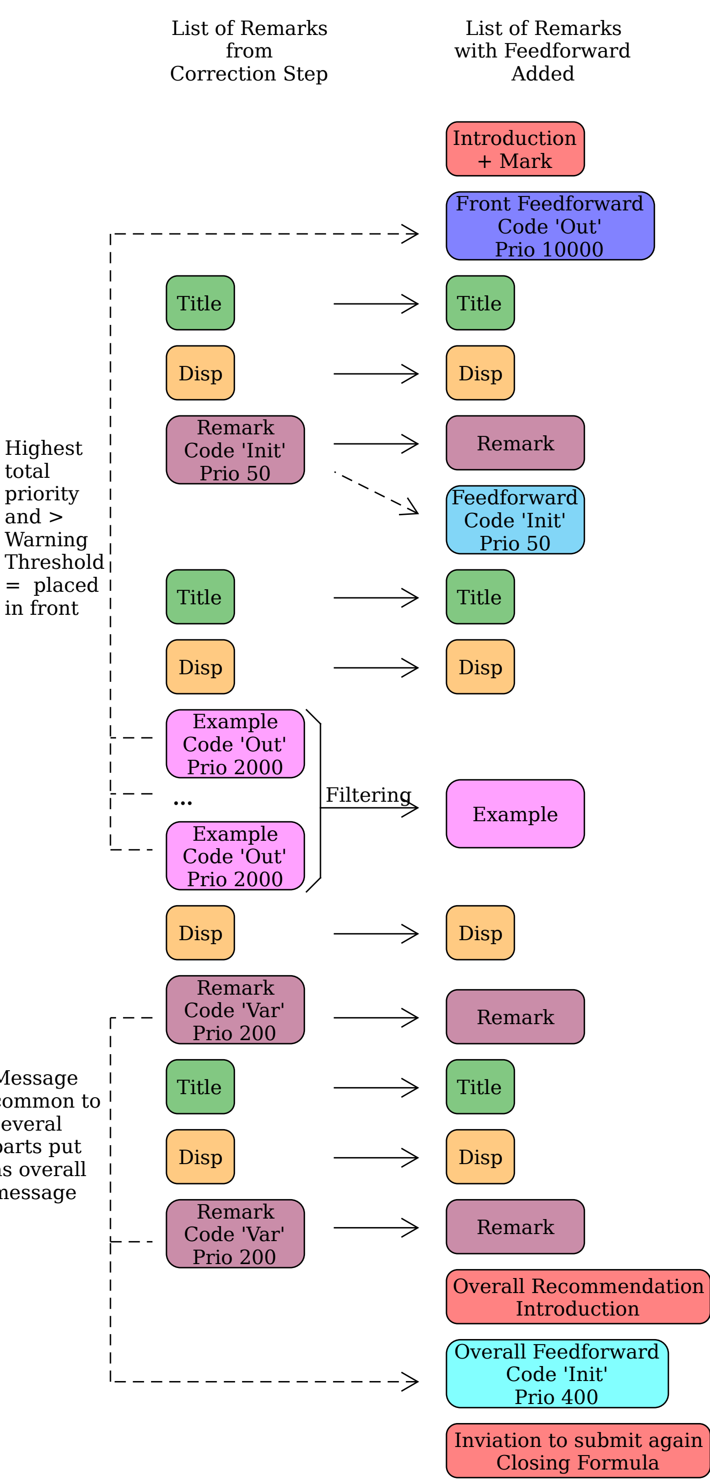


Figure 7.6: Adding Feedforward during Message Generation

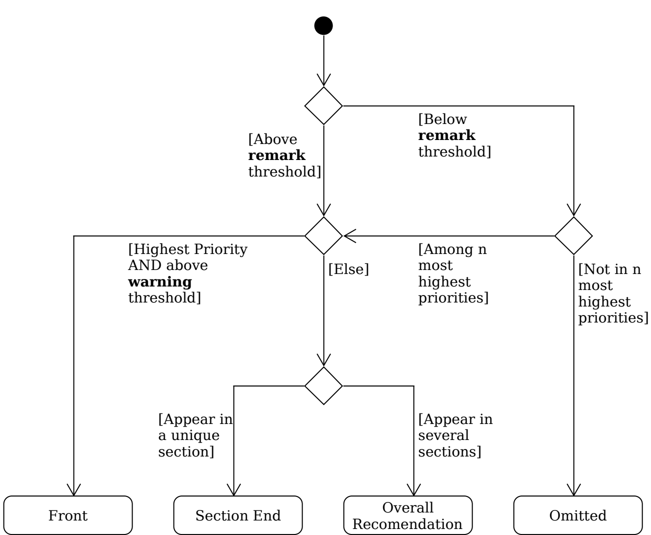


Figure 7.7: Determining the position of a feedforward message

a delegation of students during a focus group (see App. B). In case of numerous remarks of high priority, that are above a **remark threshold**, the filtering is by-passed and they are still printed.

The feedforward messages can be inserted at several places in the message: in front, at the end of a section, as an overall recommendation or omitted, as described below and illustrated in Fig. 7.7.

In Front The message corresponding to the highest total priority that is above a **warning threshold** is placed in at the very beginning of the message, as a warning, to draw the student's attention. For example, in Fig. 7.6, this is the case of the feedforward labelled "Out", because of its highest priority that is above the warning threshold.

At the end of a section All the feedforward messages that correspond to a unique section of the message (defined by a TITLE, see above) are displayed as close as possible to the question they are related to, i.e., at the end of the section

As an overall recommendation All the feedforward messages that correspond to several sections are placed at the end of the message as an overall recommendation.

Omitted All the other feedforward messages that are not among the highest priorities nor above the remark threshold are simply omitted.

At present, all the remarks particular values (*i.e.*, their priorities as well as the remark and warning thresholds) must be defined by the programmer of an exercise correction. From an exercise to another, the same remarks could have different priorities depending on the focus the corrector wishes to give to a certain subject in the context of the exercise.

7.6 Plagiarism Detection

CAFÉ does not detect plagiarism itself. However, it can preprocess students submissions files to create for each student, a syntacly correct C source file (which the original student file is not) that can be understand by a plagiarism detector, such as MOSS [166].

7.7 Conclusion and Future Work

This chapter presented in depth how CAFÉ works and provided a lot of implementation details. CAFÉ not only correct and grade student programs but also assess the use of GLIBP methodology and make possible to enforce programming constrains, be they lexical or related to the code complexity.

Some verifications require even to modify the student code and these operations are hidden from them and performed seamlessly, while enabling to better assess their performance. It is worth noting that some errors, such as infinite loops and out-of-bounds accesses would be much more difficult to systematically detect if the correction was performed by hand.

Eventually, CAFÉ provide the student with a message containing feedback and feedforward that is tailored from the student's own performance.

As a future work, extending further CAFÉ's features would always be an interesting idea but we think that these points deserve to be addressed in the first place:

- Give CAFÉ some memory to remember the errors a student committed in previous exercises. It would help to provide them with a better feedback and feedforward, *e.g.*, by focusing on the errors they keeps doing or to personally congrats them when they eventually get rid of an error theystruggled with.
- Change the way the student submit its Graphical Loop Invariant and its code. Graphical Loop Invariant Drawing Editor (GLIDE) (see Sec. 3.1) could be adapted to serve as a GUI to submit Graphical Loop Invariant to CAFÉ.
- Upgrade the form of the message provided by CAFÉ. A better message would contain colours, image (*e.g.*, of the Graphical Loop Invariant), better text, hyperlink to the course web page, etc.

CHAPTER 8

CAFÉ EVALUATION

THE PREVIOUS CHAPTER introduced the program Correction Automatique et Feedback des Étudiants (CAFÉ) which provides, as the name indicates, *feedback* message to students. This chapter tackles the answer to the research question:

RQ 2. 2 *How the feedback produced by* CAFÉ *is received by students?*

To answer this question, we analyse performance and perception data. The rest of the chapter is organised as the following: Sec. 8.1 introduces our methodology, Sec. 8.2 presents the results and Sec. 8.3 discusses them. Finally, Sec. 8.4 draws a conclusion.

8.1 Methodology

Following Verpoorten et al. [185] framework (See Sec. 4.2), we analyse performance and perception data. However, we do not include participation data to our analysis. We believe Participation data is not relevant here[25].

8.1.1 Performance Data

Here, by performance, we mean the success in submitting an answer to CAFÉ, regardless its correctness (which is further tackled in Chap. 10). Before any correction step, a submission process can fail in three main ways:

1. The file submitted to CAFÉ is not correctly formatted (*e.g.*, submitting a `docx` file instead of a `txt` file);

[25] Chapters 10 and 11 analyse these kind of data in the context of a teaching activity build upon CAFÉ.

2. The instructions that are specific to a particular programming activity were not followed (*e.g.*, the submitted file has a wrong file name);

3. The code that was submitted could not be compiled.

CAFÉ has been used to correct students answer for 5 years in the context of a Programming Challenges Activity (PCA) (see Chap. 9 and following). We collected and analysed the messages delivered by CAFÉ after each submission between academic years 2016–2017 and 2019–2020 for 6 Challenges each year.

8.1.2 Perception Data

Table 8.1: Surveys Respondents. N is the total number of students. # is the number of respondents. Two percentages are provided: %(Total) is computed on the total number of enrolled students and %(Part.) on the number of Participants in the Final Exam

		Respondents		
Year	**N**	#	**% (Total)**	**% (Part.)**
2017–2018	72	28	38.9	50.0
2018–2019	76	22	28.9	47.8
2019–2020	82	16	19.5	26.7

Between academic years 2017–2018 and 2019–2020, surveys were carried out after the final exam. The surveys were online and anonymous in order to let the students express themselves freely. These are the same *one time* surveys as the ones presented in Chap. 4 but we focus here on the effect of the message delivered by CAFÉ on students perception and behaviours. Table 8.1 recalls the number of respondents to each survey and compare them to other course statistics[26].

8.2 Results

8.2.1 Performance

The students performance in submitting files to CAFÉ is shown in Fig. 8.1. As it can be seen in the Figure, the majority of submissions are successful, except for Challenges #0 and #1 in 2018–2019.

As far as academic year 2016–2017 is concerned, the number of formatting errors is quite constant over the Challenges (see Fig. 8.1a). The instructions issues mainly concern the Challenge #0. Compiling error show a peak for Challenge #1 then slightly increase from Challenges #2 to #4.

From 2017–2018 to 2019–2020, a decrease in the number of submission problems can be

[26] This Table is identical to Table 4.3.

(a) 2016–2017

(b) 2017–2018

(c) 2018–2019

(d) 2019–2020

Figure 8.1: Submissions problems using CAFÉ

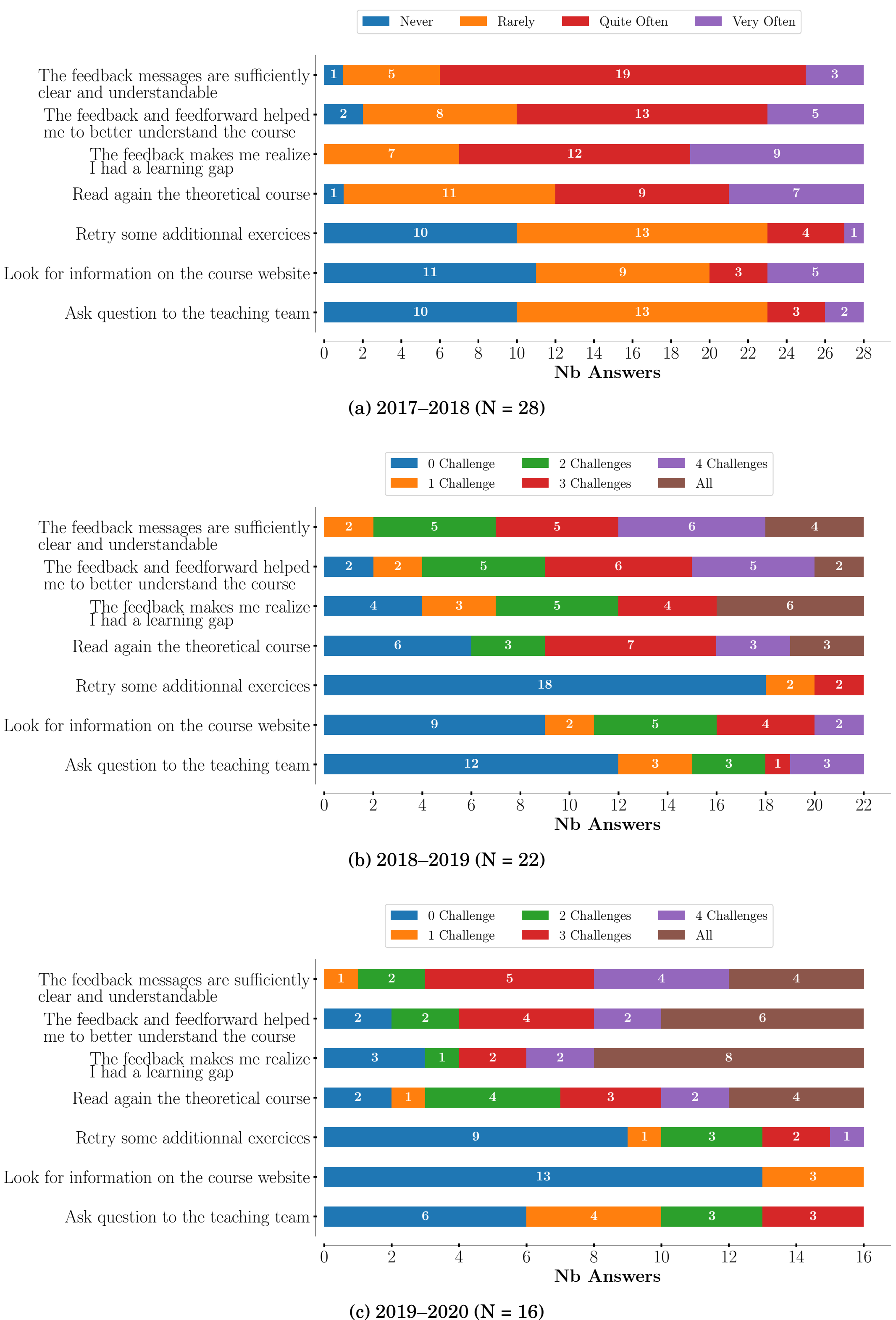
observed as the Challenge number increases (see Fig. 8.1b, Fig. 8.1c and Fig. 8.1d). The decreases in formatting and instructions issues are steeper as these problems arise mainly in the first Challenges. In 2018–2019 and 2019–2020, one can observe an increase for the last Challenge #5, that was problem-free the previous years.

8.2.2 Perception

Fig. 8.2 shows the students opinions about the message they received from CAFÉ.

First, the students acknowledge the messages are sufficiently clear and understandable (22/28, "quite often" and "very often" in 2017–2018; 15/22 for 3 Challenges or more in 2018–2019; 13/16 for 3 Challenges or more in 2019–2020 – see Fig. 8.2, 1st lines).

Second, When asked if the feedback and *feedforward* helped them to better understand the course content, 18/28 respondents reported that this was the case "quite often" and "very often"



(a) 2017–2018 (N = 28)

(b) 2018–2019 (N = 22)

(c) 2019–2020 (N = 16)

Figure 8.2: Students' survey responses. The first three lines correspond to information the students received through feedback from CAFÉ. The last four lines correspond to the students' reported reactions prompted by feedback.

in 2017–2018; most of them reported that this was the case for three Challenges or more (13/22 in 2018–2019, 12/16 in 2019–2020 – see Fig. 8.2, 2nd lines).

When asked if the feedback made them realise that they had a learning gap, a majority of them answered it was so in 2017–2018 (21/28, “quite often” and “very often”) and in 2019–2020 (12/16 for 3 Challenges or more). The figures nearly reach half of the respondents in 2018–2019 (10/22 for 3 Challenges or more – see Fig. 8.2, 3rd lines).

The rest of Fig. 8.2 gives insight into how the students used the feedback/feedforward between consecutive submissions. In fact, most of the respondents reread the theory course (16/28 quite and very often in 2017–2018, 13/22 in 2018–2019 and 9/16 in 2019–2020 for three Challenges or more – see Fig. 8.2, 4th lines). Much less students retried some additional exercises (Fig. 8.2, 5th lines), looked for information on the course website (Fig. 8.2, 6th lines) or asked the teaching team questions (Fig. 8.2, 7th lines).

8.3 Discussion

8.3.1 Performance

Trends in 2016 are very different from the other years (see Fig. 8.1a) because it was the first year CAFÉ was used and the program itself was modified in order to be more resilient to submission problems. Some recurring formatting issues were indeed addressed and solved thanks to submissions failure reports. The message from CAFÉ in case of submission problem also has been improved. This explains that the next years, the number of formatting errors drastically decreases after the first Challenges.

There was no compiling errors in Challenge #5 in 2016–2017 and 2017–2018 just because the activity did not consist in programming. This was no more the case the following years. The new version of the activity includes exercises about dynamic allocation and pointers (more information in Chap. 9) that may confuse students and make the compiling errors number rise.

The down trends from the first Challenge to the last, observed from 2017–2018 to 2019–2020 (see Fig. 8.1b, Fig. 8.1c and Fig. 8.1d) can be explained in two ways. First, the students learn from CAFÉ message how to submit properly and try to compile their code before submitting it to CAFÉ. Second, we should mention that the number of students submitting to CAFÉ decrease from an Challenge to another (this is not visible in Fig. 8.1 but is studied in Chap. 11). With this in mind, one can argue that only engaged students keep participating in the Challenges and try to make as few errors as possible while submitting to take benefit from the message CAFÉ delivers.

8.3.2 Perception

Concerning the clarity and readability of the message generated by CAFÉ, students acknowledge their quality but there is still room for their improvement.

As for the students’ reactions to the feedback and feedforward (see Fig. 8.2, lines 4 to 7), it is

not surprising that a majority of the respondents reread the theory course for more than three Challenges because CAFÉ directs them specifically to the course (e.g., gives the exact location of the relevant subsection). The other actions are less often explicitly suggested. Contact with the teaching team is the last resort if the student has a question about the feedback or a problem with the CAFÉ system itself. However, CAFÉ has been designed to limit that kind of interaction.

With regard to the feedback portion of the information transmitted by CAFÉ about the students' performance (see Fig. 8.2, lines 2 and 3), the data tends to show that it is useful for the students to know where they went wrong, if we follow the classification by Keuning et al. [101].

8.4 Conclusion

This chapter studied how the message generated by CAFÉ was received by the students. Data shows that during the year, it helps students reducing the number of submission errors.

In most cases, students acknowledge the clarity and the understandability of the message they received. It helps them to understand the course and to realise whether they have a learning gap, even providing them with references to the course notes to overcome it, thus helping them in their learning.

In the future, to further improve the message from CAFÉ, one could think to no longer use a text output. Historically, the message was mailed to the student but nowadays, students read it on the website of the submission platform on which CAFÉ is run. Giving up text for a better format would allow for better graphics and colours in the message.

As far as the feedback and feedforward are concerned, they are designed by the teaching team for each particular Challenge and it may be a time consuming task. A trade-off should be found between the quality of the message and the time taken to program it.

If CAFÉ was more integrated to the GLIBP methodology, *e.g.*, if the Graphical Loop Invariant Drawing Editor (GLIDE) (see Sec. 3.1) and CAFÉ were used together, one could more easily measure thanks to GLIDE the effect of the message received from CAFÉ on the student's Graphical Loop Invariant and codes. This would enable to tailor CAFÉ's message even more for a particular programming exercise and a particular student.

Part III Programming Challenges Activity

CHAPTER 9

Introduction to the Programming Challenges Activity

THIS CHAPTER presents the evolution of our *CS1* course taught at the University of Liège, Belgium. Over the last seven years several teaching activities have been thought to complement traditional theoretical courses and exercise sessions in order to promote students' engagement. The result is aligned with (i) the principles of Assessment for Learning (AfL), which consists in leveraging the assessment to improve the students learning, and (ii) the concept of blended learning, (*i.e.*, mixing in-class and on-line teaching activities). This chapter describes the difficulties the students faced and what we implemented to assist our course evolution, in particular a Programming Challenges Activity (PCA). The subsequent chapters in this part provide an evaluation of the PCA. Chap. 10 focuses on the students' reception of the PCA and Chap. 11 analyses the contribution of the PCA to the course in line with the AfL.

9.1 Need for a PCA

Teaching Introduction to Programming (*i.e.*, CS1) is known to be a difficult task for many students and has been the topic of a large number of research studies [131, 139]. In recent years, various studies have demonstrated that, often, students following a CS1 class encounter difficulties in understanding how a program works [168], how to design an efficient and elegant program [45], dealing with loops and conditional [37] , problem solving and mathematical ability [139], and in checking whether a program works correctly [21]. These are particular examples of difficulties faced by students we also observed in our CS1 class.

For a long time, the failure rate and withdrawal ratio of our students have been high, as it is the case with other CS1 courses [17, 187]. Although the situation has slightly positively evolved in the recent years [23], this is not the case in Belgium where the failure rate in Belgium for 1st year student is still around 70% [8].This figure can be partially explained by the open access

policy for higher education in Belgium. This has meant that some students come to university without important background skills for particular courses of study. However, this situation should not be taken as a pretext for inaction but rather as a call for the improvement of courses and teaching.

Literature in educational psychology has highlighted many predictors of student success in higher education [59] . Among the various parameters favouring it, some can be influenced by the teacher practices, namely those that lead to more student engagement [127].

A few years ago, we decided to make our CS1 course change from a traditional course (*i.e.*, ex cathedra theoretical lessons with exercises sessions) towards a new approach that would promote engagement, potentially be more motivating for first year students, and that would provide them with more feedback. Additionally, we also wanted students to practice more.

When we began the design of Correction Automatique et Feedback des Étudiants (CAFÉ) (see Chap. 7), we identified we could leverage its usage to mould our course on the AfL approach [147, 164, 189], which seemed to us a promising way to achieve our goals. Sambell et al. [164] describes AfL as:

> "[A]n integrated approach to teaching, assessment and supporting student learning. Our view of assessment is broad. It includes summative assessment activities but also assessment which plays a vital role in improving students' progress and attainments and is embedded in teaching." (p. 147)

This part discusses that evolution of our CS1 course over the last seven years. We believe that experience gained with this course evolution is quite general and could be easily transferred by any educational team eager to adopt AfL in their course.

9.2 Evolution of the CS1 Course

Our CS1 course is provided during the first semester at the University of Liège (between mid-September until mid-December). During the All-Saints week, mid-term exams (Mathematics, Physics, CS1, and Foreign Language) are organised for first year students. After two revision weeks in December, the final exam is held in January. The CS1 course has been offered to students since 2013 and, originally consisted of traditional theoretical lectures, exercises (*i.e.*, programming tasks on paper), and lab sessions (*i.e.*, programming tasks in front of computers).

Regarding the assessment, we followed Faculty guidelines by organising the mid-term exam and the final exams. In order to ensure the students learn the theory (*i.e.*, the rules of the programming language that must be known to ensure the course understanding as the semester goes on), five short multiple choices questions (MCQ) tests [38] were organised during the semester. In 2015, due to the low passing rate in the Computer Science section, remedial classes were introduced and made mandatory for students failing at the mid-term exams (not only for the CS1 course but also in Mathematics, Physics, and Foreign Language).

Assessment	Weight
MCQ	10%
PCA	10%
Mid-Term Exam	15%
Final Exam	65%

Table 9.1: Weights of the assessment in the final mark

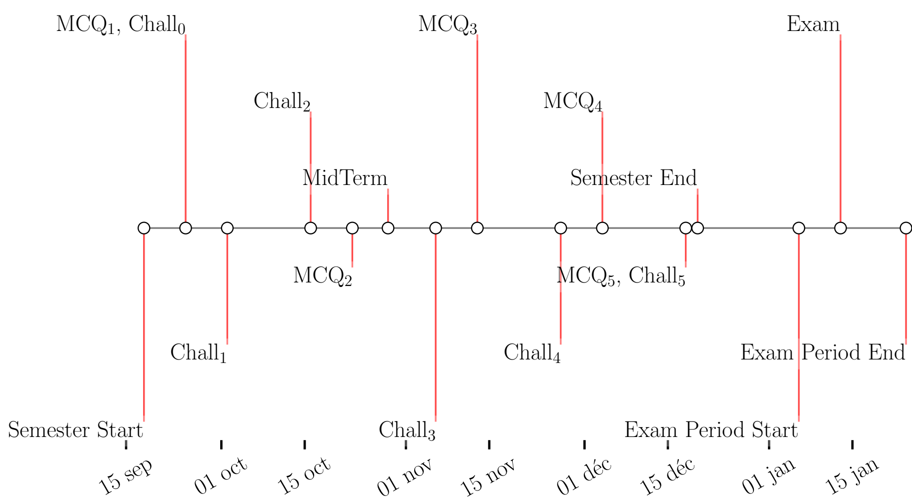


Figure 9.1: Example of a semester timeline (here, the academic year 2019–2020)

The core change took place in 2016 when we introduced a Programming Challenges Activity (PCA, see the next section), *i.e.*, a teaching activity spread over the whole semester and consisting in submitting small pieces of code – called Challenges – on a platform running our program CAFÉ (see Chap. 7). The introduction of the PCA marked the beginning of the course transformation into blended learning [28, 78], which is defined as

> "[T]he combination of traditional face-to-face and technology-mediated instruction." [78]

In 2018, the subject covered in the last Challenge was extended to the last chapter of the course (*i.e.*, dynamic memory allocation) because we observed the students had struggled to master it.

To sum up, Table 9.1 lists all the assessments and their respective weights in the final mark and Fig. 9.1 shows an example of semester timeline including all the assessments: the MCQ, the PCA and the two exams.

9.3 Programming Challenges Activity

The Programming Challenges Activity (PCA) consists of six assignments called **Challenges** distributed throughout the first semester, approximately every two weeks. Challenges account

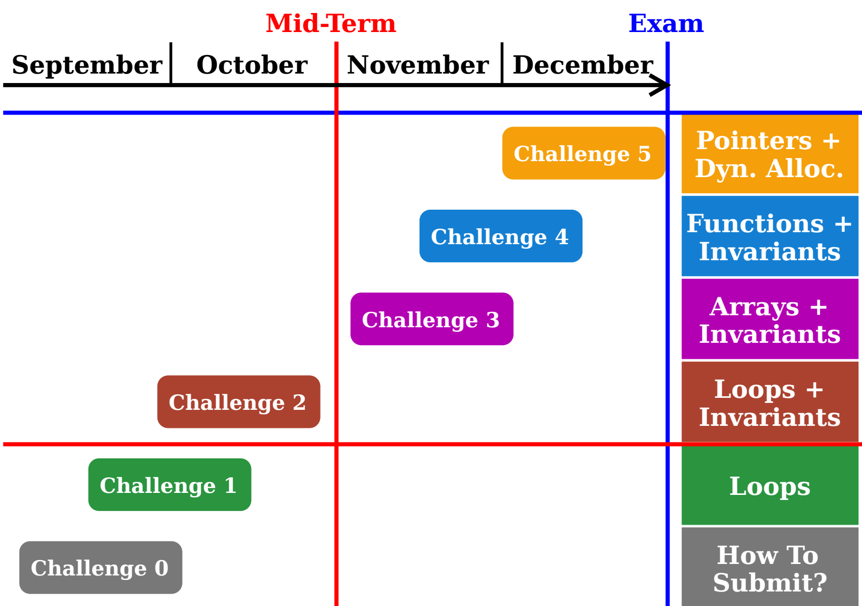


Figure 9.2: Challenges timeline. The column at the right shows the Challenges' subjects.

for 10% of the final grade, each Challenge having the same weight (*i.e.*, 2%). Each Challenge consists in a programming/algorithmic task.

Fig. 9.2 shows the Challenges timeline. The first Challenge (called "Challenge 0") allows students to feel comfortable with CAFÉ (in particular the *feedback*), but also on how to submit a challenge and, consequently, does not account in the final mark. The five subsequent Challenges do account in the final mark. They are of increasing complexity (from a simple loop to write – Challenge 1 – to a modular program solving a reasonably complex problem – Challenge 4). The last Challenge is dedicated to pointers and dynamic allocation, as we noticed those particular topics appear to be difficult for students. The particular shape of Fig. 9.2 also represents, on the right side, a knowledge gauge filling up during the semester: Challenge 2 relies on knowledge/understanding acquired with Challenge 1, Challenge 3 on Challenge 1 and 2, and so forth.

9.3.1 Challenges

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 (see Chap. 7). They can then consider these feedback

and feedforward to improve their answer and submit it again [62]. 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 [29]. Doing so prevents also the student from being bogged down.

Students can submit again up to two times, hence totalling three submissions. Karavirta et al. [98] have shown that students submitting a lot of time pieces of code for online assessment without necessarily getting good grades are inefficient in their work. As a conclusion, Karavirta et al. [98] suggested to limit the number of resubmissions in order to "guide [students'] learning process" (p. 238). This is exactly what has been done with CAFÉ. We decided to limit the number of submissions to three to avoid 'trial and error' process that is in contradiction with our programming methodology (see Chap. 2). Students have three days to complete a Challenge. At the end, only the last submission accounts for the mark.

9.3.2 Dealing with absences and Trump Cards

During the semester, students are allowed to skip one of the Challenges (except the Challenge 0 designed to get students familiar with the submission process on CAFÉ), what we call 'playing one's *Trump Card*'. This means that, when the Trump Card is played by a student, the Challenge does not account in the student's final mark. It is enough not to submit any response to a Challenge to play the Trump Card. This possibility was set up for two reasons: first, to avoid student excuses when not submitting their Challenge [32], second, to increase students' perception of controllability, inducing higher motivation [186].

9.4 Assessment for Learning

Since the introduction of the PCA, all teaching activities have complemented each other to align with the six principles of Assessment for Learning (AfL) presented by Sambell et al. [164].

1. The PCA and both mid-term and final exams consist of solving genuine programming tasks. These do not only assess the final product (*i.e.*, the program) but also the use of a proper programming methodology that led to the formation of the program. This refers to the AfL principle of "authentic assessment" [164, p. 6].

2. CAFÉ offers students the possibility to have up to two free submissions (they receive feedback about their performance without affecting their grades). Moreover, prior to MCQ, students have the opportunity to train themselves with mock MCQ, available on the course blackboard. This ensures to "balance summative and formative assessment" [*ibid.*].

3. Students may submit an answer up to three times during each Challenge [98] and receive feedback after each submission. Mock MCQ provide students with immediate feedback.

By doing so, we "create opportunities for practice and rehearsal" in "low-stakes teaching activities" [164, p. 6]. Even though both MCQ and PCA are graded, they do not account for a large amount in the final mark (10% for the MCQ and 10% for the PCA).

4. The feedback provided by CAFÉ after each submission, the correction of the mid-term exam, and the mandatory remedial courses provide students with "formal feedback to improve learning" [*ibid.*]. Formal feedback encompasses teacher's comments on student's work as well as and "self- and peer review and reflection" [*ibid.*].

5. The PCA introduction has enabled us to also change the way we organise the exercise and lab sessions. The students work in small groups in the classroom and can discuss with each other and with the pedagogical team gaining "opportunities for informal feedback" [*ibid.*].

6. As the course focuses on a programming methodology, it also covers the good practices of a programmer: how to test one's code, how to efficiently search for help in the proper resources (documentation, Internet, etc.) in order to turn students into "effective lifelong learners" [164, p. 7].

9.5 Conclusion and Research Questions Introduction

This chapter describes why we decided to implement a Programming Challenge Activity (PCA) and how we did so. That enabled us to align our CS1 course to the six principles of the Assessment for Learning. We hypothesise these six principles will contribute to increasing students' engagement. In the following of this document's third part, we addresses the following research questions:

RQ 3.1: *Does the* PCA *and its multiple features promotes student's engagement during the semester?*
This question is studied in Chap. 10.

RQ 3.2: *Does the course transformation through* AfL *lead to a stronger students' engagement and a higher success rate?*
This question is studied in Chap. 11.

Programming Challenges Activity Evaluation

THIS CHAPTER presents the students' reception of the Programming Challenges Activity (PCA) through the analysis of participation, performance and perception data collected from academic years 2016–2017 to 2019–2020. Data shows that students do take opportunity to submit several times and benefit from it. Observed submission behaviours enable us to conclude on the soundness of the PCA parameters (duration, number of resubmissions, difficulty). In addition, figures exhibit the same trends from year to years hence strengthening our conclusions. Sec. 10.1 introduces the methodology and the data we used to evaluate the PCA. Sec. 10.2 presents the results and Sec. 10.3 discusses them. Sec. 10.4 concludes the evaluation. Chap. 11 focuses on the integration of the PCA in our *CS1* course and the effect of the course transformation it enabled (see Chap. 9).

10.1 Methodology

This chapter reports on students' exposure to the PCA. The results are presented and discussed according to the 3P framework [185] which recommends consistent analysis of any pedagogical innovation by gathering and meshing three types of data that reflect aspects of the students' learning experience: Participation, Performance and Perception. Our investigation is guided by the question:

RQ 3. 1 *Does the* PCA *and its multiple features promotes student's engagement during the semester?*

These effects will be explored through three hypotheses:

Participation: all students seize the learning opportunity represented by the PCA

Performance: participation in the PCA leads to learning gains in programming
Perception: students report satisfaction regarding their experience with the PCA

10.1.1 Cohorts Presentation

Table 10.1 presents statistics about the students who took the course from academic years 2016–2017 to 2019–2020. The audience of our course rose from 54 students in 2016–2017 to 82 in 2019–2020. Most of them are freshpeople (*i.e.*, they were in their first year at the university). The table also provides the final exam participation and success rate figures. The success rates figures are consistent with the success rate for first year university students in Belgium [8].

10.1.2 Data Sources

10.1.2.1 Participation Data

The submission platform on which CAFÉ is run allows us to gather various pieces of data: submission time, number of submissions, and the student's mark given by CAFÉ. The various submission times for a particular Challenge and a particular student can be used, for instance, to determine the amount of time between two submissions.

10.1.2.2 Performance Data

As this chapter is focused on the PCA, we use, as performance data, the grades automatically computed by CAFÉ (see Sec. 7.4.1) after each submission during the PCA. Comparing the first

Table 10.1: Statistics about the students who took our CS1 course from Academic years 2016–2017 to 2019–2020. "N" is the total number of enrolled students. "Fresh people" refer to new students. "Repeat." refers to repeater. "Transfer" are students who changed their programs. "Part." refers to students who took the final exam, "Sign." are student who were present without taking the exam (in French, we call it a "signature"). "Abs." refers to the absentees. The raw final pass rate counts all the student while the adjusted one only counts the participants. "SR" and "FR" stands for Success and Failure Rates, respectively.

Academic Year	N	Origin						Final Exam Participation			Final Exam Pass Rate			
		Fresh People		Repeat.		Transfer					Raw		Adjusted	
		#	%	#	%	#	%	Part.	Sign.	Abs.	SR	FR	SR	FR
2016–2017	54	42	78	3	6	9	17	88.9	5.6	5.6	38.9	61.1	43.8	56.2
2017–2018	72	53	74	9	13	10	14	77.8	9.7	9.7	25.0	75.0	32.1	67.9
2018–2019	76	64	84	6	8	6	8	60.5	21.1	18.4	23.7	76.3	39.1	60.9
2019–2020	82	59	72	8	10	15	18	73.2	15.9	11.0	19.5	80.5	26.7	73.3

Table 10.2: Surveys Respondents. N is the total number of students. # is the number of respondents. two percentages are provided: %(Total) is computed on the total number of enrolled students and %(Part.) on the number of Participants in the Final Exam

		Respondents		
Year	N	#	% (Total)	% (Part.)
2017–2018	72	28	38.9	50.0
2018–2019	76	22	28.9	47.8
2019–2020	82	16	19.5	26.7

and last grade obtained by a student during the same Challenge enables us to observe their progression.

10.1.2.3 Perception Data

Between academic years 2017–2018 and 2019–2020, surveys were carried out after the final exam. The surveys were online and anonymous so that the students could express themselves freely. These are the same surveys as the ones already presented in Chapters 4 and 8 but we focus here on the PCA reception. Table 10.2 recalls the numbers of respondents each year and compares them to other course statistics[27].

[27] This Table is identical to Table 4.3 and Table 8.1

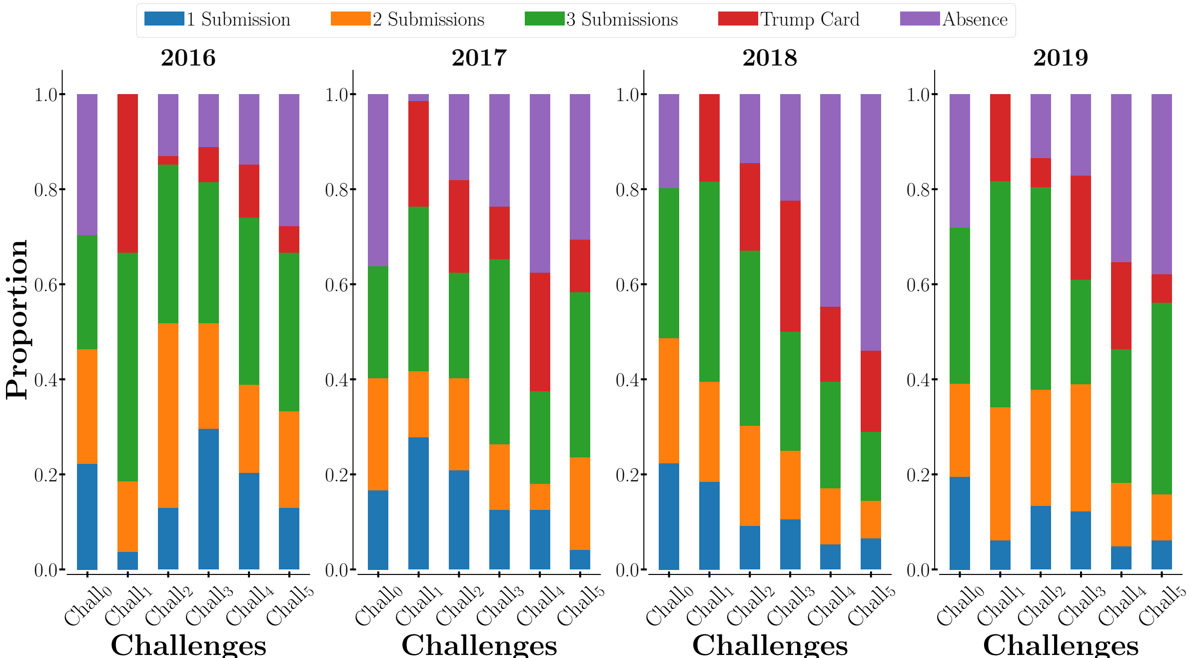


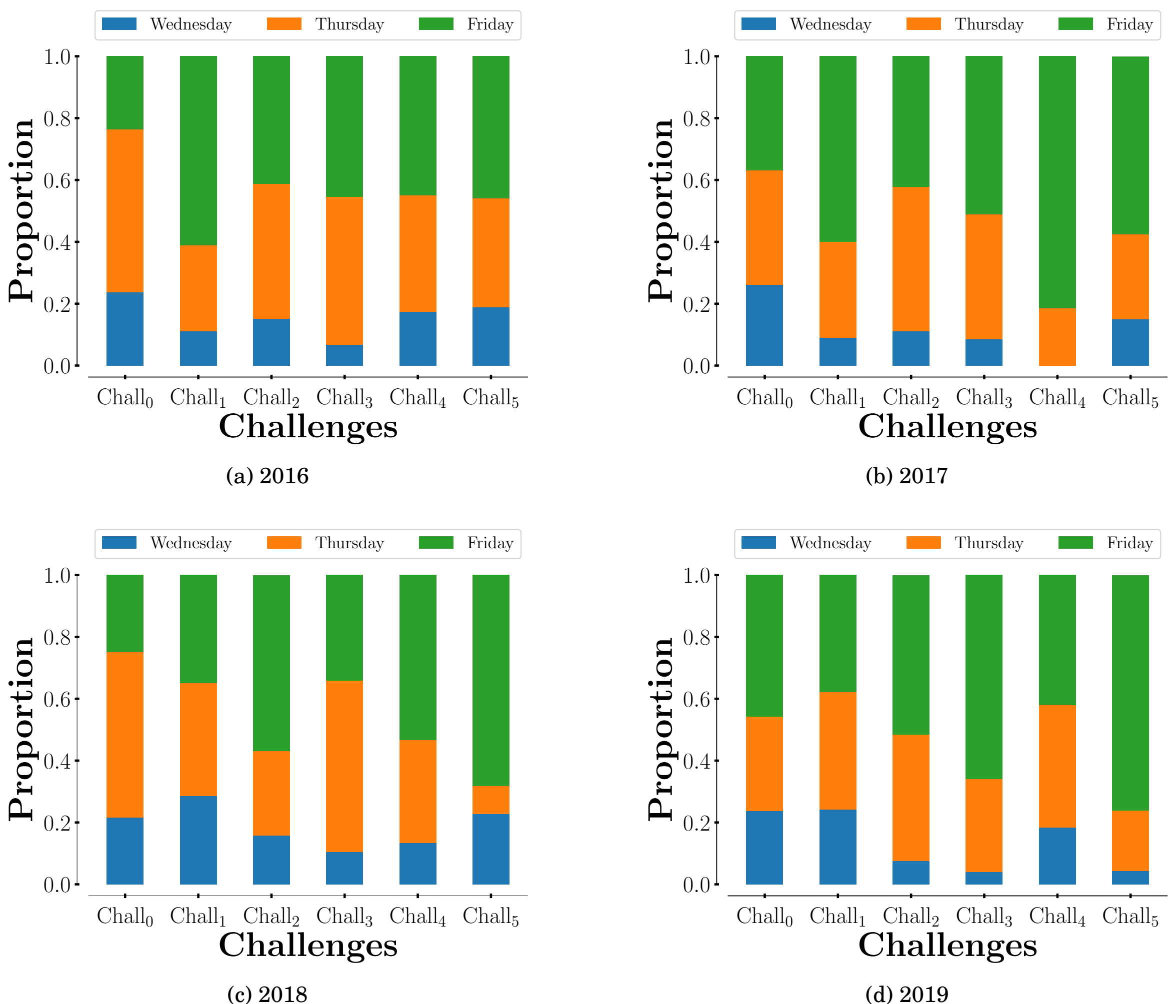
Figure 10.1: Distribution of students' participation in the PCA over the semester. "Trump Card" refers to students not submitting for the first time and "Absence" to students not submitting for at least the second time (or, in the case of Challenge 0, for the first time, since the Trump Card was not available for that Challenge). The absence for Challenge 1 in 2017 refers to a student who enrolled later in the semester.

10.2 Results

10.2.1 Participation: all students seize the learning opportunity represented by the PCA

10.2.1.1 Taking Challenges and Using *Trump Cards*

Fig. 10.1 shows the students' participation in the PCA over the semester. Each year, the participation decreased during the semester in absolute numbers from 85% (peak for Chall. #2) to 72% in 2016; from 74% to 60% in 2017; from 83% to 29% in 2018 and from 83% to 56% in 2019 (see blue, orange and green in Fig. 10.1). Logically, we can see a parallel rise in absences from 15% to 26% in 2016; from 3% to 28% in 2017; from 16% to 54% and from 13% to 38% in 2019 (see purple in Fig. 10.1). As far as the "Trump Cards" are concerned (see red in Fig. 10.1), they are played in majority for the Challenge 1 and one can also observe a peak for Challenge 3 (in 2016 and 2018 and 2019) and Challenge 4 (in 2017). Overall, 46% of the students did all of the Challenges in 2016; 12.5% in 2017; 7% in 2018 and 30.5% in 2019. The downtrend in participation did not affect triple submissions, which stood at 50% of participants of most of the graded Challenges (see green in Fig. 10.1).



(a) 2016

(b) 2017

(c) 2018

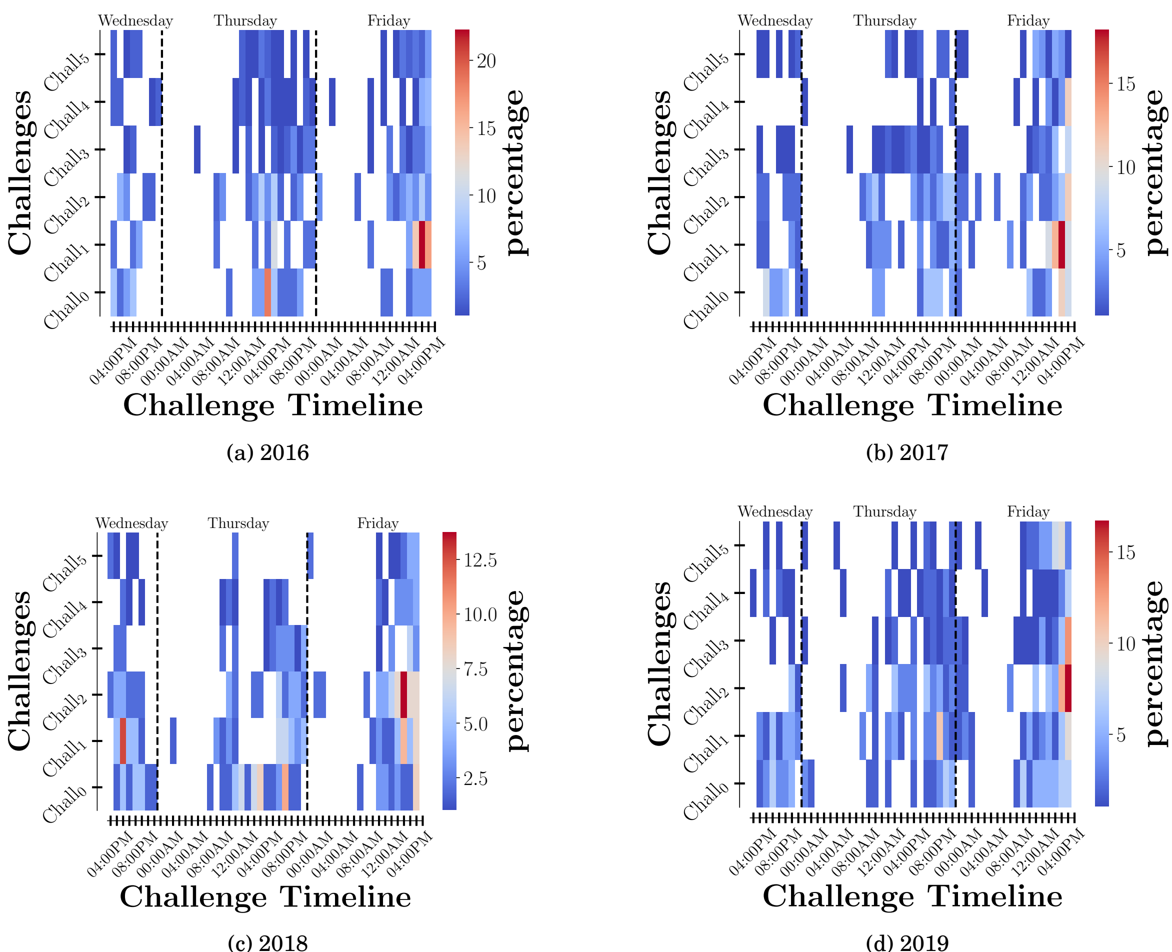
(d) 2019

Figure 10.2: Distribution of submissions per day during the PCA

10.2.1.2 Submission time

Time of submission during the Challenge Each Challenge begins on a Wednesday at 4:00 p.m. and ends on Friday at 6:00 p.m. Fig. 10.2 shows that Friday (in green) is the preferred day for the first submission for a majority of the graded Challenges, rising even up to 60% of the submissions for Challenges #1 in 2016; #1 and #4 in 2017; #2, #4, and #5 in 2018 and #3 and #5 in 2019. Thursday (in orange) is the second most popular day for the first submission, mainly for Challenges #1 (in 2018 and 2019), #2 (in 2016 and 2017) and #3 (in 2016, 2018 and 2019). For nearly all of the Challenges, fewer than 20% of the students made their first submission on Wednesday.

Fig. 10.3 deepens this analysis by using a heat map graph to provide an overview of the hourly distribution of submissions. On Wednesday, the first submissions occur mainly before 9:00 p.m only in 2018 (see Fig. 10.3c) while other years show night work (*i.e.*, after 10:00 p.m.).



(a) 2016

(b) 2017

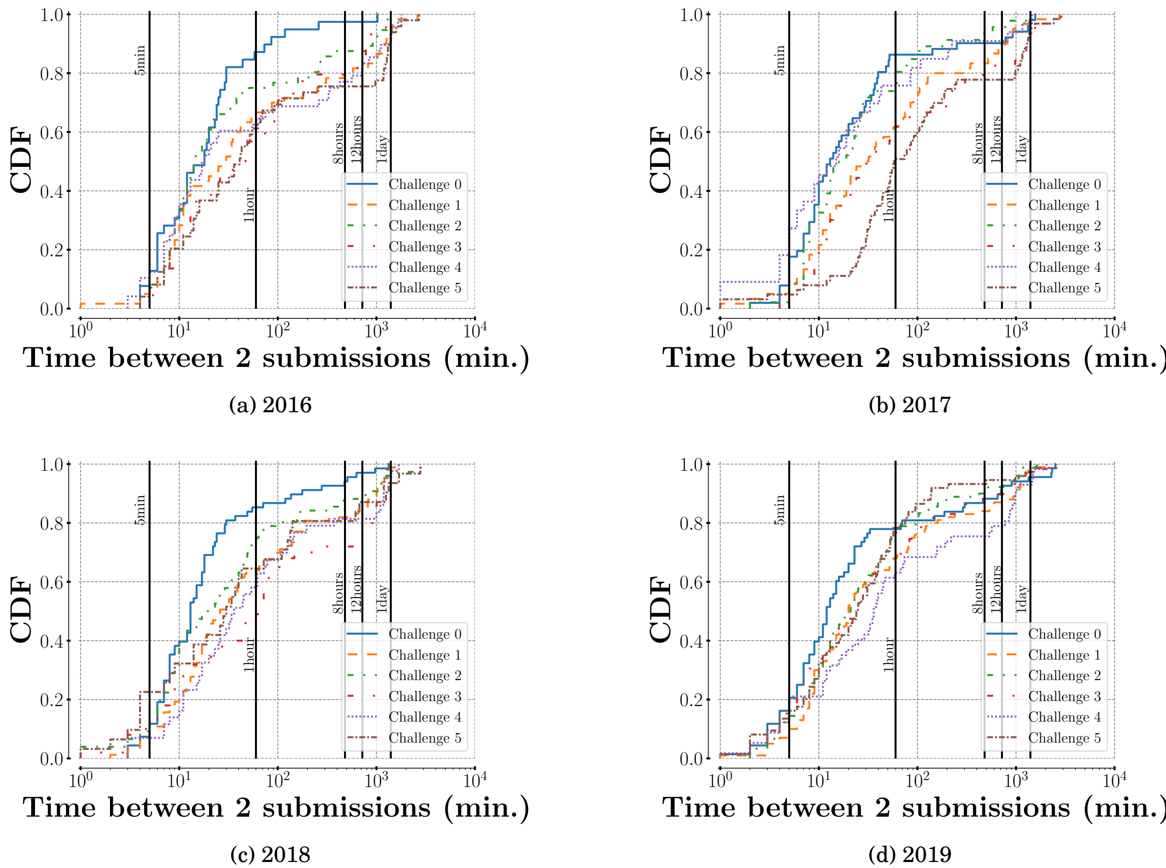
(c) 2018

(d) 2019

Figure 10.3: First submission time heat map. The color represents the percentage of submissions (N is the total number of students). Red indicates more submissions; blue indicates fewer. The reader should be warned that each heat map exhibits its own percentage range.

Thursdays show more night work, for nearly all Challenges and all years. Fridays show a large number of last-minute first submissions (*i.e.*, 2 or 3 hours before the deadline). Finally, there are also some students who submit on Thursday and Friday mornings, when other lectures are scheduled.

Time between submissions One of the main features of CAFÉ is that it provides the students with *feedback* and *feedforward*. To take advantage of this information, students should spend some time between submissions. Fig. 10.4 provides a degree of insight into the time intervals between consecutive submissions for all Challenges. All of the curves follow the same profile from year to year, except for Challenge 0. During Challenge 0, almost 80% of the resubmissions occurred within 1 hour. For the other Challenges, between 10% and 15% of consecutive submissions are less than 5 minutes apart. As well, about 60% of the resubmissions occurred



(a) 2016

(b) 2017

(c) 2018

(d) 2019

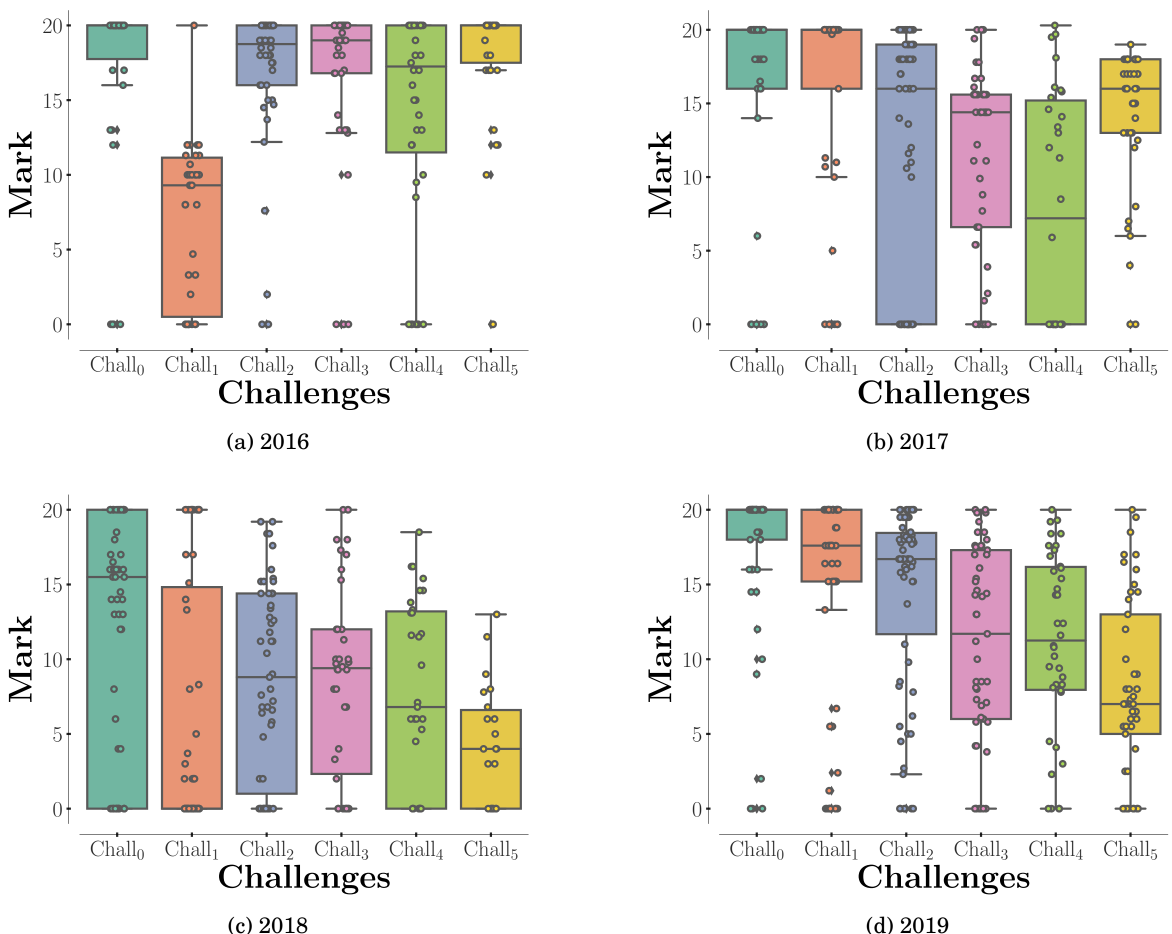
Figure 10.4: Time between submissions for a Challenge. This figure plots an empirical cumulative distribution function of the time difference in minutes between consecutive submissions. Note that a logarithmic scale is used for the x-axis: time intervals (5 min, 1 h, 8 h, 12 h, and 1 day) are represented by black vertical bars to make the figure easier to read

between 5 minutes and 1 hour. The plateaus between 4 hours and 12 hours indicate that very few students wait that length of time before submitting again. Finally, between 10% and 20% of the consecutive submissions are more than 12 hours apart. Challenge #5 in 2017 may seem to be an exception but it exhibits in fact the same trend shifted to the right (50% of the resubmissions within 1 hour, nearly 30% between 1 and 5 hours and a plateau between 5 and 16 hours).

10.2.2 Performance: participation in the PCA leads to learning gains in programming

10.2.2.1 Inter-Challenge Scores

Fig. 10.5 shows a box plot of the students' marks obtained for their first submission in each Challenge. Considering the four quartiles of the mark distribution, the box plots reveal that first-submission results tend to decrease as the semester goes along for years 2017 to 2019. One can also see that in 2016, the grades are not decreasing. In 2016 and 2017, Challenges #5 show



(a) 2016

(b) 2017

(c) 2018

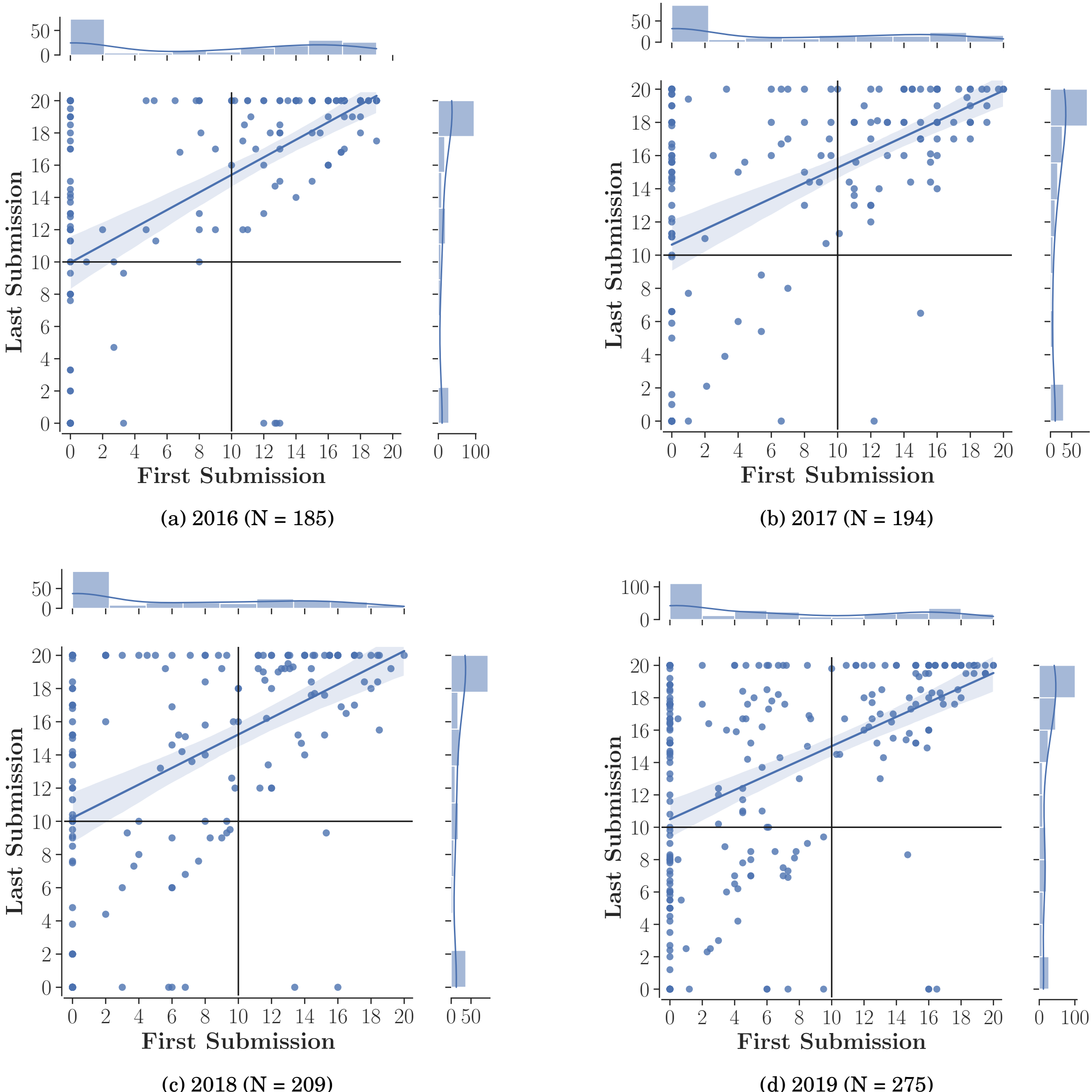
(d) 2019

Figure 10.5: Box plot of the students' results for their first submission for each Challenge. Each dot represents a result. The boxes and whiskers represent the four quartiles

more success than in the next years.

10.2.2.2 Intra-Challenge Scores

Since CAFÉ allows students to resubmit enhanced solutions to a Challenge, it is worth making sure that multiple submissions for a given Challenge do indeed enable students to improve their results. Fig. 10.6 shows, for each year, the relationship between the first and the last submission result for all the Challenges in the year. The X-axis is the result (over 20) of the first submission and the Y-Axis is the result (over 20) of the last submission (either the second or the third one). Over the grade scale (20), 10 is the minimum to succeed and values < 10 correspond to a failure. All the dots above the bisector of the first quadrant are related to an improvement between the first and the last submissions. One can see that is is the case for all the years. The year 2019 (see Fig. 10.6d) shows more points in the bottom left part of the graphs, referring to students who failed at a Challenge, even a majority of them still improved their results.



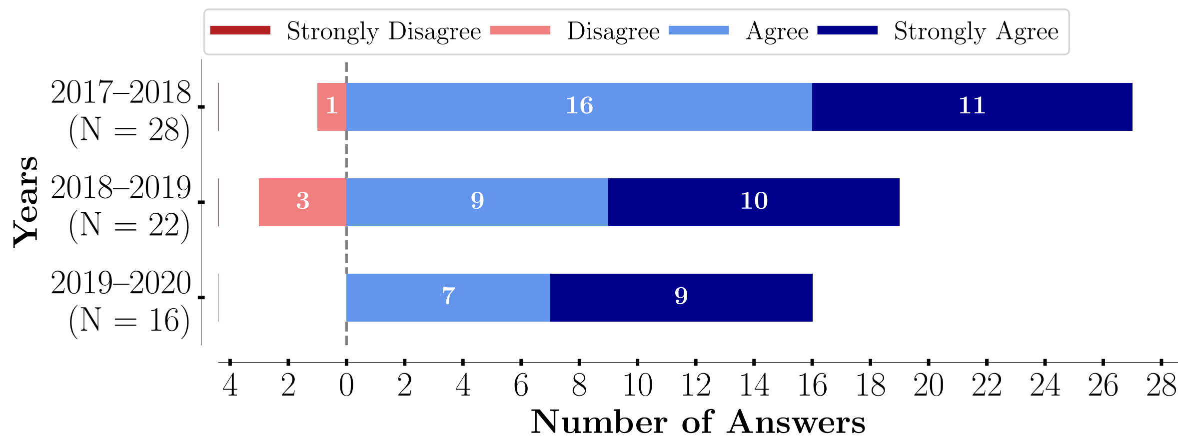
(a) 2016 (N = 185)

(b) 2017 (N = 194)

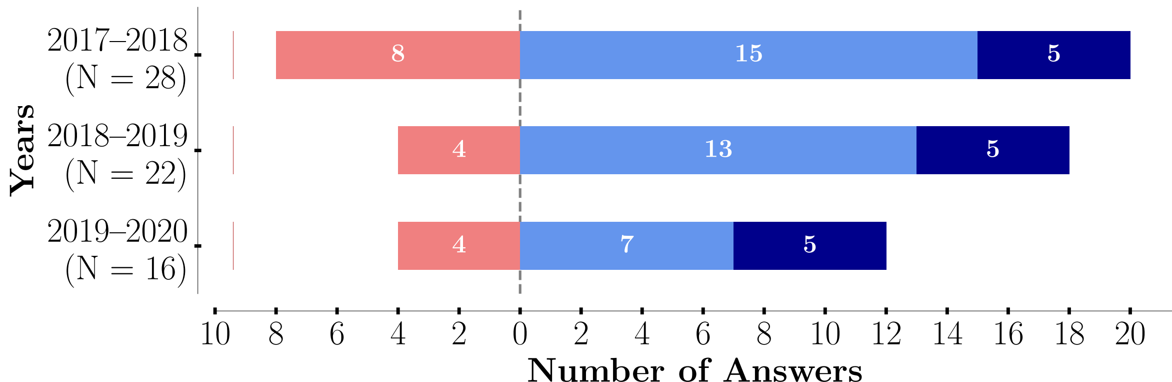
(c) 2018 (N = 209)

(d) 2019 (N = 275)

Figure 10.6: Improvement through multiple submissions. The X-axis is the result after the first submission and the Y-axis is the result of the last submission. The results (over 20) are both computed by CAFÉ. Each dot represent a Challenge and all the Challenge of a same semester are plotted on the same figure. The subplots above and at the right of the graphs represent the distribution of the Challenge grades for the first and last submissions, respectively.



(a) "Submitting five Challenges consisting in writing code to solve a problem was a good way to make me work regularly."



(b) "Submitting five Challenges consisting in writing code to solve a problem made me feel confident about my programming skills."

Figure 10.7: Responses to the surveys from 2017–2018 to 2019–2020. All the plots use a Likert [126] scale.

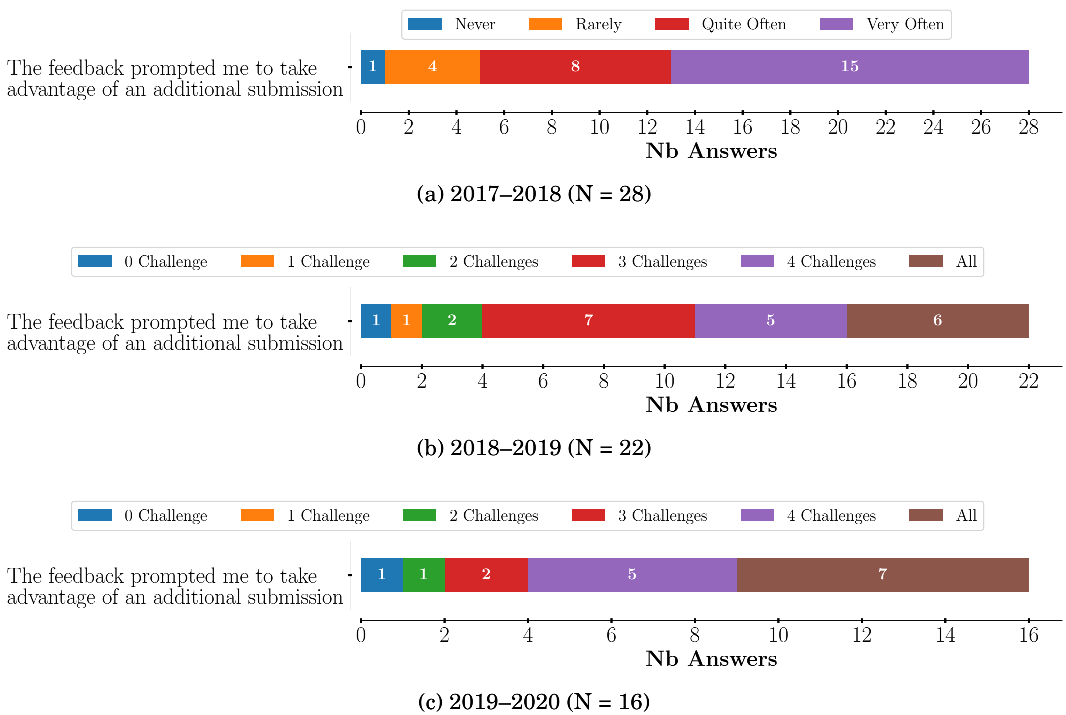
Additionally, one can observe each year a vertical "bar" at 0 on the X-Axis: this corresponds to student who failed at the first submission with the lowest grade but eventually finished with succeeding.

10.2.3 Perception: Students Report Satisfaction Regarding their Experience with the PCA

The survey received 28 responses in 2017–2018, 22 responses in 2018–2019 and 16 responses in 2019–2020 from students who took part in the Challenges (see Table 10.2).

10.2.3.1 Overall Benefits of the PCA

Respondents claim to benefit from the PCA, since a large majority of them (27/28 in 2017–2018, 19/22 in 2018–2019 and 16/16 in 2019–2020) agree with the statement "Submitting 5 Challenges



(a) 2017–2018 (N = 28)

(b) 2018–2019 (N = 22)

(c) 2019–2020 (N = 16)

Figure 10.8: Students' survey responses to the question: for how many Challenges the feedback message prompted you to take advantage of an additional submission?

Table 10.3: Reasons given in response to the question: "Why did you play your Trump Card when you did?" ($N_{2018-2019} = 22, N_{2019-2020} = 16$)

	# Chall. 2018–2019						# Chall. 2019–2020					
	1	2	3	4	5	Σ (%)	1	2	3	4	5	Σ (%)
Organisational problem		2	2	2	2	8 (36)			1	1	2	4 (25)
To save my grade (not lower it)					6	6 (27)						
"Laziness"		1			1	2 (9)						
"Too complicated" (perceived difficulty)				1		1 (4)			1	1	1	3 (19)
Never played it						5 (23)						9 (56)

consisting in writing code to solve a problem was a good way to make me work regularly" (see Fig. 10.7a). Most respondents (20/28 in 2017–2018, 18/22 in 2018–2019 and 12/16 in 2019–2020) also agreed with the statement "Submitting 5 Challenges consisting in writing code to solve a problem made me feel confident about my programming skills" (see Fig. 10.7b).

10.2.3.2 Reasons for Playing the Trump Card

The students were asked why they played their Trump Card when they did. As shown in Table 10.3, in 2018–2019, 23% (5/22) of the respondents never played their Trump Card. A deeper analysis of the students who never played their Trump Card reveals that that year, all of

them responded to the survey. As the year 2019–2020, we can see in Table 10.3 that a majority of respondents never played their Trump Card (56%, 9/16) while the proportion of non played Trump Card was of 30.5% (25/82) of the students.

In 2018–2019, the most common reason to play one's Trump Card (8/22, 36%) was an organizational problem (e.g., lack of time to complete the Challenge) and was quite well distributed throughout the semester (the Trump Cards played by the respondents that gave this reason are spread over all of the PCAs – see Table 10.3). The second most common reason was always mentioned in relation to Challenge 5: 27% of the students (6/22) were afraid that they would lower their grade if they took the Challenge or were already satisfied with their grade. They adopted a strategy that gave more weight to the score already "earned" than to the experience they could gain by completing Challenge 5. Apart from these main reasons, one student mentioned the difficulty of Challenge 4 and two of the respondents confessed to some "laziness."

In 2019–2020, respondents who played their Trump Card mention also organisational problems (25%, 4/16) and their perception of the difficulty of the Challenges (19%, 3/19) as a reason to use their Trump Card.

10.3 Discussion

10.3.1 Participation: all students seize the learning opportunity represented by the PCA

10.3.1.1 Taking Challenges and Using Trump Cards

For all of the Challenges, a majority of participants actually took advantage of resubmitting to improve their performance (see Fig. 10.1). This is exactly what the PCA was designed for, in line with the creation of "opportunities for practice and rehearsal" espoused by the AfL [164]. Similarly, participants took the opportunity to play their Trump Card regularly throughout the semester. In this respect, two very different behaviours could be distinguished: some students used it as it was intended and others simply dropped the course, in which case their first absence was labelled as a Trump Card. These latter students did not participate in subsequent Challenges. Each year, the number of drop-outs increased regularly over the semesters. A steep increase can be seen for Challenges 3 or 4 (depending on the year), which came right after the mid-term exam (see Fig. 10.1). This suggests that the mid-term exam results made some students decide to drop the course (and perhaps change their program)[28]. By the end of the semester, the absence rates for the Challenges #4 and #5 seem to be a good proxy to measure the dropouts: each year, those figures are aligned with the final exam participation rate (see Table 10.1 and Fig. 10.1).

[28] It is worth noting that a mid-term exam is organized for several courses: Maths, Physics, CS1, and English. This means that dropping out may not necessarily be related to results for our CS1 mid-term exam in particular.

10.3.1.2 Submission time

The Results section presents data regarding the submission times during the course of a Challenge and the time between consecutive submissions. The following discusses the PCA parameters related to this data, i.e., the length of the Challenge and the number of submissions allowed.

Time of submission during the Challenge Currently, each Challenge takes place over three days: from Wednesday at 4:00 p.m. to Friday at 6 p.m. At first glance this may seem short, but the problems to be solved are not that difficult. In fact, designing a Challenge consists in finding a good balance: the problem should be difficult enough to challenge the students without overwhelming them with work. An increase in length could be considered, but Challenges could not last for more than one week; otherwise, they might interfere with the students' workload for their other first year courses. It is also worth noting that each Challenge is aligned with other teaching activities in our CS1 course (theory lectures, practical sessions, multiple choices questions (MCQ)) that all address the same topic. To a certain extent, and following Nicol's second principle of good assessment and feedback practice consisting in encouraging students to spend "time and effort" on challenging learning tasks" on a regular basis [147, p. 32], the Challenge's rhythm helps to pace the students' study and work. Keeping this in mind, if a Challenge lasted much longer, there would be a risk of some students being left behind through procrastination and putting off the moment they decide to focus on a Challenge. In addition, one could think about just adding the weekend to the Challenge (i.e., from Wednesday, 4:00 p.m. to Sunday, 6:00 p.m.), but the various data discussed in this paper show that this would not be effective. Indeed, we have shown that a majority of students are more likely to make their first submission on Friday (currently the last day of the Challenge) than on any other day (see Fig. 10.2). It is worth noting that the lectures are held on Wednesdays. If the students are tired, this could explain why they do not begin to submit on that day. If the Challenge included the weekend, we could assume that a certain number of first submissions would be delayed until the Saturday or even Sunday, since the students tend to submit close to the deadline (see Fig. 10.3). While an automatic correction system could clearly handle such a schedule, the teaching team would not be available in case of problems (e.g., technical issues with the submission platform). Finally, the students' quality of life would be affected if they chose poorly and put off doing their work until the Sunday. Taking all this into consideration, extending the length of the Challenges does not seem advisable.

Time between submissions The number of available submissions was meant to be a scarce resource so that students would think twice before submitting and take time to reflect on the feedback and feedforward they received. Currently, we allow up to three submissions per Challenge, following the recommendations of Karavirta et al. [98], who clustered several behaviours in a group of students using an automatic assessment tool with an unlimited number

of submissions. Among these, they identified those they call “the iterators” who submit a high number of times without necessarily getting good grades, which indicates that they are not working effectively. Karavirta et al. thus recommend limiting the number of resubmissions to “guide [them] in their learning process.” We could increase the number of resubmissions to 4 or even 5. However, increasing the number of submissions too much would decrease the risk per submission. That would allow students to perform a kind of “test-driven development” (i.e., submitting quickly while the tests generate errors), which is contrary to the programming methodology taught in the course. This behaviour can, in fact, be observed in Challenge 5 (see Fig. 10.4): more than 20% of students resubmitted within 5 minutes, even if, at first glance, this is not the best way to benefit from the feedback and feedforward provided. Moreover, the CAFÉ system was not designed to be used like this, since it also emphasizes the programming methodology being learned by the students. On the other hand, if the number of submissions is increased and the students do take time and make use of the opportunity to close the gap between their last submission and a better outcome, thus completing the feedback loop [163], it means that more time should be given for the Challenges. The previous section explains why this is not desirable. Finally, if the students still need more time and more feedback for a particular Challenge, it would indicate that the Challenge was too difficult and should be redesigned.

10.3.2 Performance: participation in the PCA leads to learning gains in programming

10.3.2.1 Inter-Challenge Scores

The decreases observed throughout the semester (see Fig. 10.5) could be explained by the increase in the tasks covered by the Challenges, which are cumulative: students might experience a snowball effect if they progressively accumulated learning gaps. But it should be kept in mind that Fig. 10.5 shows students’ marks for the first submission, and CAFÉ allows several submissions. For Challenges 0 and 1, nearly 20% of students made a single submission: students getting a good grade on the first attempt do not need to resubmit. For Challenges 2 to 5, the lower results make resubmission essential in order to make progress and accumulate knowledge. This is not surprising. In fact, it is exactly why multiple submissions were allowed in the first place, followed by feedback that the students could use to progressively improve their performance. In 2016 and 2017, the Challenge #5 did not address the dynamic memory allocation. This can explain the increase in the results.

On the other hand, the results in 2016 may be explained by two factors. First, it was the first year the PCA and CAFÉ were proposed to students and some bugs and problems arose during the first Challenge explaining the lower grades at the first submissions. Second, the plagiarism detection was not yet implemented and systematically used (see Sec. 7.6). This was the case for the next years. However, the plagiarism detection does not allow to spot all the cheaters and the fraud prevention must be repeated each year. This is probably why the first Challenge results are higher than the rest, *e.g.*, in 2017. Each year, we unfortunately caught a few students in the act.

10.3.2.2 Intra-Challenge Scores

Data show that the multiple submissions opportunities offered by the PCA enable students to improve their performance (see Fig. 10.6). Yet, data also show students do not necessary use their three submissions (see Fig. 10.1). We conclude that they may be satisfied with their score. On the other hand, students who were not satisfied could submit again, since the PCA allows this, establishing itself as a tool likely to have a positive impact on the goals students set for themselves and their ability to feel highly committed, reducing so one of the seven cause of student withdrawal pointed by Tinto [182].

As the vertical bar at 0 on the X-axis is concerned, this can be explained by compiling or formatting issues (see Chap. 8) at the first try. Such errors are graded with a 0/20 and PCA's multiple submissions allow students to correct their file. Of course, if a student encounters a problem with CAFÉ for which they is not responsible, the teaching team ensures that they does not lose a submission. This kind of situation is less likely to happen since we learned from four years of PCA and adapted CAFÉ accordingly.

10.3.3 Perception: Students Report Satisfaction Regarding their Experience with the PCA

10.3.3.1 Limitations Due to the Number of Answers

Every year, while the survey link was sent to all enrolled students, we had no means of collecting opinions from those who chose not to respond (whether they left the program or not). These opinions would have been valuable in our analysis.

In 2017–2018 and 2018–2019, we collected 28 answers that is roughly half of the students who continued to follow the CS curriculum during the second semester The second semester of academic year 2019–2020 was marked by the COVID-19 pandemic [190]. This may explain the low number of respondents that year.

In 2018–2019, all the students who never played their Trump Card answered the survey and in 2019–2020 a majority of respondents never played it (see Table 10.3). This suggest we collected the opinion of rather highly committed students.

However, we do not observe discrepancies in the responses collected in the three years we conducted the surveys (see Fig. 10.7 and Fig. 10.8).

10.3.3.2 Overall Benefits of the PCA

The students' perceptions tend to confirm that the PCA achieved one of its primary goals, i.e., to make them work on a regular basis. Students also state that they gained confidence in their programming skills (see Fig. 10.7). This result is consistent with the purpose of "creating opportunities for practice and rehearsal" from the AfL [164]. Moreover, they acknowledged that the feedback itself encouraged them to take advantage of an additional submission (see Fig. 10.8, first lines). As for the students' reactions to the feedback and feedforward (see Fig. 10.8, lines 4

to 7), it is not surprising that a majority of the respondents reread the theory course for more than three Challenges because CAFÉ directs them specifically to the course (e.g., gives the exact location of the relevant subsection). The other actions are less often explicitly suggested. Contact with the teaching team is the last resort if the student has a question about the feedback or a problem with the CAFÉ system itself. However, CAFÉ has been designed to limit that kind of interaction. With regard to the feedback portion of the information transmitted by CAFÉ about the students' performance (see Fig. 10.8, lines 2 and 3), the data tends to show that it is useful for the students to know where they went wrong, if we follow the classification by Keuning et al. [101]. This result is also aligned with the AfL principle of providing students with "formal feedback to improve learning" [164]).

10.3.3.3 Reasons for Playing the Trump Card

Introducing the Trump Card mechanism is a double-edged sword. On one hand, it was thought that the Trump Card would make students take responsibility for their learning, avoid making excuses for not submitting, and increase their perception of control over the course (and thus their engagement and motivation). Data shows that some students indeed used their Trump Card to cope with organizational issues (see Table 10.3).

On the other hand, it allows students to develop short-term strategies to maximise their PCA grade (e.g., by avoiding a bad mark on a Challenge perceived as too difficult, see Table 10.3) that can lead them to avoid practising the task featured in the Challenge that they discarded using the Trump Card. This is illustrated by Challenge 5 in 2018–2019. According to the survey, at 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?", some students (4/22) "regretted using [their] Trump Card" and recognized that "taking the Challenge would have helped them for the final exam". If we deepen this analysis by looking at the final exam results for the questions addressing the same subject as Challenge 5 (pointers and dynamic memory allocation), we observe that every student who succeeded in Challenge 5 also got these questions right. It should be noted that they first improved their score in Challenge 5 through multiple submissions. However, those who did not submit code for Challenge 5 failed at the same kind of questions on the final exam.

We still believe that the Trump Card system must be maintained. However, this means, first, that the students must be made more aware of the potential consequences of their choices (i.e., applying a poor/short-term strategy) and, second, that there must be debunking of any rumours about the difficulty of a given Challenge that would discourage students from even trying it. This has been done in 2019–2020 and the use of the Trump Card decreased, especially for the fifth Challenge (see Fig. 10.1), thus validating this approach.

10.4 Conclusion

This chapter discussed students' acceptance of computer Programming Challenge Activities (PCA) in our CS1 Course and presented data about the students' participation, performance, and perception. The PCA enables the students to work on a regular basis. Five times during the semester, they submit programming exercises (called Challenges) on a web platform and automatically get feedback and feedforward, which they can take into account to improve their solution. They say that doing this gives them more confidence in their programming skills.

These promising results encourage the continued use of the PCA in the future. Also, providing every student with individual feedback up to three times for each Challenge would not be feasible without an automatic system like CAFÉ.

Furthermore, the data allows us to validate the PCA parameters, such as the schedule, number of submissions, Trump Card and so on. For instance, the Trump Card system can be maintained if the students are made more aware of the consequences of using it; the length of the Challenge (three days) can be retained since a majority of students start to work on the last day of a Challenge.

The analysis of participation data revealed a trend in dropping the course, with a peak just after the mid-term exam. This phenomenon is not a particularity of the PCA, as it will be shown in next chapter, in which we evaluate the impact of our CS1 course transformation on student's engagement and success rate.

Programming Challenges Activity and Assessment for Learning

THIS CHAPTERaims at evaluating the course evolution described in Chap. 9 that enabled us to align the course to Assessment for Learning (AfL) principles. We hypothesised that such modifications would lead to a increase in course engagement that eventually could lead to better course performance. hence we formulate the following research question::

RQ 3. 2 *Does the course transformation through* AfL *lead to a stronger students' engagement and a higher success rate?*

The chapter is organised as the following: Sec. 11.1 introduces the methodology, Sec. 11.2 presents the results and Sec. 11.3 discusses them. Finally, Sec. 11.4 draws a conclusion.

11.1 Methodology

In order to measure the impact of the course transformation, we analyse data collected before and after the introduction of the Programming Challenges Activity (PCA), namely from academic year 2013–2014 to 2019–2020. The students cohorts that are considered are presented in Table 11.1 (that contains, for the academic years 2016–2017 to 2019–2020 the same information as Table 10.1). The total number of students rose from 2013 to 2019 (a growth rate of 41%). All these years, more than 70% of students came for the first time to the university (labelled as fresh people), almost 20% of the students reoriented from another curriculum (labelled as transfer students) and the rest were repeaters.

In contrast to the other chapters in which we divided our data in three categories (Participation, Performance and Perception [185]), we do not include in this study Perception data as we did not collect any before the introduction of the PCA.

Table 11.1: Populations of the *CS1* Course from Academic Years 2013–2014 to 2019–2020.

Academic Year	Total	Fresh People		Repeaters		Transfer Students	
		#	%	#	%	#	%
2013–2014	58	43	74	4	7	11	19
2014–2015	58	42	72	5	9	11	19
2015–2016	52	41	79	1	2	10	19
2016–2017	54	42	78	3	6	9	17
2017–2018	72	53	74	9	13	10	14
2018–2019	76	64	84	6	8	6	8
2019–2020	82	59	72	8	10	15	18

11.1.1 Data Sources

Numerous data sources are considered, *i.e.*, data directly retrieved from Correction Automatique et Feedback des Étudiants (CAFÉ), the course Blackboard page, and the results of the assessments. Except for teaching activities results, the analysis is focused on data about students' engagement. First, the platform on which CAFÉ is run enables us to retrieve data about students' submissions, namely the number of submissions per Challenge and the students' results for each Challenge. Second, the Challenge subjects can be downloaded from the Blackboard platform of the course. The Blackboard usage statistics enable us to collect data about mock multiple choices questions (MCQ). Finally, data about participation in all the assessments are used: there are five MCQ, five Challenges, the mid-term exam, and the final exam.

11.2 Results

11.2.1 Participation

Fig. 11.1 presents the MCQ participation rate from 2013 to 2019. Since the MCQ take place at the beginning of the session, these figures are a proxy of the course attendance. Taking each year individually, the participation rates exhibit the same trends from year to year: the first MCQ have a participation rate higher than 80% and it decreases a bit for the third MCQ. The participation rate drops for MCQ4 and the decrease continues for MCQ5.

In Fig. 11.1, the participation rates are shown as decreasing over the years (except in 2016–2017).

Fig. 11.1 shows also the participation to mock MCQ (at the right of the Figure) in 2018 and 2019. In 2018, the training rate is 10% below the participation rate for the MCQ 1 and 2. It drops to 20% below the participation rate for the next MCQ. In 2019–2020, the participation

[29] This is the same figure as Fig. 10.1

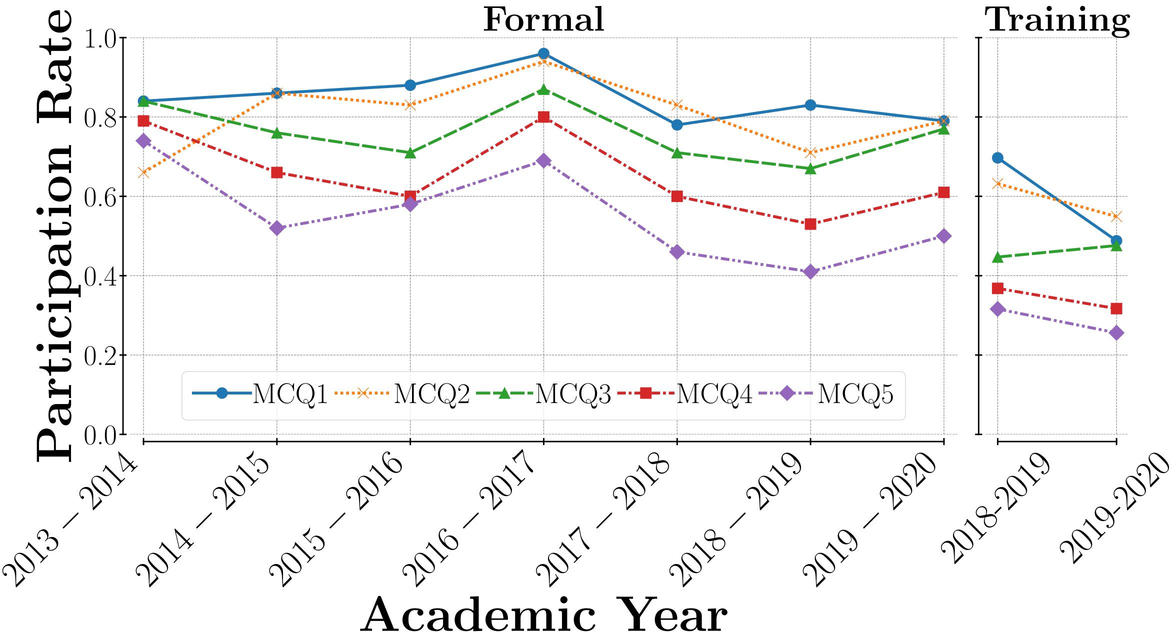


Figure 11.1: Participation rate in the Multiple Choice Questions (MCQ) Tests from academic year 2013–2014 to 2019–2020. The right part of the Figure also show the participation rate in the mock MCQ collected in 2018 and 2019.

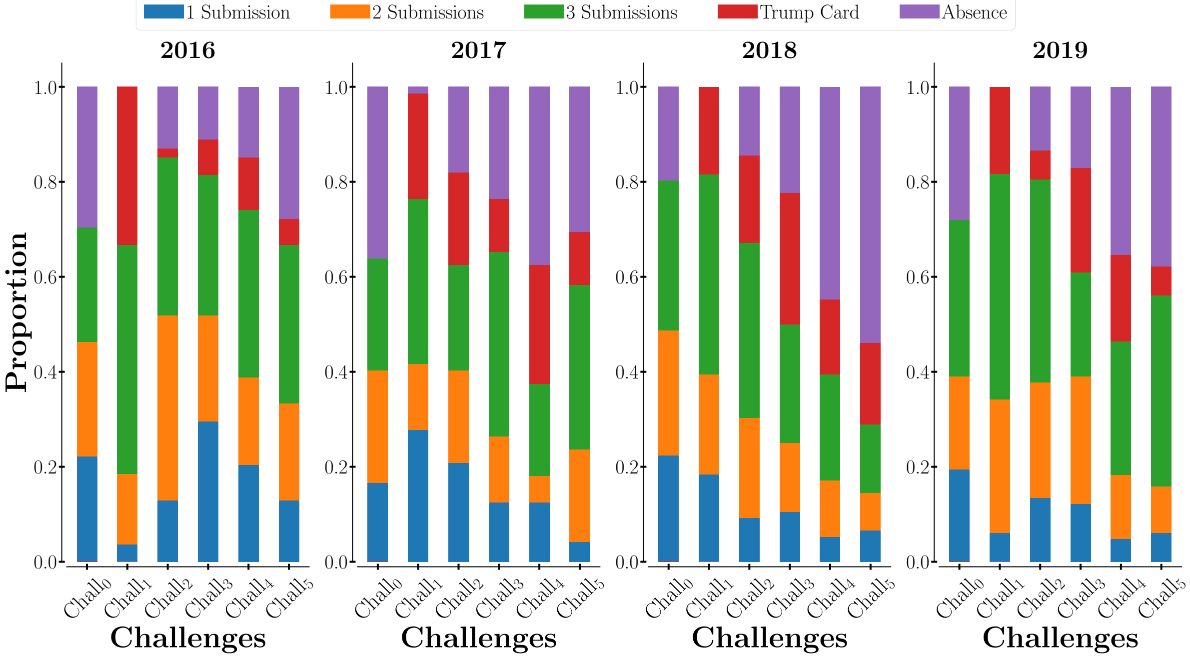


Figure 11.2: Participation to the PCA, on a Per Challenge Basis, from Academic year 2016–2017 to 2019–2020. In 2017, the absence for the Challenge 1 refers to a student that joined the curriculum later in the semester.

Table 11.2: Participation in the Mid-Term and Final Exam from Academic year 2013–2014 to 2019–2020. "Part." is the proportion of students taking the exam; "Sign." is the proportion of students that are present without taking the exam (they sign the presence sheet); "Abs." is the proportion of absentees and "Exc" stands for "Excused" (*i.e.*, students who cannot be present for a good reason, generally a medical one).

Academic Year	Mid-Term Exam				Final Exam			
	Part.	Sign.	Abs.	Exc.	Part.	Sign.	Abs.	Exc.
2013–2014	91%	0%	9%	0%	87%	4%	9%	0%
2014–2015	88%	0%	10%	2%	83%	5%	12%	0%
2015–2016	77%	0%	21%	2%	77%	6%	15%	2%
2016–2017	89%	2%	9%	0%	89%	6%	6%	0%
2017–2018	86%	0%	13%	1%	78%	10%	10%	3%
2018–2019	80%	0%	15%	5%	61%	21%	18%	0%
2019–2020	86%	0%	10%	4%	73%	16%	11%	0%

rate rose while the training rate decreased, leading to a difference of 20% or more between the training and the corresponding MCQ.

With respect to Challenges participation, Fig. 11.2 [29] shows the proportion of the number of submissions, as well as the proportion of students who did not submit the Challenge either because they were absent or they played their *Trump Card*. Students who take the Challenges mainly submit three times. On the other hand, the number of absentees grows as the semester goes and there is often a big rise between the Challenge 3 and 4, *i.e.*, when the results of the mid-term exams are published to the students.

Participation figures in the mid-term and final exam are presented in Table 11.2 This table shows that the participation in the midterm exam is fairly constant and always above 80% except in 2015–2016 (77%). There are few students that sign the presence sheet without taking the exam at mid-term and the proportions of absences vary between 9% and 15%, except in 2015-2016 (21%). On the other hand, the final exam participation is decreasing over time (except in 2016-2017 that is an outlier, as for Challenge and MCQ participation). The mean participation rate is 82.3% between 2013–2014 and 2015–2016 and 75% between 2016–2017 and 2019–2020.

11.2.2 Performance

Table 11.3 shows the link between participation to Challenges and the students' performance from Academic Year 2013–2014 to 2019–2020. For each year the PCA has been organised, the table shows the cumulative number of students having submitted Challenges and the mean number of completed Challenges per student (*i.e.*, "Challenges Participation" columns). The table also shows the cumulative number of Challenge submissions over the whole PCA, the cumulative average number of submissions per student over the whole PCA, and the mean number of submissions per individual participation (*i.e.*, "PCA Participation" columns). Table 11.3 also provides data

Table 11.3: Link Between Students Participation and Performance from Academic Years 2013–2014 to 2019–2020. “#” refers to cumulative values. “Per Stu.” refers to the average per student. “Per Part.” is the mean number of submissions per Challenge participation. SR and FR stand respectively for Success Rate and Failure Rate. Total figures include students that did not take the exam while “Adjusted” figures only include the students that took the Final Exam (in January). There is no data from 2013 to 2015 regarding the Challenges and PCA since CAFÉ was not yet implemented.

Academic Year	N	Chall. Part.		PCA Part.			Course Performance			
							Total		Adjusted	
		#	Per Stu.	#	Per Stu.	Per Part.	SR	FR	SR	FR
2013–2014	58	-	-	-	-	-	25.9	74.1	33.3	66.7
2014–2015	58	-	-	-	-	-	12.1	87.9	14.6	85.4
2015–2016	52	-	-	-	-	-	17.3	82.7	22.5	77.5
2016–2017	54	240	4.44	535	9.91	2.23	38.9	61.1	43.8	56.2
2017–2018	72	214	2.97	478	6.64	2.23	25.0	75.0	32.1	67.9
2018–2019	76	202	2.66	473	6.22	2.34	23.7	76.3	39.1	60.9
2019–2020	82	267	3.21	647	7.80	2.42	19.5	80.5	26.7	73.3

Table 11.4: Description of the categories used in Fig. 11.3 in term of performance markers of our error taxonomy introduced in Chap. 4

Category	Description	Markers
Correct	The code has the intended behaviour	CODE_1
Incorrect	The student committed a minor mistake	Any markers except CODE_1, CODE_3, CODE_4 or CODE_7
Very Incorrect	The code presents syntax errors or does not make any sense or contains a major mistake (either an infinite loop or a Buffer Overflow)	At least one among CODE_3, CODE_4 or CODE_7

about students’ performance: the success and failure rate (resp. SR and FR) in January (students can retake the exam in June and August). Focusing on the students that take the exam enabled us to evaluate more precisely the effect of the PCA introduction on the success rate. The success rate dropped between 2013 to 2016. Starting from 2016–2017, the success rate increases. From 2015–2016 to 2016–2017, it almost doubled although the year 2016–2017 seems to be an outlier in term of students’ participation (4.44/5 challenges per student) and success (43.8% actually taking the exam).

Another way to assess the effect of the PCA is to look at the quality of code written by the students at the Final Exam, in January. This is shown in Fig. 11.3. The figure shows the proportion of codes that are Correct (*i.e.*, the code has the intended behaviour), Incorrect (*i.e.*, the

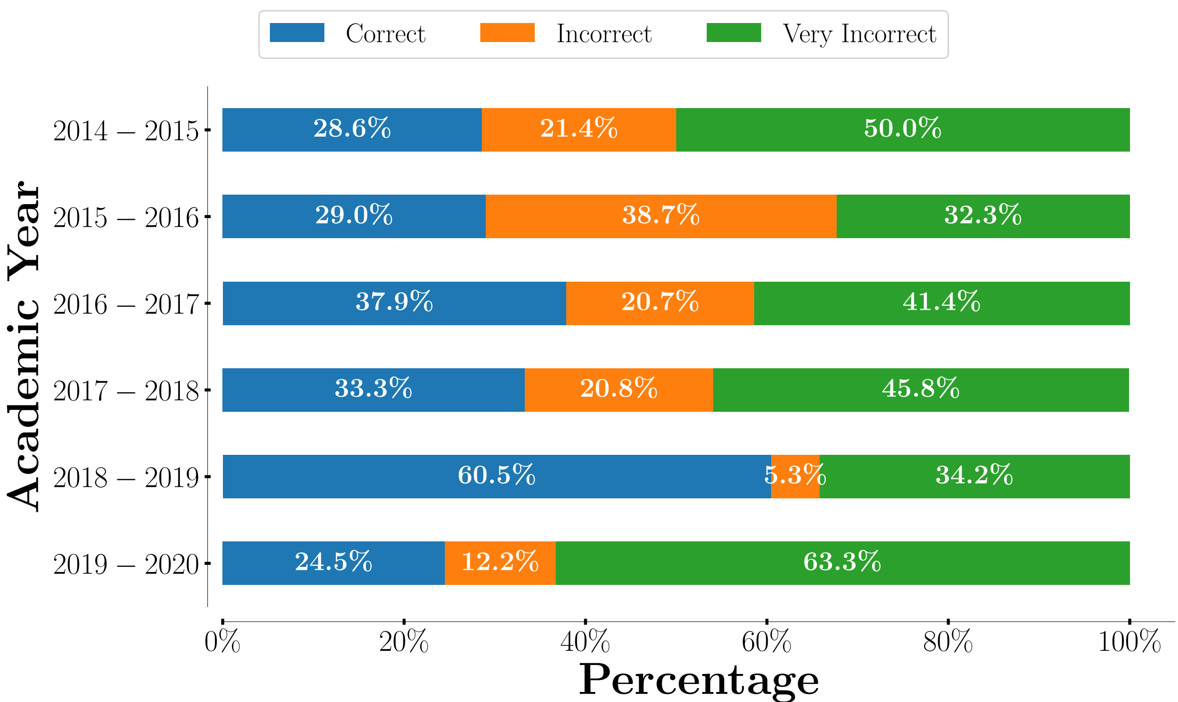


Figure 11.3: Final Exam Code Quality from Academic Years 2014–2015 to 2019–2020. “Correct” refers to codes that are (nearly) correct. “Incorrect” refers to codes that contain minor mistakes and “Very Incorrect” refers to codes either that do not make any sense, presents syntax errors, infinite loops or out-of-bound memory accesses.

student committed a minor mistake), or Very incorrect (*i.e.*, the code presents syntax errors or does not make any sense or contains a major mistake). Table 11.4 links those three categories with code performance markers introduced in Chap. 4. The proportion of code labelled as Correct increases from 28.6% (2014–2015) and 29.0% (2015–2016) to 37.9% (2016–2017) and 33.3% (2017–2018). However, 2018–2019 and 2019–2020 seem to be outliers.

11.3 Discussion

11.3.1 Participation

In Fig. 11.1, the decreasing participation in the third MCQ that take place shortly after the mid-term exam (see Fig. 9.1) but before the results are made available to the students suggests that the mid-term exam has an effect on students participation. Once the results are published, one can see that the participation rate decrease continues for the subsequent MCQ. That rising number of absences through the semester is indicative of abandonment since the MCQ take place at the beginning of exercises sessions and are a proxy of the course attendance.

The introduction of the PCA (in 2016–2017) does not seem to have modified that trend. On the contrary, the participation rates in the MCQ are lower since then suggesting that, while the

student numbers increased during that period (as can be seen in Table 11.1), the absenteeism rate increased as well. This may be partially explained by the open access policy to higher education in Belgium: students may enter the CS curriculum lacking some fundamental skills (*e.g.*, in maths) and are discouraged (*e.g.*, by maths and physics courses that are also assessed during mid-term exams).

As far as the PCA itself is concerned, Fig. 11.2 shows that students do take the opportunity to submit multiple times and may benefit from the *feedback*. This is exactly what CAFÉ was made for, with AfL in mind, *i.e.*, to offer “opportunities for practice and rehearsal” (AfL, 3rd principle).

However, they do not take all the opportunities to train since a majority of students play their Trump Card during the PCA. In the same way, the mock MCQ participation rates are below 50% (see Fig. 11.1). On one hand, taking all the opportunities to train could lead to better results. On the other hand, the introduction of non-mandatory activities was thought to increase students' perception of controllability that induces higher motivation [186]. We then can expect that a student will not take all the offered exercises. In order to help them to make informed choices, we deliver regularly during the semester feedback and *feedforward* about their performance (AfL, 4th and 5th principles).

Regarding the participation rates in the exams (see Table 11.2), the introduction of the PCA does not seem to have affected the mid-term one. On the other hand, the final exam participation is decreasing over time (except in 2016-2017 that is an outlier, as for Challenge and MCQ participation). The introduction of the PCA does not seem to have reduced that trend. On the contrary, the mean participation rate is 82.3% before the PCA and 75% after its introduction. Note that this drop in the participation to the final exam is actually counter-balanced with an increase in the success rate (see next section)

11.3.2 Performance

Table 11.3 shows that the academic year 2016–2017 (*i.e.*, when the PCA was introduced) is a the tipping point when the success rate started to increase above 30%. The year 2016–2017 was previously identified as an outlier in term of participation but the high participation is also associated with an higher success rate (43.8%). The shift of the success rate may be explained both by the fact that students can work on a regular basis thanks to the Challenge and because the PCA grades take into account in their final mark, allowing them to increase it in a low-stakes activity (AfL, 3rd principle).

As far as the quality of the code is concerned (see Fig. 11.3), one can first conclude that years 2018–2019 and 2019–2020 are outliers: the exam question may have been too easy in 2018–2019 while it may have been too difficult in 2019–2020, leading to respectively more Correct (60.5%) and Very Incorrect codes (63.3%). Designing an exam question of the same difficulty from year to year is a difficult task. Both questions in 2018–2019 and 2019–2020 required to browse an array. The main difference between the two problems is that the 2019–2020 one asked to write a result

in the last index of the array (*i.e.*, `A[N-1]`) and a lot of students wrote `A[N]`, leading their code to be labelled with the CODE_4 – Buffer Overflow, that we consider a major mistake.

Nevertheless, the proportion of Correct code increases after the introduction of the PCA (from 28.6% (2014-2015) and 29.0% (2015-2016) before to 37.9% (2016-2017) and 33.3% after). Consequently, those who seriously engage in the PCA (and thus have practised and take advantage of feedback and feedforward throughout the semester) are more likely to write a correct piece of code at the end of the course.

11.4 Conclusion

The data presented in this chapter shows that the evolution of the CS1 course and alignment with AfL principles have increased student engagement. Students have received earlier feedback and feedforward to improve their performance.

However, the data has shown that the students do not take all the offered opportunities to train themselves as far as, for example, the mocks MCQ are concerned.

Besides, the mean success rate (in January – students can retake the exam in June and August) has risen from 18.4% between 2013 and 2015 to 26.8% (+ 45%) between 2016 and 2019 (See Table 11.3). It is encouraging to think that these figures can partially be attributed to the course evolution, as also suggested in the Table.

The number of students exiting during the semester is still high, in particular after the mid-term exam results are reported. This was one of the reasons to introduce AfL principles in the course, especially feedback that may "Encourage positive motivational beliefs and self-esteem" according to Nicol and Macfarlane-Dick [148]. Those students dropping out cannot be totally eliminated, as their actions may be the result of the open-access policy to higher education in Belgium.

From our perspective it is unfortunate to lose such a large portion of our students. We hope that the CS1 course helps students understand the expectations and requirements of a university curriculum, in terms of work and commitment and that they go on to succeed.

In our faculty, the enrolment in the Computer Science curriculum meets the general open-access rule of Higher Education in Belgium. As far as the other curriculum organised in the faculty is concerned, *i.e.*, in Engineering, students must take an entrance exam (especially to test their maths skills). The engineering faculties of the French part of Belgium claim that this exam is responsible for their success rates that are significantly higher than in the other curricula [1] and that passing this exam is a proxy for success in the studies university, regardless of their subject [*ibid.*]. This tend to suggest the entrance examination actually ensure the motivation of the students before they enter the curriculum. We do not think that this entrance policy would be fit for the CS curriculum as it may discourage a lot of students profiles to enrol.

Chapter 12

Conclusion

Learning how to program a computer is at the core of the first course of Computer Science curricula [131], widely referred as *CS1*. In the context of the Higher Education in Belgium, where the open-access education is the rule and thus one cannot make any assumptions on students background, especially concerning their maths skills, we developed a programming methodology based on Graphical Loop Invariant.

With the idea of getting the students work in a regular basis, we designed a teaching activities consisting for students in submitting small pieces of code regularly during the semester: the Programming Challenges Activity (PCA). Such an activity enabled us to align our CS1 course towards the principles of the Assessment for Learning (AfL) [164].

Since the students were required to submit pieces of code during the semester and due to the lack of human resources, we developed a program capable of automatically assessing their work and providing them with *feedback* and *feedforward*, making them able to close the feedback loop [29]. Thus was born Correction Automatique et Feedback des Étudiants (CAFÉ) [119, 123].

This document, divided in three parts, firstly describes our programming methodology and its perception by the students. Secondly, the program CAFÉ is introduced and finally, we describe and evaluate the PCA, as well as the contribution of the AfL alignment to our CS1 course. Here is a summary of this work.

12.1 Part I: Graphical Loop Invariant Based Programming

The first part presents our programming methodology, called Graphical Loop Invariant Based Programming (GLIBP) that consists in drawing a Graphical Loop Invariant, which is a graphical representation of the variables that are manipulated in a loop, as well as the relationship they share. This picture must be drawn before writing the code, that is then deduced thanks to the

Graphical Loop Invariant.

To help students to draw useful Graphical Loop Invariant (*i.e.*,, that will actually help them in deducing their code), we provide them with seven rules: (1) The drawing shall correspond to the problem and be labelled; (2) The boundaries of the problem shall be provided; (3) One (or more) dividing line(s) shall be provided; (4) Each dividing line shall be properly labelled; (5) The drawing shall be labelled for explaining what has been achieved so far; (6) The drawing shall be labelled to indicate what should still be done; (7) All the named structures and variables shall be present in the code.

We also explain how to leverage the Graphical Loop Invariant to deduce the code instructions and illustrate on several examples how various data structures can be represented in our framework.

We also introduce two tools that were though to ease the teaching of the GLIBP methodology. First, Graphical Loop Invariant Drawing Editor (GLIDE) is an application that enables to draw Graphical Loop Invariant and to receive feedback about the syntax of the drawing. GLIDE also embeds a feature to help writing the code by supporting the graphical transformations described by the GLIBP methodology.

We then address the reception of our methodology by answering the research questions RQ 1. 1 to RQ 1. 4. Thanks to data collected from CAFÉ and GLIDE usages, we show that student can practise the GLIBP methodology all along the semester. However, even if GLIDE enables them to get preliminary feedback on their Graphical Loop Invariant, most of the students do not take this chance.

After having introduced a taxonomy of the errors that was built upon the GLIBP's rules, we show that errors committed depend, of course, of the problem being solved but some of them are recurrent, as the problem boundaries being missing; the lack of *Dividing Line*(s) labels and the lack of relationship with the code. The number of errors does not seem to decrease during the semester, as if all the programming activities were independent.

Students globally acknowledge the utility of the Graphical Loop Invariant to understand how a loop works. In majority they feel they are capable of using the GLIBP methodology to write a loop. However, they recognize they did not fully apply the methodology when they were supposed to. This seem to suggest that the Graphical Loop Invariant is perceived as the vegetables in a healthy diet: everybody will recognize they are useful but some would not complain if they are absent. As the Graphical Loop Invariant is concerned, this explains why students do not take the opportunity to validate their drawing in GLIDE or keep doing avoidable errors (like a missing Dividing Line label) during the semester and even at the exam. This suggest, when presenting the GLIBP methodology, to keep explaining how to leverage the Graphical Loop Invariant to deduce the code instructions.

As data suggested that an incorrect Graphical Loop Invariant would be correlated to an incorrect code and that a correct Graphical Loop Invariant would be more likely correlated to

a correct code, we formulated the RQ 1. 5 to investigate the link between the Graphical Loop Invariant and the code. The COVID-19 pandemic [190] forced us to lower our ambitions and we had to focus in assessing the impact of the Blank Graphical Loop Invariant on the code quality. We presented a Crossover Randomized Controlled Trial (CRCT) whose most of the results turned out to not be significant enough to conclude. Still, we set the course for future experiment to assess the GLIBP methodology.

12.2 Part II: Automatic Correction and Feedback to the Students

The second part introduces CAFÉ, the system we have developed to assess both students program and Graphical Loop Invariant and provide them with feedback and feedforward. CAFÉ is capable to detect coding errors that are difficult to detect by hand (*e.g.*, special cases of infinite loop) or even when the code is run (*e.g.*, out-of-bounds accesses) and is therefore valuable from a teacher point of view. Moreover, CAFÉ provides to students a service that would be unfeasible without such an automatic tool, although it requires for the teaching team to spend some times on correction programming.

The reception of the response message generated by CAFÉ is addressed by the research question RQ 2. 2. The submitting errors to CAFÉ decrease as the semester goes. CAFÉ's message are acknowledged to be clear, to help in course understanding and to make aware of an eventual learning gap.

12.3 Part III: Programming Challenges Activity

The third part introduces the Programming Challenges Activity (PCA) and positions it with respect to the other teaching activities of our CS1 course. The PCA enabled us to align the course to the Assessment for Learning (AfL) principles.

The research question RQ 3. 1 addresses the evaluation of the PCA. Results show that students do submit several times during the challenges and use their additional submission to increase their performance. However, only a minority of them take the opportunity to take several days to think about their work. Students recognise the PCA is a good way to make them work on a regular basis and make them feel confident about their programming skills.

The research question RQ 3. 2 focuses on the evaluation of the course transformation. Data show that the introduction of the PCA lead to higher success rate and seem to have increased the quality of the code. However, the PCA does not seem to have risen the participation in the final exam in January. While the number of students rose in recent years, the participation rate in the final exam followed a reverse trend.

The regular measure of students participation, enabled by the multiple activities such as the Challenges or the MCQs, reveal that the mid-term exams is the tipping point of the year, where students decide to drop out.

12.4 Future Work and Research Directions

12.4.1 Push forward the GLIBP methodology Assessment

Our attempt to experiment whether the Graphical Loop Invariant enables to write better code (see Chap. 5) was not successful due to the second lock-down of the COVID-19 pandemic [190].

The experiment methodology described in Chap. 5 is a starting point to push further the GLIBP methodology assessment. Once the effect of Graphical Loop Invariant on the code will be clearly established, it will be possible to assess each component of the GLIBP methodology to potentially adjust them: the number and the formulation of the rules, the shapes of the data structures representations, the way we explain how to deduce code instructions from a Graphical Loop Invariant, etc.

12.4.2 The Dropouts Case

The participation data analysed through the document show a large number of dropouts that can be observed after the mid-term exam in November. The surveys were not able to collect the opinions of the students who drop out during the semester. As such, the course abandonments are a blind spot of the evaluation of new teaching initiatives, not only for our CS1 course but even at the level of the first year of the Computer Science curriculum. Therefore, this problem should be addressed at the faculty level.

12.4.3 GAMECODE: GLIBP methodology exercises

During COVID-19 pandemic [190], the first lock-down forced us to switch to remote teaching. In the context of a CS2 course that continues to use the GLIBP methodology (but also introduces recursion, and basic data structures such as Files, Lists, Queues, and Stacks) we developed new homework exercises instead of giving students yet another podcast in their course schedule.

We called these exercises GAMECODEbecause they are inspired from GameBooks in which the reader can choose the path they takes to complete the story. With GAMECODE, students can choose their own solving path for each exercise. Moreover, they could solve them at their own convenience.

A GAMECODE exercise meets the following principles:

Stand-alone book a GAMECODE exercise is self-sufficient and contains the minimal information to complete the exercise;

Just-in-time theory the theoretical reminders are placed where they are needed and are as short as possible

“No spoilers hints" hints given to the students never reveal a solution, nor a part of it;

No single solution several solutions are always possible and a GAMECODE exercise always references the course forum to discuss them with the teachers or pairs.

When the pandemic is over, the effectiveness and interest of these exercises should be evaluated.

12.4.4 Integrating CAFÉ, GLIDE and even the GAMECODES

In addition to provide CAFÉ and GLIDE with better GUIs, one could think to merge them into a more powerful tool. The GUI of such a tool would show the Graphical Loop Invariant next to the code and enable to go more intuitively from the Graphical Loop Invariant to the code and vice versa. by doing so, the new tool would become a real IDE (Integrated Development Environment)[1].

12.4.4.1 Overview of such a Tool

The new tool could be used in four main modes:

PRACTISE This mode would proposed scripted exercises, inspired by the GAMECODE, of increasing difficulties that would depend on the student level. Here, the tool would perform formative assessment.

CHALLENGE This mode would embed all the Challenges of the PCA, providing summative assessment in the context of low-stakes activity. The PCA could be gamified [52, 53, 84, 90, 175]: *e.g.*, students would earn badges upon Challenges successes or other achievements, such as properly use dynamic allocation function, declaring and using arrays, et cætera.

EXAM This mode could be use for summative assessment at the end of the semester. The tool would display less feedback but would register carefully students coding behaviour (*e.g.*,, it could enforce the fact that the Graphical Loop Invariant is drawn before the code is written.

COMPANION This mode would enable the user to freely draw an Graphical Loop Invariant, check its syntax (as GLIDE does), and then write the corresponding code. The code content could also be checked to verify whether the data structure drawn and the variables in the label are present in the code (Rule 7 of the GLIBP methodology, see Sec. 2.1)

All theses activities are already possible but the new tool would go further thank to the synergy of CAFÉ and GLIDE features. For example, syntax highlighting in the Graphical Loop Invariant could be rendered in the code too, the Graphical Loop Invariant could be used to illustrate to the students why some errors occur such as infinite loops or out-of-bounds accesses.

As far as scripted exercises, challenges and exams are concerned, the tool could implement the Blank Graphical Loop Invariant in a much more user-friendly way than today.

The communication with the teaching team would also be eased as the tool would simplify the sharing of both the Graphical Loop Invariants and their respective codes with a teacher.

[1] By the way, the GL**IDE** acronym would still fit perfectly.

12.4.4.2 Data-Driven features

During all the programming activities, the new tool would keep track of all the mistakes committed by the users to customise the feedback and feedforward, as well as to recommand tailored practise exercise in PRACTISE mode. This would require Artificial Intelligence Capabilities. The information about common mistakes could also be retrieved by the teaching team, enabling to regulate the lectures content or pace if they detect students collectively struggle on a particular subject.

The time students take to correct a bug could be monitored to encourage struggling students (like what Marwan et al. [137] propose).

An AI-capable tool could also ask users to rate some feedback and feedforward messages to continually assess their quality.

12.4.4.3 Beyond Programming

Later, it would be interesting to study in what extent such a system capabilities could be reused in the context of other courses and not only programming ones. The tool could leverage on one hand the graphical manipulations and on the other hand, the assessment of reasoning about the graphics. The first non-CS candidates would be geometry, physics but one could also think about graph manipulation in economics.

Part IV | Appendices

APPENDIX

Guarded Commands Metalanguage and Weakest Precondition

This appendix presents the metalanguage introduced by Dijkstra [56] and compares it with our pseudocode, especially in terms of the rules to compute weakest precondition. Dijkstra's metalanguage comprises variable assignment, alternative and repetitive constructs. These instructions are listed in Listing A.1. It is worth noticing that the alternative and repetitive constructs may have several guards (denoted by B_i in Listing A.1)

Concerning the alternative construct, here is how it is evaluated: all the B_i guards are evaluated. Among all the guards that are evaluated to true, one of them is selected in a non-deterministic way and then, its corresponding statement list is executed. If no guard is true, the if..fi construct is equivalent to an abort instruction.

As far as the iterative construct is concerned, all the B_i guards are evaluated. Among all the guards that are evaluated to true, one of them is selected in a non-deterministic way and then, its corresponding statement list is executed. Then, this process is reiterated until all guards are false. If no guard were true at the beginning, the do..od construct is equivalent to a skip instruction (*i.e.*, an instruction that does nothing).

```
1  skip               // Do nothing
2  abort              // Abort
3  x := a             // Assignment
4  x := 0; y := 1     // Composition
5
6  // Alternative construct (IF)
7  // If all guards are false, equivalent to abort
8  if B1 → SL1
9   □ B2 → SL2
10      ...
```

```
11  □ Bn → SLn
12 fi
13
14 // Repetitive construct (DO)
15 // If all guards are false, equivalent to skip
16 do B1 → SL1
17  □ B2 → SL2
18      ...
19  □ Bn → SLn
20 od
```

Listing A.1: Guarded commands metalanguage “cheatsheet”. The B_i stand for Boolean expressions and the SL_i for statement lists

In the rest of the document, we use simpler alternative statement and loops. Listing A.2 presents the differences: our if statement always implies that, if its guard is false and no else-statement has been specified, nothing is done. Our loops only have a unique guard (*i.e.*, the Loop Condition) and its corresponding statement list is the *Loop Body*

```
1 // Alternative construct (IF)
2 if (B)
3     SLthen
4 [else
5     SLelse
6 // Corresponds to:
7 if B → SLthen
8  □ !B → SLelse // A missing else clause corresponds to a skip
9 fi
10
11 // Repetitive construct (DO)
12 while(B)
13     SLbody
14 // Corresponds to:
15 do B → SLbody
16 od
```

Listing A.2: Guarded commands metalanguage version of our instructions

A.1 weakest precondition calculation

The semantics of the commands is documented thanks to the weakest precondition. As a reminder, wp(S, R) is the weakest precondition upon which, the activation of the program S will terminate and lead to the postcondition R. For each statement of Dijkstra [56]'s metalanguage, we provide the weakest precondition calculation on a postcondition R as it appears in his book, on the left-hand side of the page. When applicable, we also provide a translation in our own pseudocode on the right-hand side of the page.

A.1.1 skip

For any condition R,

$$\mathrm{wp}(\texttt{skip},\ R) = R \qquad (A.1) \qquad\qquad \mathrm{wp}(;,\ R) = R \qquad (A.2)$$

In other words, skip does nothing.

A.1.2 abort

For any condition R,

$$\mathrm{wp}(\texttt{abort},\ R) = F \qquad (A.3) \qquad\qquad \text{Not applicable} \qquad (A.4)$$

Where F is "the predicate that is false in all point of the state space." [56, p. 14]. Our pseudocode does not need for an explicit abortion statement.

A.1.3 Assignment

For any condition R,

$$\mathrm{wp}(\mathrm{x} := \mathtt{E},\ R) = R[\mathrm{E/x}] \qquad (A.5) \qquad\qquad \mathrm{wp}(\mathrm{x} = \mathtt{E},\ R) = R[\mathrm{E/x}] \qquad (A.6)$$

$R[E/x]$ denotes the condition R where all the occurrences of x were replaced by E.

A.1.4 Composition

For any condition R,

$$\mathrm{wp}(\mathrm{S}_1;\mathrm{S}_2,\ R) = \mathrm{wp}(\mathrm{S}_1,\ \mathrm{wp}(\mathrm{S}_2,\ R)) \qquad (A.7) \qquad\qquad Idem \qquad (A.8)$$

In other words, the statement S_1 will be executed before S_2.

A.1.5 Alternative construct

$$\begin{aligned}\mathrm{wp}(\mathtt{IF},\ R) = {} & (\exists j, 1 \le j \le n, B_j) \\ & \wedge (\forall j, 1 \le j \le n, \\ & B_j \Rightarrow \mathrm{wp}(\mathrm{SL}_{\mathtt{j}},\ R))\end{aligned} \qquad (A.9)$$

$$\begin{aligned}\mathrm{wp}(\mathtt{if}\ (\mathrm{E})\ \mathrm{SL}_{\mathtt{then}}\ \mathtt{else}\ \mathrm{SL}_{\mathtt{else}},\ R) = {} & \\ (E \Rightarrow \mathrm{wp}(\mathrm{SL}_{\mathtt{then}},\ R)) & \\ \wedge (\neg E \Rightarrow \mathrm{wp}(\mathrm{SL}_{\mathtt{else}},\ R)) &\end{aligned} \qquad (A.10)$$

A.1.6 Iterative construct

Let $BB = (B_1 \vee B_2 \vee \cdots \vee B_n)$

If, for all states

$$(P \wedge BB) \Rightarrow \text{wp}(\texttt{IF},\ P) \tag{A.11}$$

Then

$$(P \wedge \text{wp}(\texttt{DO},\ T)) \Rightarrow \text{wp}(\texttt{DO},\ P \wedge \neg BB) \tag{A.12}$$

for all states.

This result is referred as the "fundamental invariance theorem for loop". In the equation A.11, the `IF` denotes a iterative construct where the `do` and `od` were replaced by `if` and `fi`, respectively. In the equation A.12, wp(`DO`, T) means a terminating loop (In practise, the loop termination is investigated separately, using the Loop Variant). To sum up, A.11 requires that P does not vary upon the activation of one iteration of the loop and A.12 concludes it will not vary upon the whole iteration (that does terminate). And yes, P is the Loop Invariant.

A.1.6.1 Using our pseudocode

Our loops have only one guard: `while(B)` $\texttt{SL}_{\texttt{body}}$

If, for all states

$$(Inv \wedge B) \Rightarrow \text{wp}(\texttt{if(B)}\ \texttt{SL}_{\texttt{body}},\ Inv) \tag{A.13}$$

Then

$$(Inv \wedge \text{wp}(\texttt{while(B)}\ \texttt{SL}_{\texttt{body}},\ T)) \Rightarrow \text{wp}(\texttt{while(B)}\ \texttt{SL}_{\texttt{body}},\ Inv \wedge \neg B) \tag{A.14}$$

for all states.

Results of the Focus Group on CAFÉ Messages

THIS APPENDIX contains the transcript of the recording of the focus group on CAFÉ messages, presented in Sec. 7.5. The font size was reduced on purpose to shorten the text size.

Introduction

IFRES Supervisor – Bonjour à tous, Je m'appelle Laurent Leduc, je ne suis pas de votre fac, je viens de l'IFRES, je suis un chercheur en pédagogie. L'IFRES, c'est un institut de formation et de recherche en enseignement supérieur de l'université et je travaille comme d'autres membres de notre équipe comme conseiller pédagogiques avec les profs pour optimiser la qualité des dispositifs d'enseignements que vous avez. Et le professeur Donnet et Simon nous consultent régulièrement pour discuter de la qualité de ce qui se fait dans votre cours de 1re année de M. Donnet. Donc l'idée, aujourd'hui, c'est pour autant que possible améliorer la qualité des cours, c'est de prendre l'avis des étudiants. Vous avez été choisi, apparemment, je ne sais pas très bien comment ça s'est passé, pour participer à cette séance et je vous en remercie parce que l'idée, c'est vraiment que vous donniez votre avis sur - on va parler plus spécifiquement - de la plateforme CAFÉ. Parlez vraiment très librement, c'est pas une évaluation, c'est en dehors du cours, l'idée est vraiment que sur bases des informations que vous nous donnez, on puisse faire autant que possible, c'est pas toujours possible mais évoluer éventuellement le dispositif CAFÉ. Donc, sentez-vous vraiment libres de parler même si Simon est là. Il est là parce qu'il a besoin d'entendre, lui qui en est notamment le concepteur. C'est au bénéfice des étudiants de votre années et des étudiants des années ultérieures que vous parliez librement, donc n'hésitez pas à la faire. Alors on va se centrer sur CAFÉ, qui est donc cette plateforme de soumission de réponses à une série de devoirs (sic.). Le but, quand ils ont créé la plateforme CAFÉ, c'est de vous donner l'occasion, à intervalles réguliers, de faire des exercices de codages. Ça vaut pour un certain nombre de points mais l'une des forces de ce dispositifs...

Unidentified – (on frappe à la porte)

IFRES Supervisor – Oui, entrez !

Student 3 – Excusez-moi pour le retard

IFRES Supervisor – ... Donc j'expliquais que c'est une consultation d'étudiants pour rendre un avis sur la plateforme CAFÉ, pour essayer de la faire évoluer. L'une des forces du dispositif CAFÉ, c'est qu'il vous permet notamment, pour chaque devoir (sic.), d'avoir trois soumissions, à chaque fois. Et donc, pour chacune de ces soumissions, vous recevez un feed-back, c'est sur le feedback qu'on va se concentrer aujourd'hui. Ce qu'on va faire, c'est qu'on va vous présenter - c'est ce que Simon vous a distribué, il va y revenir dans un instant - on a subdivisé les feedbacks en différentes parties et ce qui nous intéresse, nous, c'est de voir, chacune de ces sous-composantes du feedback que vous recevez, si ils (elles) vous apparaissent, pour vous, étudiants, utiles. Utiles, d'une part,

pour tirer le meilleur parti des trois soumissions qui sont à votre disposition, utiles pour vous rendre compte de ce qui est bon et pas bon dans votre réponse : cela vous permet de comprendre ce qui marche bien ou marche pas bien dans votre réponse. Et enfin, est-ce que ce feedback vous aide à vous améliorer, à rectifier le tir. Chaque fois qu'on va voir une des sous-composantes, c'est un peu les questions qu'on aura à l'esprit. Y'en a d'autres et c'est Simon qui va vous les présenter.

Teaching Assistant – Donc il y a cinq critères, on a envie de mettre le focus sur ces critères-là : est-ce que les informations sont données en quantité suffisantes ? Est-ce que, quand quelque chose est indiqué, il y a un lien explicite avec les critères d'une bonne performance = un challenge qui aurait un résultat satisfaisant. Est-ce que vous dit le feedback est bien en lien avec ça ? Est-ce que ce n'est pas heurtant / Est-ce que ça ne vous rebute pas ? Est-ce que c'est assez clair ? Dans la clarté, il y a parfois des informations qui vous être données, mais est-ce qu'il faut aussi une phrase d'introduction pour vous expliquer ce qui est mentionné dans le feedback. On y reviendra. Quand il y a une recommandation, elles sont souvent encadrées, est-ce qu'elles ne sont pas trop générales ?

Teaching Assistant – Alors, je vous ai donné deux feuilles, la première feuille, en couleur, c'est une soumission pour le Challenge # 2, elle est recto/verso. C'est un exemple de soumission et vous avez un feedback correspondant. Ici, dans les slides, on va passer sur quelques sous-parties de ce feedback et à chaque fois, on va poser les différentes questions concernant ce que Laurent vous a déjà expliqué.

Clarté de l'encodage de l'Invariant

Teaching Assistant – Donc, tout d'abord, une première question, qui est moins sur le feedback, sur l'activité en elle-même, vous avez un rappel de la consigne avec l'invariant muet, et la façon de l'encoder, ici, dans le squelette, est-ce que pour vous, la consigne est claire ? Première question, est-ce qu'il n'y a pas des points d'achoppement, ce genre de choses.

Several Students – Non

Student 1 – C'est ce que j'aurais fait si j'étais sur une feuille.

IFRES Supervisor – Tout le monde, parce que, exprimez vous tous, parce que c'est l'avis de tout le monde de fonctionner comme cela ?

Several Students – Ouais

Student 1 – Ça a été

IFRES Supervisor – Sans réserves ? Vous n'avez pas de copains qui vous ont déjà dit « ouais, c'est pas si clair » ?

Student 2 – Il m'a fallu un petit peu de temps, après, c'est moi aussi qui me suis pas très bien concentré dessus, pour comprendre qu'il y avait les mots en dessous et qu'il fallait compléter de cette manière-là. Après, c'est chacun sa manière de l'interpréter, c'est peut-être moi qui ai eu du mal à réfléchir comme ça. Mais y'a pas spécialement moyen de faire mieux que cela. Donc, à part ça...

IFRES Supervisor – OK donc la consigne est claire ?

Students Delegate – Ouais, c'est clair.

IFRES Supervisor – Parfait !

Feedback sur l'Invariant

Au sujet des titres

Teaching Assistant – On va tout de suite passer à la suite. Il y a tout d'abord ici, en jaune, dans l'ellipse : le feedback est subdivisé en sections et chaque section a un titre, est-ce que, à chaque fois qu'il y a un titre, est-ce que vous voyez bien à quoi cela correspond dans le feedback, quand vous voyez les différents titres ?

Several Students – Oui

Teaching Assistant – Globalement. De toute façon, il y aura plusieurs sections, y'aura plusieurs titres, on reviendra dans la suite, on reviendra sur chacun d'eux.

IFRES Supervisor – Sur celui-ci en particuliers, vous voyez ce qui se...

Several Students – Oui

Au sujets du feedback sur l'Invariant

Teaching Assistant – Je vais faire apparaitre ici différents rectangles, avec différentes parties, le rectangle bleu « voici comment le système comprend ... » (lecture). La boite rouge : « Rappelez-vous, l'Inv ... ». Et ensuite dans la boite orange : « La variable i ne semble pas ... ». Pour ces boites-là, en quoi cela vous permet de tirer partie de vos trois soumissions ? Est-ce que cela vous permet de vous rendre compte de ce qui est bon/pas bon, est-ce que cela vous aide à rectifier le tir ?

Student 1 – J'ai eu justement un problème, j'ai eu la même erreur dans mon Challenge, j'avais pas bien compris la dernière case avec la flèche (« la variable i ne semble ...»). Moi, dans ma tête, i = 1, si la boite 5, c'est 1, cela doit normalement être juste. Comme vous m'avez expliqué, c'est initialisé à 1, du coup c'est la même valeur, c'est ça que je n'avais pas bien compris. [Note : j'ai déjà affiché 1 ligne alors que rien n'a été affiché].

Teaching Assistant – Du coup, il n'y a pas assez d'explications à ce niveau-là ?

Student 1 – C'est moi qui avait mal compris.

Teaching Assistant – Si j'avais affiché l'Inv, en traduisant avec i qui vaut 1, pour montrer en quoi c'était incohérent, ça aurait peut-être ?

Student 1 – Oui, voilà, comme quand j'étais venu vendredi, vous m'aviez expliqué en quoi c'était faux.

Teaching Assistant – OK...

Student 1 – C'était plus complet que juste « ça ne semble pas initialisé à la valeur suggérée par l'Inv ».

Proposition de traduction de l'invariant dans la situation initiale

Teaching Assistant – Ça va. Peut-être une traduction pourrait être envisageable. À chaque fois qu'il y a des choses [remarques], ici on va collecter des données, et on ne promet pas que chaque remarque va donner lieu à une amélioration, on va faire ce qu'il est possible de faire.

IFRES Supervisor – Tout ce qu'on peut, c'est de prendre votre avis, ce n'est pas nécessairement possible, après de le faire mais au moins, on l'entend. C'est ça l'idée. Si on prend un peu cadre par cadre, le cadre bleu, vous voyez à quoi ça peut vous servir de recevoir cette info-là ?

Student 1 – Ben, oui.

Student 2 – Ouais, voir si la manière dont le système comprend notre Inv est la manière dont nous on le comprend, voir si c'est la même chose des deux côtés.

Students Delegate – Voir ce qu'on attend de nous en fait.

Student 1 – Si ce qu'on pensait dans notre tête a été retranscrit par la machine et donc...

IFRES Supervisor – Utile, donc cela

Several Students – Oui, oui, utile.

Suite des cadres

IFRES Supervisor – OK. Cadre vert ? Qu'est-ce que vous ? À quoi ça rime de vous dire ça ?

Student 1 – C'est nos erreurs, c'est ce qu'on a mal fait.

Teaching Assistant – OK. Est-ce qu'il faudrait une phrase qui introduit en disant : « passons en revue toutes les boites » ou est-ce, pour vous, c'est pas nécessaire ?

Several Students – Non

Students Delegate – Tant qu'il est écrit « boite 5 », donc voilà.

Teaching Assistant – Tant que la boite est indiquée. OK. Le fait que rien ne soit dit au niveau des autres boites, boites 1,2,3,4, est-ce que vous voudriez, est-ce que ce serait plus utile pour vous de savoir que, la [les] boites 1 à 4 sont correctes? Est-ce que le fait que c'est sous-entendu...

Several Students – C'est implicite

Student 1 – oui, si c'est un feedback, ça doit nous donner [...] pour nous améliorer.

Teaching Assistant – Dans votre tête, un feedback, ça vous dit juste ce que vous avez mal fait ?

Several Students – Ouais

Unidentified – Un feedback automatisé, en tout cas, oui.

Teaching Assistant – OK. Si on prend maintenant le cadre rouge, « Rappelez-vous ... », L'indentation de ce commentaire, est-ce que vous voyez que c'est en rapport avec la boite 6 ou pas ?

Student 1 – J'aurais pris plutôt global, moi.

Teaching Assistant – OK. C'est vrai que cette remarque-là elle est, enfin... mais elle s'affichait surtout si la boite 6 était incorrect en fait. Le « Rappelez-vous », c'est un peu un rappel théorique, est-ce que cela vous semble assez explicite ? Assez long ? Est-ce qu'il faudrait détailler ce qu'on entend par là ?

Student 3 – Étant donné qu'on a le cours comme support, je pense qu'un rappel comme ça suffit. Ce n'est pas la peine de donner énormément d'information, de tout servir sur un plateau. Je pense que c'est assez clair mais sans... Ça pourrait toujours être plus complet, on pourrait mettre entre parenthèses, quoique non, c'est déjà affiché au dessus de toute façon.

Student 2 – Il faut que ça reste reste concis quand même.

IFRES Supervisor – C'est l'avis de tout le monde ça ? J'entends bien votre avis mais...

Student 1 – Je suis d'accord.

Unidentified – Moi aussi

Student 3 – Je suis d'accord avec ça, sachant qu'il y a le cadre bleu en plus qui appuie un peu avec le « $1 \leq i \leq N+1$ » (Plusieurs étudiants : Oui). Avec le cadre rouge en rappel, je pense que, oui, c'est complet. C'est mon avis, en tout cas.

IFRES Supervisor – OK

Student 1 – Peut-être nous donner des pistes, par ce que je me rappelle que, à un précédent Challenge, y'avait le numéro du/des slides concernant l'erreur, pour aller voir dans le cours, ça peut faire gagner du temps et se focus sur d'abord notre erreur. Ça pourrait être pourquoi pas utile mais c'est, d'un autre côté, c'est vraiment nous mâcher le travail en mode « Va voir slide 54 et relis ! ». Ça peut nous aider.

Student 3 – Rajouter des références.

Student 1 – Autant nous aider pour qu'on comprenne mieux plutôt que nous laisser patauger et au risque d'apprendre de mauvaises choses.

Student 3 – Ce serait bien qu'il y ait un lexique des slides.

IFRES Supervisor – Oui, donc vous êtes en faveur de cela quand même, que l'information qui vous est donnée vous oriente d'autant que possible sur ce dont vous avez besoin.

Several Students – Mmmm

IFRES Supervisor – OK

Teaching Assistant – Pour l'instant, tel que c'est codé, il y a un nombre limité de références au cours qui sont données pour ne pas renvoyer des références vers 15 slides s'il y a 15 erreurs. Donc, ici à chaque fois, c'est dans les recommandations. Les recommandations « va voir le cours », elles sont limitées à 3 par Challenge, je pense qu'à la fin de l'entrevue, on verra si 3, c'est suffisant ou pas. On discutera peut-être là-dessus [C'est passé à 5 après e focus groupe]

Commentaires sur l'initialisation de la variable

Teaching Assistant – Sur l'initialisation de la variable, on va passer au slide suivant. Là, j'ai mis le code en question. C'est le code tel que l'étudiant l'a soumis. J'ai rappelé aussi le morceau d'Inv. Il y a la remarque qui dit « la variable i ... ». Et ensuite, il y a la recommandation « Initialisation des variables selon l'INV ... ». Première chose, je vous ai mis en parallèle ici, le code, est-ce que pour la question sur l'Inv, est-ce que vous avez besoin que code vous soit rappelé dans le feedback ou est-ce que pour vous, c'est clair que ... Quand vous lisez le feedback,qu'est-ce que vous avez sous les yeux quand vous lisez le feedback ?

Unidentified – Ouais, la question quoi [l'énoncé, on suppose]

Teaching Assistant – Y'a pas besoin de rajouter la ligne de code, là, qui soit ?

Students Delegate – Je ne pense pas que ça fasse du mal...

Student 2 – Après, il faut pas que ça embrouille, si on a peut-être plusieurs pistes et qu'on se dit : « c'est peut-être dans le code que j'ai mal fait un truc » alors que c'est pas spécialement ça, faut pas avoir 10000 pistes non plus.

Student 1 – Si c'est pas trop long comme le Challenge # 2, ça va : on peut aller chercher, si y'a 30/40 lignes grand max...

Students Delegate – Maintenant, si c'est quelque chose de, je ne sais pas, 100 lignes, vaut mieux avoir le bout de code qui pose problème.

Teaching Assistant – OK

IFRES Supervisor – Ça, ça vous permettrait de cibler l'endroit du code qui pose problème ?

Several Students – Ouais, Mmmm

IFRES Supervisor – Faut voir si on pourrait le faire, ça.

Teaching Assistant – À ce niveau-là, c'est relativement possible. C'est codable [# TODO]

IFRES Supervisor – Mademoiselle, vous alliez parler ?

Students 5 & 6 – Non, non, j'écoutais...

IFRES Supervisor – OK

Student 3 – Donc, la suggestion, c'est d'avoir cette aide-là à partir du moment où un code prend beaucoup de lignes ou un certain nombre de lignes, c'est ça ?

Student 2 – Comme dans le terminal : quand on fait une erreur de syntaxe ou quoi

Student 3 – Pour le Challenge # 2, le code n'était pas long, on peut voir tout de suite...

Student 2 – Si t'as des erreurs de syntaxe dans ton code...

Unidentified – [... pas compris]

Student 2 – Pour des erreurs comme celle-là, avoir l'équivalent du terminal qui nous cible bien où est la faute, ça pourrait aider.

Teaching Assistant – Et la recommandation dans l'encadré ? Est-ce que c'est pas trop général, trop bateau ? Si c'est cette erreur-là, est ce que ça ne vous semble pas... La recommandation qui est donnée, est-ce que cela vous semble intéressant ? Ou pas tellement, pour vous, c'est de toute façon ce qu'il faut faire dans ce cas d'erreur-là et c'est clair dans votre tête ?

Student 1 – Moi, ça me parait logique de faire cela. C'est le concept de base de l'Inv, oui, qui est ré-expliqué. Est-ce que ça peut être utile ? Oui. Ça dépend des gens, pour moi, personnellement, je sais qu'il faut que je refasse un dessin pour essayer de bien comprendre... Enfin, avoir ce message-là, ça ne va pas m'avancer dans la résolution de mon erreur.

Teaching Assistant – OK

IFRES Supervisor – C'est un peu trop général ?

Student 1 – Oui, ça ne me donne pas des pistes sur mon erreur à moi, c'est le concept de base de comment construire mon truc.

IFRES Supervisor – Les autres ? Par rapport à ce qu'il a dit ?

Students Delegate – Je suis d'accord avec lui, parce que c'est vrai qu'on nous dit juste ce qu'on doit faire, pas vraiment la voie qu'on doit prendre. On dit : « fais ça » et voilà, « fais le dessin ».

IFRES Supervisor – Est-ce que c'est mieux que rien ? Je veux dire, est-ce que c'est quand même une petite bouée pour vous ? Au moins, quand vous lisez ça, vous ne vous sentez pas abandonnés ou...

Student 3 – C'est un peu comme je le vois. C'est un commentaires qui pourrait être utile pour les personnes qui, par exemple, en début d'année, ne comprennent pas encore bien le principe des invariants et n'ont pas forcément assimilés la matière.

IFRES Supervisor – OK

Teaching Assistant – Et si, à la place, il y avait par exemple « Initialisation des variables selon l'INV », je vous traduis l'Invariant dans la situation initiale et ça donnerait : « J'ai déjà affiché 1 première ligne du damier », est-ce que ça pourrait permettre de mieux corriger ? Parce que, s'il y a déjà 1 ligne alors que c'est au début du code, et qu'il n'y a encore rien d'affiché, ça montre qu'il y a un problème dans le potage (sic.) ?

Student 3 – Ça aurait aidé plus, je pense. [# TODO]

Feedback sur le code

Affichage du résultat attendu, de l'étudiant et différence des deux

Teaching Assistant – OK. On passe à la [Suite]

IFRES Supervisor – Oui

Teaching Assistant – Ici, vous avez, sur la droite, le code qui a été produit par un étudiant et qui est relativement incorrect parce que quand on prend le code, la plateforme le compile et l'exécute, et il y a la section « Vérification du dessin des damiers. ». Le titre, est-ce que vous voyez bien à quoi cela correspond ?

Several Students – Ouais

Teaching Assistant – Ensuite, dans la suite, il y a ensuite ici la ligne ici en cyan : « Pour $N = 10$, $c1 = \{$, $c2 = \%$, à la ligne 3 ». La ligne 3, ça fait réf... Le code va imprimer tous les damiers et le comparer avec le damier attendu, et ensuite, il va sélectionner au hasard des lignes qui ne sont pas cohérentes et là, dans le cadre rose, il vous montre une ligne qui était le résultat attendu, le résultat de l'étudiant et ensuite, il imprime la différence. Est-ce que vous aviez tous bien compris ça comme ça ? Là, que la ligne 3 fait référence à la 3e ligne du tableau de l'étudiant ou est-ce qu'il faut le préciser, par exemple, à la 3e ligne du damier.

Student 2 – Non, C'est bon.

Student 4 – ? Ça prête à confusion quand même parce qu'on utilise déjà les numéros de lignes pour le code. Si on utilise les mêmes notations pour les deux, ça peut prêter à confusion.

Several Students – Ouais

Teaching Assistant – On pourrait confondre ça avec la troisième ligne de votre code ?

Student 1 – C'est vrai

Student 2 – Oui, je suis d'accord.

Paramètres utilisés pour le test

Teaching Assistant – OK, ça va. J'en prend note. Vous voyez tous pourquoi est-ce que je vous ai mis « Pour $N = 10$, $c1 = \{$, $c2 = \%$ » ?

Student 1 – Oui

IFRES Supervisor – À quoi ça rime, à vos yeux de vous le donner, pourquoi est-ce Simon a décidé de vous donner ça comme feedback ? C'est quoi l'intérêt de vous donner ça ?

Student 1 – C'est un test. Si on était l'utilisateur et qu'on testait le programme de notre côté, on aurait ça comme... c'est comme chez nous, moi, je le teste d'abord de mon côté pour voir si ça marche et donc.

Student 3 – Donc on peut retester avec exactement les mêmes valeurs, pour voir ce que ça donne.

IFRES Supervisor – C'est bien ça. Et ça vous aide de recevoir ça ? C'est utile, par rapport aux questions qui sont là [Au tableau, présentées dans l'intro] ? C'est utile pour savoir ce que vous avez bien fait/pas bien fait, par exemple ?

Several Students – Oui, c'est assez clair

Teaching Assistant – La ligne différence, est-ce qu'elle est utile ou est-ce que si on l'avait pas mise, vous auriez pu voir vous même la différence entre le résultat attendu et votre résultat.

Students Delegate – Elle est utile parce lorsqu'il y a une seule différence, voilà mais si y'en a plusieurs...

Student 1 – Pour ce cas-là, c'est utile mais pour d'autres, peut-être que ce sera moins utile.

Students Delegate – Ouais

Student 1 – À voir pour l'autre Challenge. Si c'est pas un damier ou autre chose, à voir avec ce qui est demandé. Là, on voit clairement la différence. Pour d'autres choses, on le verra peut-être pas et ça peut être utile d'avoir la différence.

IFRES Supervisor – Y'a aussi l'autre question. Est-ce que c'est utile pour rectifier le tir ? Ça n'a pas marché, donc vous allez refaire une deuxième soumission. Oui? Vous pouvez développer un peu ? En quoi est-ce utile ?

Students Delegate – On sait qu'on a faux et on sait plus ou moins où on a faux. Là, on voit qu'on a un % de trop, donc on va aller regarder dans le code ce qu'on doit faire pour ne pas l'avoir. Donc c'est utile, c'est comme si on compilait le programme et qu'on le lançait et on voit bien qu'on a pas ce qu'on attendait. Donc on essaie de voir ce qu'on peut faire pour avoir ce résultat attendu.

Student 3 – Ça nous met la comparaison sous les yeux, clairement. Je pense que c'est utile

Students Delegate – Oui.

IFRES Supervisor – Les filles, c'est votre avis aussi ?

Students 5 & 6 – Oui

Les étudiants et le débogage

Teaching Assistant – La démarche de débogage de votre code, ça vous parait limpide et que c'est de toute façon comme cela que vous devez procéder ? Y'a pas un moment donné, où on afficherait un résultat et d'être bloqué sur la question en ne voyant pas du tout comment procéder ou est-ce que vous voyez directement ce que vous allez devoir entreprendre pour... ?

Student 3 – Je ne pense pas que cela nous donne la méthode de résolution, ça nous donne un résultat. La méthode de résolution, je ne vois pas comment...

Teaching Assistant – Et quand vous avez ce genre d'erreur, est-ce que vous retournez voir l'Inv pour corriger le code ou est-ce que vous allez directement dans le code ?

Student 2 – Personnellement, je touche à tout jusqu'à ce que ça marche

Unidentified – Moi aussi

Teaching Assistant – OK

IFRES Supervisor – Et vous n'avez que trois soumissions, en définitive. Ça peut être insuffisant, alors ?

Student 2 – Oui, mais on teste chez nous.

Student 1 – Moi, par exemple, quand je fais un Challenge, je regarde d'abord chez moi si ça fait le résultat attendu et si ça fonctionne, alors, là, j'envoie. Seulement si ça marche chez moi.

Student 3 – De toute façon, CAFÉ teste l'Inv et dit si..., enfin l'invariant dépend du code.

Teaching Assistant – Il teste, il vérifie si l'initialisation est correcte dans le code et les autres parties du code sont testées un peu différemment. Donc le nombre d'itérations, on va le voir ici dans la suite. On essaie un peu de tester toutes les zones. La

vrai zone qui est testée ici, c'est le corps de la boucle. Le corps de la boucle, on peut pas vérifier que tout le monde...parce que vous êtes libres de mettre ce que vous voulez dans le corps de la boucle, l'ordre des opérations, c'est vous qui décidez. Moi, je ne peux pas vérifier ça très facilement, c'est très difficile à faire.

IFRES Supervisor – Et quand vous dites que vous le testez jusqu'à ce que ça marche, vous persévérez, peut-être que vous êtes des étudiants motivés, faut être persévérant pour se dire « je vais le faire jusqu'à ce que ça marche », ou vous pensez que vos condisciples font pareils et que c'est logique de fonctionner comme ça ?

Student 3 – Après, c'est pas juste des tests aléatoires. On essaie de cibler, par exemple, c'est est-ce que j'ai mis $<$ alors que c'est $\leq$ par exemple. Plus comme ça, si j'ai ce genre de choses. Si ça se trouve, c'est un truc tout bête comme ça, ça vaut le coup d'essayer.

Teaching Assistant – Est-ce que si j'avais ajouté une recommandation, du style, bon j'aurais peut-être pu faire un test, par exemple pour voir si votre résultat était plus long que le résultat attendu et donc il y aurait eu, dans une boite comme ça en disant « Tiens, il semble avoir trop de caractères, ça veut peut-être dire que vous faites trop d'itérations sur la ligne, vérifiez votre gardien ».

Students Delegate – Ouais, vérifier le gardien

Teaching Assistant – Est-ce que cet indice-là est utile ou pour vous, c'est de toute façon le gardien qu'il faut aller voir ?

Students Delegate – En voyant ça, ...

Student 1 – En voyant ça, c'est déjà expliqué

Student 3 – C'est affiché plus graphiquement, on va dire

Students Delegate – Y'en a qui ne comprennent pas forcément parce qu'ils commencent la programmation maintenant et j'en connais, je vais pas donner des noms, j'en connais qui voient ça et qui ne pensent pas forcément que c'est le gardien qui est mal fait, ou des choses comme ça. Mettre « vérifiez le gardien », ou quelque chose du genre pourrait aider certains.

IFRES Supervisor – C'est déjà, pour un bon étudiant, de n'avoir pas, entre guillemets, que ça ?

Several Students – Ouais

Students Delegate – Enfin, quelqu'un qui a plus ou moins, qui comprend ce que c'est une boucle. Parce que tout le monde ne le fais pas. Même si on est censé le faire. Tout le monde ne comprend pas exactement ce que c'est une boucle et comment elle fonctionne.

Student 3 – On est pas tous égaux face aux (autres?)

IFRES Supervisor – Mesdemoiselles ?

Students 5 & 6 – Oui

IFRES Supervisor – Monsieur, je vous sens plus, mitigé.

Student 3 – Non, je processais l'information que je viens d'entendre.

IFRES Supervisor – (rires) OK

Cas particuliers et nombre d'itérations

Teaching Assistant – Élide suivant ? Donc on a encore ici deux titres : le « cas particuliers N = 0 » et « Vérification du nombre d'itérations ». Vous voyez toujours à quoi correspondent les titres ?

Several Students – Moui

Teaching Assistant – OK. Pour le cas particulier, il va y avoir encore des boites qui vont apparaitre ici, en vert et en violet. Vous voyez pourquoi est-ce qu'on teste de manière indépendante et à quoi ça correspond ?

Student 1 – Je voulais, enfin, ça, à moins que je me trompe, dans le Challenge, ce serait peut-être intéressant de nous dire « fais attention ». Moi quand j'ai vu ça, j'ai vu « cas particulier », j'ai eu juste, tant mieux, dans ma tête, je n'y avais pas pensé.

Student 2 – Je n'y avait pas fais attention non plus.

Student 1 – C'est peut-être intéressant de ...

Teaching Assistant – En première approche, à la première soumission, vous faisiez pas attention au cas particulier

Unidentified – (V et PV) Non

Unidentified – Moi, j'ai pas du tout pensé

Student 2 – Moi tant mieux, j'ai pas pensé, j'aurais pu perdre des points là-dessus... Ces deux points-là, ce serait intéressant de préciser de faire attention.

Teaching Assistant – Une ligne, par exemple dans l'énoncé qui dirait « attention, testez bien N = 0 » ?

Student 2 – Ou alors, résoudre en dessous d'un certain nombre d'itérations. Pour, je sais pas.

Student 1 – « Faire attention aux cas particuliers et au nombre d'itérations », ça pourrait être...

IFRES Supervisor – Ça, pour le coup, c'est un critère d'une bonne performance. Est-ce que vous seriez intéressé de recevoir une espèce de liste, des points d'attention que vous devriez à l'esprit ?

Students Delegate – Oui, ça serait bien.

Student 1 – Oui, parce que là, vraiment, quand j'ai rendu, j'y ai pas pensé d'un coup. Et après, j'ai vu « ha ouais, c'est juste ».

IFRES Supervisor – Quelqu'un y avait pensé ?

Student 3 – Dans l'énoncé, ça parlait de nombres entiers supérieurs, c'est ça ? Ça parlait pas...

Teaching Assistant – Nombre entier, N pair

Student 1 – 0, C'est pair, donc...

Teaching Assistant – Si vous ratez, si on vous dit « c'est des nombres entiers pairs », vous perdez des points parce que N = 0 et que vous avez complètement oublié de tester le cas, est-ce que vous vous sentiriez, je ne sais pas ? volé ?

Student 2 – Oui

Unidentified – Mmmmm

Student 1 – J'aurais le démon (sic.).

Student 2 – Moi, non

Student 4 – Ça n'a juste pas de sens de faire un code qui ne fait rien, en fait c'est pour ça. On n'imagine pas cette possibilité, qui voudrait essayer de faire ça ? Pour moi, c'est un peu ça en tout cas.

Student 3 – Ce serait décevant de perdre des points pour un cas, enfin, c'est pas dans l'énoncé.

Unidentified – On aurait pas pensé

Student 3 – Voilà, si on respecte les consignes de l'énoncé, qu'on nous informe pas, c'est dommage. Faut informer, en fait.

Teaching Assistant – Vous avez l'impression de vous faire entuber, « Ah, il a été trop spépieux » ?

Student 3 – Un petit peu.

Student 4 – C'est un cas qui doit être quand même pris en compte.

Students Delegate – Je pense ça aussi.

Teaching Assistant – Oui

Student 4 – C'est pas pour ça qu'on écrit le code.

Teaching Assistant – Faut vérifier...

Student 3 – On est en 1re année, on est en train d'[emmerder ?] les étudiants

Unidentified – Peut-être qu'ils ont fait des codes de merde ?

Teaching Assistant – Pas forcément, Je l'ai vu après mais il y a des gens qui font des gardiens avec N - 1 avec N unsigned, N - 1 unsigned, ça fait 4 milliards, ils essaient d'imprimer un damier de 16 milliards de milliards de caractères et donc la machine de soumission crache. Donc, c'est vrai, qu'à un moment donné, il vaut mieux attirer l'attention là-dessus.

Student 2 – Après, c'est pas une question sur laquelle il faut compter des milliards de points.

Teaching Assistant – Non, il y a 2 pts sur 20, c'est pas...

Student 2 – C'est un cas qu'il faut quand même prendre en compte, je trouve. On n'est pas obligé de nous pré-mâcher le travail à 200% non plus. C'est vrai que moi, j'y ai pas pensé, on aurait peut-être dû me mettre la puce à l'oreille mais...

Publicité de la répartition des points

Teaching Assistant – La part relative des différents critères, est-ce que ça, par exemple, à chaque fois que y'a un titre, je pourrais mettre la pondération en fonction du titre, est-ce que ? Il faut que je vois ça avec le Professeur Donnet, c'est pas moi qui décide si on donne cette info. Mais est-ce que c'est une info intéressante ?

Several Students – Oui [# TODO]

Teaching Assistant – Par exemple, l'Inv, pour vous, qu'est-ce qui est le plus important dans le Challenge d'après vous ? Qu'est-ce qui est le plus côté ? Est-ce que c'est la partie code ou la partie Inv ?

Students Delegate – L'invariant ?

Teaching Assistant – D'après vous, combien ? 50/50, 70/30 ?

Student 1 – 70/30 Plutôt. 60/40.

Teaching Assistant – C'est 60/40 pour l'instant. Cette info-là, est-ce que vous préférez l'avoir de base ou est-ce que vous trouvez que le discours de Benoit Donnet au cours théorique de ce qu'il dit sur l'Invariant, est-ce que vous trouvez implicite que l'Inv soit plus côté que le code ?

Students Delegate – Il l'a déjà dit au cours, donc...

Student 2 – Oui, on a eu tout un cours sur l'Inv disant que c'était.

Students Delegate – Je vois pas comment ça pourrait nous aider parce qu pour que ça marche, on doit bien faire les deux. Qu'on sache lequel des deux est plus côté ou pas, ça ne change pas grand chose à notre problème.

Teaching Assistant – Y'a des gens ...

Students Delegate – On stresse plus s'il y a un problème

IFRES Supervisor – Du point de vue pédagogique, je suis d'accord avec vous.

Teaching Assistant – C'est vrai, mais y'a différents types d'étudiants, y'a des étudiants qui, face à une difficulté, quand t'es à la dernière soumission et que t'as plusieurs choses à corriger, y'a les étudiants dits stratégiques, qui vont préférer investir plus de temps sur ce qui va rapporter le plus de point.

Publicité des points d'attention

IFRES Supervisor – Du point de vue de l'apprentissage, pour cette raison-là, c'est pas spécialement intéressant que vous ayez ça. La question, c'est de savoir : tout d'un coup, vous voyez une question sur le cas particulier, certains d'entre vous l'ont dit, vous avez été relativement surpris que ça apparaisse dans le feedback, est-ce que vous auriez besoin, plutôt d'une sorte d'énumération de l'ensemble des critères ou des points que vous devez garder à l'esprit dans votre réponse, dont, en l'occurrence, les cas particuliers.

Student 4 – Dans ce cas-ci, ça peut être intéressant d'avoir un, d'être averti à l'avance qu'il y a des cas particuliers auquel il faut penser mais je ne pense pas qu'il faut aller trop dans les détails.

IFRES Supervisor – OK

Teaching Assistant – Ça, ça rajouterai, grosso-modo, 2 tiers de page, une page dans l'énoncé, l'énoncé ne deviendrait pas trop long, si on fait toute une liste de truc tac, tac, tac, tac, tac ?

Student 3 – On peut pas lister, parler des cas particuliers en général ?

Teaching Assistant – Il serait mis par exemple « faites attention au cas part. N = 0 », ce genre de choses. Ou est-ce que vous préférez une liste de tous les points d'attention ou est-ce que s'il y a une phrase, quelque part, dans l'énoncé, qui dit « attention, tenez compte de N = 0 » en note de bas de page, est-ce que ça suffit ?

Several Students – Ouais.

Unidentified – Ça suffit.

Teaching Assistant – Et donc les recommandations, « Testez le code avec le cas part. ...», qu'est-ce que vous pensez de cette recommandation ?

IFRES Supervisor – Utile ?

Student 3 – Ouais, c'est utile, je pense.

Unidentified – Mmmm

Unidentified – C'est [bon à prendre ?]

IFRES Supervisor – Clair pas trop général ?

Student 3 – Ce serait dans l'énoncé ou le feedback ?

Teaching Assistant – Le quoi ?

IFRES Supervisor – Ça, c'est dans le feedback.

Student 3 – L'information de tester... les cas part

Teaching Assistant – L'information « attention, testez N = 0 », ce serait dans l'énoncé. Ce serait par exemple, vous avez N, pair avec un renvoi en note de bas de page, vous savez que j'aime bien les notes de bas de page, y'en a souvent une demi-douzaine. Est-ce que y'en aurait pas trop à ce moment-là ?

Student 1 – Non, c'est comique [c'est le but]

Student 4 – Ça fait partie de l'énoncé aussi.

Teaching Assistant – Oui, c'est comique, dans l'Ingénu, ils procèdent de la même manière. Hé hé

Student 3 – Si l'énoncé est bien, on va dire, sectionné, les informations affichées clairement...

Teaching Assistant – Et pour l'instant, c'est clair à vos yeux, vu qu'on en parle ?

Several Students – Oui

Student 3 – L'énoncé , ça va.

Nombre d'itérations

Teaching Assistant – Pour le nombre d'itérations, on met « N = 18, y'a 360 itérations., votre code ... », qu'est-ce que vous pensez de ce genre de message ? Est-ce que vous voyez directement à quoi ça peut correspondre ?

Students Delegate – Le gardien, il est mal fait.

Teaching Assistant – Oui, on a pas envie d'être heurtant, on va pas mettre « c'est mal fait ! ».

Student 1 – Moi, je vois qu'il y a 360 itérations. OK, très bien, mais je ne vois pas à combien d'itérations je devrais être

techniquement. Ça m'avance pas. Ça produit trop mais combien de trop ? beaucoup trop ou ?

Student 3 – À combien on doit s'y attendre ?

Teaching Assistant – OK. Je pense que dans le Challenge qui arrive demain, il est marqué dans l'énoncé le nombre d'itérations auquel on doit s'attendre. C'est dans l'énoncé. Ce sera testé de cette manière là. Donc là ça serait... Et la recommandation pour limiter le nombre d'itérations, « inspectez son gardien »,

Students Delegate – Ça c'est

Student 1 – Ça c'est nickel.

Teaching Assistant – Ça vous partait utile ? Ou trivial, comme recommandation ?

Student 2 – D'une manière, c'est pas déroutant non plus, donc pourquoi est-ce qu'on l'enlèverait ?

Teaching Assistant – OK

Student 2 – Si y'en a qui pensent pas à ça à première vue.

IFRES Supervisor – Le commentaire n'est pas long

Student 2 – Oui, c'est 2 lignes.

Recommandation finale

Teaching Assistant – Slide Suivant. La dernière recommandation finale, dans le la page recommandations - c'est expliqué dans le manuel « Comment comprendre le feedback » - s'il y a des recommandations qui sont communes à plusieurs sous-questions, plusieurs titres, elles sont mises à cet endroit-là du document. Vous avez la recommandation, là : « Pour corriger vos Inv .. ». Là potentiellement, les deux invariants étaient incorrects et l'information est mise à cet endroit-là du feedback. Est-ce que c'est assez clair, est-ce que ce n'est pas trop loin dans le feedback, parce qu'on a d'abord parlé du code et des cas particulier ? La place de la recommandation finale ?

Student 4 – C'est un récapitulatif de tous les problèmes, finalement.

Student 1 – Mouais

Teaching Assistant – Si des problèmes plus ou moins communs parce que les autres recommandations ont déjà été données, en fait. Est-ce que ça gêne pas que ce soit trop bas, comme ça.

Student 1 – Moi, quand je vois, ici, c'est un petit feedback mais s'il y a beaucoup d'erreur et après, je vois ça, je me dis que, du coup, je dois remonter pour aller voir. Peut-être faire des recommandations par partie. Pour l'Inv , pour autre chose [# TODO : bonne idée] au lieu de faire tout un gros bloc à la fin, plutôt faire par partie comme ça on a bien les errerus qu'on a faite en haut et c'est plus simple.

Teaching Assistant – Chapitrer un peu les titres, en fait ?

Student 1 – Oui

Teaching Assistant – Avec des sections et des titres dans la sections ?

IFRES Supervisor – En même temps, vous trouvez que c'est un gros bloc, ce n'est pas très très long ?

Teaching Assistant – Il pourrait en avoir 3. Il pourrait avoir 3 blocs.

Student 1 – S'il y a une erreur, si c'est un Challenge plus long, et qu'il y a plus d'erreurs...

IFRES Supervisor – Ça sera long, oui, OK. Au niveau de la clarté ?

Unidentified – Clair

IFRES Supervisor – Pas trop général ?

Student 3 – C'est un peu le but de cette dernière recommandation [Non...]

Unidentified – Oui

IFRES Supervisor – Ok

Student 3 – Je trouve que c'est bien que ça se trouve à la fin. Ce format là, c'est pas très long non plus.

Conclusion et resoumission

Teaching Assistant – Ensuite,il y a le dernier cadre : « N'hésitez pas à soumettre si ... »

IFRES Supervisor – Ça vous incite ?

Student 1 – Ouais

Student 3 – On nous briefe, on nous a déjà bien briefé au sujet des re-soumissions, je pense que c'est pas forcément utile comme

info, mais ça clôture bien le feedback.

Student 2 – Ça clôture, ça résume bien

Student 1 – Il faut voir ça comme une fin de lettre, « Veuillez agréer, Monsieur/Madame, voilà », c'est emballé

Student 3 – Une petite note sympathique

Cotation sur les 3 soumissions

Teaching Assistant – Au niveau de la re-soumission, est-ce qu'il y a des choses qui vous incitent, qui vous freinent de manière générale, à re-soumettre ?

Student 1 – Ouais, moi j'ai un truc, c'est le fait par exemple, si on est à deux soumissions et qu'il nous reste plus qu'une soumission, que ça prenne, c'est logique mais en même temps c'est chiant, que ça prenne la dernière soumission, les points de la dernière soumission en compte. Par exemple, je prends pour le Challenge # 2, j'ai arrêté au bout de 2 soumissions, j'aurais bien voulu, 1) essayer pour du beurre après la fin du Challenge, parce que ça m'ennuie d'être en erreur, j'ai envie d'avoir juste, quoi. Donc, peut-être qu'après le temps du Challenge soit (sic.) fini, pouvoir quand même le faire pour du beurre, et que [2)], ça ne prenne que la plus grosse note au lieu que ça prenne la dernière note en compte. J'ai fait deux soumissions le # 3 et je suis resté sur une erreur. C'est embêtant mais c'est logique d'un autre côté sinon, c'est trop facile.

Student 3 – Personnellement, j'aurais apprécié, enfin, je sais pas si on peut obtenir une bonne note par hasard, mais je me suis fait avoir par ce système que la dernière soumission compte et j'ai réussi à perdre le peu de points que j'avais lors du 1er Challenge parce que j'étais stressé pour la dernière soumission et j'ai changé complètement de stratégie.

Teaching Assistant – Ouais

Student 3 – Moi, je pensais, enfin, j'avais en tête que le programme, CAFÉ, prenait peut-être la meilleure soumission, je ne sais pas si c'est applicable ? Ça pourrait être utile, je trouve, que CAFÉ garde, comment dire, la... compte-tenu du fait que CAFÉ est performant et objectif, est censé être objectif par rapport aux points qu'il donne, qu'il garde, peut-être, la meilleure note ou la meilleure performance. Le gars qui va oublier un « ; » pour sa dernière soumission et qui a fait un truc parfait pour la soumission 2, un truc qui marche et qui veut l'améliorer et qui a, je ne vais pas le dire, pas de chance parce qu'on est censé être des professionnels, enfin, on est des étudiants, on est censé devenir professionnels, on peut vraiment perdre tout.

Teaching Assistant – Vous trouver ça un peu trop punitif, en fait ?

Students Delegate – La moyenne peut-être ?

Unidentified – Ça pourrait être ça.

Student 3 – C'est... je trouve ça un peu vache mais je comprends le principe.

Student 1 – La moyenne, je suis pas pour, enfin ...

Teaching Assistant – Ou des genre de points de participation si vous foirez à la dernière soumission ? Du style, vous faites 15, 18 et ensuite vous oubliez un « ; », vous avez pas 0, vous avez, je ne sais pas, un petit quelque chose, quand même, pour ne pas...

Several Students – (rires)

Student 1 – Passer de 0 à 2

Teaching Assistant – Pas 2 mais bon...

Le stress

Student 3 – Y'a des élèves qui peuvent, un peu comme moi, se planter sur deux soumissions et puis, pour la troisième, recommencer from scratch, recommencer tout un code et du coup, faire moins bien. À cause du changement de stratégie. J'ai eu ce problème parce que, dans la boucle for, j'avais mis une virgule et CAFÉ ne voulait pas me donner de feedback.

Teaching Assistant – Ouais.

Student 3 – Donc j'étais complètement stressé ce jour-là et puis, j'avais tout recommencé. Bon là, j'avais quand même eu plus de points lors de la dernière soumission.

Teaching Assistant – Dans ce cas là, c'est un peu un stress inhibiteur ?

Student 3 – Oui, ça ne donne pas envie de resoumettre trois fois.

Student 4 – Après, dans ce cas-là, ton compilateur doit te le dire à l'avance. Si tu testes pas ton code et que tu l'envoies directement, c'est ta faute aussi je pense.

Teaching Assistant – Non, c'est ...

Student 3 – Le compilateur sait pas si ton gardien il est bon ou s'il a changé.

Teaching Assistant – Mais là, l'erreur au niveau de la virgule, ça a été corrigé dans CAFÉ

Student 3 – Ah, excellent.

Teaching Assistant – Y'a du code - c'est pas pour ça qu'il faut mettre des virgules, c'est très mauvais. -

Student 3 – J'en mettrai plus jamais

IFRES Supervisor – Les trois soumissions, c'est pas du tout l'idée que la 3e génère du stress, c'est de vous donner plusieurs possibilités.

Teaching Assistant – C'est un peu dommageable que la 3e génère ça. Peut-être, on en discutera avec le prof [# TODO]. Parce que les politiques : moi, je suis ouvert à vraiment tout, au niveau de CAFÉ et tout au niveau du cours, on peut même discuter de l'éventualité de ne pas faire d'examen, avec moi, on peut parler de ça, avec le prof, je ne pense pas... faut voir un peu ce que lui...

IFRES Supervisor – Il est très constructif par rapport à ça, il sait vraiment entendre les... oui oui.

Teaching Assistant – En tout cas mes remarques... celles des étudiants, peut-être pas. N'allez pas le dire.

Student 3 – J'ai donné ma suggestion...

Teaching Assistant – Non mais non [dans le sens oui], ça peut... d'avoir quand même des...

Student 3 – Ça peut rendre ou pas service

Reprise des questions générales

Teaching Assistant – Parce que pour l'instant, le fait que le dernier point qui est fait apparait au niveau de la soumission [la cote], c'est codé dans la plateforme et pas dans CAFÉ en fait mais on a quand même l'historique de vos points, donc on sait quand même voir et on pourrait vous calculer une cote, c'est dans l'ordre du possible.

Student 3 – Ou refaire un calcul, pas forcément garder la dernière note, faire un calcul ...

Teaching Assistant – ... alambiqué. En fonction if machin else la face de la lune est dans le premier quartier, c'est la racine cubique de la moyenne.

Several Students – (rires)

Student 4 – Ça peut avoir l'effet inverse si j'ai raté la première soumission pour une raison ou une autre, parce que j'ai bien foiré quelque chose de bête, si je refais le 2e parfaitement j'ai 20/20. Si on prend la moyenne des deux, j'aurai 10/20. Alors que la première...

IFRES Supervisor – Absolument

Student 1 – La moyenne, c'est pas fou

Teaching Assistant – Ce serait pas forcément la moyenne, ce serait un truc intelligent, vous me connaissez.

Student 4 – Perdre des points parce que j'ai envoyé une bêtise ou mal zippé mon truc ou quoi, c'est un peu dommage aussi.

Student 3 – Le but aussi, c'est de s'améliorer à chaque fois.

Teaching Assistant – Ouais

Unidentified – C'est trop punitif.

Teaching Assistant – Ça, à ce niveau-là, le niveau punitif au niveau du nom de l'archive, on a changé « Comment comprendre le feedback », donc maintenant, dès qu'il y a un message qui vous arrive en anglais, c'est vraisemblablement parce qu'il y a une erreur dans le nom de l'archive. Donc ça, il faut être très [vigilant]. Mais ça, peut vraiment rien faire car c'est pas nous qui contrôlons le fait que la machine crashe si vous avez mis une parenthèse dans le nom de l'archive. Ce n'est pas de mon ressort. Les autres qui programment cette partie-là, je leur ai déjà dit que c'était bête mais, ça ne devrait pas ... [changer] mais bon, j'en peut strictement rien. Sinon, au niveau général, maintenant qu'on a balayé tous les éléments du feedback, de manière plus globale, si on interroge les questions qui sont là.

IFRES Supervisor – La première ici, on vient de la poser un peu. La seconde, est-ce que dans l'information générale, dans l'ensemble du feedback, vous avez le sentiment d'avoir tout ce qu'il faut pour savoir ce qui est bon et pas bon dans votre code. Ou il y a des informations qui vous paraissent encore vous manquer ?

Student 2 – Non, c'est assez explicite. Je pense que quand on a, dans le cas, ici, avec l'accent circonflexe qui montre où est l'erreur, y'a pas photo, quoi.

Students Delegate – Ouais

Student 2 – Y'a pas moyen de le mettre plus sous le nez que ça, je pense.

Student 1 – Pour le code, oui, je suis d'accord qu'on voit bien où on a faux, c'est noté noir sur blanc. Peut-être que, comme je disait tantôt, pour l'Inv, pour le cas de ce Challenge-ci, j'ai pas trouvé ça assez explicite pour bien me montrer où j'avais faux. Donc voilà . Mais pour le code, ça va.

IFRES Supervisor – Monsieur ?

Student 4 – J'ai oublié ce que je voulais dire.

IFRES Supervisor – Est-ce qu'il y a des infos qui vous paraissent manquer à un moment ou à un autre ?

Student 1 – Non, parce qu'on nous précise bien correctement qu'est-ce qu'on doit aller revoir. Du coup, techniquement, ça doit nous aider à rectifier le tir, vu que... Pour la grosse recommandation là, on voit bien qu'est-ce qu'on doit aller revoir et quelle est la méthode à suivre.

IFRES Supervisor – Et vous Monsieur, vous avez commencé votre phrase par « même si »

Student 2 – J'ai dit, ouais, même si on est fainéant comme moi et lire une grande feuille de texte, c'est pas toujours facile, les conseils sont quand même encadrés et c'est clair quoi. Donc y'a Inv SP1, Inv SP2, y'a ça, ça, ça... Par contre, des fois, c'est dans l'énoncé où j'ai un peu du mal à m'y retrouver mais pour une fois, c'est moi qui lit en diagonale.

Nombre de recommandations

Teaching Assistant – J'ai plusieurs questions à ce niveau-là. Les recommandations, pour l'instant, au niveau du logiciel, je les ai limitées à 3 pour qu'il n'y en ai pas plus mais est-ce que je devrais en mettre 5 ? Est-ce que 10, ce serait pas trop ? Ou ? Qu'est-ce que vous en pensez ?

Student 1 – Pour moi, si y'a des erreurs, pour toutes les erreurs, faut mettre des recommandations

Student 3 – Plus y'a de recommandations, plus ça aide, mais une manière de les placer intelligemment par rapport aux contenus, à la vérification. Je trouve que dans des cadres, c'est bien mis en évidence, ça ne gêne pas l'information, le feedback.

Students Delegate – Tout dépend de la qualité de l'erreur, si c'est une erreur de syntaxe ou on a mis un « ; » là où il ne fallait pas ou quelque chose comme ça, mettre chaque fois où il y avait un « ; » mal placé, c'est dommage. Mais si c'est quelque chose de plus grave et de plus complexe, comme dans l'Inv. Imaginons qu'on doive faire un programme qui doive faire deux boucles et deux Inv, et ce sont des fautes qui sont...

Teaching Assistant – Communes ?

Students Delegate – Peut-être pas communes, parce si c'est commune, on peut les rassembler mais si c'est des fautes différentes et importantes, on peut pas les zapper pour mettre juste trois recommandations. Peut-être en mettre plus, tout dépend de la gravité de l'erreur et de la complexité du Challenge. [# TODO: priorité au delà de laquelle la recommandation s'affiche de toute façon]

Student 3 – Pas plusieurs fois la même recommandation.

Teaching Assistant – C'est déjà géré par le code. Ça été pensé dès le départ, heureusement. Y'aurait moyen de mettre plus de recommandations, sachant que y'en aura jamais 2 les mêmes et s'il y en a deux qui portent la même partie, ce serait plus en fin de document. C'est envisageable d'en mettre plus.

Student 3 – De toute façon, je pense pas qu'il y ait des centaines de recommandations différentes qui peuvent survenir. C'est souvent la même chose.

Student 2 – Le compilateur et les tests qu'on fait à la maison filtrent le plus gros des erreurs. Non ? Toutes les recommandations ont lieu d'être.

Student 3 – On compile pas forcément, y'a peut-être des gens qui...

Teaching Assistant – Ça s'est vérifié

Student 3 – ... qui vont soumettre comme ça sans ...

Students Delegate – Peut-être dire dans l'énoncé « compilez avant »

Student 2 – Ça a déjà été dit, ça non ? Il me semble.

Teaching Assistant – C'est pas trivial de dire « compilez votre code avant » ?

Student 2 – Ouais, c'est ça essayer de voir si ça fonctionne.

Student 1 – Y'a beaucoup de rappels qui ont été faits et qui sont trivial, ça ne fait pas de mal. Moi, ça me parait logique, peut-être qu'à d'autres non.

Au sujet du squelette

Teaching Assistant – Le squelette, que vous avez sous les yeux, en bleu, il y a à chaque fois les rappels de la manière de soumettre, ou presque, avec un rappel d'énoncé ? Est-ce que c'est pas trop indigeste, par rapport à l'énoncé ? J'ai parfois l'impression d'écrire les conditions générales d'une assurance ou d'un placement bancaire avec des...[Notes de Bas Pages] ?

Student 2 – Du moment que c'est structuré comme ça, non. C'est pas indigeste. Si on veut aller voir si on a oublié, « ah merde, comment est-ce que je vais faire », bam : c'est directement mis, souligné et on n'a plus qu'à aller regarder.

Teaching Assistant – Parce que là, ici, j'ai en plus intentionnellement activé la coloration syntaxique avant de vous l'imprimer. Est-ce que vous le faites aussi chez vous en coloration syntaxique pour voir la différence entre ?

Student 4 – Je pense que tout le monde le ferai

Students Delegate – Je le fais pas moi

Student 3 – Pour faire le code ?

Unidentified – Non...

Teaching Assistant – Le squelette. Quand tu prends le fichier chez toi Challenge3.txt et que tu affiches la coloration syntaxique C, c'est ça que tu va avoir sous les yeux ou avec les couleurs que tu choisis dans ton logiciel.

Student 3 – À partir du moment où je copie-colle dans le squelette, la coloration syntaxique... je ... pas.

Teaching Assistant – Pour vous, c'est assez clair, j'ai parfois des étudiants qui ont du mal à repérer l'endroit où on place une réponse, par exemple.

Student 3 – Voilà, ça, je vais pas dire que j'ai des difficultés à placer une réponse mais ça peut arriver, je pense. Là, il y a 9 points et au dessus, il y en a 6, mais ne scrolant on peut ...

Teaching Assistant – confondre les deux ?

Student 3 – ... à confondre. Y'a pas vraiment de séparation entre Inv SP1 et SP2

Student 2 – Oui, peut-être mettre une ligne de tirets dans le squelette.

Student 3 – Pour les distraits, y'aurait moyen de confondre. Ça m'est djà presque arrivé.

Teaching Assistant – Mettre une ligne de tiret. Ce serait les mêmes couleurs que ça, y'aurait moyen. Segmenter, avoir presque le même titre « Inv SP1 » et vous verriez « Correction Inv SP1 » dans le feedback, ça, ça pourrait être utile aussi. [# DONE]

Unidentified – Mmmm

Teaching Assistant – OK

Recommandations ? Demandes ? Imperfections ?

IFRES Supervisor – D'autres choses, que vous voulez dire sur le système CAFÉ en général ? Des recommandations ? Des demandes ? Des imperfections ?

Soumettre après le timeout

Student 1 – Je crois que je l'ai dit : d'avoir la possibilité, après le temps de soumission, que la machine puisse quand même évaluer notre Challenge. Je prends mon cas du Challenge 2, je suis resté à 2 soumissions, j'ai pas tenté la 3e mais j'aurais bien voulu quand même que la machine corrige la 3e soumission pour du beurre entre guillemets. Pour pas que je reste sur une erreur et que je comprenne mon erreur.

Student 3 – Voilà, après le timeout ...

Student 1 – Après le vendredi, 18h.

Student 3 – ... que CAFÉ accepte encore des soumissions mais sans enregistrer le résultat.

Student 1 – Juste pour tester notre code et voir si c'est juste.

Student 3 – Une possibilité en plus, oui.

Remerciements, etc.

Teaching Assistant – Ok, en tout cas, un grand merci. On va voir ça. Le check pour du beurre après la deadline, je viens de le noter, ça peut... J'en discuterai avec le prof. On va utiliser les données soit à des fins de recherche de notre côté soit pour améliorer la plateforme. Y'aura peut-être des choses qui d'ici à demain... Demain, c'est le Challenge 3, je vais peut-être déjà mettre des choses...

Students Delegate – Une question pour demain...

Teaching Assistant – Est-ce qu'il y a le QCM 3 ?

Students Delegate – ... est-ce qu'il y a QCM ou pas demain ?

Teaching Assistant – Non, y'a une erreur sur eCampus apparemment.

Students Delegate – Donc y'a pas QCM demain ?

Teaching Assistant – Non. Ça [les docs papiers], je vais le recycler moi-même, vous n'avez pas tellement besoin de ça... Et

merci à tous !

IFRES Supervisor – Merci pour votre temps !

BIBLIOGRAPHY

[1] Les études d'ingénieur civil – Programme de l'examen spécial d'admission. Booklet edited by UCL, ULB, ULg and UMONS, 2015. URL https://www.facsa.uliege.be/upload/docs/application/pdf/2015-12/brochure_admission_2015_web1.pdf.

[2] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. MIT Press McGraw-Hill, Cambridge, Mass. New York, second edition, 1996. ISBN 0262011530. URL https://web.mit.edu/alexmv/6.037/sicp.pdf.

[3] Jean-Raymond Abrial. *The B-Book. Assigning programs to meanings*. Cambridge University Press, August 2005. ISBN 0521021758.

[4] Alfred Aho and Jeffrey Ullman. *Concepts fondamentaux de l'informatique*. Dunod, Paris, 1993. ISBN 2100031279.

[5] Peter Aitken and Bradley L. Jones. *Le langage C*. CampusPress, Paris, 2003. ISBN 2744015881.

[6] Suad Alagić and Michael A. Arbib. *The design of well-structured and correct programs*. Springer-Verlag, New York, 1978. ISBN 0387902996.

[7] Michael Anderson, Bernd Meyer, and Patrick Olivier, editors. *Diagrammatic Representation and Reasoning*. Springer London, 2002. doi: 10.1007/978-1-4471-0109-3.

[8] ARES. Indicateurs de l'enseignement supérieur. Web page. URL https://www.ares-ac.be/en/statistiques/indicateurs.

[9] Ken Arnold, Gosling James, and Holmes David. *The Java programming language*. Addison-Wesley, Upper Saddle River, NJ, 2006. ISBN 9780321349804.

[10] Owen Astrachan. Pictures as invariants. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 112–118, 1991.

[11] Ralph-Johan Back. Invariant based programming revisited. In *Proc. International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (Petri Nets)*, June 2006.

[12] Ralph-Johan Back. Invariant based programming: basic approach and teaching experiences. *Formal Aspects of Computing*, 21(3):227–244, 2009.

[13] Ralph-Johan Back, Johannes Eriksson, and Linda Mannila. Teaching the construction of correct programs using invariant based programming. In *Proc. of the 3rd South-East European Workshop on Formal Methods*, 2007.

[14] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer Berlin Heidelberg, 2005. doi: 10.1007/978-3-540-30569-9_3.

[15] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free c programs. *Communications of the ACM*, 64(8): 56–68, aug 2021. doi: 10.1145/3470569.

[16] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 1.17 edition, 2021. URL `http://frama-c.com/download/acsl.pdf`.

[17] Theresa Beaubouef and John Mason. Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin*, 37(2):103–106, 2005. doi: 10.1145/1083431.1083474.

[18] Theresa Beaubouef, Richard Lucas, and James Howatt. The unlock system: enhancing problem solving skills in cs-1 students. *ACM SIGCSE Bulletin*, 33(2):43–46, 2001. doi: 10.1145/571922.571953.

[19] Kent Beck. *Test-Driven Development: By Example*. Addison Wesley Professional, December 2002. ISBN 9780321146533.

[20] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-69061-0.

[21] Yifat Ben-David Kolikant and Meirona Mussai. “so my program doesn’t run!” definition, origins, and practical expressions of students’(mis) conceptions of correctness. *Computer Science Education*, 18(2):135–151, 2008. doi: 10.1080/08993400802156400.

[22] Eli Bendersky. `pycparser`. complete C99 parser in pure python. Github Repository, 2020. URL `https://github.com/eliben/pycparser`.

[23] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming – 12 years later. *ACM Inroads*, 10(2):30–36, apr 2019. doi: 10.1145/3324888.

[24] Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. The effect of a web-based coding tool with automatic feedback on students' performance and perceptions. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, feb 2018. doi: 10.1145/3159450.3159579.

[25] Pierre Berlioux and Philippe Bizard. *Algorithmique : construction, preuve et evaluation des programmes*. Dunod, Paris, 1985. ISBN 2040157719.

[26] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the leon verification system. In *Proceedings of the 4th Workshop on Scala - SCALA '13*. ACM Press, 2013. doi: 10.1145/2489837.2489838.

[27] Conrad Bock, Steve Cook, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. OMG® Unified Modeling Language® (OMG UML®) Version 2.5.1. Technical Report formal/2017-12-05, Object Management Group, 2017. URL `https/www.omg.org/spec/UML/`.

[28] Curtis J. Bonk and Charles R. Graham, editors. *The Handbook of Blended Learning: Global Perspectives, Local Designs*. John Wiley and Sons, 2006. doi: 10.5465/amle.2008.31413871.

[29] David Boud. Sustainable assessment: Rethinking assessment for the learning society. *Studies in Continuing Education*, 22(2):151–167, August 2000. doi: 10.1080/713695728.

[30] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007.

[31] Achille Braquelaire. *Méthodologie de la programmation en C : norme C 99 - API POSIX*. Dunod, Paris, 2005. ISBN 2100490184.

[32] Markus Brauer. *Enseigner à l'Université: Conseils Pratiques, Astuces, Méthodes Pédagogiques*. Armand Colin, 2011. ISBN 9782200254582.

[33] Géraldine Brieven, Simon Liénardy, and Benoit Donnet. Lessons learned from 6 years of a remote programming challenge activity with automatic supervision. In *Annual Conference of European Distance and E-Learning Network*, pages 63–79. Springer, 2022. URL `https://hdl.handle.net/2268/292692`.

[34] Géraldine Brieven, Simon Liénardy, Lev Malcev, and Benoit Donnet. Graphical Loop Invariant Based Programming. In Catherine Dubois and Pierluigi San Pietro, editors, *Formal Methods Teaching. FMTea 2023*, volume 13962 of *Lecture Notes in Computer Science*, pages 17–33, Cham, 2023. Springer. doi: 10.1007/978-3-031-27534-0_2.

[35] Malcolm C. Brown. Using gini-style indices to evaluate the spatial patterns of health practitioners: Theoretical considerations and an application based on Alberta data. *Social Science & Medicine*, 38(9):1243–1256, may 1994. doi: 10.1016/0277-9536(94)90189-9.

[36] Burning Glass Technologies. Beyond point and click: The expanding demand for coding skills. Online report, June 2016. URL `https://academy.oracle.com/pages/Beyond_Point_Click_final.pdf`. [Online; accessed: 24 august 2021].

[37] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. Identifying challenging CS1 concepts in a large problem dataset. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 695–700, 2014. doi: 10.1145/2538862.2538966.

[38] Vincent A. Cicirello. On the role and effectiveness of pop quizzes in CS1. *ACM SIGCSE Bulletin*, 41(1):286–290, mar 2009. doi: 10.1145/1539024.1508971.

[39] Cineflix Productions. Mayday: Air Disaster. TV Show, 2003–2020. URL https://www.cineflixrights.com/our-catalogue/215. Aired in Belgium on Disscovery Channel (in French).

[40] David R. Cok. Openjml: Software verification for java 7 using jml, openjdk, and eclipse. 2014. doi: 10.48550/ARXIV.1404.6608.

[41] David R. Cok. JML and OpenJML for java 16. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*. ACM, jul 2021. doi: 10.1145/3464971.3468417.

[42] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[43] Michelle Craig and Andrew Petersen. Student difficulties with pointer concepts in c. In *Proceedings of the Australasian Computer Science Week Multiconference*, pages 1–10, 2016. doi: 10.1145/2843043.2843348.

[44] Michelle Craig, Andrew Petersen, and Jennifer Campbell. Answering the correct question. In *Proceedings of the ACM Conference on Global Computing Education*, pages 72–77, 2019.

[45] Nell B Dale. Most difficult topics in cs1: results of an online survey of educators. *ACM SIGCSE Bulletin*, 38(2):49–53, 2006. doi: 10.1145/1138403.1138432.

[46] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. How can automatic feedback help students construct automata? *ACM Transactions on Computer-Human Interaction*, 22(2):1–24, apr 2015. doi: 10.1145/2723163.

[47] Stephen Randy Davis. *C++ pour les nuls*. First Interactive, Paris, 2001. ISBN 2844278965.

[48] Pierre-Arnould de Marneffe. *Introduction à l'Algorithmique I*. Centrale des Cours de l'A.E.E.S., 1998.

[49] Paul J. Deitel and Harvey Deitel. *Intro to Python for Computer Science and Data Science: Learning to Program with Ai, Big Data and the Cloud*. PEARSON, February 2019. ISBN 0135404673.

[50] Claude Delannoy. *Programmer en langage C : cours et exercices corrigés*. Eyrolles, Paris, fifth edition, 2012. ISBN 9782212125467.

[51] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic grading of programming exercises in a MOOC using the INGInious platform. In *Proceedings of the European MOOC Stakeholder Summit 2015*, pages 86–91, 2015. URL https://www.info.ucl.ac.be/~pvr/DervalEMOOCS2015.pdf.

[52] Christo Dichev and Darina Dicheva. Gamifying education: what is known, what is believed and what remains uncertain: a critical review. *International journal of educational technology in higher education*, 14(1):9, 2017. doi: 10.1186/s41239-017-0042-5.

[53] Darina Dicheva, Keith Irwin, and Christo Dichev. Oneup: Engaging students in a gamified data structures course. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 386–392, 2019. doi: 10.1145/3287324.3287480.

[54] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. doi: 10.1145/362929.362947.

[55] Edsger W. Dijkstra. Notes on Structured Programming. Circulated privately, April 1970. URL `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`.

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

[57] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer New York, 1990. doi: 10.1007/978-1-4612-3228-5.

[58] Benoit Donnet. INFO0946-1 Introduction to computer programming. Course Program, 2021. URL `https://www.programmes.uliege.be/cocoon/20212022/en/cours/INFO0946-1.html`. [Online; accessed: 9 august 2021].

[59] Serge Dupont, Mikaël De Clercq, and Benoît Galand. Les prédicteurs de la réussite dans l'enseignement supérieur. *Revue française de pédagogie*, (191):105–136, jun 2015. doi: 10.4000/rfp.4770.

[60] Bruce Eckel. *Thinking in Java*. Pearson Education (US), March 2006. ISBN 0131872486.

[61] Stephen H. Edwards and Manuel A. Perez-Quinones. Web-CAT: automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 328–328, 2008. doi: 10.1145/1384271.1384371.

[62] Nickolas Falkner, Rebecca Vivian, David Piper, and Katrina Falkner. Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 9–14, 2014. doi: 10.1145/2538862.2538896.

[63] Peter Farrell. *Math Adventures with Python: An illustrated guide to exploring math with code*. No Starch Press, January 2019. ISBN 1593278675.

[64] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Programming Languages and Systems*, pages 125–128. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-37036-6_8.

[65] Timothy Fisher. *Java : l'essentiel du code et des commandes*. CampusPress, Paris, 2007. ISBN 9782744021626.

[66] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *ACM SIGPLAN Notices*, 37(5):234–245, may 2002. doi: 10.1145/543552.512558.

[67] Robert W. Floyd. Assigning meanings to programs. In Jacob T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*. American Mathematical Society, 1967. ISBN 9780821892343.

[68] Northern Illinois University Center for Innovative Teaching and Learning. Instructional scaffolding. In *Instructional guide for university faculty and teaching assistants*. Online Instructional Guide, 2012. URL `https://www.niu.edu/citl/resources/guides/instructional-guide`. [Online; accessed on 4 August 2021].

[69] Jicheng Fu, Farokh B Bastani, and I-Ling Yen. Automated discovery of loop invariants for high-assurance programs synthesized using ai planning techniques. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 333–342. IEEE, 2008.

[70] Carlo A Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys (CSUR)*, 46(3):1–51, 2014.

[71] Tony Gaddis. *Starting out with programming logic and design*. Pearson, NY, fifth edition, 2019. ISBN 9780134801155.

[72] Tony Gaddis. *Starting Out with Visual C#*. Pearson, fifth edition, March 2019. ISBN 0135183510.

[73] Tony Gaddis, Judy Walters, and Muganda Godfrey. *Starting Out with C++*. Pearson Education (US), February 2019. ISBN 0135235006.

[74] Richard Gerber, Aart J. C. Bik, Kevin B. Smith, and Tian Xinmin. *The software optimization cookbook : high-performance recipes for IA-32 platforms*. Intel Press, Hillsboro, Or, second edition, 2006. ISBN 0976483211.

[75] David Ginat. On novice loop boundaries and range conceptions. *Computer Science Education*, 14(3):165–181, 2004.

[76] Wolfram Gloger, Doug Lea, and The Contributors to the Gnu C Library. *malloc, free, calloc, realloc – allocate and free dynamic memory*, 2021. URL `https://man7.org/linux/man-pages/man3/malloc.3.html`. Release 5.11 of the Linux man-pages project.

[77] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, fourth edition, 2006. ISBN 987-0-471-73884-8.

[78] Charles R. Graham, Wendy Woodfield, and J. Buckley Harrison. A framework for institutional adoption and implementation of blended learning in higher education. *The Internet and Higher Education*, 18:4–14, jul 2013. doi: 10.1016/j.iheduc.2012.09.003.

[79] Vincent Granet. *Mini manuel d'algorithmique et de programmation : cours + exos corrigés*. Dunod, Paris, 2012. ISBN 9782100573509.

[80] Yves Granjon. *Informatique : algorithmes en Pascal et en langage C : rappels de cours, questions de réflexions, exercices d'entraînement*. Dunod, Paris, 2004. ISBN 2100485288.

[81] David Gries. *The Science of Programming*. Springer, 1987.

[82] Thierry Groussard. *Java 6 : les fondamentaux du langage Java*. Éditions ENI, St-Herblain, 2009. ISBN 9782746047617.

[83] Brian H. Hahn and Daniel T. Valentine. *Essential MATLAB for engineers and scientists*. Elsevier/Academic Press, Amsterdam Boston, 2010. ISBN 9780123748836.

[84] Brian Harrington and Ayaan Chaudhry. Tracademic: improving participation and engagement in cs1/cs2 with gamified practicals. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 347–352, 2017. doi: 10.1145/3059009.3059052.

[85] Michael T. Helmick. Interface-based programming assignments and automatic grading of java programs. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '07*. ACM Press, 2007. doi: 10.1145/1268784.1268805.

[86] David Hemmendinger, Paul A. Freiberger, William Morton Pottenger, and Swaine Michael R. History of computing – The personal computer revolution. Encyclopaedia Britannica article, 2021. URL `https://www.britannica.com/technology/computer/The-personal-computer-revolution`. [Online; accessed: 12 August 2021].

[87] Colin A. Higgins, Geoffrey Gray, Pavlos Symeonidis, and Athanasios Tsintsifas. Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing*, 5(3):5, sep 2005. doi: 10.1145/1163405.1163410.

[88] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259.

[89] Silas Hsu, Tiffany Wenting Li, Zhilin Zhang, Max Fowler, Craig Zilles, and Karrie Karahalios. Attitudes surrounding an imperfect AI autograder. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, may 2021. doi: 10.1145/3411764.3445424.

[90] Maria-Blanca Ibanez, Angela Di-Serio, and Carlos Delgado-Kloos. Gamification for engaging computer science students in learning activities: A case study. *IEEE Transactions on learning technologies*, 7(3):291–301, 2014. doi: 10.1109/TLT.2014.2329293.

[91] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, January 2013. ISBN 859037985X.

[92] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*. ACM Press, 2010. doi: 10.1145/1930464.1930480.

[93] Jean-Louis Imbert. *Algorithmes fondamentaux et langage C : programmation : codage, alternatives, boucles, tableaux, modularité*. Ellipses, Paris, 2008. ISBN 9782729838676.

[94] Emmanuel Jakobowicz. *Python pour le data scientist : des bases du langage au machine learning*. Dunod, Malakoff, 2019. ISBN 9782100801626.

[95] Jacques Julliand. *Cours et exercices corrigés d'algorithmique vérifier, tester et concevoir des programmes en les modélisant*. Vuibert, Paris, 2010. ISBN 9782311000207.

[96] B. L. Juneja and Anita Seth. *Programming with C*. New Age International (P) Ltd., Publishers, New Delhi, 2009. ISBN 9788122426137.

[97] Ville Karavirta, Petri Ihantola, Juha Helminen, and Mike Hewner. js-parsons – a JavaScript library for Parsons Problems. Github Webpage. URL `https://js-parsons.github.io/`. [Online; accessed: 27 July 2021].

[98] Ville Karavirta, Ari Korhonen, and Lauri Malmi. On the use of resubmissions in automatic assessment systems. *Computer science education*, 16(3):229–240, 2006. doi: 10.1080/08993400600912426.

[99] Ville Karavirta, Juha Helminen, and Petri Ihantola. A mobile learning application for parsons problems with automatic feedback. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research - Koli Calling '12*. ACM Press, 2012. doi: 10.1145/2401796.2401798.

[100] Brian W. Kernighan and Dennis M. Ritchie. *Le langage C norme ANSI*. Dunod, Paris, second edition, 2004. ISBN 9782100487349.

[101] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43, 2018. doi: 10.1145/3231711.

[102] Jesse M. Kinder and Philip Nelson. *A student's guide to Python for physical modeling*. Princeton University Press, Princeton, 2015. ISBN 9780691170503.

[103] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27 (3):573–609, may 2015. doi: 10.1007/s00165-014-0326-7.

[104] Avraham N. Kluger and Angelo S. De Nisi. The effects of feedback interventions on performance: A historical review, a meta-analysis, and a preliminary feedback intervention theory. *Psychological Bulletin*, 119(2):254–284, March 1996. doi: https://psycnet.apa.org/doi/10.1037/0033-2909.119.2.254.

[105] Cyril H. Knoblauch and Lilian Brannon. Teacher commentary on student writing: The state of the art. *Freshman English News*, 10(2):1–4, Fall 1981. URL `http://jstor.org/stable/43518564`.

[106] Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley, second edition, April 1998. ISBN 0201896850.

[107] Stephen G. Kochan. *Programming in C*. Developer's Library. Sams Publishing, Indianapolis, Ind, third edition, 2005. ISBN 0672326663.

[108] L Kovacs and Tudor Jebelean. Automated generation of loop invariants by recurrence solving in theorema. In *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04)*, pages 451–464. Mirton Publisher Timisoara, Romania, 2004.

[109] Laura Ildiko Kovacs and Tudor Jebelean. An algorithm for automated generation of invariants for loops with conditionals. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pages 5–pp. IEEE, 2005.

[110] Amruth N Kumar. Using problets for problem-solving exercises in introductory c++/java/c# courses. In *2013 IEEE Frontiers in Education Conference (FIE)*, pages 9–10. IEEE, 2013. doi: 10.1109/FIE.2013.6684774.

[111] Michael Kupferschmid. *Classical FORTRAN : programming for engineering and scientific applications*. CRC Pressr, Boca Raton, 2009. ISBN 9781420059076.

[112] Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In Rudolf Fleischer, Bernard Moret, and Erik Meineche Schmidt, editors, *Experimental Algorithmics. From Algorithm Design to Robust and Efficient Software*, volume 2547 of *Lecture Notes in Computer Science*, pages 78–92. Springer, Berlin Heidelberg, 2002. ISBN 9783540363835. doi: 10.1007/3-540-36383-1_4.

[113] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[114] K. Rustan M. Leino and Valentin Wüstholz. The dafny integrated development environment. *EPTCS 149, 2014, pp. 3-15*, April 2014. doi: 10.4204/EPTCS.149.2.

[115] Abe Leite and Saúl A. Blanco. Effects of human vs. automatic feedback on students' understanding of AI concepts and programming style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, feb 2020. doi: 10.1145/3328778.3366921.

[116] Jean-Michel Léry. *Le langage C*. Pearson Education France, Paris, 2005. ISBN 2744070866.

[117] Simon Liénardy. Learning computer programming around a CAFé. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 318–319. ACM, aug 2020. doi: 10.1145/3372782.3407119.

[118] Simon Liénardy. Ça ferait presque le CAFÉ : une évaluation continue, à distance, fournissant du feedback et automatisée – Retour sur 4 ans d'évolution du cours d'Introduction à la Programmation en Sciences Informatiques à l'ULiège. *Rallye Péda.9 – Vers une évaluation 2.0 ? De la menace à l'opportunité*. Youtube Video and slides, 2020. URL `http://hdl.handle.net/2268/253206` and `https://youtu.be/NTV5al5c4PQ`.

[119] Simon Liénardy. CAFÉ Automatic Correction and Feedback to Students. Github Repository, 2021. URL `https://github.com/slienardy/CAFE`.

[120] Simon Liénardy and Benoit Donnet. GameCode: Choose your Own Problem Solving Path. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, page 306. ACM, aug 2020. doi: 10.1145/3372782.3408122.

[121] Simon Liénardy, Laurent Leduc, and Benoit Donnet. CAFÉ: an automatic and on-line learning system to guide freshmen towards the meeting of higher education requirements. *European First Year Experience Conference 2018*. Slides, 2018. URL http://hdl.handle.net/2268/221117.

[122] Simon Liénardy, Lev Malcev, and Benoit Donnet. Graphical loop invariant programming in CS1. *Grascomp Doctoral Day 2019 (GDD'19)*. Long Abstract and Slides, 2019. URL http://hdl.handle.net/2268/241671.

[123] Simon Liénardy, Laurent Leduc, Dominique Verpoorten, and Benoit Donnet. Café: Automatic correction and feedback of programming challenges for a CS1 course. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pages 95–104, 2020. doi: 10.1145/3373165.3373176.

[124] Simon Liénardy, Benoit Donnet, and Laurent Leduc. Promoting engagement in a CS1 course with assessment for learning. *Student Success*, 12(1):102–111, mar 2021. doi: 10.5204/ssj.1668.

[125] Simon Liénardy, Laurent Leduc, Dominique Verpoorten, and Benoit Donnet. Challenges, multiple attempts, and trump cards: A practice report of student's exposure to an automated correction system for a programming challenges activity. *International Journal of Technologies in Higher Education*, 18(2):45–60, 2021. ISSN 1708-7570. doi: 10.18162/ritpu-2021-v18n2-03.

[126] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[127] Alf Lizzio, Keithia Wilson, and Roland Simons. University students' perceptions of the learning environment and academic outcomes: Implications for theory and practice. *Studies in Higher Education*, 27(1):27–52, feb 2002. doi: 10.1080/03075070120099359.

[128] Simon Liénardy, Laurent Leduc, and Benoit Donnet. CAFÉ : un système d'évaluation et de feedback automatique des étudiants en science informatique*Colloque du DIdactifen 2018 : Les disciplines enseignées : des modes de penser le monde*. Slides, 2018. URL http://hdl.handle.net/2268/222640.

[129] Richard Lobb and Jenny Harlow. Coderunner: A tool for assessing computer programming skills. *ACM Inroads*, 7(1):47–51, 2016. doi: 10.1145/2810041.

[130] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. Programming, problem solving, and self-awareness: effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 1449–1461, 2016.

[131] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda M. Ott, James H. Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pages 55–106, 2018. doi: 10.1145/3293881.3295779.

[132] Sze Yee Lye and Joyce Hwee Ling Koh. Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41:51–61, dec 2014. doi: 10.1016/j.chb.2014.09.012.

[133] Rémy Malgouyres, Rita Zrour, and Fabien Feschet. *Initiation à l'algorithmique et à la programmation en C cours avec 129 exercices corrigés*. Dunod, Paris, third edition, 2014. ISBN 9782100710010.

[134] José Marien. *Éléments de thermodynamique*. Centrale des Cours de l'AEES asbl, 2009.

[135] Victor J. Marin, Maheen Riaz Contractor, and Carlos R. Rivero. Flexible program alignment to deliver data-driven feedback to novice programmers. In Alexandra I. Cristea and Christos Troussas, editors, *Intelligent Tutoring Systems*, pages 247–258. Springer International Publishing, 2021. doi: 10.1007/978-3-030-80421-3_27.

[136] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson, 2009. ISBN 9780134695822.

[137] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W. Price, and Tiffany Barnes. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ACM, aug 2020. doi: 10.1145/3372782.3406264.

[138] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, January 2012. ISBN 0521173019.

[139] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90, 2018. doi: 10.1109/TE.2018.2864133.

[140] Michael Metcalf, John Reid, and Malcolm Cohen. *Modern Fortran explained*. Oxford University Press, Oxford New York, 2011. ISBN 9780199601424.

[141] Janet Metcalfe and Arthur P. Shimamura, editors. *Metacognition: Knowing about knowing*. MIT press, 1994. ISBN 9780262279697. doi: 10.7551/mitpress/4561.001.0001.

[142] Bertrand Meyer. A basis for the constructive approach to programming. In *IFIP Congress*, pages 293–298, 1980. URL `http://se.ethz.ch/~meyer/publications/constructive/constructive_ifip.pdf`.

[143] Carroll Morgan. *Programming from specifications*. Prentice-Hall, 1990. ISBN 978-0131232747. URL `https://www.cs.ox.ac.uk/publications/books/PfS/`.

[144] F. Lockwood Morris and Clifford B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984. doi: 10.1109/MAHC.1984.10017.

[145] Susanne Narciss. *Handbook of research on educational communications and technology-tasks*, chapter Feedback strategies for interactive learning tasks, pages 125–144. Routledge, third edition, 2008. ISBN 9780415963381.

[146] Susanne Narciss and Katja Huth. *Instructional Design for Multimedia Learning*, chapter How to Design Informative Tutoring Feedback for Multimedia Learning, pages 181–195. Waxmann, 2004. ISBN 3-8309-1384-2.

[147] David Nicol. *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, 2009. ISBN 9781844829019. URL `http://dera.ioe.ac.uk/11605`.

[148] David J. Nicol and Debra Macfarlane-Dick. Formative assessment and self-regulated learning: a model and seven principles of good feedback practice. *Studies in Higher Education*, 31(2):199–218, apr 2006. doi: 10.1080/03075070600572090.

[149] Linda B Nilson. *The graphic syllabus and the outcomes map: Communicating your course*, volume 137. John Wiley & Sons, 2009. ISBN 978-0-470-18085-3.

[150] Seymour Papert. *Mindstorms : children, computers, and powerful ideas*. Basic Books, New York, 1980. ISBN 0465046274. URL `https://mindstorms.media.mit.edu/`. Text freely available online (see URL).

[151] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, jun 2017. doi: 10.1145/3059009.3059026.

[152] Jack Parkinson and Quintin Cutts. Investigating the relationship between spatial skills and computer science. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 106–114, 2018. doi: 10.1145/3230977.3230990.

[153] Nick Parlante. CodingBat: Code Practice, 2011. URL `https://codingbat.com`. [Online; accessed: 30 March 2019].

[154] Pearson. My Lab Programming. URL `https://www.pearsonmylabandmastering.com/northamerica/myprogramminglab/`. [Online; accessed: 30 March 2019].

[155] George Pólya. *How to Solve It*. Princeton University Press, 1945. ISBN 9780691164076.

[156] James Prather, Raymond Pettit, Brett A Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 531–537, 2019. doi: 10.1145/3287324.3287374.

[157] Danijel Radošević, Tihomir Orehovački, and Alen Lovrenčić. Verificator: educational tool for learning programming. *INFORMATICS IN EDUCATION*, 8(2):261–280, 2009. ISSN 1648-5831. URL https://ssrn.com/abstract=2505746.

[158] Alain Rey. *Dictionnaire des expressions et locutions*. Dictionnaires Le Robert, Paris, 2006. ISBN 9782849023174.

[159] Kelly Rivers and Kenneth R Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, volume 50, 2013. URL http://ceur-ws.org/Vol-1009/0900.pdf.

[160] Enric Rodríguez-Carbonell and Deepak Kapur. *Theoretical Aspects of Computing - ICTAC 2004*, volume 3047 of *Lecture Notes in Computer Science*, chapter Program verification using automatic generation of invariants, pages 325–340. Springler, 2004. ISBN 978-3-540-25304-4. doi: 10.1007/978-3-540-31862-0_24.

[161] Joanne K. Rowling. *Harry Potter and the philosopher's stone*. Bloomsbury Pub, London, 1997. ISBN 0747532699.

[162] Sandra Milena Merchán Rubiano, Orlando López-Cruz, and Esteban Gómez Soto. Teaching computer programming: Practices, difficulties and opportunities. In *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–9, 2015. doi: 10.1109/FIE.2015.7344184.

[163] D. Royce Sadler. Formative assessment and the design of instructional systems. *Instructional Science*, 18(2):119–144, jun 1989. doi: 10.1007/bf00117714.

[164] Kay Sambell, Liz McDowell, and Catherine Montgomery. *Assessment for Learning in Higher Education*. Routledge, 2013. ISBN 9780415586580. doi: 10.4324/9780203818268.

[165] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 318–329, 2004. doi: 10.1145/964001.964028.

[166] Saul. Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, June 2003. doi: 10.1145/872757.872770. See https://theory.standford.edu/~aiken/moss/.

[167] Peter H Schmitt and Benjamin Weiß. Inferring invariants by symbolic execution. In Bernhard Beckert, editor, *Proceedings of 4th International Verification Workshop in connection with CADE-21*, volume 259, pages 195–210, 2007. URL http://ceur-ws.org/Vol-259/paper16.pdf.

[168] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 308–311. IEEE, 2017. doi: 10.1109/ICPC.2017.9.

[169] Michael L. Scott. *Programming language pragmatics*. Morgan Kaufmann Pub, San Francisco, CA, 2006. ISBN 9780126339512.

[170] Sebastian Serth, Thomas Staubitz, Ralf Teusner, and Christoph Meinel. CodeOcean and CodeHarbor: Auto-grader and code repository. In *Seventh SPLICE Workshop at SIGCSE 2021 "CS Education Infrastructurefor All III: From Ideas to Practice", SPLICE'21,*, 2021. URL `https://cssplice.github.io/SIGCSE21/proc/SPLICE2021_SIGCSE_paper_13.pdf`.

[171] Service Universitaire de Protection et d'Hygiène du Travail. Consignes d'urgence – Institut Montefiore B28. Emergency instructions posted in B28 buillding, 2017. Address: Quartier Polytech zone 1, Allée de la découverte, 4000 Liège.

[172] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, 2013. doi: 10.1145/2491956.2462195.

[173] Fujian Song, Sheetal Parekh, Lee Hooper, Yoon K. Loke, Jon Ryder, Alex J. Sutton, Caroline Hing, Chun Shing Kwok, Chun Pang, and Ian Harvey. Dissemination and publication of research findings: an updated review of related biases. *Health Technology Assessment*, 14 (8), feb 2010. doi: 10.3310/hta14080.

[174] Sheryl A. Sorby. The case for spatial skills instruction in computer science. In *Computing Education Summit White Paper*, 2013. URL `https://stacks.stanford.edu/file/druid:mn485tg1952/SorbySherylOhioState.pdf`.

[175] Gina Sprint and Diane Cook. Enhancing the cs1 student experience with gamification. In *2015 IEEE Integrated STEM Education Conference*, pages 94–99. IEEE, 2015. doi: 10.1109/ISECon.2015.7119953.

[176] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection. For GCC Version 10.3.0*. GNU Press, 2021. URL `https://gcc.gnu.org/onlinedocs/gcc-10.3.0/gcc.pdf`.

[177] David Svoboda and Wiki Contributors. MSC21-C. Use robust loop termination conditions. SEI CERT C Coding Standard wiki, 2009. URL `https://wiki.sei.cmu.edu/confluence/display/c/MSC21-C.+Use+robust+loop+termination+conditions`.

[178] John Sweller. *Instructional Design in Technical Areas*. Australian Council for Educational Research, April 1999. ISBN 0864313128.

[179] Wing C. Tam. Teaching loop invariants to beginners by examples. In *Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, pages 92–96, 1992. doi: 10.1145/134510.134530.

[180] Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces. ISO/IEC 9899:2018 programming languages — C. Technical report, ©ISO/IEC, 2018. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf. Consulted from diff marks in ISO/IEC 9899:202x draft.

[181] The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM and the IEEE Computer Society, jan 2013. doi: 10.1145/2534860.

[182] Vincent Tinto. Taking retention seriously: Rethinking the first year of college. *NACADA Journal*, 19(2):5–9, Fall 1999. doi: 10.12930/0271-9517-19.2.5.

[183] Sasha Trubetskoy. List of 20 simple, distinct colors. Web page, 2017. URL https://sashamaps.net/docs/resources/20-colors/. [Online; accessed: 26 March 2021].

[184] Göktürk Üçoluk and Sinan Kalkan. *Introduction to Programming Concepts with Case Studies in Python*. Springer-Verlag, October 2012. ISBN 9783709113431. doi: 10.1007/978-3-7091-1343-1.

[185] Dominique Verpoorten, Emmanuelle Parlascino, Marine André, Patricia Schillings, Julie Devyver, Olivier Borsu, Jean-François Van de Poël, and Françoise Jérôme. Blended learning-pedagogical success factors and development methodology. Report, IFRES-Université de Liège, 2017. URL http://hdl.handle.net/2268/209645.

[186] Rolland Viau. *La motivation en contexte scolaire*. de boeck, Bruxelles, fifth edition, May 2009. ISBN 9782804111489.

[187] Christopher Watson and Frederick W. Li. Failure rates in introductory programming revisited. In *Proc. Conference on Innovation & Technology in Computer Science Education (ITiCSE)*, pages 39–44, June 2014. doi: 10.1145/2591708.2591749.

[188] Howard White, Shagun Sabarwal, and Thomas de Hoop. Randomized controlled trials (RCTs). In *Methodological Briefs: Impact Evaluation*, number 7. UNICEF Office of Research, Florence, 2014. URL https://www.unicef-irc.org/publications/pdf/brief_7_randomized_controlled_trials_eng.pdf.

[189] Dylan Wiliam. What is assessment for learning? *Studies in Educational Evaluation*, 37(1): 3–14, March 2011. doi: 10.1016/j.stueduc.2011.03.001.

[190] World Health Organization. WHO Director-General's opening remarks at the media briefing on COVID-19 – 11 March 2020. WHO Website [Online; accessed: 27 July 2021], March 2020. URL https://www.who.int/director-general/speeches/detail/who-director-general-s-opening-remarks-at-the-media-briefing-on-covid-19---11-march-2020.

[191] Burkhard C. Wünsche, Zhen Chen, Lindsay Shaw, Thomas Suselo, Kai-Cheung Leung, Davis Dimalen, Wannes van der Mark, Andrew Luxton-Reilly, and Richard Lobb. Automatic assessment of OpenGL computer graphics assignments. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, jul 2018. doi: 10.1145/3197091.3197112.

[192] David Yevick. *A first course in computational physics and object-oriented programming with C*. Cambridge University Press, Cambridge New York, 2005. ISBN 0521827787.

[193] Juriy Zaytsev, Stefan Kienzle, and Andrea Bogazzi. Fabric.js, a powerful and simple javascript html5 canvas library, 2008. URL `http://fabricjs.com`. [Online; accessed: 24 March 2021].