



UNIVERSIDADE
LUSÓFONA

CENTRO UNIVERSITÁRIO DE LISBOA

ESCOLA DE COMUNICAÇÃO, ARQUITECTURA, ARTES E TECNOLOGIAS DA INFORMAÇÃO

DOUTORAMENTO EM INFORMÁTICA - NOVOS MEDIA E SISTEMAS UBÍQUOS

Re-Orienting Object-Oriented Programming Education: Automatic Assessment Tools and Large Language Models

Thesis presented in public examinations for the attainment of the degree of
Doctor in Informatics - New Media and Ubiquitous Systems, supervised by
Professor Dr. Pedro Hugo De Queirós Alves.

Bruno Miguel Pereira Cipriano
Nº a21908727

2025

www.ulusofona.pt



UNIVERSIDADE
LUSÓFONA

CENTRO UNIVERSITÁRIO DE LISBOA

ESCOLA DE COMUNICAÇÃO, ARQUITECTURA, ARTES E TECNOLOGIAS DA INFORMAÇÃO

DOUTORAMENTO EM INFORMÁTICA - NOVOS MEDIA E SISTEMAS UBÍQUOS

Re-Orienting Object-Oriented Programming Education: Automatic Assessment Tools and Large Language Models

"FINAL VERSION"

Thesis defended in public examinations at Universidade Lusófona, Centro Universitário de Lisboa, on January, 30th, 2025, before the jury appointed by Appointment Order No. JÚRI Nº 1073 / 2024, dated November 29, 2024, with the following composition:

President:

Prof. Doutor Luís Miguel Serra da Costa Campos (Associate Professor, Director of the PhD program in Informatics, Universidade Lusófona - Centro Universitário de Lisboa), by delegation of the Rector;

Members:

Prof. Doctor Michel Wermelinger - Associate Professor, Open University (UK) (Opponent);

Prof. Doctor Nuno Gonçalo Coelho Costa Pombo - Associate Professor, Universidade da Beira Interior (Opponent);

Prof. Doctor André Leal Santos - Auxiliar Professor, ISCTE - Instituto Universitário de Lisboa;

Prof. Doctor João Pedro Carvalho - Researcher, Universidade Lusófona – Centro Universitário de Lisboa;

Prof. Doctor Lúcio Studer Ferreira - Auxiliar Professor, Universidade Lusófona – Centro Universitário de Lisboa;

Prof. Doctor Maria Pinto Albuquerque - Auxiliar Professor, ISCTE - Instituto Universitário de Lisboa;

Prof. Doctor Pedro Arroz Correia Bonifácio Serra - Auxiliar Professor, Universidade Lusófona – Centro Universitário de Lisboa);

Supervisor:

Prof. Doctor Pedro Hugo Queirós Alves - Professor Associado, Universidade Lusófona – Centro Universitário de Lisboa.

Bruno Miguel Pereira Cipriano

Nr. a21908727

January 2025

Acknowledgements

I would like to sincerely thank...

My mother, Ana, and my father, Adelino without whom I would not be where I am, both academically and personally.

My uncle, Carlos, who gave me my first computer: his old IBM PS/1. That machine opened the door to the world of technology and computing, sparking a hobby that eventually evolved into a full-fledged career as a software developer and computer science educator.

Pedro Malta, who hired me as a teaching assistant back in 2011. Without that first opportunity, this thesis might have never been produced.

Pedro Alves, my supervisor. Pedro invited me to join the PhD program and guided my work with a lot of pragmatism – something I consider essential to the successful conclusion of this project. Pedro supported my decision to dive into the ChatGPT research thread (or rabbit hole) in 2022 — a crucial moment that led to this submission. My collaboration with Pedro started well before the PhD program: we’ve been working together since 2015, having revamped two computer science courses, Data Structures and Algorithms and Programming Languages II. Despite our frequent disagreements, this partnership has allowed me to learn and grow in multiple ways.

My collaborators, Nuno Fachada, Rodrigo Correia, and Paul Denny. Nuno was instrumental in kickstarting the productive part of my PhD with his support on my first paper and remained a consistent source of information. Rodrigo, for proofreading my presentations. And Paul, for the numerous lessons and insights that have enriched my work.

Melinda, for her patience through the many weekends lost to research that could have been spent at the beach together, and for her help in creating some of the graphical artifacts that enhanced my work.

To my grandmother, Maria Alice Pereira, who always kept moving.

To the rest of my family and friends—including the four-legged and the feathered ones—whose companionship and love are always with me, even if not all of you are named here.

Finally, to ChatGPT, for opening the very interesting research avenue that I was able to navigate, as well as for its help in shortening long sentences.

Thank you all for your part in this journey.

Abstract

Object-Oriented Programming (OOP) is a programming paradigm in which systems are created using objects representing instances of important entities in the software’s domain. These objects have mutable states. Furthermore, they communicate with each-other, affecting those states. OOP was originally created to help reduce software development cost, through the application of best practices that result in more modular code that is easier to change and adapt. Teaching (and learning) OOP has its own complexities: in OOP, merely implementing something that works is not sufficient: it is important to teach and promote the aforementioned best practices.

Automatic Assessment Tools are tools used to automatically evaluate students’ code. These tools are commonly used in Computer Science (CS) courses in order to improve student autonomy (through formative and constructive feedback) and reduce teacher workload. Such tools have existed since 1960, but certain improvements are still possible, in particular when using AATs to validate typical OOP assignments, which a) typically involve multiple classes/objects, b) are focused not only on functional validation, but also on best practices and code quality, and c) pose specific challenges when constructing feedback.

The advent of Large Language Models (LLMs) has created multiple challenges for the Computer Science Education (CSE) community. Concerns regarding students’ potential misuse of these tools have been the cause of discussions between teachers, with some of them opting for banning or restricting the usage of these tools, while others try to integrate them in their courses. Although extensive research already exists regarding the use of LLMs for introductory programming courses, a knowledge gap exists in terms of these tools’ impact on more advanced courses such as OOP. More concretely, not much is known about these tools’ capacity to generate OO-code and what can be done to responsibly integrate them in OO courses.

The development of this project contributed to the dual research goals of improving AATs (and their usage) for OOP education and addressing the knowledge gap regarding LLMs’ impact on OOP education. Furthermore, we have proposed a new visual notation for prescribing OO exercises and counter LLM over-reliance, which was well received by our students who also reported other benefits, such as increased motivation to work on video-based projects.

Keywords: object-oriented programming, automated assessment tools, large language models

Resumo

A Programação Orientada por Objectos (OOP, do Inglês Object-Oriented Programming) é um paradigma de programação no qual os sistemas são criados usando objectos que representam instâncias de entidades importantes no domínio do software. Estes objectos têm estados mutáveis. Para além disso, comunicam uns com os outros, alterando esses estados. A OOP foi inicialmente criada para reduzir o custo do desenvolvimento de software, através da aplicação de um conjunto de práticas que levam a código mais modular e que é mais fácil de mudar e de adaptar. Ensinar (e aprender) OOP tem as suas próprias complexidades: em OOP, implementar código que funciona não é suficiente: é importante ensinar e promover as boas práticas previamente referidas.

As Ferramentas de Avaliação Automática (AATs, do Inglês Automatic Assessment Tools) são ferramentas usadas para avaliar automaticamente o código dos alunos. Estas ferramentas são frequentemente usadas em cursos de Ciências de Computação para melhorar a autonomia dos alunos (através de *feedback* formativo e/ou correctivo) e para reduzir a carga de trabalho necessário por parte dos professores. Estas ferramentas existem desde 1960, mas ainda existem oportunidades de melhoria, em particular quando os AATs são usados para validar tarefas típicas de OOP, que a) tipicamente envolvem várias classes/objectos, b) são focadas não só na correcção funcional, mas também nas boas práticas e qualidade do código, e c) apresentam desafios próprios na construção de feedback.

A chegada dos Modelos de Linguagem de Grande Escala (LLMs, do Inglês Large Language Models) criou múltiplos desafios para a comunidade de Educação em Ciências da Computação (CSE, do Inglês Computer Science Education). As preocupações sobre o possível uso indevido dessas ferramentas por parte dos alunos têm levado a discussões por parte de professores, com alguns deles a optar por proibir ou restringir o uso dessas ferramentas, enquanto outros tentam integrá-las nas suas disciplinas. Apesar de existir investigação extensa sobre o uso de LLMs em disciplinas de introdução à programação, existe uma lacuna de conhecimento em relação ao impacto destas ferramentas em disciplinas mais avançadas, como a OOP. Mais concretamente, pouco é conhecido sobre a capacidade destas ferramentas gerarem código orientado a objectos e sobre o que pode ser feito para as integrar de forma responsável em disciplinas de OOP.

O desenvolvimento deste projecto contribuiu para dois objectivos de investigação: melhorar os AATs (e o seu uso) no ensino da OOP e preencher a lacuna de conhecimento que existe em relação ao impacto dos LLMs no ensino da OOP. Para além disso, foi proposta

uma nova notação visual para apresentar exercícios OO e contrariar a sobre-dependência dos LLMs, que foi bem recebida pelos nossos alunos, que também reportaram outros benefícios, tais como um aumento da motivação ao trabalhar com projectos com enunciado apresentado em vídeo.

Palavras-chave: programação orientada por objectos, ferramentas de avaliação automática, modelos de linguagem de grande escala

Contents

Acknowledgements	2
Abstract	3
Resumo	4
List of Acronyms	9
List of Tables	11
List of Figures	12
List of Listings	14
1 Introduction	15
1.1 Research goals	17
1.2 Thesis scope	17
1.3 Contributions	17
1.4 List of Publications	19
1.5 Thesis Structure	21
2 Background	22
2.1 Teaching Object-Oriented Programming	22
2.1.1 OOP Concepts	22
2.1.2 Standard representations	24
2.1.3 Inheritance and Polymorphism: a Case Study	29
2.1.4 Code organization	34
2.1.5 Teaching and Learning OOP	34
2.2 Automatic Assessment Tools	35
2.2.1 Relevant tools	36
2.2.2 AAT cheating patterns	39
2.2.3 Feedback	40
2.3 Large Language Models for CSE	40
2.3.1 Foundations of LLMs	40

2.3.2	LLMs in Practice	44
2.3.3	Large Language Models' Impact on CSE	55
2.3.4	Challenges	56
2.3.5	Current Approaches	58
2.3.6	CS Students perception and acceptance of LLMs	61
3	Automatic Assessment Tools for Object-Oriented Programming	63
3.1	Chapter summary	63
3.2	Drop Project	64
3.2.1	Introduction	64
3.2.2	Software description	65
3.2.3	Illustrative Examples	70
3.2.4	Deployment	72
3.2.5	Impact	72
3.2.6	Conclusions	74
3.3	Project-Oriented and Exercise-Oriented AATs	74
3.3.1	Introduction	74
3.3.2	Software description	76
3.3.3	Research Methods	78
3.3.4	Results	80
3.3.5	Discussion and Limitations	82
3.3.6	Conclusions	83
3.3.7	Data Availability	84
3.4	Lessons Learned in Automated Assessment	84
3.4.1	Introduction	84
3.4.2	Definitions	86
3.4.3	Lessons learned: generic recommendations	88
3.4.4	Lessons learned: specific recommendations per assessment type	96
3.4.5	Lessons learned: other recommendations	97
3.4.6	The Future of Automatic Assessment	97
3.4.7	Limitations	98
3.4.8	Conclusions	98
4	Large Language Models and Object-Oriented Programming	99
4.1	Chapter summary	99
4.2	Evaluating LLMs on OOP assignments	100
4.2.1	Introduction	100
4.2.2	The assignments	101
4.2.3	Methodology	103
4.2.4	It works but it's not OO - how LLM(s) tried to solve the "IT Com- pany" assignment	103

4.2.5	Overall Findings	107
4.2.6	Recommendations for CS Educators	109
4.2.7	On the importance of citing your sources	110
4.2.8	Limitations	111
4.2.9	Conclusions and Future Work	111
4.2.10	Data Availability	112
4.3	Diagram and Video-Based Exercises	112
4.3.1	Introduction	112
4.3.2	Notation: Diagrams	114
4.3.3	Notation: Videos	118
4.3.4	Method	118
4.3.5	Evaluation	119
4.3.6	Results	119
4.3.7	Discussion	122
4.3.8	Limitations	124
4.3.9	Conclusions	124
5	Conclusions and Future Work	125

List of Acronyms

AAT	Automatic Assessment Tool
API	Application Programming Interface
APPS	Automated Programming Progress Standard
BERT	Bidirectional Encoder Representations from Transformers
BLEU	Bilingual Evaluation Understudy
CLIP	Contrastive Language-Image Pre-Training
CoT	Chain-of-Thought
CER	Computing Education Research
CS	Computer Science
CSE	Computer Science Education
CSEDU	International Conference on Computer Supported Education
CSV	Comma-Separated Values
DP	Drop Project
EB	Exercise-based (learning)
ECTEL	European Conference on Technology-Enhanced Learning
GPT	Generative Pre-trained Transformer
ICSE	International Conference on Software Engineering
ICPEC	International Computer Programming Education Conference
IDE	Integrated Development Environment
ITiCSE	Annual Conference on Innovation and Technology in Computer Science Education
ISO	International Organisation for Standardization

JAR	Java ARchive
LaMDA	Language Model for Dialog Applications
LLaMA	Large Language Model Meta AI
LLM	Large Language Model
OO	Object-Oriented
OOP	Object-Oriented Programming
OMT	Object Modelling Technique
PB	Project-based (learning)
REST	Representational State Transfer
UML	Unified Modelling Language
VPL	Virtual Programming Lab
WAR	Web Application Resource / Web application ARchive

List of Tables

2.1	Summary and main characteristics of the discussed AATs	38
2.2	Gradient of Large Language Model releases	45
2.3	Requirements for Data, Code, and Model components of an Open-Source AI system	45
3.1	Survey results evaluating the students' experience with Drop Project	73
3.2	Project submission metrics for each course designed around DP.	73
3.3	Survey questions for students who adopted the plugin.	79
3.4	Open-ended questions posed to students in the specified target groups. . . .	80
3.5	Histogram and summary statistics for the questions posed to plugin adopters.	81
4.1	Evaluation of the 3 models' solutions for each assignment.	108
4.2	Survey questions categorized by Research Question.	119
4.3	Results of the 'vision' experiments	124

List of Figures

1.1	This thesis falls in the intersection between OOP Education, Automatic Assessment Tools, and Large Language Models.	18
2.1	UML representation of an elementary Person class	25
2.2	UML representation of a composition relationship	25
2.3	UML representation of an inheritance relationship	26
2.4	UML representation of design with Inheritance and Composition relationships	27
2.5	UML representation of a design which takes advantage of Inheritance to obtain polymorphism	28
2.6	UML representation of a possible, although non-ideal, solution to the Geometric Figures area calculation problem	30
2.7	UML representation of a better solution to the Geometric Figures area calculation problem	32
2.8	Timeline of Milestones in NLP for LLMs	43
3.1	Assignment folder structure.	66
3.2	DP's high level architecture	68
3.3	DP's evaluation pipeline.	69
3.4	Several example reports from DP	71
3.5	Overview of the solution's data flow	76
3.6	The plugin integrated into IntelliJ IDEA's main window.	77
3.7	The assignment instructions inform students of the tasks they need to perform.	77
3.8	The submission report informs students of their progress and presents them with any unit test failures.	78
3.9	Hierarchical breakdown of student participation and plugin adoption based on the survey responses.	80

4.1	Simplified UML class diagram for the “IT Company” assignment, excluding non-essential attributes, methods, and constructors. Students only receive textual instructions, not this diagram.	104
4.2	Example algorithmic exercise.	114
4.3	Example state-change function exercise.	116
4.4	Class declaration exercises.	117
4.5	Example inheritance exercise.	117
4.6	State transition rules.	118
4.7	Students’ opinions with regards to diagram-based exercises when compared to natural language exercises (RQ1)	120
4.8	Students’ opinions about each of the diagram types (RQ2)	121
4.9	Distribution of student replies to questions about video-based exercises (RQ3)	121
4.10	Students’ opinions on diagram/video exercises’ effect on LLM usage (RQ4)	122

List of Listings

2.1	An example implementation of the object oriented mechanisms which allow polymorphism.	29
2.2	An example of bad OO code	31
2.3	An example of good OO code	33
3.1	An example <i>JUnit</i> test for a project where students implement a board game.	72
3.2	An example <i>JUnit</i> test for an exercise where students must implement a function which sums two integers.	85
3.3	An example <i>JUnit</i> test for an OOP project where students must implement a board game.	85
3.4	Test function which validates a boolean function using a single assertion. . .	89
3.5	In certain cases, just referring the function name is insufficient and might cause confusion.	90
3.6	A test with more actionable feedback than the one in Listing 3.5.	90
3.7	Overfitted solution that passes the test defined in Listing 3.2	91
3.8	Test function with a progressive reduction of feedback level.	92
3.9	Example of the assert-early and often strategy.	94
3.10	Gracefully handling potential null references to provide useful feedback. . .	94
3.11	Progressive disclosure: this test is only executed after the student passes at least 4 simple tests.	95
3.12	Testing collections' contents: its better to validate a few examples before asserting the full list size.	95
4.1	Partial instructions for the "IT Company" assignment	102
4.2	A function generated by Bard for the "Condominium Mgt." assignment that fails our Code Quality validations.	108

Chapter 1

Introduction

Object-Oriented Programming (OOP) is a programming paradigm in which systems are created using objects representing instances of important entities in the software's domain. These objects usually aggregate multiple data and have mutable states. They also have specific behaviours which represent relevant actions in the system. Furthermore, they communicate with each-other, affecting those states.

Imagine, for example, a race-car computer game in which a human player drives a car around a track, trying to execute the fastest possible lap time. Such a software might contain objects to represent Cars, Engines, Wheels, the Race Track, among others. A Car will have a number of attributes (e.g., brand or maker) and will also be composed of other objects, such as the Engine, its four Wheels, and so on. Now, consider that the game starts and the player presses the key to accelerate its Car: this action will change the car itself, since the acceleration will impact the Car's Engine, which, in turn, will impact the Car's Wheels. Finally, the Race Track itself will be impacted, since the Car's position will change.

OOP was originally created to help reduce software development cost, through the application of best practices that result in more modular code that is easier to change and adapt. Teaching (and learning) OOP has its own complexities: in OOP, merely implementing something that works is not sufficient: it is important to teach and promote the aforementioned best practices. Also, typical OOP exercises involve multiple code files, which raises other challenges which will be discussed later.

Automatic Assessment Tools (AATs) have been used in Computer Science courses since the 1960s [59]. These tools validate students' code by using strategies such as unit-testing or output matching [86]. Some benefits of using AATs are the increase of students autonomy [66] and the reduction of teacher workload [139]. On the negative side, some students tend to over-rely on the AAT, neglecting proper analysis and delegating all their testing efforts to the assessment tool [86].

Research related with AATs is varied, with many studies focusing on the tools, their learning impact, advantages, disadvantages and limitations. Strategies to mitigate the disadvantages have also been proposed (e.g., the use of time penalties to prevent, or at

least reduce, student over-reliance).

Multiple such tools can be found in the literature. Some of them are based on virtual environments, implemented to abstract students from certain complexities of programming (e.g., abstract away the need to manually invoke the compiler), while others try to promote that students use real-world software development tools (e.g., version control systems such as Git).

Unfortunately, not all of these tools can easily be experimented and tested by educators wanting to use an AAT. This happens because many of them were developed for a single course or to run an experiment, not having been developed to the point of distribution. In some cases, the issue is the lack of (updated) binaries, while in other cases, parts of the source code are missing. Finally, the lack of documentation is also a problem with some tools.

Furthermore, as we have discussed before, OOP has certain requirements (e.g., support for multiple code files) which not all available tools support. Also, certain validation strategies, such as output matching, are not ideal for OOP, where merely ensuring that the code works is not sufficient.

The quality of the feedback given by AATs is also particularly relevant for OO education. The interactions between objects have the potential to create certain bugs which are hard to debug without proper feedback. This must be taken in consideration when creating test cases for supporting OO assignments and when defining the feedback given by those test cases.

As discussed, AATs are technological solutions with a notable role in CSE, designed specifically for tasks such as assessment and feedback. However, new scientific developments have brought us a different kind of technology: Large Language Models (LLMs), which, while not originally built for education, are increasingly being applied in educational contexts and influencing the way we teach and learn.

LLMs are Artificial Intelligence (AI) tools trained on very large amounts of data to understand and generate human-like text, enabling diverse applications in natural language processing tasks.

The increasing capacities of LLMs in terms of code generation has placed the CSE community on alert, spawning discussions on how to react and adapt the current teaching practices to these models, since students are likely to use and abuse these tools, possibly using ‘copy-and-prompting’¹ to pass their home-works, tests and other assignments, which results in the risk of producing graduates who are weak in fundamental computing skills such as computational thinking and program design [104]. Some educators choose to fight these models – by blocking access, deploying AI-generated content detection tools and other similar mechanisms –, while others decide to embrace them and adapt the respective courses in order to prepare students for a professional future where they are expected to interact with these and other similar tools [104].

Although the CSE community has faced technological challenges in the past, such as

¹copy-and-prompting: copying an assignment into an LLM’s input area in order to get a solution

the calculator, the Internet, Google and StackOverflow, LLMs pose a different and specific kind of challenge, since, unlike those other tools, they are not a helper, but an automation instead. This poses specific challenges which must be addressed when bringing in and accepting LLMs in the classroom.

Even though extensive research has been conducted on LLMs' capacity to handle introductory programming assignments, which are typically based on the implementation of single functions that receive an input and transform it into an output, not much is known about these models' capacities to handle OOP assignments consisting of the implementation of multiple classes which must interact with each-other.

1.1 Research goals

This research project has two main goals: 1) improve AATs and/or their usage in CSE, taking OOP's particularities in consideration, and 2) help fill the knowledge gap which exists in regards to LLMs and OOP, namely in terms of informing the CSE community of the nuances of LLM-generated OOP code, and in finding ways of responsibly integrating these models in OOP courses.

1.2 Thesis scope

This work falls under the field of computer science education (CSE), also known as computing education research (CER). This field focuses on how computer science is taught and learned. It seeks to understand how students grasp CS concepts, while also investigating and developing innovative pedagogical methods to enhance teaching practices. Additionally, it aims to improve curriculum design and course content to better support student learning and engagement.

More concretely, this thesis focuses on the Teaching of Object-Oriented Programming, on the implementation and usage of Automatic Assessment Tools, and on the integration of Large Language Models in CS courses.

Figure 1.1 visually explains the intersection between the three relevant sub-topics.

1.3 Contributions

This work makes the following contributions:

1. Presents an Open-Source AAT apt for supporting students working on OOP assignments and assisting teachers in evaluating those assignments (i.e.,an unit-tests-based AAT with support for multiple-file submissions and code quality validations), as well as a student evaluation of the tool;
2. Proposes an extension of the aforementioned AAT aimed at improving students' workflows when dealing with small assignments;

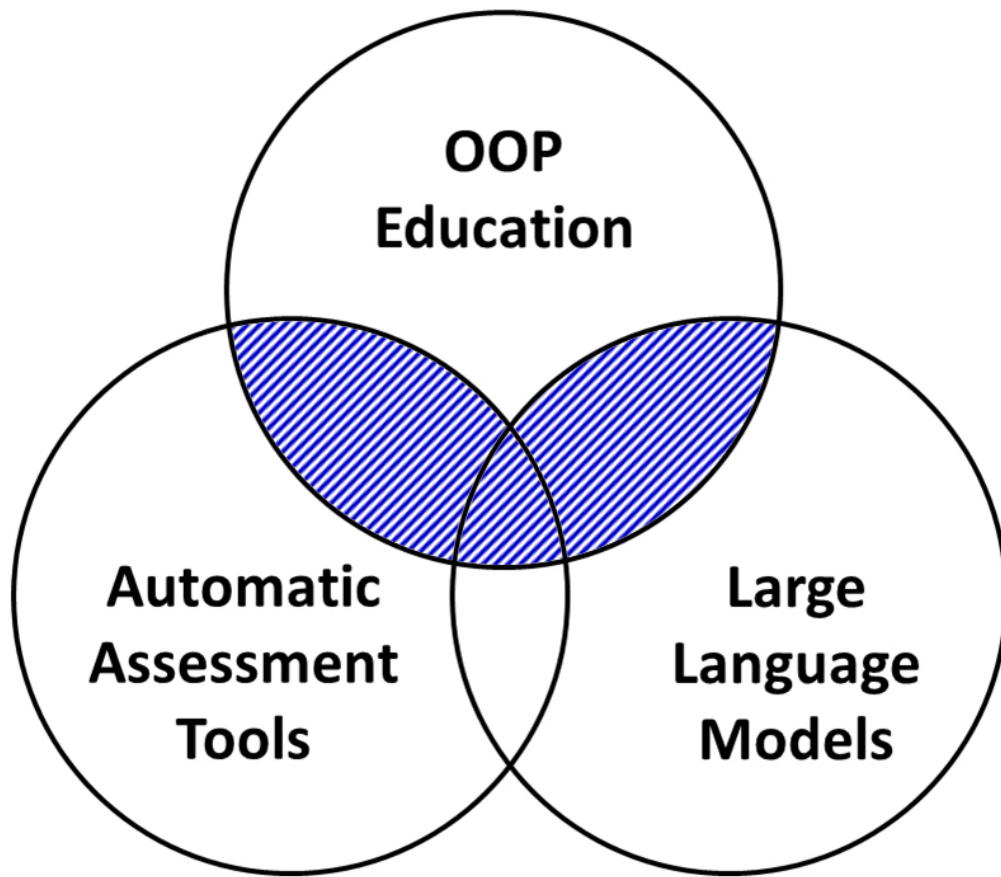


Figure 1.1 – This thesis falls in the intersection between OOP Education, Automatic Assessment Tools, and Large Language Models.

3. Presents a list of recommendations for defining unit tests for automatic assessment, aimed at enhancing student autonomy and grading fairness in multiple-types of assignments (i.e., small exercises as well as projects) in and different types of courses, including OOP;
4. Provides an overview of LLMs, their evolution, and their impact on the educational area, with a particular focus on Computer Science Education;
5. Evaluates the capacities of current state-of-the-art Large Language Models to solve OOP assignments;
6. Discusses the current limitations of said models in terms of being used to support teaching and learning of OOP;
7. Proposes a new pedagogical approach for teaching OOP, based on diagrams and videos designed for this paradigm's particularities, and presents an evaluation of its effectiveness.

1.4 List of Publications

This thesis comprises material previously published on two journal papers and four conference papers. An extended abstract (and respective poster), as well as a demo paper, were also presented at conferences. Two other papers dedicated to the intersection between CSE and LLMs were also developed, one as a first author and the other as a second author, but they were not included in the thesis document due to not being focused on OOP. Finally, a non-published paper was developed as a second author.

The list of publications and their contribution to this thesis is presented below.

Papers:

1. **Cipriano, B. P.**, Fachada, N., & Alves, P. (2022). Drop Project: An automatic assessment tool for programming assignments. *SoftwareX*, 18, 101079.
SoftwareX is a Q2 journal dedicated to publishing Open-Source educational and research software.
This paper contributes to chapter 3 of this thesis.
2. **Cipriano, B. P.**, Baltazar, B., Fachada, N., Vourvopoulos, A., & Alves, P. (2024) Bridging the Gap between Project-Oriented and Exercise-Oriented Automatic Assessment Tools. *Computers* 2024, 13, 162.
This paper was published in the special issue ‘Future Trends in Computer Programming Education’ of the ‘Computers’ journal. ‘Computers’ is dedicated to the publication of computer science papers, focusing particularly on Computer and Network Architecture (Q2 in 2023), as well as in Human-Computer Interaction (Q3 in 2023).
This paper contributes to chapter 3 of this thesis.
3. **Cipriano, B. P.**, & Alves, P. (2024, June). Seven Years Later: Lessons Learned in Automated Assessment. In *Proceedings of the 5th International Computer Programming Education Conference* (pp. 3:1-3:14). Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
ICPEC is an international conference dedicated to Computer Programming Education.
This paper contributes to chapter 3 of this thesis.
4. **Cipriano, B. P.**, & Alves, P. (2023, June). GPT-3 vs Object Oriented Programming Assignments: An Experience Report. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (pp. 61-67). ITiCSE is a B-level conference dedicated to Computer Science Education.
This paper contributes to chapter 4 of this thesis.
5. **Cipriano, B. P.**, & Alves, P. (2024, April). LLMs Still Can’t Avoid InstanceOf: An Investigation Into GPT-3.5, GPT-4 and Bard’s Capacity to Handle Object-Oriented Programming Assignments. In *Proceedings of the 46th International Conference on*

Software Engineering: Software Engineering Education and Training (pp 162-169). ICSE is an A*-level conference dedicated to Software Engineering. This paper was featured on the Software Engineering Education Track.

This paper contributes to chapter 4 of this thesis.

6. **Cipriano, B. P.**, Alves, P., & Denny, P. (2024, September). A Picture Is Worth a Thousand Words: Exploring Diagram and Video-Based OOP Exercises to Counter LLM Over-Reliance. In *Proceedings of the 19th European Conference on Technology Enhanced Learning* (pp. 75-89). Springer Nature Switzerland.

EC-TEL is a B-level conference dedicated to the presentation of topics in the field of Technology-Enhanced Learning.

This paper contributes to chapter 4 of this thesis.

7. **Cipriano, B. P.**, & Alves, P. (2025). "ChatGPT Is Here to Help, Not to Replace Anybody": An Evaluation of Students' Opinions On Integrating ChatGPT In CS Courses. *To appear in the Proceedings of 17th International Conference on Computer Supported Education, 2025.*

A paper presenting the results of a student survey evaluating students' opinions and perceptions with regards to the use of tools such as ChatGPT in CS courses. The paper will be presented in CSEDU 2025 (B-level).

8. Alves, P. & **Cipriano, B. P.** (2025). "Give Me the Code": Log Analysis of First-Year CS Students' Interactions with GPT. *To appear in the Proceedings of 17th International Conference on Computer Supported Education, 2025.*

An analysis of how students interacted with ChatGPT to implement a requirement in the scope of a Data-Structures and Algorithms project. The paper will be presented in CSEDU 2025 (B-level).

Other:

- **Cipriano, B. P.** (2024, July). Towards the Integration of Large Language Models in an Object-Oriented Programming Course. In *Proceedings of the 2024 Conference on Innovation and Technology in Computer Science Education V. 2* (pp. 832-833). This publication consisted of an extended abstract and a poster highlighting the main aspects of the LLM-component of this thesis (i.e., **Chapter 4**). It was featured in ITiCSE's 2024 Doctoral Consortium and also presented at the main conference (B-level).
- **Cipriano, B. P.**, Silva, M., Correia, R. & Alves, P. (November, 2024). Towards the Integration of Large Language Models and Automatic Assessment Tools: Enhancing Student Support in Programming Assignments. In: *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*. This publication consisted of a demo paper and a demonstration of an early stage

prototype which integrates an AAT with a LLM in order to help student get LLM-generated tips for errors and issues reported by the AAT. The presented tool was built on top of the Plugin described in section Project-Oriented and Exercise-Oriented AATs (3.3) of **Chapter 3** of this thesis. The demo was featured in the ‘posters & demos’ track of the Koli Calling Conference 2024 (B-level).

- Alves, P., & **Cipriano, B. P.** (2023). The centaur programmer – How Kasparov’s Advanced Chess spans over to the software development of the future. arXiv preprint arXiv:2304.11172. <https://arxiv.org/pdf/2304.11172>

A position paper exploring potential Human/AI collaboration models for software development. This paper was motivated by LLMs’ capacity to generate computer code.

Conference rankings obtained from ICORE².

Journal rankings obtained from Scimago Journal and Country Ranking³.

1.5 Thesis Structure

This thesis is structured as follows:

Chapter 2 - Background gives an overview of relevant literature on the three relevant topics: a) Teaching and Learning of Object Oriented Programming, b) Automatic Assessment Tools, and c) Large Language Models.

Chapter 3 - Automatic Assessment Tools for Object-Oriented Programming address the problem of creating an AAT with the necessary characteristics for teaching and learning Object-Oriented Programming. Furthermore, it presents guidelines for defining unit-tests and the respective feedback for supporting students working in multiple types of assignments and courses, including OOP.

Chapter 4 - Large Language Models and Object-Oriented Programming addresses the impact of LLMs in Object-Oriented Programming courses by presenting an evaluation of the capacity of 3 such models to generate code to solve real-world OOP assignments. Furthermore, this chapter also introduces a new diagrams-and-video-based approach for prescribing OOP exercises, which aims at reducing LLM over-reliance.

Chapter 5 - Conclusions and Future Work summarizes the work presented in this thesis, and proposes some avenues for future research.

²<https://portal.core.edu.au/conf-ranks/>

³<https://www.scimagojr.com/>

Chapter 2

Background

2.1 Teaching Object-Oriented Programming

Object Oriented Programming was developed in order to help reduce the cost of maintaining the software [196], by introducing some practices that lead to more modular code that is also easier to change and adapt, which is relevant since most software is not static and tends to evolve with newer features and bug fixes [196, 130].

In OOP, systems are created using objects which represent instances of important entities in the software’s domain [196]. For example, imagine a software whose goal is supporting architects in their workflow. In such software, objects will probably exist to represent entities such as windows, doors and walls. These objects will have properties (such as height, width and weight), but they will also have behaviours. For example, some windows can be open simply by sliding, while others require manipulation of a lever. These behavioural differences within the same object sub-type are supported in OOP by techniques such as inheritance and polymorphism. Furthermore, these objects will interact with each-other: the windows and doors will be integrated within walls and walls which have windows will be displayed differently. This leads to further complexity when designing and implement such systems.

Furthermore, although OOP is the common name of this paradigm, the ‘programming’ part is not the most important one: analysis and design are important tenets of OOP, allowing the developer to identify and define which entities exist in the system, how they communicate with each other and what is the result of that communication.

To quote Sandi Metz¹, “Object-oriented design requires that you shift from thinking of the world as a collection of predefined procedures to modelling the world as a series of messages that pass between objects.” [129].

2.1.1 OOP Concepts

The following are some of the most important concepts in OOP.

¹Author of Practical Object-Oriented Design in Ruby

Information hiding or Abstraction A programming concept where the implementation details of certain software components are concealed, allowing programmers to focus only on the aspects which are relevant at each time. Although this concept is not exclusive to OOP, it is central to this paradigm.

Object Objects encapsulate (or conceal) data and implementation.

Class A template for the creation of objects. For example, in Java, if we wanted to represent information names and dates of birth of a group of people, we could create a **Person** class containing those two attributes. We would then use this class to create (or instantiate) **Person** objects.

Method A function associated with a class. In some OO languages, a method is a function that belongs to a class and has access to its attributes. Methods implement the class's behaviours. Throughout this thesis, the words method and function are used interchangeably.

Message sending objects talk with each-others by sending messages (i.e., by calling methods). Message passing promotes decoupling of objects, since each object does not need to know how the other object will handle the request.

Composition A mechanism which allows a class to be composed of objects belonging to another class. Also known as an 'Has A' relationship. Example: *Door has a Handle*.

Inheritance A mechanism which allows a class (usually known as subclass, derived class or specialization) to inherit attributes and behaviours (methods) from another class (usually called superclass, base class or generalization). This allows the subclass to reuse code and enhance functionality without changing the superclass. Also known as an 'Is A' relationship. Examples: *Programmer is a Profession* and *Duck is an Animal*.

Polymorphism from the Greek: *poly* (many) and *morphē* (form). A concept that allows objects of different types to be treated as instances of the same class through a shared interface, thus promoting code flexibility and re-usability.

Abstract class A class which cannot be instantiated. In other words, the compiler will not allow the creation of objects of this class. These classes serve to define common attributes and behaviours which are then inherited by their subclasses.

Abstract method A method which does not have implementation. Such methods can only exist in abstract classes. Abstract methods serve the purpose of defining mandatory method protocols that the sub-classes must implement.

Dynamic dispatch The process of selecting which implementation of a (polymorphic) method/function should be called in runtime, considering the type(s) of the method's arguments.

2.1.2 Standard representations

Attempts to create standardized ways of describing and documenting OO-systems started in the early 1990s, with the appearance of methodologies such as the Booch Method (initially developed by Grady Booch at Rational Software) and the Object Modelling Technique (OMT, developed by Rumbaugh, et al.). Both those methodologies defined specific notations for describing classes, their relations, as well as more dynamic components such as state-transition diagrams.

In circa 1995, Rational Software proposed the Unified Modelling Language (UML), which features graphical elements from both Booch Method and OMT, and was also complemented with a notation to describe certain dynamic aspects of software behaviour: use cases, proposed by Jacobson in 1987.

In 2005, UML became an ISO standard, a status which it maintains to this date.

UML contains a wide range of diagram types to describe OO-systems:

- Class diagrams: diagrams which represent the classes that exist in a system, their attributes, methods (or functions) and relations with other classes (e.g., composition, inheritance);
- Activity diagrams: visual representations of the workflow of an activity (e.g., process) of the system. Used for modelling the dynamic aspects of the system.
- Sequence diagrams: representation of how objects interact in a particular scenario, focused in the order of the messages passed between those objects. Valuable for understanding the detailed functioning of the system.
- State transition diagrams: visual representation of the various states that an object goes through in response to events.
- Object diagrams: allow the representation of how a certain set of instantiated objects relate to each other in the computer's memory.
- Use Case diagrams: diagrams used to represent interactions between users (or actors) and the specific parts of the system.

Examples

Class diagram - single class representation

Figure 2.1 presents an example of how a single class would be represented using UML class diagrams. The class is represented in UML by a rectangle which is divided in three sections: the first section displays the class name, the second section displays the class' attributes (i.e., data) and the third section the classes methods (i.e., behaviours).

In this example, a **Person** class is described. This class is characterized by 3 attributes: the Person's name, phone number and email address. Note that the data-types are also present: the Person's name is a String, the phone number is an Integer and the email is a

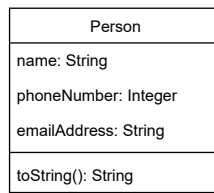


Figure 2.1 – UML representation of an elementary Person class. This class has 3 attributes and 1 method.

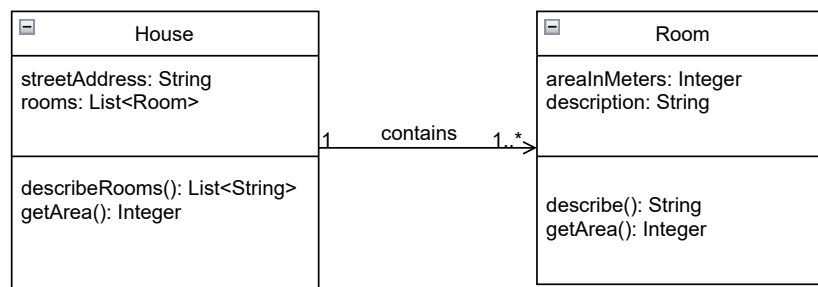


Figure 2.2 – UML representation of a composition relationship. In this case, a House is composed of Rooms.

String. Finally, this class has a single method: the `toString()` method, which is usually used to obtain a String representation of the object.

Composition relationship

Figure 2.2 presents an example of how to present a composition relationship in UML. In a composition relationship, one of the classes is the container, while the other is the contained (or part).

In this case, the **House** class is the container, while the **Room** class is the contained. The diagram also informs us of how many objects can participate in the relationship. In this example, the label "1..*" placed to the left of the **Room** class informs the reader that the House will have at least one Room. On the other hand, the label "1" placed to the right of the **House** class indicates that the Room will belong to exactly one House. This information is the relationship's multiplicity. If we look at the behaviours, we can see that the **House** class has a `describeRooms()` method which returns a List of Strings describing the house. Comparatively, the **Room** class contains a `describe()` method, which will return a String describing each specific room. The house object can ask each of its rooms for the information (String) and then combine all the information in a list. Similar logic will exist for the `getArea()` function: when called with a Room object, it will return the area of that room; When called with a House object, it will return the area of the whole house.

Inheritance relationship

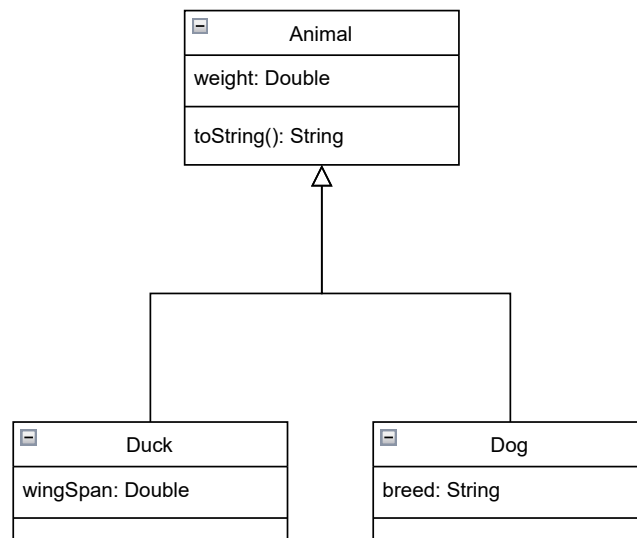


Figure 2.3 – UML representation of an inheritance relationship. In this case, *Animal* is the superclass, while *Duck* and *Dog* are the subclasses.

Figure 2.3 displays an example of how UML can be used to represent inheritance relationships.

In this case, the **Animal** class assumes the role of superclass, while the **Duck** and **Dog** classes are its subclasses. The triangle denotes these relationship by pointing from the subclass towards the superclass. These relationships should be read as: *a Duck is an Animal* and *a Dog is an Animal*, respectively. The two subclasses will receive all attributes and methods defined in the superclass, and have the possibility of adding their own. As such, both **Duck** and **Dog** will inherit the `weight` attribute and the `toString()` method from their superclass. In summary, both subclasses will have two attributes each: the **Duck** will have `weight` and `wingSpan`, while the **Dog** will have `weight` and `breed`.

Example with inheritance and composition

Figure 2.4 presents a design which contains both composition as well as inheritance relationships. The composition relationship is the same that was previously presented, with the **House** class being the container and the **Room** class being the contained. However, this time the model allows for multiple types of **Rooms** to be created: **Kitchens**, **Bedrooms**, **Livingrooms** and **Bathrooms**.

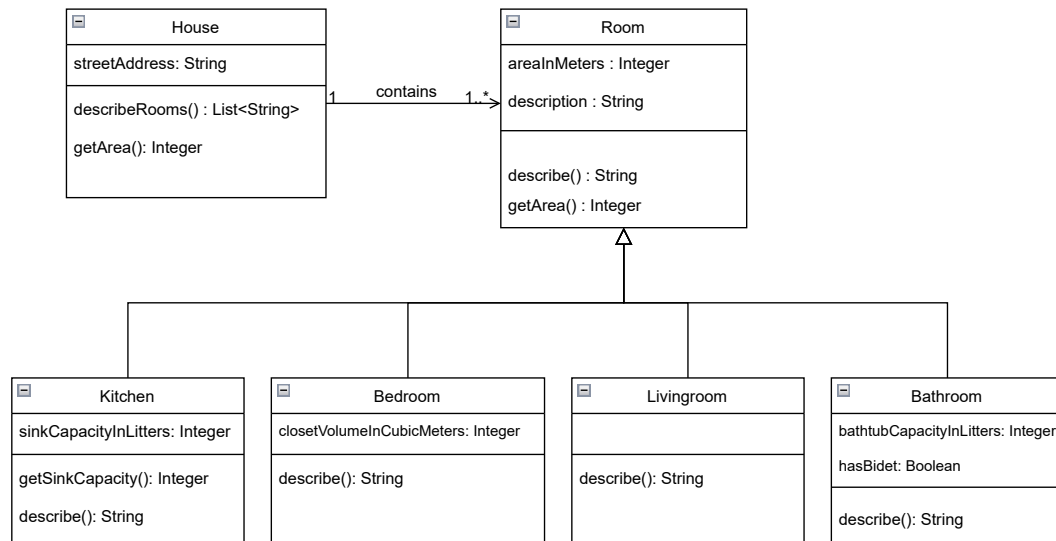


Figure 2.4 – UML representation of a design which contains both composition elements (a **House** contains **Rooms**) as well as inheritance elements (a **Kitchen** is a **Room**, a **Bedroom** is a **Room**, a **Livingroom** is a **Room** and a **Bathroom** is a **Room**. Each sub-class has a `describe()` method which will combine values of inherited attributes with the values of its specific attributes.)

Polymorphism

As previously mentioned, polymorphism is the concept that allows objects of different types to be treated as instances of the same class through a shared interface. In some OO languages (e.g., Java), there is a caveat: those types must be related, usually via inheritance².

Let's go back to the **Animal/Duck/Dog** example to see a possible UML representation of polymorphic behaviour. In Figure 2.5, the **Animal** class is now abstract (and its name is italicized, as per UML rules) and defines an abstract `makeSound()` function/method. Furthermore, the classes **Duck** and **Dog** override `makeSound()` and implement their own version. An implementation of this model is available in Listing 2.1. By analyzing `Main.main()`, we can observe two practical consequences of polymorphism: 1) we can use the same data structure – the array of **Animal** objects – to store objects of different, but related, types, and 2) we can call the `makeSound()` function on those same objects, without worrying about their concrete type.

²It can also be achieved by using interfaces, a mechanism which allows specifying that a class must implement a certain method.

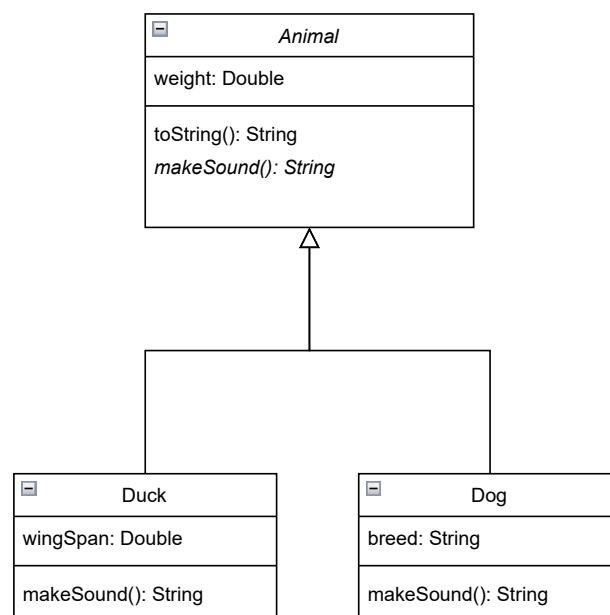


Figure 2.5 – UML representation of a design which takes advantage of Inheritance to obtain polymorphism. The **Animal** class is abstract and defines an abstract method `makeSound()` which is then overridden by both sub-classes.

Listing 2.1 – An example implementation of the object oriented mechanisms which allow polymorphism. The `makeSound()` function can be called for multiple types of objects (i.e., Duck, Dog and any other Animal sub-classes).

```

1  abstract class Animal {
2      public abstract String makeSound();
3  }
4
5  class Duck extends Animal {
6      // Overridden method in Duck class
7      @Override
8      public String makeSound() {
9          return "Quack, Quack!";
10     }
11 }
12
13 class Dog extends Animal {
14     // Overridden method in Dog class
15     @Override
16     public String makeSound() {
17         return "Bark, Bark";
18     }
19 }
20
21 class Main {
22     public static void main(String[] args) {
23         Animal[] myPets = new Animal[]{new Duck(), new Dog()};
24         for(Animal pet : myPets) {
25             pet.makeSound();
26         }
27     }
28 }

```

2.1.3 Inheritance and Polymorphism: a Case Study

To better understand the advantages of **inheritance** and **polymorphism**, consider the following example, which mimics a software where geometric figures are created and manipulated. This software is able to store lists of user-defined geometric figures and perform calculations of properties related with the geometric figures stored in those lists.

A class was created to represent the most generic geometric figure (i.e., **GeometricFigure**), from which all the actual figures (e.g., **Square**, **Triangle**, and so on) should inherit.

Now imagine a more concrete requirement: given a list of geometric figures, the software must be able to calculate the sum of the areas of all figures contained in the list. Figure 2.6 presents a UML class diagram with a possible solution, while Listing 2.2 presents a possible implementation based on the same UML.

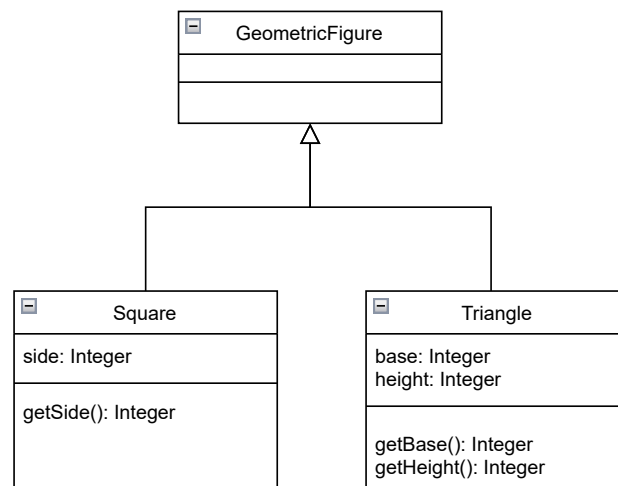


Figure 2.6 – UML representation of a possible, although non-ideal, solution for the Geometric Figures area calculation problem. The 3 classes are being used as data-containers and have no relevant behaviour.

Listing 2.2 – An example of bad OO code: 1) information hiding is poor, and 2) does not take advantage of polymorphism.

```

1
2 class GeometricFigure { ... }
3
4 class Square extends GeometricFigure {
5     int side;
6     // readers and writers such as getSide() omitted for brevity ...
7 }
8
9 class Triangle extends GeometricFigure {
10     int base, height;
11     // readers and writers, such as getBase() and setBase() omitted for
        brevity ...
12 }
13
14 class Main {
15
16     public static double sumAllAreas(List<GeometricFigure> figures) {
17         double sum = 0;
18         for(GeometricFigure figure : figures) {
19             double area;
20             if(figure instanceof Square) {
21                 int side = ((Square)figure).getSize();
22                 area = side * side;
23             }
24             else if(figure instanceof Triangle) {
25                 Triangle triangle = ((Triangle)figure);
26                 int base = triangle.getBase();
27                 int height = triangle.getHeight();
28                 area = 1 / 2 * base * height;
29             }
30             sum += area;
31         }
32         return sum;
33     }
34
35 }

```

If we focus on the `sumAllAreas()` function, we can observe a set of conditional statements (i.e., `if/else if`) which decide the program's behaviour based on the object's type. This solution has a number of disadvantages:

- If we need to extend the code to support other geometric figures (e.g., `Rectangle`), besides creating the new class, we will have to adjust this code by adding another `else if` block;
- Knowledge related with the calculation of each type's area is defined outside the

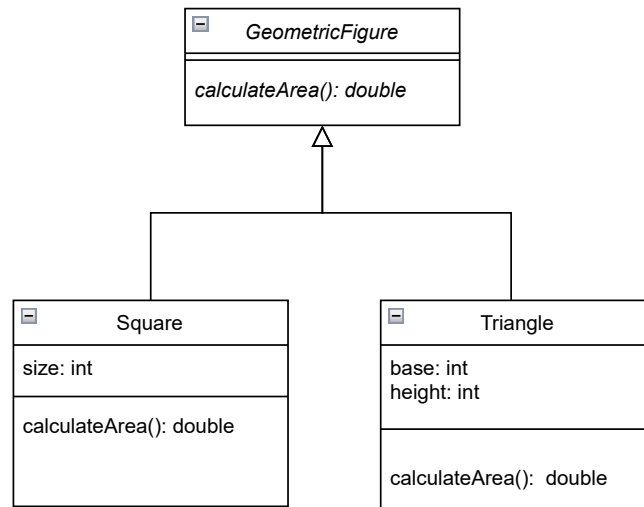


Figure 2.7 – UML representation of a better solution to the Geometric Figures area calculation problem. The `GeometricFigure` class declares an abstract method called ‘`calculateArea()`’ which the sub-classes must implement. Each of the classes now has a specific behaviour.

respective class, thus breaking the information hiding principle.

We can improve this code and prevent these issues by following OO principles:

1. Make class `GeometricFigure` abstract;
2. Add to it an **abstract** `calculateArea()` method, so that all subclasses will have to implement it. This will promote information hiding, since each class will only know its own area calculation logic. It will also allow taking advantage of polymorphism to simplify the `sumAllAreas()` function;
3. Re-write function `sumAllAreas()`, taking advantage of the previous changes.

Figure 2.7 presents a UML class diagram with an improved OO model following the first two recommendations, and Listing 2.3 shows an implementation which takes advantage of the polymorphism mechanism.

Listing 2.3 – An example of good OO code, taking advantage of inheritance and polymorphism to promote dynamic dispatch.

```
1 abstract class GeometricFigure {
2     abstract double calculateArea();
3 }
4
5 class Square extends GeometricFigure {
6     int side;
7     double calculateArea() {
8         return side * side;
9     }
10 }
11
12 class Triangle extends GeometricFigure {
13     int base, height;
14     double calculateArea() {
15         return 1/2 * base * height;
16     }
17 }
18
19 class Main {
20
21     public static double sumAllAreas(List<GeometricFigure> figures) {
22         double sum = 0;
23         for(GeometricFigure figure : figures) {
24             sum += figure.calculateArea();
25         }
26         return sum;
27     }
28
29     public static void main(String[] args) {
30         // ...
31     }
32 }
```

This solution has some advantages:

- If we need to support another figure type (e.g., Rectangle), we only need to define that class and make sure it contains a ‘calculateArea()’ function;
- The area calculation formula is hidden within the respective class (i.e., encapsulation, information hiding)
- The calculation function has much less code and is much simpler;

This example can also be used to illustrate the idea of dynamic dispatch: by removing the `if / else if` statements, we leave to the Java Virtual Machine (JVM)³ the respon-

³A software system which allows a computer to execute Java programs.

sibility of figuring out which method will be called, considering the type (class) of the object.

2.1.4 Code organization

One of the important characteristics of OO exercises is the need to use multiple code files. This need arises from the fact that a multi-class system will be better organized if each class is contained within a single file, since the code that is strongly-related will be contained in the same file.

2.1.5 Teaching and Learning OOP

Some OOP courses are strictly based on the interpretation of UML diagrams: the teacher first explains students how to interpret UML diagrams, and then students have to interpret those diagrams and convert them to a programming language [138]. While this technique has its merits, it results in very directed exercises, where the student does not do a significant amount of domain analysis and design. Instead, the student acts as an implementer of the class models devised by the teachers [138].

In a real-world scenario, the client will typically not inform the software developers about which classes and methods should exist in the system. Instead, developers will infer or derive that information from reading requirements and/or from talking with the client and analysing the problem domain. This reality has led some educators to create OO courses in which students learn to analyse problem descriptions and map them into class models describing entities and behaviours which reflect the analysed problem's domain and allow achieving the desired software functionality [41, 43]. These design-oriented exercises are important, since designing software systems is recognized by the CSE community as a common student difficulty, even for more experienced students [207].

Furthermore, the teaching of OOP must not be solely focused on training students to produce working code: the students' code solutions should follow the previously discussed best practices (i.g. information hiding, inheritance, polymorphism, and so on) in order to best take advantage of the paradigm and produce code that is easy to change [41, 43], since a large part of the total cost of a software system is adding new features and adapting the original requirements as time goes by [130].

Finally, the teaching of OOP benefits from both exercise-based and project-based teaching methodologies [109]. Project-based (PB) learning describes a course structure in which students work towards finding solutions to real-world problems [109, 167]. These problems are represented as projects that often span the entire course duration. On the other hand, in exercise-based (EB) courses, students solve questions with predetermined answers (i.e., small coding questions) [109]. In EB courses, individual assignments have shorter durations and are typically less creative than those used in PB learning [109]. PB learning is not a new idea, and research has shown that engineering students tend to be more motivated when working with real-life case studies and challenging open-

ended questions [20, 77], which are more easily implemented with projects rather than small exercises. A study by Mills et al. [132] evaluated PB learning practices in several engineering degrees and noted that while students who participate in PB learning are generally motivated and demonstrate better teamwork and communication skills, they also display a less rigorous understanding of engineering fundamentals. The same study concluded that a mixed approach of traditionally lectured courses with both PB and EB components might be the best way to satisfy industry needs without sacrificing knowledge of engineering fundamentals.

These ideas can easily be used to structure an OOP course: small exercises with 2-3 classes can be used to facilitate learning the basic concepts and the respective syntax, while larger projects will force students to deal with more complex design decisions and implementation details.

2.2 Automatic Assessment Tools

AATs are educational software systems that automate the assessment (or evaluation) of student-developed computer programs. AATs have been around since at least 1960 [59]. These tools allow students to check if their code solutions satisfy the requirements of the assignment with full autonomy [66], without assistance from the teacher. Furthermore, AATs reduce teacher workload [140, 119]. This is usually done by automating the compilation and execution of the student's code in a restricted environment and validating its correctness through output matching, unit testing, or other techniques [12, 86]. Depending on the tool, students may or may not receive rich feedback about their submissions' failures. By providing quick feedback and highlighting mistakes, these tools can be used to foster computer science students' engagement in programming courses and increase their motivation [107]. Finally, most of these tools automatically grade student solutions, a process which tends to promote grading consistency when compared to manual grading [48].

The use of AATs is common in Computer Science courses [87], and there are a variety of commercial and academic systems. However, many of them are closed-source and/or not packaged in a production-ready fashion, making their adoption in other institutions impractical. This is partially confirmed by previous research, which has shown that some of these AATs are created as part of a thesis or for a particular course [86]. They are finished enough to study a certain research question or to support the needs of a particular course, but are generally not suitable for widespread deployment. This situation might be improved in the future, since, in the most recent years, a greater number of the most recent developments have been published as open-source software [139].

2.2.1 Relevant tools

Several relevant tools should be considered when adopting an AAT for programming exercises and projects.

A first example is repl.it, a proprietary online tool where teachers create assignments that can be evaluated with unit testing or with input/output matching, depending on the programming language [176]⁴. Students interact with this platform using a web-based IDE. HackerRank [120] is a similar platform, originally developed for competitive programming and programming interviews. While such services might be practical for small exercises, they are not ideal for mid- to large-sized, multi-file projects, as they offer environments far removed from common professional development workflows [173].

Some universities use open-source systems originally developed for setting up programming contests. An example of this is Mooshak [106, 78]. Students interact with Mooshak by uploading code files, which are then tested using input/output matching. Raccode, Mooshak’s companion plugin, allows students to submit a code file directly from the NetBeans IDE [173]. Mooshak is geared towards the validation of assignments that can be solved with a single source file, and thus better suited for simple exercises.

Web-CAT [64] is an open-source automated grading system in which solutions can be uploaded via browser or submitted using a plugin for the NetBeans IDE. It is able to evaluate the code coverage of the student’s own tests. Assignment creation implies the manual upload of the unit tests using web forms. Although Web-CAT is oriented for projects, its NetBeans plugin makes it apt for small exercises [47]. Web-CAT’s recommended installation method is a pre-built WAR file⁵ that can be downloaded from a Sourceforge page⁶. However, it should be noted that the WAR file was last updated in 2008. The GitHub repositories⁷ contain commits after that date, thus the WAR file does not seem to have the latest changes.

Another relevant tool is check50 [168], an open-source command line tool for assessing the correctness of students’ code using tests written in a custom language. This tool does not enforce any specific code structure (e.g., mandatory functions), which makes it inappropriate for large projects, which need to be well-structured following software engineering best practices [47]. The same authors developed a set of separate companion tools for checking adherence to code style guidelines (i.e., style50), performing the actual submissions (i.e., submit50), and plagiarism detection (i.e., compare50) [122].

The Canary Framework (CF) is a set of six independent programs based on real workplace tools, such as GitHub, Jenkins, Checkstyle and Jacoco [79]. Like Web-CAT, Canary

⁴As of August 2024, repl.it continues to exist but has shifted away from its original focus on educational use, transitioning towards a more general-purpose, AI-based coding platform. The information presented in this section was accurate as of 2022, when the respective publication was initially submitted. We have chosen to retain this entry to provide a historical perspective on the tools available to educators and students at that time.

⁵WAR files—Web Application Resource or Web application ARchive—are used to distribute a collection of Java classes, HTML pages and other resources that together constitute a web-application.

⁶<https://Web-CAT.org/projects/Web-CAT/InstallWebCat.html>.

⁷<https://github.com/Web-CAT>.

is able to measure the coverage of the student’s test code. This framework follows a continuous integration approach, which is more suitable for PB assignments [47]. The Canary Framework only supports Git submissions, which may be a limiting factor for first-year students. Although Canary Framework’s tools are announced as open-source, only half of these seem to have their source code available⁸.

Drop Project (DP) is an AAT originally developed to support the development of projects [48], offering submissions through ZIP file upload or via a Git repository, making it less than ideal for smaller exercises. DP is fully open-source and can be readily deployed using, for example, a Docker image [30]. This tool is complemented by an IDE plugin which aims at improving its usability for small exercises. An in-depth discussion of this tool (and the respective IDE plugin) will be presented in Chapter 3 of this thesis.

Finally, some AATs are integrated out-of-the-box with learning management systems such as Moodle [126]. An example is CodeRunner, which allows students to write and submit code directly from the browser [116]. A distinguishable characteristic of this tool is that it is able to assess OpenGL code [201]. Another example is the Virtual Programming Lab (VPL), which also allows students to write and submit code directly in the browser [185]. It can also be configured to disable certain text edition capabilities, like disallowing copy paste from external sources.

Table 2.1 presents a comparison of the main characteristics of the discussed tools using multiple relevant criteria.

When analysing the comparison table, consider the following informal definitions of EB and PB suitability:

We consider an AAT as suitable for EB assignments if it allows submissions without significant legwork. AATs based on web-editors are good examples of this, as submitting code usually requires just a button click on the same interface where the code was written. Conversely, AATs that require manual submission of a package (e.g., using a ZIP file) are excluded, as this process is impractical for small exercises. Just imagine a student implementing a `int findMax(int[] numbers)` function. Having to package their code into a ZIP file and manually upload it to an online validation platform — and potentially repeat this process if the code has issues — involves significant submission effort relative to the size of the exercise.

An AAT is considered suitable for PB assignments if it supports multiple files and is not based on web-editors. The reason for this exclusion of web editors is that, although some of them support multiple files, they are not an ideal option from a usability and tool perspective [47]. This is because they typically lack the feature-rich code navigation and refactoring capabilities of traditional desktop-based IDEs, which are particularly helpful for handling larger code bases composed of multiple files.

⁸<https://sos-cer.github.io/projects/auto-grading>

Table 2.1 – Summary and main characteristics of the discussed AATs. “Open?” indicates whether the tool is open-source. “EB?” and “PB?” refer to suitability for exercise-based and project-based approaches, respectively. The term “web-IDE” is used as a shorthand for “online code editors”.

Tool	Open?	Installation	EB?	PB?	Beginner-Friendly?
repl.it	No	N/A	Yes	No	Yes (web-IDE)
HackerRank	No	N/A	Yes	No	Yes (web-IDE)
Mooshak	Yes	Pre-compiled JAR file	Yes	No	Yes (submissions via upload)
Web-CAT	Yes	Pre-compiled WAR file (from 2008); source install requires compilation of multiple sub-systems	Yes ^a	Yes	Yes (submissions via upload and IDE plugin)
check50	Yes	Requires installation of multiple tools (e.g check50, style50, submit50)	Yes	No	Yes
Canary framework	Partially ^b	Requires installation of multiple tools	No	Yes	No (only Git submissions)
VPL	Yes	Must be installed by Moodle’s administrator	Yes	No	Yes (submissions via upload and web-IDE)
CodeRunner	Yes	Must be installed by Moodle’s administrator	Yes	No	Yes (web-IDE)
Drop Project ^d	Yes	WAR file compiled from latest sources/one-click deploy to Heroku (single tool)	Yes ^c	Yes	Yes (submissions via upload and Git)

^a With EB plugin for NetBeans IDE. ^b Since the code for half the repositories is not available. ^c With the proposed plugin for IntelliJ IDEA. ^d DP and the respective IDE plugin will both be presented in Chapter 3 of this thesis.

From our investigation, the AAT that comes closest to being suitable for both small exercises and projects is Web-CAT. However, it does not appear to be easy to install, deploy, and/or use due to a lack of updated binaries and documentation. In summary, all of the AATs discussed so far have one or more types of limitations, which can be summarized as follows:

- They are difficult to install, deploy, and/or use;
- They are inappropriate for beginner students, e.g., by only allowing submissions using Git;
- They are unsuitable for assessing and evaluating multi-file projects (i.e., PB learning);
- They are unsuitable for assessing and evaluating small exercises (i.e., EB learning).

2.2.2 AAT cheating patterns

Research has also delved into students' ability to work around automatically graded assessments and get successful scores without performing the assignment as prescribed by the teachers. The authors of [101] analysed students' code submissions in two AAT-based courses⁹, identifying and categorizing cheating attempts into 4 distinct patterns:

- **Overfitting:** Students' code is hardcoded to match expected output; ineffective beyond the specific assignment's tests. Overfitting corresponded to 63% of the cheating attempts.
- **Problem evasion:** Students' code fails to meet non-functional requirements (e.g., the use of recursion). It accounted for 30% of the cheating attempts.
- **Redirection:** Students call the reference solution from their own code. Redirection corresponded to 6% of the cheating attempts;
- **Injection:** Malicious code is injected to change the assignment's validation process. This pattern corresponded to 1% of the cheating attempts.

Some of these issues, particularly Redirection and Injection, are highly dependent on the AAT's architecture. For instance, AATs that isolate the reference solution from the students' code, such as [48], indirectly prevent Redirection. However, both Overfitting and Problem evasion are cheating patterns which do not directly exploit AATs' weaknesses.

⁹Both courses used VPL to perform automated assessment.

2.2.3 Feedback

Feedback is considered a facilitator of learning and performance [172]. In a literature review focused on feedback, Shute defined formative feedback as *information communicated to the learner that is intended to modify his or her thinking or behaviour to improve learning* [172]. Formative feedback can be presented to the learner using information (e.g., verification of response accuracy, explanation of the correct answer, hints) and can be administered at various times during the learning process (e.g., immediately following an answer, after some time has elapsed) [172]. Shute’s research also offered guidelines for creating formative feedback. In a recent literature review focused on formative feedback for computer science students [182], the authors concluded that, to better assist novice programmers in learning programming, formative assessments must improve the presentation of error messages. Finally, in a study evaluating the use of GPT-3.5 for generating hints for problems detected by an AAT [141], students were divided into two groups – one receiving GPT-based hints and the other only receiving regular AAT feedback. The usefulness of the GPT hints was assessed using a 5-point Likert scale; 46% of students rated them highly (4 or 5), while 19% gave low ratings (1 or 2). The study found that students with access to GPT-based hints tended to use the AAT’s default feedback less. However, the findings also suggest some potential over-reliance on the GPT-generated feedback.

2.3 Large Language Models for CSE

2.3.1 Foundations of LLMs

Natural Language Processing

Natural Language Processing (NLP) is a field of Artificial Intelligence that focuses on the interactions between computers and human languages, such as English or Portuguese. NLP algorithms try to understand, interpret, generate and manipulate natural language text or speech. The goal of this field is to enable computers to perform a wide range of language-related tasks, allowing humans to interact with computers using natural language.

In 1957, Noam Chomsky published a book called “Syntactic Structures” [39], in which he proposed the notion that, for a computer to understand a language, the sentences would have to be converted to a specialized grammar. Furthermore, Chomsky also created a type of grammar which could methodologically translate natural languages into a format that would be usable by computers. The concepts explored in this book were influential to the future development of NLP.

Later, in 1958, John McCarthy (one of the original proposers of the term artificial intelligence [128]) released the LISP programming language. LISP stands for “LISt Processor,” and Linked Lists are one of its main data structures [127]. Furthermore, LISP code is written using Symbolic Expressions (commonly known as S-expressions), a paren-

thesized notation for representing data and code as nested Lists [127]. This representation is at the core of LISP’s design, enabling one of its most distinctive properties: homoiconicity. Since LISP’s source code is itself represented using the language’s fundamental data structure, S-expressions, programs can be easily manipulated and self-modified.

In 1966, Joseph Weizenbaum developed ELIZA, a computer program which allowed natural language conversations between its user and itself. ELIZA allowed the implementation of ‘scripts’ which would interact with the user. These scripts were implemented using a lisp-like representation (i.e., S-expressions). ELIZA used pattern matching to identify keywords in the users’ sentences. Those keywords would then trigger specific behaviours. One of the most famous ‘scripts’ was called the DOCTOR, and would simulate a psychotherapist, reflecting the user’s words back to them. This program would give the user an illusion of understanding, but it had no actual understanding of what was said by either party [198].

Later, in 1971, Terry Winograd developed SHRDLU, a computer program partially written in LISP which allowed the user to interact with objects inside a virtual box filled with different blocks. Users could move, name, collect and query the state of the objects. This interaction was done using a conversational-style interface [200].

In the 1980s, scientific consensus existed with regards to the importance of building large datasets to support research initiatives in NLP [125]. One of the produced datasets was the Penn Treebank project (1989). This project produced a dataset of 4.5 million English words annotated with tags representing their part-of-speech information (e.g., noun, verb, etc.) [125].

The following decade saw the introduction of statistical methods in NLP. A notable work was Peter F. Brown et al’s “A Statistical Approach to Machine Translation”, published in 1990 [32], which presented a groundbreaking approach to machine translation using probabilistic models.

The early 2000s saw the appearance of Conditional Random Fields, a technique supporting the prediction of structured objects, such as sequences, grids or trees, rather than scalar or real values [103].

In 2006, Google launched Google Translate¹⁰, a statistical machine translation service, originally developed using documents and transcripts from the United Nations and European Parliament.

Tomas Mikolov et al. introduced Word2Vec in 2013. This technique allows for efficient representation of words as vectors which capture information about the meaning of a word considering the words surrounding it in a sentence [131].

In 2015, Dzmitry Bahdanau et al. published a paper which touched on the concept of ‘attention’, a mechanism by which certain parts of the source sentence are given more or less importance. This led to improved results when translating long sentences [19].

Finally, in 2017, Ashish Vaswani et al. published a paper called “Attention is All You Need”, which proposes a new neural network architecture – the Transformer –, based

¹⁰<https://translate.google.com/>

solely on attention mechanisms [189]. This paper is considered to have spawned the LLMs revolution.

Refer to Figure 2.8 for a presentation of the aforementioned works and papers in a **timeline** format.

Language Models

Language Models (LMs) are tools which, after being trained with textual data (e.g., books, websites, and so on), are able predict the next token in a sequence (e.g., the next word in a sentence). These models are built from training data, usually composed of large body of text, gathered from multiple sources such as books, the Wikipedia, online newspapers and articles, and so on. Imagine a dataset where multiple instances of “Feed the cat” and “Walk the dog” exist, but contains no instances of “Feed the dog” and “Walk the cat”. If a model trained on such dataset is asked to complete the sentence “Feed the ?”, it will likely provide “cat” as the most probable next completion, while “dog” would be a less likely completion, since the model has never seen such use.

Multiple techniques exist for building language models. One example is the technique known as n -gram language model, which is considered the simplest kind of language model by some authors [93]. This type of language model assumes that the probability of the next word in a sequence depends only on a fixed size window (i.e., n) of previous words.

Multiple practical applications of such technologies exist. An example would be a word processor or e-mail client which displays possible completions based on the user’s first few letters or words. Applications also exist in software development. For instance, the authors of [161] experimented the use of n -gram language models to add API-based auto-completion features to an Integrated Development Environment (IDE).

However, this technology is not without faults. If the training dataset contains mistakes, such as sentences with incorrect grammar or misspelled words, then it is likely that the model will generate predictions with similar mistakes. This means that the models are sensitive to their training data. Also, language models do not verify the respective generated content: just because a certain prediction is likely, it does not mean it’s factually correct.

Large Language Models

Building on the concept of LMs, researchers have developed an architecture called *Transformer* [189]. This architecture was first proposed in 2017 and displayed significant improvements in natural language translation, managing to surpass the state-of-the-art models in English to German and English to French translations, with the added benefit of lower training costs [189]. This was done by the application of the *self-attention* (also known as *intra-attention*) mechanism by which each token in a sequence is able to influence other tokens, helping the model capture long-range dependencies and relationships and understand how different words in a sentence relate to each other.

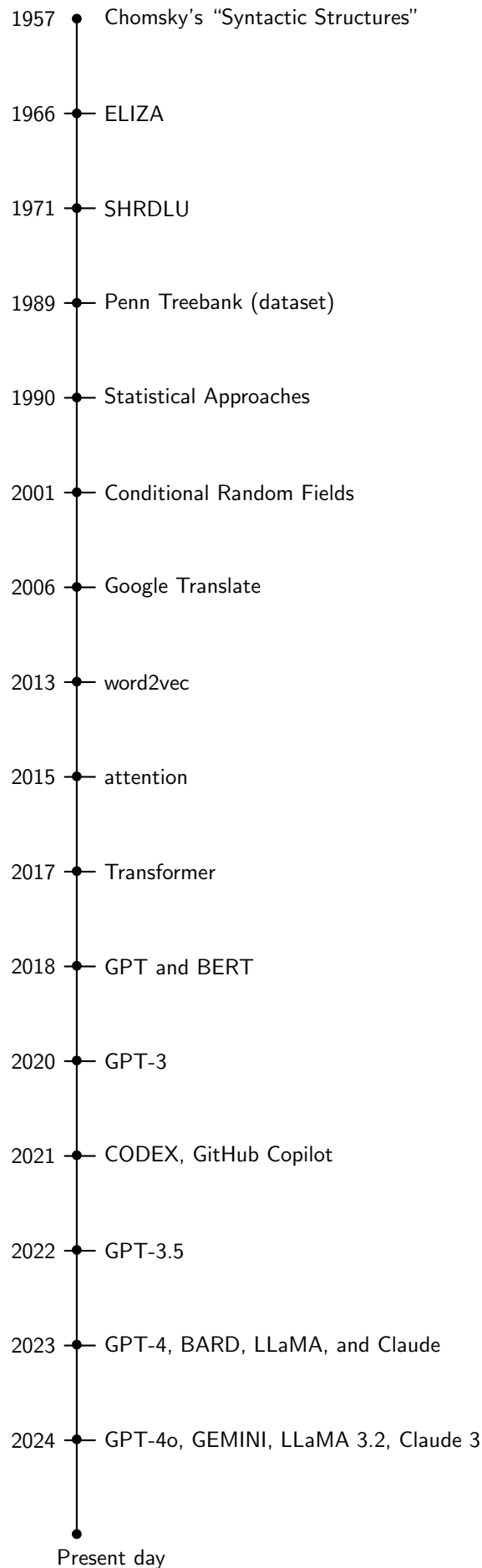


Figure 2.8 – Timeline of Key Milestones in Natural Language Processing Relevant to the Development of Large Language Models.

In order to understand the mechanism, consider the following simplified example. Take the sentence “My car is blue”. When the language model is processing this sentence, a vector representation is created for each of the four words. Afterwards, the relationship between each word is computed. In this step, each word vector influences the vectors of the words that come after it in the sentence. This means, for example, that the vector for ‘blue’ is changed by the words: “My”, “car” and “is”. This process results in stronger relationships between the words (and their vectors).

Multiple tools based on the *transformer* architecture have since been developed and released. Some notable examples are OpenAI’s GPT (Generative Pre-trained Transformer) [155], Google’s BERT (Bidirectional Encoder Representations from Transformers) [57] and Google’s LaMDA (Language Models for Dialog Applications) [186].

2.3.2 LLMs in Practice

Landscape Overview

At the time of writing this thesis, OpenAI’s GPT-4 is considered the state-of-the-art model, but it is not available for free (i.e., *gratis*). OpenAI currently allows free access to their GPT-3.5 model (unlimited if used from the respective ChatGPT interface; API usage is free up to a certain amount of tokens). Other Closed-Source alternatives, such as Google’s Bard (based on their LaMDA model), Google’s Gemini [180] and Anthropic’s Claude [5] exist. Moreover, multiple non-commercial models have also been made available. Some notable examples are GPT-J [194], Mistral [92] and Meta’s Llama (Large Language Model Meta AI) [187].

Commercial vs Open vs Open Source Models

As previously mentioned, several non-commercial tools also exist, with some of them claiming to be Open Source. However, there is currently no universally accepted definition of what constitutes an Open Source LLM or Artificial Intelligence.

We can start by presenting a sort of *gradient* of model releases which has been used to describe LLMs in several publications: Closed models, Open-weight models and Fully open models. This definition uses as its differentiating criteria the access to the model’s weights as well as the openness of the training data [9]. Table 2.2 details this classification as described by [9].

While this classification provides a helpful framework for understanding different levels of openness in AI models, it also exposes a challenge: multiple levels of ‘openness’ exist, each with its own criteria. This tiered approach can blur the lines between what is considered ‘open,’ leading to varying interpretations of how accessible and transparent a system truly is.

The Open Source Initiative¹¹ has recognized this gap and is currently working on

¹¹<https://opensource.org/>

Table 2.2 – Gradient of Large Language Model releases

Model Type	Description	Examples
Closed models	Model weights are not available	GPT-3.5 [33] GPT-4 [8] Bard [179] Gemini [180] Claude [5]
Open-weights models	Model weights are available, but training data or code are not available	LLaMA [187] Mistral.AI [92] deepseek [28]
Fully open models	Full access to model weights, code, and training data	Bloom [105] OLMo [76] StarCoder2 [117]

establishing a definition of Open Source AI¹² [183]. This effort arises because many AI systems, especially LLMs, consist of multiple components, not all of which are software. Consequently, existing OS definitions and licenses are not always applicable. For example, the datasets used to train LLMs are governed by different legal frameworks laws around the world, which raises particular challenges when trying to open source such systems [183].

The most current working draft (v.0.0.9)¹³ identifies 3 types of mandatory required components in order for a data-based AI system to be considered Open Source: Data, Code and Model. For each component there exists a more specific sub-list of what must be made available under an OSS compatible license. The full list of mandatory components is the following:

Table 2.3 – Requirements for Data, Code, and Model components of an Open-Source AI system

Component	Requirements
Data	At least one of the following must be made available: (1) datasets, (2) research paper, (3) technical report, (4) data card
Code	All of the following must be made available: (1) data pre-processing code, (2) training, validation and testing code, (3) inference code, (4) supporting libraries and tools
Model	All of the following must be made available: (1) model architecture, (2) model parameters (e.g., weights)

As we can observe, the data component is the most flexible one, with multiple options

¹²<https://opensource.org/deepdive>

¹³published on August 22, 2024

for open publication being given. This is, in fact, the biggest difference between the classification ‘gradient’ proposed by [9], which only considers a model as ‘fully open’ if the respective training data is also open.

Evaluating LLMs

Since LLMs give non-deterministic (i.e., probabilistic) results, it is possible that some of their attempts at solving a problem yield incorrect or incomplete results. As such, multiple metrics have been developed to evaluate these tools, taking in consideration their particularities. Some examples are:

- Perplexity - measures how well a language model can predict the expected next token (or word) in a sample sequence. The lower the perplexity value, the better the model is performing [91].
- Bilingual Evaluation Understudy (BLEU) - a metric used to measure the quality of text which was machine translated from one language to another [142].
- Pass@k - measures the likelihood that at least one of the top-k generated solutions contains a correct solution to a given problem or task. This metric was proposed by OpenAI in the context of evaluating LLMs’ capacity to generate computer code [38]. It is calculated by generating multiple LLM-solutions based on a given prompt and, running a set of unit tests over those solutions. If any of the generated solutions passes all the unit tests, then it is considered a success. The value k gives further information. If $k=1$ (i.e., pass@1), then the model’s first solution was correct. If $k=10$ (i.e., pass@10), then at least one of the model’s first 10 solutions was correct.

Generic benchmarks

Furthermore, several benchmarks and question/problem datasets have been proposed and used for evaluating LLMs. A first example is LLMsys, an online service where users can write prompts that are simultaneously given to two randomly selected LLMs¹⁴. Afterwards, the service displays the results for each of the anonymous models, and allows users to vote on the output of their preference. These votes then calculate a score which is used to rank the LLMs. Another alternative has been the usage of standardized tests usually taken by humans wishing to access certain universities or professions. For example, OpenAI reported having tested GPT-4 on questions appearing in Scholastic Assessment Tests (SAT) such as the SAT Math test¹⁵. They also reported testing the model with questions appearing on the Bar Exam¹⁶, as well as a variety of other certification-based tests regularly taken by humans [137].

Benchmarks and Datasets for Programming Tasks

¹⁴At the time of writing this thesis, LLMsys supports more than 50 LLMs

¹⁵The SAT is a standardized test widely used for college admissions in the United States.

¹⁶The Bar Exam is taken by students wishing to become lawyers in the United States

A number of datasets have also been proposed for the evaluation of LLMs’ programming abilities. One example is the Automated Programming Progress Standard (APPS), which consists of 10 thousand examples of Python coding problems [80]. APPS problems were collected from varying coding websites such as Codeforces [1] and Kattis [6, 66]. Unlike other datasets, the majority of the problems is not based on single-function synthesis, focusing instead on the generation of full programs which handle user input (stdin) and output a result (stdout) [38]. A possible drawback of using this dataset is that it uses problems from Codeforces and Kattis. Considering that it is a known fact that some GitHub users have public repositories containing solutions for problems from these sources [38], it is possible that an LLM trained on code publicly available on GitHub has seen some of the problems and solutions during its training phase, which could distort its evaluation.

Another example is HumanEval, an hand-written dataset consisting of 164 Python problems created for the validation of LLM-base code generation. Each problem includes a function signature, a docstring (i.e., problem description), and several unit tests. The problems were hand-written in order to promote some level of novelty, although it is not ensured [38].

Also relevant is StudentEval, a dataset containing 1749 prompts which were written by students with one semester of Python programming experience. The novelty of this dataset is the fact that it contains prompts written by non-experts, unlike the majority of other datasets [18].

Building on the fact that the majority of the proposed datasets is English and Python centric, some researchers have started to produce alternative datasets containing multiple programming languages and/or multiple natural languages. For example, the authors of [17] have converted HumanEval to 10 programming languages. Furthermore, the authors of [146] created a multi-language dataset, both in terms of natural languages (having problems described in 23 languages) as well as in terms of programming languages (12 of them).

Specialized language models: Code LLMs

Although general purpose models have demonstrated good performance on multiple topics [8], researchers have also been working on the tuning and/or creation of models which are specialized to certain knowledge domains.

One of the knowledge domains where this has happened is computer programming. These models aim at supporting computer programmers with the generation of computer code.

A Closed-Source example is OpenAI’s Codex. This model was based on GPT-3, but it was additionally trained on code available on GitHub. Codex is the model behind the original version of GitHub Copilot, a plugin which offers code generation capability for IDEs such as Visual Studio Code and IntelliJ IDEA. Copilot’s code generation is spawned by inputting code comments such as “compute the sum of an array”, which the tool then uses

to generate a code block that attempts to satisfy the comment prompt. Also dedicated for computer code generation are the open source models CodeParrot [188] (based on GPT-2), SantaCoder [10], StarCoder [111], StarCoder2 [117] and Code Llama [159].

The authors of [202] performed a systematic evaluation of some of these Code LLMs (e.g., CodeParrot and Codex), using HumanEval as a benchmark. This study used several versions of the tested models, with varying sizes (e.g., CodeParrot 110M, CodeParrot 15B, Codex 300M, Codex 2.5B, and Codex 12B). The best model was Codex 12B, with a Pass@1 of 28.81%, a Pass@10 of 46.81% and a Pass@100 of 72.31%. Furthermore, even the smaller Codex 300M had better performance in the 3 metrics than the other non-Codex tested models, showing that model size is not the single contributing factor to these models performance.

Although these specialized LLMs have been successful at code generation, the reasons for this are currently not well understood, leading some researchers to try and bridge this knowledge gap. An example is [81], which evaluated 10 models (e.g., StarCoder, Code Llama, among others) to determine how well they understand programming concepts such as control flow, data flow and identifier names. This study concluded that the studied models lack understanding of programming concepts related to control-flow and data-flow.

Other types of generative models

The models presented so far are known as text-to-text models, since they receive textual input and return textual output. However, other types of LLMs exist. Two examples are text-to-image and text-to-speech. An example of a text-to-image model is OpenAI's Dall-E, which is based on the previously presented transformer architecture [157].

Models with the opposite capability (i.e., image-to-text) also exist. A case in point is CLIP (or Contrastive Language-Image Pre-Training), developed by OpenAI [154].

Multi-modal tools

Some of the most recently available tools are now capable of handling multiple input and output types. Two examples of this are Google's Bard/Gemini [102, 85] and OpenAI's ChatGPT with GPT-4 [136], which support both textual as well as image input.

Training Data

As previously mentioned, LLMs are sensitive to the contents and quality of their training data. However, the details of the data used to train them are not always known. This is particularly true for commercial models such as the GPT family, Bard and Claude.

For example, it is known that the GPT-3 model was trained using a filtered version of the Common Crawl dataset¹⁷ (60%), an expanded version of the WebText dataset¹⁸ (22%), two collections of books available online (8% each), and finally, the English Wikipedia

¹⁷<https://commoncrawl.org/the-data>

¹⁸<https://paperswithcode.com/dataset/webtext>

(3%) [33], while Google’s Bard was trained on dialogs from public forums (50%), code documents from programming question/answer sites (12.5%), as well as web documents (6.25% in English and 6.25% in other languages)[186]. However, these descriptions are very high level and do not allow discerning the actual training data which was used.

This opacity raises some issues. First, it is possible that the training data contained copyrighted works (e.g., literary works) [22], which, in turn, might lead the models to output significant parts of said works, eventually resulting in legal issues such as copyright infringement. This issue has prompted researchers to attempt to determine if these models have been trained on copyrighted content and to what extent they can reproduce that content.

For example, in [36], researchers performed *data-archaeology* using a technique called ‘membership inference query’ [171] to infer books that are known to ChatGPT (using GPT-3.5), GPT-4 and BERT. The study used books which, under the U.S. law, are in the public domain (e.g., ‘Alice in Wonderland’ by Lewis Carroll) as well as books that are still under copyright (e.g., ‘Harry Potter and the Sorcerer’s Stone’ by J.K. Rowling and ‘Fifty Shades of Grey’ by E.L. James). In summary, these researchers reported that OpenAI’s models (in particular, GPT-4) have likely been training with copyrighted works. The authors of [94] evaluated multiple LLMs, such as LLaMA, GPT-3.5 and Claude, and determined that the models appear to have ‘memorized’ large chunks of certain literary works, such as ‘Harry Potter and the Sorcerer’s Stone’. While it is possible that these were not memorized from the books, since sometimes fans of these works of fiction will post portions of the books in online discussion forums, the fact that models are able to reproduce their learned material verbatim might still be problematic.

This might result in both legal issues as well as some delay in the adoption of these technologies by people and companies who might be afraid of the resulting legal issues.

Second, we are unable to determine if the dataset was curated in order to avoid biases, incorrect but common practices, and so on.

It should be noted that some models claim transparency in terms of reporting their training data. Two examples are GPT-J, which is reported to having been solely trained on the dataset known as The Pile [194, 193, 73, 72], and StarCoder, which is reported to have been trained on the dataset known as The Stack [111, 99].

Prompt Engineering

A prompt is a a natural language textual description of a task that an AI system should perform. Although current LLMs, such as GPT-3.5, GPT-4 and Bard/LaMDA, are able to perform some tasks from a textual description and without any extra training (i.e., zero-shot learning), research has found that their performance can be improved by the application of multiple techniques, such as one-shot learning (a technique where the model receives a single example case) and few-shot learning (similar to one-shot, but the model is given a small number of example cases) [33].

To better understand these ideas, let's see an example of zero-shot learning in action (model = GPT-4).

User:

```
Classify the following sentence as neutral , positive or
negative :
* The Foosball game was okay .
```

And here's the model's reply:

GPT-4:

```
The sentence "The Foosball game was okay." conveys a
neutral sentiment. It suggests that the experience
was neither particularly good nor bad, implying a
middling or average reaction to the game.
```

The model's pre-training was sufficient to make it 'understand' what a neutral sentiment sentence looks like, and thus it is capable of classifying this sentence without needing extra information.

Let's try another variant of this task:

User:

```
Classify the following sentence as neutral , positive or
negative :
* The Foosball game was beautiful .
```

Let's observe the model's reply:

GPT-4:

```
The sentence "The Foosball game was beautiful." is
classified as positive.
```

Again, the models' pre-training was sufficient to allow it to understand what a positive sentiment sentence looks like.

Now imagine a similar classification task where the classification is not something standard. For example let's consider that we have a list of words, and our goal is to classify them into 3 groups, called A, B and C. We expect group A to contain words which

represent animals, group B to contain the words which represent colours and group C to contain the words representing means of transportation.

User:

Classify the following words into groups A, B and C:
yellow, boat, red, dog, horse, car

Here's GPT-4's reply:

GPT-4:

We can classify the given words into three groups based on common categories:

- **Group A: Colors**
 - Yellow
 - Red
- **Group B: Animals**
 - Dog
 - Horse
- **Group C: Vehicles**
 - Boat
 - Car

This classification sorts the words based on their general types: colors, animals, and vehicles.

In this case, the LLM proposed a logical categorization, but it does not match the desired one. In order to get a result which is closer to our goals, we can try to give the LLM some examples.

Here's an adjusted prompt with one example added (i.e., one-shot learning):

User:

Classify the following words into groups A, B and C:

yellow, boat, red, dog, horse, car

Notes:

- words similar to 'dog' should be placed in group A.
- words similar to 'yellow' should be placed in group B.

And the model's output is now:

GPT-4:

To classify the given words based on the notes provided:

- **Group A** includes words similar to 'dog', which suggests grouping animals or living beings. Hence, 'horse' also belongs in Group A.
- **Group B** includes words similar to 'yellow', which suggests grouping colors. Therefore, 'red' also fits in Group B.
- **Group C** would then logically contain words that do not fit into the categories of living beings or colors. This includes 'boat' and 'car', which are types of vehicles.

So, the classification is:

- **Group A:** dog, horse
- **Group B:** yellow, red
- **Group C:** boat, car

As the chat log shows, the GPT-4 model was able to use the extra information in order to inform its categorization process, and was also able to give a correct reply to the original exercise. We performed the same experience with Google's Gemini, and the results were equivalent ¹⁹.

In this experiment, two examples were given to inform the model: the groups which should contain the words 'dog' and 'yellow'. If we would have given the model a single example, this prompt would be considered one-shot. As we gave it more than one example, it is considered to be few-shot prompting. In classification tasks such as the one in the example, supplying one example per label is common. However, this might not be

¹⁹<https://g.co/gemini/share/99a90d6fc237>

enough. In some cases, more than one example per label might be needed.

With the knowledge that it is possible to influence LLM's behaviour with the addition of information to the prompts, researchers have investigated and developed multiple other prompt engineering techniques. A few notable examples are:

- Chain-of-Thought (CoT) prompting is a technique where the prompt explicitly guides a language model through a series of intermediate steps or reasoning processes to generate more complex or accurate answers [197]. In practical terms, this is a type of few-shot prompting, in which the extra examples present the models with reasoning processes that break the example problem down. This has been shown to elicit the models to 'think' step-by-step which, in turn, leads to improved results.
- Self-consistency - An adaptation of CoT which was developed based on the idea that people think differently and that there are multiple different ways of reaching the same result [195]. To implement this technique, the model should be supplied with the CoT prompt a number of times. Afterwards, the model's reasoning paths are ignored, and the most consistent result among the sample set is selected.
- Role Playing is a prompting technique which aims at augmenting zero-shot attempts at a problem [100]. To apply this technique, users should prefix their prompts with a description of the role which the LLM should assume (e.g., a profession). An example would be: 'From now on, you are a math teacher. Please answer the following math question' [100].
- Emotional stimuli - researchers have demonstrated that it is possible to get better results from LLMs by including prompts which elicit some kind of emotional stimuli [110]. An example of this would be suffixing a prompt with 'Write your answer and give me a confidence score between 0-1 for your answer'.

Limitations

Although they have impressive capacities, LLMs are not without faults. These models are very sensitive to their training data: if the training data contains factual errors or other types of mistakes (e.g., spelling and grammar), it is possible that the models will generate similar mistakes. Moreover, just because a certain completion is likely or probably, it does not mean that it is factually correct [113]. In fact, LLMs are known for generating content that is coherent but factually incorrect. When this happens, it is said that the model 'hallucinated' [113]. An example of this is the generation of false/non-existing references (e.g., book titles and authors) when the model is asked to provide sources for its information [50].

Furthermore, if the training data is somehow biased, it is likely that the model will also display similar biases [27]. Some examples are gender-biases, such as having a higher likelihood of associating certain professions with one of the genders [181], and cognitive

biases (e.g., against certain people groups), possibly leading to unfair decisions [61]. This fact has led some researchers to investigate ways of reducing such biases from LLM's output [61].

In the particular case of computer programming code, it is possible that the models were trained in low quality code [43]. In fact, some studies have determined that LLMs sometimes suggest code with known vulnerabilities [144, 16]. Furthermore, although LLMs such as GPT-4 have been shown to be able to detect some common security vulnerabilities [67], they have demonstrated poor capacity to handle obscure and context-specific vulnerabilities [67].

This makes it important to fact-check any information which is given by GPT, at least when dealing with professional and other-wise serious subjects (e.g., by searching for alternative sources). In the case of computer code, it is advisable to thoroughly test and validate the output before using it in production environments. This can be achieved by means of code reviews [67], unit testing [67], debugging, taking in consideration any relevant best practices [43, 67] among other criteria [67].

Difficulties and Challenges for Researchers

Since these models have a probabilistic component, their behaviour is not deterministic [27, 160]. As such, the same experiment repeated twice might yield different results, which creates some difficulties in terms of reproducing scientific results [160].

Although some models allow controlling the level of randomness, this is not possible in all models or use cases. For example, GPT-based models have an hyper-parameter called 'temperature' [135]. In short, the lower the value of this parameter, the more deterministic the model will behave, since the model will always select the more probable next token [135]. Increasing the value of this parameter will induce more randomness, in the sense that less probable tokens will be selected more frequently [135]. However, this parameter can only be changed when using GPT from the respective API. That being said, it should be noted that ChatGPT has more functionality than the underlying GPT models, since ChatGPT is specialized application of GPT that's fine-tuned for dialog applications [65]. As such, since the majority of end users will probably use the models from the respective chat interfaces, since it's much easier to access and use, it is important to study these interfaces and not only the models. Two common strategies that are being employed by researchers to minimize issues raised by the lack of determinism is the execution of multiple trial attempts for the same task and the execution of multiple similar tasks.

Another challenge arises from the fact that these model's behaviour is not static, since they are under continuous development. For example, research has shown that GPT-4's accuracy in the task of identifying prime vs. composite numbers dropped from 84% in March 2023 to 51% in June 2023 [37]. This lead the same researchers to suggest that regular monitoring of these tools should be performed, in order to evaluate their progress over time.

More recently, some researchers have started to propose strategies to address and mitigate some of these challenges. The authors of [160] proposed an initial set of guidelines for LLM producers, as well as for researchers working with or evaluating LLMs within the scope of Software Engineering research. For example, they recommended that LLM providers make their training data more transparent, by creating API's which allow checking if a certain source was included in the training dataset. Furthermore, they recommend the presentation of versioning information in order to allow users to discern the significance of model version changes. Furthermore, they recommend that researchers evaluating LLMs use multiple runs with the same prompts in order to evaluate the models' output variability. They also recommend that researchers make available execution metadata, such as model version, exact prompts and execution dates.

Educational impact

Some studies [205, 178, 68] have evaluated students' acceptance of ChatGPT using surveys based on frameworks such as the technological Acceptance Model (TAM) [190, 192]. This framework is a theoretical model which attempts to explain how users come to accept and adopt a certain technology. This is done by the means of structured questionnaires which attempt to evaluate how the potential users perceived the technology's usefulness and ease of use, amongst other factors. These studies reported generally positive outlooks towards using this tool: for example, in the first study (N=239), participants graded ChatGPT's perceived usefulness with 67.6% and their behavioral intention to use ChatGPT was 60.9%, while, in the third study (N=277), only 28% of participants had "low to no" intention of using ChatGPT. The third study also examined participants' views on the ethics of using ChatGPT and found that some students had ethical concerns about its impact on academic integrity. Despite those ethical concerns, 75% of students agreed that the tool should be used in accordance with academic integrity guidelines, as a support tool.

2.3.3 Large Language Models' Impact on CSE

The previous section briefly explored these tools' impact on education in general. This section will be focused on their impact on computer science education.

Much research has focused on LLM's capacity to solve exercises typically used in computer science courses, with the goal of determining to which extent LLM's can be used to pass CS courses with minimal or low effort being employed by the students. This has happened across multiple CSE topics, such as introductory programming [153, 158, 164, 163, 162, 199, 203], data-structures [70], object-oriented programming [164, 138, 41, 162, 43], functional programming [149], and information security [123].

Research has also evaluated CS educators' perceptions and opinions about these tools and how to react to their existence [104]. In a study with 20 introductory programming instructors from across 9 countries, researchers evaluated instructors plans to react to LLMs, both in the short-term and in the long-term. In the **short-term**, all inquired

instructors displayed concerns about cheating, which led them to opt for one or more of the following: a) give more weight to exam scores, b) ban the use of AI tools, and, c) expose students to AI’s capabilities and limitations. However, with regards to the long-term plans, instructors were divided in two groups: some of them wanted to **resist** the use of AI tools and continue teaching programming fundamentals, while others wanted to **embrace** these tools by integrating them in their classes, to help both students and instructors [104].

More recently, researchers surveyed 92 students enrolled in an OOP course in order to determine what kind of resources they use to get help during that course and in previous courses [145]. This study was not focused on LLMs, since it evaluated the use of multiple resources, such as (getting help from) colleagues, (paid) tutoring, communities, auto-complete, Copilot and LLMs. The study concluded that 74/92 (84.8%) of the students used some kind of such resource, both before and during the OOP course. Furthermore, LLMs were identified as the most common resource used (before OOP: 50/92, during OOP: 70/92), followed by colleagues (before: 45/92, during: 62/92), communities (before: 42/92, during: 67/92) and auto-complete (before: 34/92, during: 37/92). The study also reported that usage of GitHub Copilot and paid tutoring was considered residual.

2.3.4 Challenges

Programming Courses

Much research has investigated the impact of GPT-based tools, such as Codex, GPT-3.5 and GPT-4 in solving the types of assignments which are usually common in introductory programming courses, which are usually solved with a single function that receives a set of parameters and transforms them into an expected output. Such exercises are usually prescribed using natural language descriptions (e.g., “Create a function that receives an integer array and returns the sum of its elements”). Students have to interpret the textual description of the problem, devise an algorithm to solve it, and then create computer code that implements the algorithm. This is particularly common for introductory programming exercises [11]. For example, in [38], researchers managed to solve 70% of 164 Python programming exercises using code generated by Codex, a GPT-based language model trained on code publicly available in GitHub. In [69], Codex was evaluated using 23 introductory programming assignments, and managed to make the 17th position when ranked alongside 71 students.

Research has also evaluated how these tools handle exercises prescribed in alternative ways to the classical natural language descriptions. One example is [158] focused on Codex’s capacity to handle Python-based Parsons problems [143] and revealed that the model solved 50% of the cases when indentation errors were considered, and 80% when those errors were ignored.

These tools have also demonstrated some success when handling exercises related with more advanced topics, such as recursion, searching, sorting and program efficiency [70].

GPT-3 and GPT-4 have also been shown to be able to handle object-oriented programming exercises [164, 138, 41, 162, 43], although not always following OOP's best practices [41, 43]. For example, [138] evaluated ChatGPT-3.5 and ChatGPT-4 using 80 diverse Java programming exercises, 28 of which were related with OOP. These OOP-related exercises were mostly directed, since they consisted on 1) instantiating and modifying objects considering an API's documentation as well as some example source code, and, 2) implementing classes considering a supplied UML diagram. This study reported that the studied LLMs were able to generate partially correct solutions to the exercises. In [164], researchers investigated how well GPT-3 ("`text-davinci-003`") solves Python programming exercises, including some OOP-related ones. They reported that the LLM solved 84.4% of the OOP-related exercises. However, similarly to the previous study (i.e., [138]), these exercises were very directed, with students receiving direct instructions of which classes, constructors and methods they should create and in which files they should be placed. The same researchers repeated their study using GPT-4 in [162] and reported that the LLM was now able to solve 100% of the OOP-related exercises. Other studies have focused on less directed exercises and on the models' ability to respect OOP's best practices [41, 43]. More concretely, those two studies evaluated GPT-3.5 ("`text-davinci-003`") and GPT-4 using both functional as well as code quality criteria, using assignments which require some level of inference about how classes should be related and in which classes some of methods should be implemented in, and reported that GPT-3.5 was able to pass an average of 68.66% of the functional requirements, but it only passed an average of 50% of the code quality requirements. Furthermore, this study reported that GPT-4 achieved better results, passing 81.44% of the functional requirements and 66.67% of the code quality requirements. Another piece of research evaluated the capacity of GPT-4 with 'vision' support to generate code from UML class diagrams, and reported that the model was able to generate partially correct code for an average of 88.25% of the elements presented in the diagram. The study concluded that model proficiently handles single-class UML diagrams (frequently generating all the required elements), but struggles with multi-class UML diagrams [13].

Although research on Google's Bard is more limited, it has also been the subject of a couple of papers. In [56], researchers studied Bard's capacity to implement single-function Java exercises (which are similar to what is used in introductory programming classes) and reported that Bard solved 53.1% of the single-function exercises, while GPT-3.5 solved 90.6%. The authors of [43] evaluated Bard's capacity to handle multiple real-world object-oriented programming assignments both in terms of functional requirements as well as in terms of code quality, reporting that the model was able obtain an average score of 65.15% in the functional requirements evaluation and 50% in the code quality evaluation.

Research has also demonstrated that these tools are also able to handle multiple programming languages. In particular, that proficiency has been demonstrated for at least two very popular teaching languages, Java and Python [174, 25]. Also, the interaction language does not seem to be a limiting factor for GPT, since some of these studies have

been done in languages other than English, such as Portuguese [41, 43] and Czech [123].

Other CS courses

LLM's impact in CS is not limited to programming. Studies have shown their proficiency in other CS fields, such as, for example computer security [123] and bio-informatics [148].

2.3.5 Current Approaches

Resist a.k.a. Fight!

Some educators have chosen to oppose the transformations introduced by these tools, implementing strategies to curb their utilization [104]. Common strategies are the use of blocking technologies (i.e., prevent students from accessing ChatGPT or Bard's websites from the University's Network), and the use of AI-generated-text detection tools.

While the first approach may work within the university's physical boundaries, it is not assured, since some students might have their own dedicated Internet connections, which will not be subject to the same blocks and filters. Also, that solution will not work outside the university, since students will be able to access these tools from their own homes or from other locations where unfiltered Internet access is available. Filtering the access from within the universities premises also has the potential of creating unfair situations, where students with better economical situations have access to the tools, while students who cannot afford their own Internet connections will be at a disadvantage.

With regards to the second approach, it is not yet clear if it is actually possible to implement such a detector [71]. Researchers have experimented tools such as Measure of Software Similarity (MOSS), to try to detect code generated by the Open-Source GPT-J LLM and reported that, not only did MOSS fail to trigger any plagiarism suspicion, but the generated code is diverse in structure, lacking any particular tells that could inform the development of new detection tools [29]. Also, several news articles have been published about students who had been incorrectly accused of using Generative AI to produce some of their school work [98]. Some news outlets have also reported cases of Universities abandoning Generative AI Detection tools [35, 49, 7].

Moreover, OpenAI – the developers of the GPT models family –, which had a project to develop a classifier for AI-written text, has decided to abandon that project a few months after its original deployment, a decision which was justified with the tool's 'low rate of accuracy' [90].

More recently, studies such as [82] have tried to use code quality and style metrics to detect AI generated code, being able to signal, with some confidence, that the quality of a certain piece of code is much higher than the expected for novice programmers. However, in this thesis' author's opinion, such approaches also have their limitations, since 1) students' coding style tends to evolve over time, and 2), in some more advanced topics and courses, such as Object-Oriented Programming, students' code is expected to be of higher quality, potentially making style detection less effective.

Another suggested approach is to remove certain detail from the assignment's text, in order to make it more difficult for the LLM to solve the problem [191]. This idea is actually supported by older CSE research (1979) which recommended that some exercises left out details in order to force students to consider the possible alternatives for the omitted details and make reasonable decisions [165]. The authors of [191] experimented the performance of Copilot with various levels of suppressed instructions. This study was based on 153 exercises from the CodeCheck website [83]. In the first evaluation, they only fed Copilot with the function signature. In the second evaluation, they also added example test-cases until Copilot was able to give a correct solution. In the first evaluation, Copilot was able to pass 78 of 153 problems (slightly above 50%). In the second evaluation, they analysed the 75 problems which Copilot could not give a solution by augmenting the request with one or more examples. Of those 75, 56 required an average of 1.66 examples. By joining the results of both experiments, this study determined that Copilot was able to solve 134/153 problems (87.6%) requiring 0.7 examples on average. These authors concluded that it is unlikely that LLMs such as Copilot can be fooled by simple reduction of problem details, and recommended that future research focus on other variant of code-writing problems.

Another strategy to fight LLMs is to increase the weights given to proctored evaluations, and proportionally reduce the weight of practical or laboratory evaluation components, which are probably done in conditions more favorable to cheating and abusing from LLMs (e.g., outside the classroom, at home, and so on) [104]. The authors of [208] conducted semi-structured interviews with 18 participants (12 students and 6 instructors) in order to investigate the respective experiences and preferences regarding the use of LLMs in computing courses. Amongst other findings, they reported that students and instructors disagreed on how to adapt the assessment, with instructors wanting to give more weight to proctored evaluations and reduce the weight of lab exercises, while students considered that the weight of programming labs should not be changed, indicating that those materials are valuable to their understanding [208]. In the opinion of this thesis' author, this seems to indicate that performing profound changes to the grading schemes might have negative consequences. One possibility is that significantly reducing the weight of the practical components might reduce students' motivation to work on them, possible meaning that students will make higher efforts on rote memorization than on computing skills development (e.g., computational thinking).

Finally, some educators plan to or are experimenting with ways of prescribing exercises which do not use natural language descriptions [104]. One example is [44], which proposed a new visual notation for describing OOP exercises and projects.

Embrace

Multiple studies have proposed utilizing Large Language Models (LLMs) in Computer Science Education (CSE). The suggested applications are diverse, including classroom integration, content generation (such as explanations), question generation, and the de-

velopment of digital teaching assistants or tutors.

Educators who opted to integrate these technologies within the classroom had plans such as giving personalized help to students, helping instructors with time-consuming tasks, shifting introductory courses from code writing to code reading and critique, creating more open-ended design assignments, and having students work collaboratively with AI [104].

On the topic of helping students and instructors, some research was published with regards to LLM's capacities to generate pedagogical content. An example of this is [108], which compared code explanations generated by LLMs with those created by students and concluded that the LLM-produced explanations are rated as being easier to understand and more accurate than the student-produced ones.

On a similar note, the authors of [60] compared Multiple Choice Questions (MCQs) generated by GPT-4 with those created by humans. This study concluded that GPT-4 is capable of generating MCQs with clear language, a single correct choice, and high-quality distractors [60].

Some researchers have analysed the logs of interactions between students and LLMs. The authors of [96] analyzed the interaction logs of learners aged 10 to 17, recruited from programming bootcamps, and verified how the availability of Codex (a GPT-based LLM) impacted their ability to independently learn Python. They found evidence of learner's self-regulation, with some learners actively adding code to test the AI-generated code, as well as over reliance, with some learners submitting the assignment's instructions in a single prompt. In [153], researchers studied how first year students use Copilot (a GPT-based code generation code, trained on code publicly available on GitHub) to solve a typical assignment on an introductory programming course, and found that these novice students struggle to understand and use Copilot, and are wary of the tool's implications (i.e., such as not being able to understand the generated code), but are optimistic about integrating the tool into their future development workflow. Finally, the authors of [151] analyzed the GPT interactions of students in an upperlevel course on applied logic and formal methods, using an IDE plugin. They reported that the most common interaction types were 1) asking the LLM to write a formal specification, which accounted for 31% of all prompts, with the LLM's response considered relevant in 78% of cases; (2) asking the LLM to explain assignment-related concepts, which made up 15% of prompts, with the LLM's response deemed relevant in 91% of instances; and (3) asking the LLM for a text specification, which accounted for 11% of prompts, with the LLM's response considered relevant in 97% of cases. The authors of [18] asked 80 students with a single semester of programming background to write prompts for 48 problems and found that the students' prompts can be ambiguous to LLMs.

Some research has also been done on the topic of student evaluation. For example, the authors of [75] experimented an evaluation pipeline in which GPT-4 performed a first evaluation of the student's work in a parallel functional programming class. Afterwards, they compared GPT-4's assessments with that of human graders, concluding, among other findings, that 1) the tested LLM has the capacity to accurately identify correct code sub-

missions in 98.21% of the times, 2) human oversight during the assessment process is still needed, since the model sometimes marks correct code as incorrect, and 3), the tested evaluation pipeline resulted in grading time being decreased by 81.2%. LLMs have also been used to objectively grade “Explain in plain English” questions, which have traditionally been challenging to manually grade [55].

Finally, some researchers go a step further and propose changing course contents in order to teach students how to interact and use these tools in a correct and realistic way which prepares them for the future job market. A notable example is the work on Prompt Problems, a novel pedagogical instrument which presents students with diagrams containing input/output pairs which students must analyse in order to determine what the function or program should do [52, 53]. The same researchers also implemented Promptly, a web-based tool which presents students with one of the mentioned diagrams, allows students to write an LLM prompt describing the exercise, and then submits the prompt to the LLM. Finally, it runs the LLM’s output on an AAT, and uses the AAT’s results to evaluate the quality of the prompt [52, 53]. This tool was well received by students, who identified some benefits, such as the exposure to new coding constructs, and enhancement of computational thinking, due to the need to break down the problem in order to write a prompt with precise instructions.

2.3.6 CS Students perception and acceptance of LLMs

A few recent works have evaluated CS students’ opinions on ChatGPT and similar tools [208, 152, 114, 175, 156, 206]. The authors of [208] conducted semi-structured interviews with 18 participants (12 students and 6 instructors) in order to investigate the respective experiences and preferences regarding the use of LLMs in computing courses. Amongst other findings, they reported that 1) students were interested in having LLM-oriented classes (consistent with [175] which surveyed 430 CS MSc students), 2) students and instructors were concerned about the trustworthiness of these tools, as well as the potential for students to become over-reliant on them, 3) instructors had less experience with LLMs than students, and, 4) students and instructors disagreed on adapting the assessment, with instructors wanting to give more weight to proctored evaluations and reduce the weight of lab exercises, while students considered that the weight of programming labs should not be changed, indicating that those materials are valuable to their understanding. In [152], researchers used surveys to evaluate the opinions of 57 instructors and 171 students. A significant amount (72%, approximately 123) of those students were enrolled in computer science or other related majors (e.g., software engineering). Amongst other findings, this research found that 1) students had slightly more experience using generative AI for code generation than instructors, which is coherent with the findings of [208], 2) both instructors and students strongly agree that generative AI cannot replace human instructors, and 3) first-year students seem to prefer getting help from their peers, while upper-level students seem to prefer using generative AI over other resources. The authors of [206]

assessed 41 students' views on using ChatGPT for learning programming, noting benefits like quick responses and debugging aid, and drawbacks such as encouraging laziness and giving wrong answers. Finally, the authors of [114] used a 3-question survey to assess the efficacy of CodeHelp, a tool that acts as an intermediate between the student and GPT, in order to prevent students from over-relying on the LLM. The survey yielded mostly positive student feedback regarding the usefulness of the tool.

Chapter 3

Automatic Assessment Tools for Object-Oriented Programming

3.1 Chapter summary

This chapter addresses the problem of creating an AAT which has the necessary characteristics (or requirements) for OOP education:

- supports multiple-files
- evaluates the code's correctness using unit tests
- checks the code's quality
- supports both exercise-based as well as project-based learning methodologies

This chapter is divided in three parts.

The first part (section 3.2) is strongly related with the publication titled **Drop Project: an automated assessment tool for programming assignments** (published in *Software X*, Q2, 2022). It describes an Open-Source AAT called Drop Project (DP), whose characteristics are suitable for OOP education: it handles submissions with multiple files, supports unit testing instead of input/output matching and it allows checking not only the correctness of the student's code but its quality as well. As such, it fulfils the first three requirements of an AAT for OOP education.

Afterwards, the second part (section 3.3), is based on the publication titled **Bridging the Gap Between Project-Oriented and Exercise-Oriented Automatic Assessment Tools** (published in *Computers*, Q2/Q3, 2024). It explores the idea that some AATs are more suitable for small exercises, while others are better at handling mid-to-large size projects. Building on this idea, we present an IDE plugin which allows students to submit their code to the DP AAT without leaving their work environment. This plugin makes DP more apt at supporting students with the development of small exercises, contributing to easier implementations of mixed learning environments which is an advantage for OOP

education. DP, paired with this plugin, fulfills the fourth requirement of an AAT for OOP education.

Finally, the third part (section 3.4) strongly relates with the publication titled **Seven Years Later: Lessons Learned in Automated Assessment** (presented in the 5th International Computer Programming Education Conference, 2024). Our multiple years of experience with automatic assessment allowed us to learn and gather a set of rules and recommendations about test-case construction and feedback definition. These are rules which we share with new teachers who join our courses and have to prepare new automatically evaluated assignments, since we have observed multiple situations where educators without experience in automatic assessment define assignments which do not give students enough support or which expose too much information, facilitating certain types of cheating. This publication presented those rules and recommendations to the CSE community, in an attempt to raise the community's awareness to these problems.

3.2 Drop Project: An automatic assessment tool for programming assignments

3.2.1 Introduction

This section presents Drop Project (DP), an Automated assessment tool (AAT) which addresses a number of issues found in existing systems, and in which students can submit projects and get detailed feedback on their quality.

DP is in production use since 2018 at the Lusófona University's Bachelor in Computing Engineering¹ (BSc), and its development was motivated by a number of factors. The first was the increase of enrolled students year over year, which was resulting in the grading process getting progressively harder and more time-consuming. It was particularly hard to ensure that all of the students' projects were subject to fair and equivalent assessment by the different teachers. Thus, we decided to use, adapt or implement such a system. Submission flexibility was an important software selection criteria, i.e., the tool should ideally accept both beginner-friendly web upload submissions and submissions via a Git repository, thus being useful for students and courses of all levels. Unfortunately, the AAT landscape did not offer a free and open-source, production-ready solution with this flexibility at the time, which prompted the initial development of DP.

Refer to table 2.1 for an overview of some relevant AATs as well as their limitations and adaptability to our use cases.

DP is based on the principles and technology underlying most continuous integration servers. It relies on Maven [14], the most popular build automation tool for the languages that run on the Java Virtual Machine (JVM), such as Java, Kotlin or Scala. In DP, assignments must conform to the Maven standard format. Thus, they are easily shared and reused across several courses, eliminating a common complaint in AATs: the lack of a

¹<https://www.ulusofona.pt/en/undergraduate/computing-engineering>

uniform format or structure that facilitates the portability of assignments and submissions (e.g., for sharing of assignments between teachers and institutions) [62].

In DP, feedback is based on multiple types of validations. For example:

- Unit tests, to validate functional requirements.
- Code quality, e.g., by checking if all variables are using the expected casing and if the code is respecting the indentation standards.
- Execution time and memory usage, useful for courses where computing efficiency is relevant.

With this tool, students can check with full autonomy if their work satisfies the requirements that have been defined for the assignment. This gives them the opportunity to improve their work before the deadline, as opposed to the more traditional method where students would only receive feedback after the work had been delivered and usually in the form of a grade.

This section makes the following contributions:

- Proposes an open-source, production ready, easy to deploy tool geared towards mid-size, multi-file projects - called Drop Project;
- Presents the results of a student survey evaluating the proposed tool;
- Presents an overview of DP's impact in our educational practice.

This section is organized as follows: the software is thoroughly described in 'Software description' (3.2.2), and a number of examples are presented in 'Illustrative Examples' (3.2.3). Afterwards, in 'Deployment' (3.2.4), we explore how DP can be experimented with and deployed to production. The impact DP has had on our teaching practice is discussed in 'Impact' (3.2.5). Finally, 'Conclusions' (3.2.6) closes the section.

3.2.2 Software description

Assignments and Submissions

The two most important concepts in DP are assignments and submissions. Assignments are created by teachers and represent a problem (e.g., exercise, project, etc.) to be solved by the students. The assignment contains a set of instructions that the students must follow, as well as any code or configuration that is needed in order to evaluate the student's work, such as unit tests. Submissions are performed by students, according to the assignment instructions, with the primary goal of passing all the unit tests.

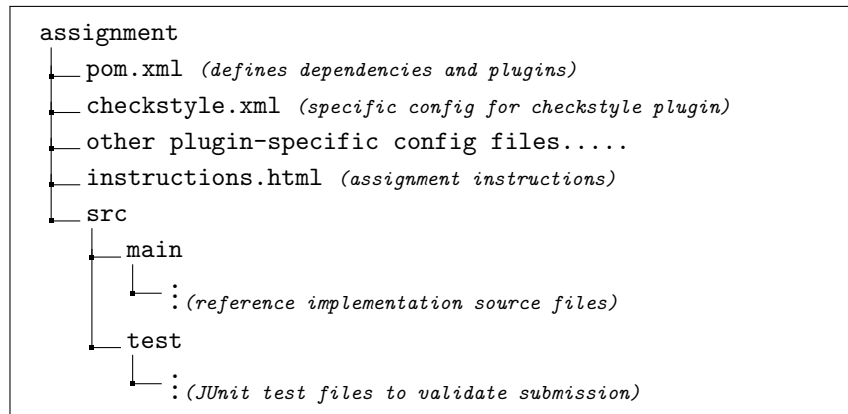


Figure 3.1 – Assignment folder structure.

Workflow

Teachers create assignments as standard Maven projects, as illustrated in Fig. 3.1. The `pom.xml` file defines the Project Object Model, which contains information about the project, as well as any configuration details used by Maven during the build process. This includes dependencies from third-party libraries and the project’s build life cycle. In its most simple form, the build life cycle fetches dependencies, compiles and packages the project. Nonetheless, depending on the plugins configured in the `pom.xml` file, the build cycle can perform dozens of other tasks, such as checking compliance with coding guidelines, unit testing, integration testing, calculating quality metrics, determining test coverage, deploying to the cloud, and more. Some plugins require additional configuration files. For example, the Checkstyle [88] plugin—which can, for example, check if all variable names are at least 3 characters long—requires the `checkstyle.xml` file. The assignment must include at least a JUnit [74] test class in the `/src/test` folder and a `Main` class with the reference implementation, that must pass all the tests. This is important since it reduces the possibility of errors in the test cases, which happen more frequently than desired [87].

Assignments are stored in a private Git repository to which DP must be connected using the standard access keys made available by the repository provider. Any changes made to the assignment are easily updated in DP through a simple refresh button.

A video demonstration of the workflow from both the teacher’s and the student’s point view is available at <https://www.youtube.com/watch?v=65IwIBuMD1E>.

Software Architecture

The application is developed in Kotlin and uses the Spring Boot framework [184]. This framework handles several facets of an application such as request dispatching, authentication, form validation, HTML rendering, error handling, asynchronous execution and integration tests.

DP’s architecture, presented in Fig. 3.5, is based on the MVC (Model View Controller) design pattern and integrated with Spring Boot. Students and teachers interact with the

system via a web application. HTTP(S) requests are handled by a set of controller classes and respective methods. For example, the `AssignmentController` manages the creation and edition of assignments. Services use automatically injected Repository classes that handle all database operations and model classes that mimic the database tables (i.e., for each table there is a corresponding model class). Relevant examples of services are the `AssignmentValidator`, which checks assignments for consistency and correction (e.g., is the reference implementation passing the assignment’s tests?), as well as the `GitClient` and the `ZipService`, which support the Git and Upload submission methods, respectively. The `JacocoResultsParser` handles the parsing of students test coverage results, and the `JUnitResultsParser` manages the parsing of JUnit test results for each submission. Some services interact directly with the file system, such as the `JacocoResultsParser` and `JUnitResultsParser`, which read the files produced by the `MavenInvoker` class. The latter interacts with the Maven Invoker open-source library [166], allowing programmatic execution of Maven builds. This library runs as a service, as shown in Fig. 3.5, and is a core component of DP’s workflow, discussed in the next paragraph. As mentioned above, the model component mimics the database and represents domain concepts such as `Assignment`, `Submission` and `Author` (i.e., `Student` or `Teacher`). Data persistence is achieved through the Java Persistence API (JPA).

Fig. 3.3 presents the evaluation pipeline followed by DP. Maven executes in its own JVM, properly sandboxed by a custom Security Manager that acts at the Java API level (e.g., checking unauthorized file access or system calls). This sandbox has the advantage of isolating student’s code from the teacher’s code, preventing security problems that could arise if the former could invoke the latter [101].

Additionally, DP provides a web interface where (1) students can submit their projects by uploading a ZIP file or connecting to a Git repository, (2) students can see the corresponding build reports, as shown in Fig. 3.4a, and, (3) teachers can create and configure assignments by connecting to a Git repository containing the Maven “master” project. Teachers also have access to several reports such as submissions for a given assignment and the submission history of a given group.

DP’s code is unit tested and, as of July 2021, the test suite is achieving a code coverage above 75%.

Software Functionalities

DP provides teachers and students with the following features:

Supports submissions with multiple code files Students can submit multiple code files. Using multiple files is common a way to improve code organization, particularly in mid-to-large scale projects.

Standard assignment format Assignments are Git repositories with a Maven project, following a standard (i.e., not specific to DP) configuration.

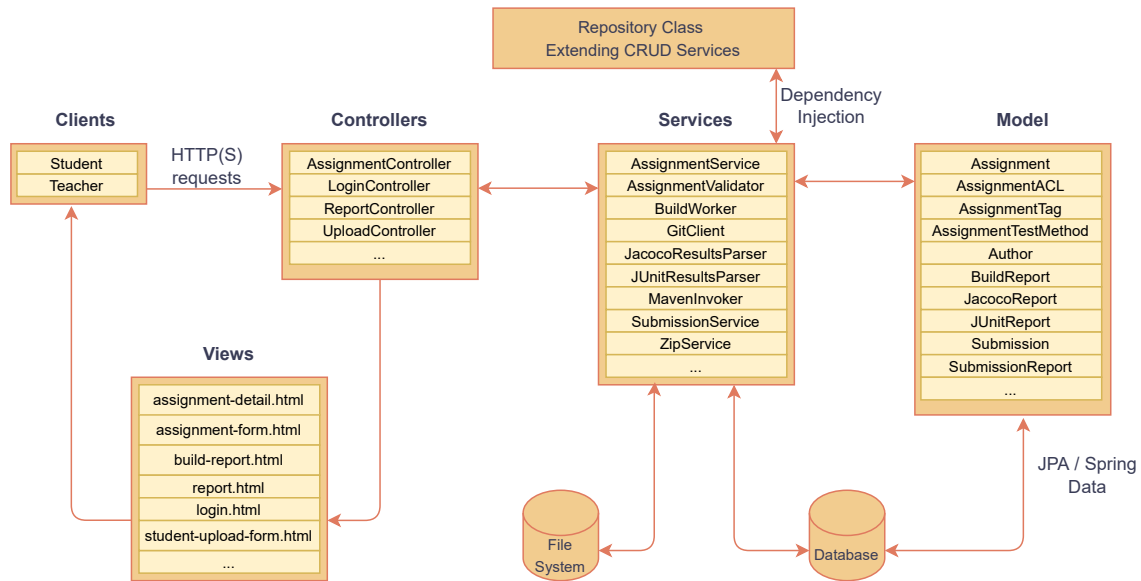


Figure 3.2 – DP’s high level architecture, based on the MVC pattern and integrated with Spring Boot. Only the most relevant classes are presented.

Individual and group submissions Besides standard individual submissions, DP supports submissions performed by groups.

Programming Languages Java and Kotlin are supported out-of-the-box. Since Maven runs on the JVM, integration of other JVM languages, such as Clojure and Scala, is trivial.

Cool-off period Teachers can define an optional waiting period between submissions in order to minimize “trial and error” approaches by the students [23].

Access-control lists DP supports (optional) per assignment access-control lists in order to ensure that only authorized students have access to specific assignments.

Multiple submission methods DP supports Git (e.g., GitHub) and ZIP file submissions, allowing teachers to adapt the assignment submission procedure to the student’s experience level.

Public tests Unit tests for evaluating submissions that give complete feedback to students. Note that only the feedback is public, i.e., the test code itself, is not public.

Private/hidden tests Unit tests for evaluating submissions that offer no feedback or partial feedback to students.

Student’s own tests DP is able to run and evaluate unit tests defined by the students.

Submission timeout detection DP is able to detect and abort the evaluation process when a student’s submission is taking too long to execute, which allows interrupting solutions that have infinite loops or solutions that are not optimized.

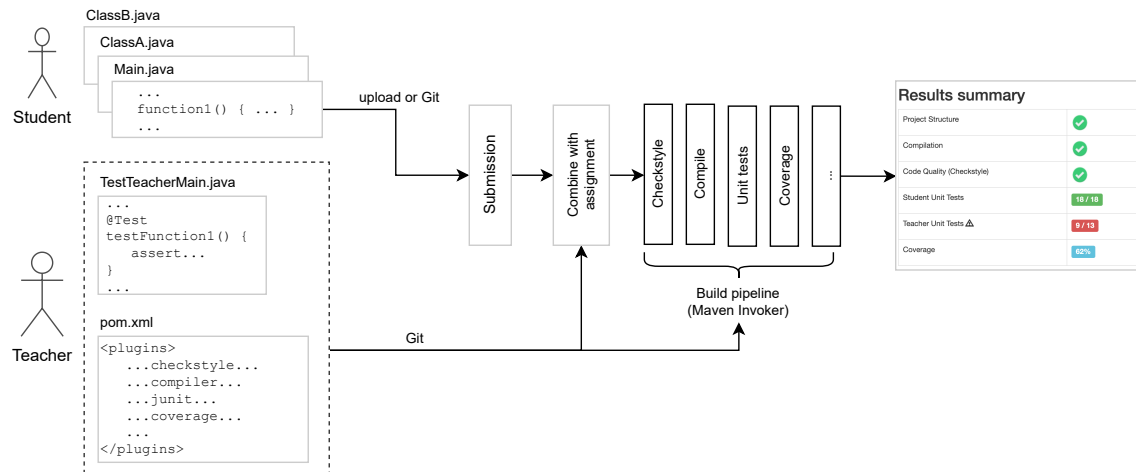


Figure 3.3 – DP’s evaluation pipeline. The student’s submitted files are combined with the assignment’s files and then processed by the build pipeline. In the end, the evaluation report is generated.

Multiple testing approaches Teachers can define assignment’s tests using all strategies available in JUnit, such as unit tests, input/output matching tests, and so on. Since the tests are implemented by code, the teacher has full control on the level of feedback given to students.

Code style evaluation DP is able to check the student’s code using teacher-defined style rules. Style validation can also be configured to forbid certain keywords and apply restrictions to the student’s code, such as disallowing loop-related keywords in exercises that require recursive implementations.

Code coverage evaluation DP can evaluate student’s own tests in terms of coverage of the respective code, thus promoting unit testing.

Execution time DP measures the amount of time a submission takes to run, thus allowing for code efficiency evaluation.

Submission signalling The capability of signalling “interesting” submissions. For example, reporting groups that fail the same tests or groups that pass the majority of tests with a relatively low number of submissions, facilitating plagiarism detection.

Leaderboard Teachers can configure a leaderboard where the students’ projects and respective results are listed. DP supports three types of leaderboard, which differ in the way projects are sorted: number of public tests passed, test coverage percentage and execution time.

Test matrix DP displays a matrix/table where teachers can check which students are passing each unit test, either public or private. This is useful, for example, to detect tests that are too hard or tests that should be reviewed because they might be incorrect.

Submission download Teachers can download student’s submissions in standard Maven format. The submission can then be imported with a single click into any IDE, thus allowing easy debugging of the student’s failing tests.

Data exporter DP can export student’s results to a CSV file², simplifying the grading process and allowing for statistical analysis.

Assignment exportation Assignments and submissions can be exported, allowing them to be ported to other DP installations or to the teachers’ own computers to be subject to further analysis. This is also useful for DP’s developers to test their additions to the tool.

Mark submission as final Teachers can mark which of a student’s submission should be exported. This gives some extra flexibility in deciding which submission will be graded, since the last submission might not be the best one.

Rebuild submission DP is able to rebuild submissions considering the latest test code. This is useful to rebuild students’ submission when bugs in the assignment are found and fixed after the students’ last submissions.

REST API

DP also provides a REST API that can be used to integrate external tools [3]. This API provides functionalities such as assignment listing, assignment presentation, and solution submission. The API implements the same security and access control measures as DP itself. For example, the ‘List of public assignments’ endpoint only returns assignments to which all students have access. To access a private assignment, the student must supply the respective assignment ID, and the API will check if the given student has access to that assignment before returning its contents. If an assignment has a defined “cool-off” period, the API will check if the time for a certain student has passed before allowing that student to perform new submissions.

3.2.3 Illustrative Examples

Several DP reports are displayed in Fig. 3.4. Fig. 3.4a presents a submission report from the student’s point of view. In this example, the submission is validated by Checkstyle, all of the student’s own tests are passing, but only 20 out of 24 teacher tests do so. Note that these 24 tests are the public ones. Fig. 3.4b also shows a submission report, this time from the teacher’s perspective. It contains more information than the student’s version, indicating how many hidden tests the submission is passing and how much code coverage the student’s own tests have reached. Students and teachers also have access to the detailed JUnit report, which contains the reasons for any potential failures (Fig. 3.4c). The

²CSV (comma-separated values) files are plain text files that contain information separated by commas and are easily imported in most spreadsheet applications.

Build report for submission 17655

Assignment: aed-1819-projecto-parte-2 | Submitted: 30-06 21:08:39

Group elements

a2180xxxx	Paulo xxx
a2181xxxx Submitter	Bruno xxxx

Results summary

Project Structure	✓
Compilation	✓
Code Quality (Checkstyle)	✓
Student Unit Tests	17 / 17
Teacher Unit Tests ⚠	20 / 24

(a) A submission report from the student's perspective.

Group elements

a21	Submitter	Student 1
a22		Student 2

Results summary

Project Structure	
Compilation	
Code Quality (Checkstyle)	
Student Unit Tests	
Teacher Unit Tests 	
Teacher Hidden Unit Tests 	

JUnit Summary (Student Tests)

Coverage: 1% (Only visible to teacher)

Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.002 sec

(b) A submission report from the teacher's perspective.

```
JUnit Summary (Teacher Tests)
```

Tests run: 24, Failures: 4, Errors: 0, Time elapsed: 21.816 sec

```
FAILURE: pt.ulusofoa.aed.songsExpert.TestTeacherMainP2.test_11-G
_ET_UNIQUE_SONGS_BY_ARTIST_01
java.lang.AssertionError: GET_UNIQUE_SONGS_BY_ARTIST (0) expecte
d:<25> but was:<43>
    at pt.ulusofoa.aed.songsExpert.TestTeacherMainP2.test_11
_GET_UNIQUE_SONGS_BY_ARTIST_01(TestTeacherMainP2.java:214)

FAILURE: pt.ulusofoa.aed.songsExpert.TestTeacherMainP2.test_15-T
OP_N_ARTISTS_WITH_LOCATION_01
java.lang.AssertionError: TOP_N_ARTISTS_WITH_LOCATION (1)
    at pt.ulusofoa.aed.songsExpert.TestTeacherMainP2.test_15
_TOP_N_ARTISTS_WITH_LOCATION_01(TestTeacherMainP2.java:400)
```

(c) Excerpt of JUnit report (student's perspective).

[illegible]

(d) Test matrix for a recent assignment (teacher's perspective). Each line displays a student and their result for each unit test.

Figure 3.4 – Several example reports from DP. On the left side are two reports from the student’s perspective, while the right side shows two reports from the point of view of the teachers. The displayed student IDs have been anonymised.

displayed information is fully controlled by the teachers when defining the assignment’s tests, taking advantage of JUnit’s API. The feedback can range from the standard JUnit output, e.g., “Expected 10 but was 20”, to more textual hints and/or tips, such as “Function `getMovies()` returned too many elements. Beware of duplicated entries in the input file.”. Listing 3.1 shows an example JUnit test similar to the ones we define to validate student’s submissions.

Listing 3.1 – An example *JUnit* test for a project where students implement a board game. The messages are only displayed when the respective assertion fails.

```
@Test
public static void testMovePlayer001() {
    GameManager manager = new GameManager("game-001.txt");
    assertEquals("The function getCreatures() returned a list of incorrect
        size", 2, manager.getCreatures().size());
    assertFalse("Cannot move creatures to the outside of the board!",
        manager.movePlayer(0, 0, -1, 0));
    assertTrue("A Turtle should be able to move from 0,0 to 0,1",
        manager.movePlayer(0, 0, 0, 1));
}
```

Sample assignments with introductory programming exercises for both Java ³ and Kotlin ⁴ are available from DP’s GitHub account.

3.2.4 Deployment

DP’s code includes an executable demo containing an in-memory database with fake/test data (teachers, students and submissions), thus allowing the experimentation of the system with minimal time investment. This demo can be executed either by clicking a “Deploy to Heroku” button available on DP’s GitHub repository which deploys the demo to Heroku, a cloud provider⁵⁶ or by installing a Docker [30] image which can be used to run the demo locally or to install a production-ready version of DP on a server.

3.2.5 Impact

In the 2018/19 edition of Algorithms and Data Structures, an anonymous student survey was conducted in order to understand the students’ perception of using the tool, considering aspects such as usability, motivation and fairness. The survey was optional and was completed by 22 students after the main project had been graded. Students were asked to specify their level of agreement with the statements listed in Table 3.1 using a 5-point Likert scale (1=Strongly disagree, 5=Strongly agree). In general, the majority of students agree with the statements, indicating that DP has a positive impact on their learning experience. This is particularly clear in the final question, where most students strongly agree with the prospect of using Drop Project again.

DP is currently being used in four course units of the aforementioned BSc: Programming Fundamentals (1st year, 1st semester), Algorithms and Data Structures (1st year,

³<https://github.com/drop-project-edu/sampleJavaAssignment>

⁴<https://github.com/drop-project-edu/sampleKotlinAssignment>.

⁵<https://www.heroku.com>.

⁶As of August 2024, Heroku continues to exist, but does not provide free accounts anymore. As such, this button is not available anymore. This information was accurate in 2022, when the respective publication was originally submitted, reviewed and published. We have chosen to retain this information in order to provide a historical perspective on the tool’s capacity at that time.

Statements evaluated	Mean response
1. I felt more confident in my programming abilities	3.36
2. I felt the grading was fair	3.64
3. I didn't need much teacher support to complete the project	3.77
4. It increased my fondness of programming	4.09
5. I am looking forward to use this tool again	4.82

Table 3.1 – Survey results evaluating the students' experience with Drop Project using a Likert scale (1-Strongly disagree, 5-Strongly agree)

Academic year	Metric	Course units		
		PF	ADS	PL2
2019/20	Students	115	72	71
	Submissions	4188	1901	1544
	Teacher Tests	28	13	24
	Cool-off (min.)	5	45	60
2020/21	Students	112	108	88
	Submissions	3810	2935	2705
	Teacher Tests	28	24	25
	Cool-off (min.)	0	30	30

Table 3.2 – Project submission metrics for the projects of each course designed around DP that is described in the 'Impact' section, considering the 2019/20 and 2020/21 academic years. The presented courses are Programming Fundamentals (PF), Algorithms and Data-Structures (ADS), and Programming Languages II (PL2). Only the submissions for the course's main project / assignment are presented.

2nd semester), Programming Languages II⁷ (2nd year, 1st semester), and Mobile Computing (3rd year, 2nd semester). DP is a core pedagogical tool in the first three of these courses, which are heavily built on top of it. It has been successfully used to support multiple types of pedagogical activities, such as classroom exercises, homework assignments, short class assessments, projects (larger evaluations that usually span weeks or months), and project defenses (small evaluations where the students have to show mastery of their own project's code). As an example, in the 2020/21 edition of Algorithms and Data Structures, a total of 40 assignments were devised. Table 3.2 presents statistical information about submissions performed in the context of the courses' main assignments/projects in the last two academic years. These numbers clearly demonstrate the impact that DP had on these courses, and consequently, on our teaching and assessment practices.

The tool's automated and immediate rich feedback helps and guides the students—as demonstrated by the survey results presented in Table 3.1— while also allowing teachers to understand where students are struggling, facilitating teacher intervention and student support, while reducing the evaluation workload. This has allowed us to progressively increase the difficulty levels of the courses' projects. Also, the grading process has become

⁷A course mostly dedicated to teaching Object-Oriented Programming and Design

more efficient and consistent between different teachers. Furthermore, every interaction between the students and DP is fully logged. Thus, DP's submission database contains data that can be mined to study the way students use the system. This data has the potential to be used in multiple research avenues, such as identification of errors in the teacher's tests, plagiarism detection, authorship attribution and/or identification of struggling students, in a similar fashion to what was done in references [115, 177, 58], for example.

3.2.6 Conclusions

This section presented DP, an open-source AAT with a wide range of features that can be used to support programming courses of varying complexity and students with different backgrounds. The usage of DP has allowed us to reduce evaluation workload, improve grading consistency and augment the project's difficulty levels.

3.3 Bridging the Gap Between Project-Oriented and Exercise-Oriented Automatic Assessment Tools

3.3.1 Introduction

Courses that employ Automatic Assessment may follow different pedagogical approaches. Some courses use an exercise-based approach, requiring students to solve a number of small assignments (e.g., weekly) [134, 109], while others use a project-based methodology, where students develop larger projects that can take multiple weeks or even months to complete [118, 109]. This duality is also observable in many AATs available in the market or described in the literature. For example, most web-based AATs are better suited for small exercises, since larger projects usually require multiple files. Although some web editors support multiple files, we argue that this is not ideal from a usability and tool perspective—not to mention their inadequacy in preparing students for the job market, as they create artificial environments that do not compare with the software development workflows commonly found in professional settings. On the other hand, AATs designed for project assessment often require impractical submission legwork for the grading of small exercises.

In turn, several courses follow a mixed approach, using both small exercises and larger projects as teaching and assessment methods [31, 109, 63]. This is the case for several courses taught by the authors, where automatic assessment is performed with Drop Project (DP), an AAT originally designed to support the development and grading of larger projects. Furthermore, educators using this teaching approach sometimes have to use multiple AATs within the same course [63], requiring both students and teachers to interact with multiple systems.

DP supports submissions via web upload or using a Git repository, requiring the inclusion of an AUTHORS.txt file with the name of the student or students. Additionally,

in the web upload method, code and supporting files must be submitted in a ZIP file. Therefore, DP is not ideal from a usability perspective in scenarios involving smaller exercises. For example, one of our courses employs a ‘mini-test’ in which students are asked to implement several functions in one or—at most—two files. These ‘mini-tests’ are taken during laboratory classes, and students have up to one hour to solve them. Students submit their solutions through the web upload method, since version control with Git is yet to be introduced. Also, given the pressure of being assessed, students sometimes make small, time-wasting mistakes, potentially resulting in grades that do not reflect their actual level of knowledge. Examples of such mistakes include creating the `AUTHORS.txt` file with an incorrect format or problems creating the ZIP file, which easily lead to precious minutes being wasted.

To streamline this procedure and bridge the gap between a project-oriented AAT and its exercise-oriented utilization, we propose the DP Plugin for the IntelliJ IDEA IDE. This plugin automates submission-related perfunctory tasks, such as creating the required documentation files and compressing and uploading the solution, while displaying assignment instructions and submission reports directly within the IDE. The plugin reduces the cost of context switching, allowing students to focus on the exercises without leaving the IDE. The proposed plugin was developed using an agile methodology, with students directly involved in the coding and testing process. This work aims to address whether it is possible to improve the usability of a project-oriented AAT via an IDE plugin, enhancing its capability in assisting students in solving small exercises.

This section makes the following contributions:

- Proposes the idea that some AATs for small exercises should have different characteristics than AATs for mid-sized projects.
- Presents a new open source, production ready tool, called Drop Project Plugin, which allows students to submit their work to the DP open source AAT, with reduced manual work and without leaving their work environment (IDE).
- Presents the results of a student survey which evaluates the proposed tool.

This section is organized as follows. The DP Plugin for IntelliJ IDEA is introduced in ‘Software description’ (3.3.2), which provides technical details and illustrates how the plugin aims to address the aforementioned issues with DP by keeping students engaged within the IDE and minimizing cursory submission tasks. Afterwards, ‘Research Methods’ (3.3.3) describes the research methodology, which consists of a student survey to assess the plugin’s impact on students’ assignment submission experiences. The results of this survey are presented in ‘Results’ (3.3.4), while ‘Discussion and Limitations’ (3.3.5) offers a discussion of these results, considering their implications and potential limitations. Finally, ‘Conclusions’ (3.3.6) summarizes our findings and offers directions for future research.

3.3.2 Software description

The presented DP Plugin for IntelliJ IDEA leverages DP’s REST API, providing an abstraction layer on top of DP’s web upload submission method, simplifying parts of this process. Figure 3.5 presents an overview of how the various components integrate and communicate with each other.

The plugin is integrated into IntelliJ IDEA’s main window (Figure 3.6), allowing students to simultaneously view their code, the assignment instructions (Figure 3.7), and the assessment results of their latest submission (Figure 3.8). The plugin uses assignments created for DP [48]. Therefore, there is no additional assignment creation overhead for teachers already familiar with DP, while students can still use DP’s standard submission methods if they so desire.

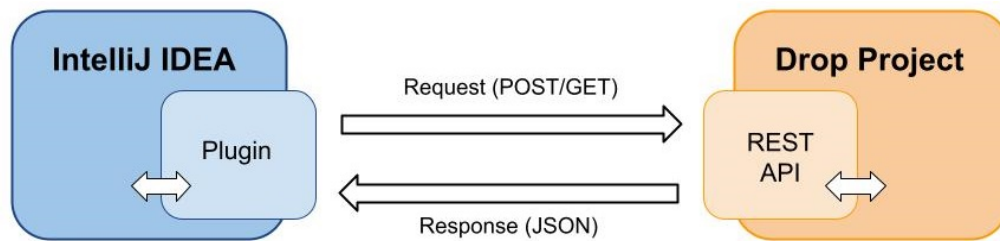


Figure 3.5 – Overview of the solution’s data flow. The plugin integrates with IntelliJ IDEA and connects to DP via its REST API.

The plugin also comes with its own security measures. For instance, the feature that updates the assignment list will only make an API request if a specific number of seconds has passed since the last update request.

The plugin was developed during the 2022/23 academic year. In both semesters, development versions of the plugin were deployed and distributed among students during selected classes of specific courses to test and obtain feedback. These efforts yielded a production-ready version of the plugin, which is available for download at IntelliJ’s marketplace [2]. The tool’s source code is also available on GitHub [4]. A video demonstration of the proposed plugin is available [21].

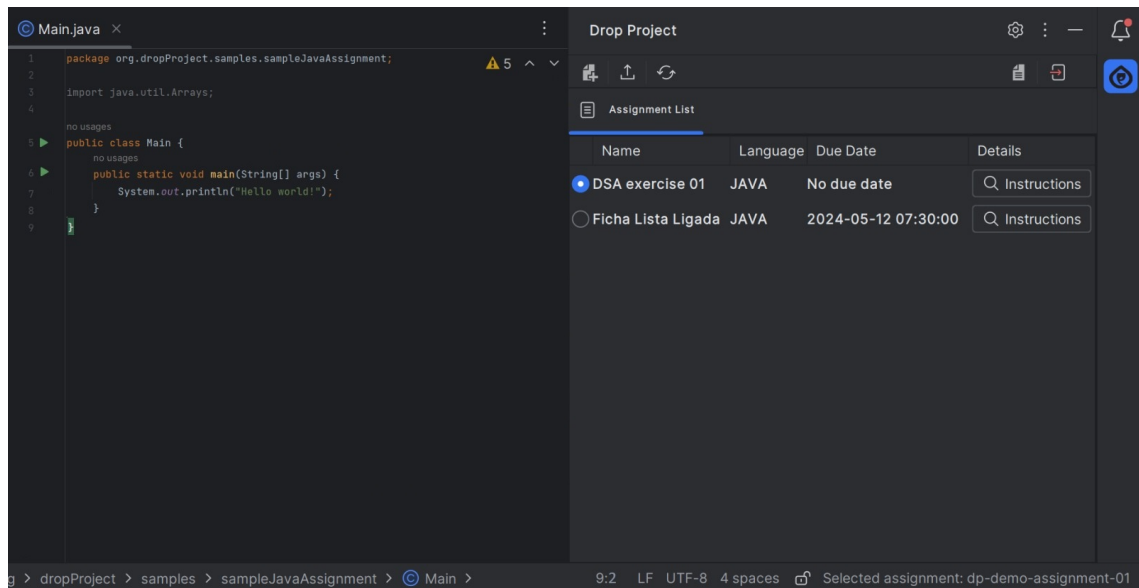


Figure 3.6 – The plugin integrated into IntelliJ IDEA’s main window. The plugin lists the assignments available to the student. The ‘Instructions’ button allows students to access the instructions for each available assignment (see Figure 3.7).

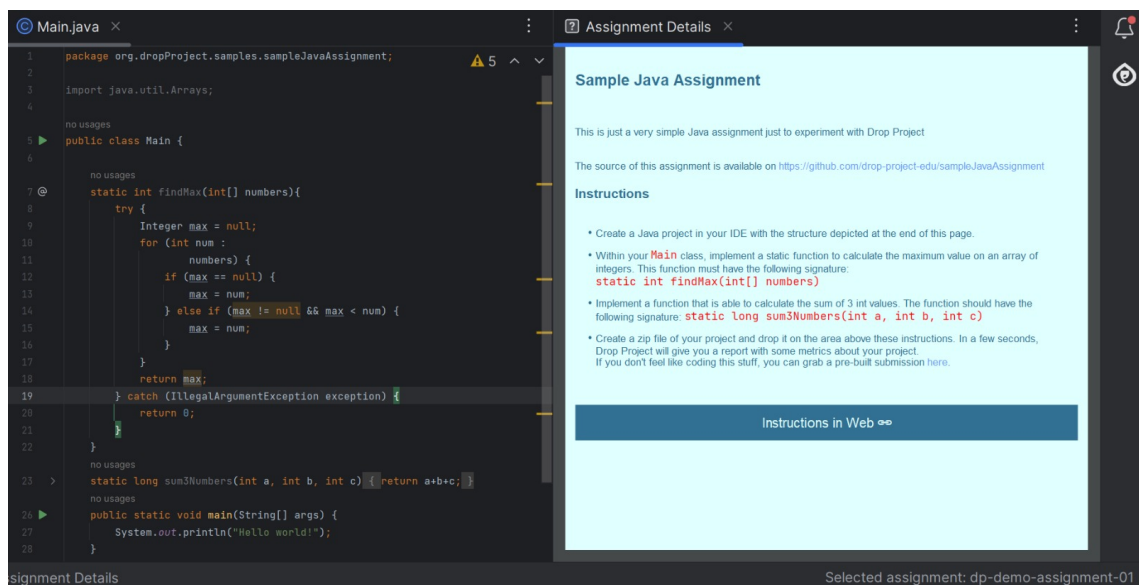


Figure 3.7 – The assignment instructions inform students of the tasks they need to perform. Students can see their code and the assignment instructions in the same window, reducing the need for context switching. Note that the comment about the ZIP file creation is not relevant for users of the plugin, but the assignment (and its instructions) remain the same that is displayed in the web version.

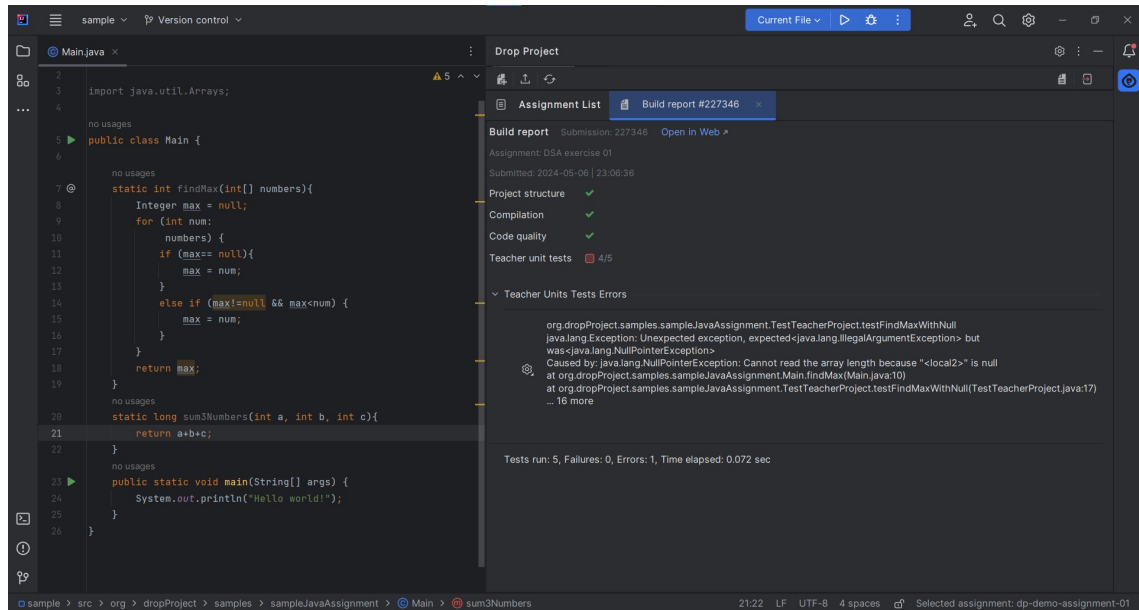


Figure 3.8 – The submission report informs students of their progress and presents them with any unit test failures. Students can see their code and its issues in the same window.

3.3.3 Research Methods

A student survey was conducted to evaluate the plugin in the context of a Data Structures and Algorithms course during the second semester of the 2022/23 academic year, i.e., during the final stages of the plugin’s development. The survey comprised both qualitative and quantitative questions. The questions aimed to assess students’ perceptions regarding various aspects of the plugin, including its impact on reducing submission time, students’ confidence in the plugin, the value of having specific reporting functionalities integrated into the IDE, and the psychological impact of using the plugin in an evaluation environment. The research context is further detailed in Section 3.3.3, while the survey questions are presented in Section 3.3.3.

Research Context

The Data Structures and Algorithms course follows a mixed approach, utilizing both PB and EB learning methodologies, with DP used for automatic assessment. All students in this course have interacted with DP, using its web interface, as DP is also used in the previous programming course of the curriculum (i.e., Programming Fundamentals⁸).

This course employs multiple evaluation mechanisms. One of them—weekly assignments—aims to promote student participation and engagement, contributing 5% to the final course grade. The course also includes ‘mini-tests’, which are one-hour assessments where students implement various related data structures and functions under the supervision of a teacher. Each ‘mini-test’ has a weight of 6.75% in the final course grade.

⁸Programming Fundamentals also utilizes both PB and EB learning methodologies.

Survey Questions

Students were divided into the following two main groups, with specific questions for each group:

Adopters: Students who identified as regular plugin users;

Non-adopters: Students who reported not having used the plugin.

Students who self-identified as regular plugin users (i.e., adopters) were asked multiple plugin-specific questions, with answers given on a 5-level Likert scale, as shown in Table 3.3. Considering that students who self-identified as regular plugin users may not have actually used the plugin during the ‘mini-test’ assessment, questions 7 and 8 in Table 3.3 were only available for students who did so.

Table 3.3 – Survey questions for students who adopted the plugin. Questions 1–6 were answered by all adopter students, while questions 7–8 were only answered by students who used the plugin during the second ‘mini-test’. Questions were answered on a 5-level Likert scale.

ID	Question
1	Rate the plugin’s effectiveness in simplifying the DP submission process.
2	Rate the plugin in terms of saving time and/or effort when compared to the traditional method (web upload).
3	Rate the relevance of the automatic creation of the AUTHORS.txt file and the automatic submission of the ZIP file with the ‘/src’ folder and AUTHORS.txt.
4	Rate your confidence in submitting exercises through the plugin.
5	Rate the positive effects on your productivity from the ability to see assignment instructions directly in the IDE.
6	Rate the positive effects on your productivity from the ability to view a submission’s build report directly in the IDE.
7	Rate the contribution of using the Plugin in reducing anxiety and/or nervousness during the ‘mini-test’.
8	Rate the contribution of using the Plugin in improving your performance during the ‘mini-test’.

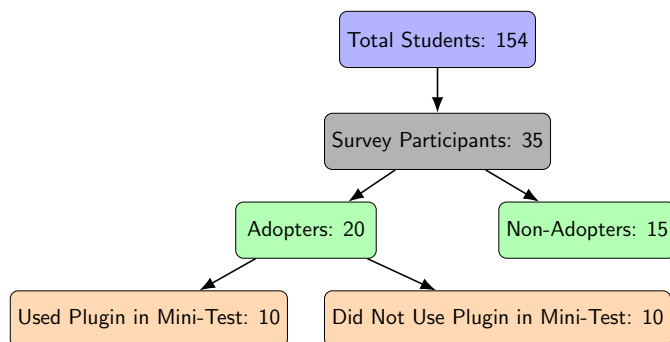
To gain additional insights into plugin usage (or lack thereof) during the semester, three open-ended questions (OE1–OE3), as shown in Table 3.4, were posed to specific student groups. First, to understand why some students who self-identified as plugin users (i.e., adopters) did not use the plugin during the ‘mini-test’, we asked for their reasons (open-ended question OE1, Table 3.4). Second, all adopters answered a qualitative question to leave any relevant comments (open-ended question OE2, Table 3.4). Finally, we asked the students who had not tried the plugin (i.e., non-adopters) for their reasons for not doing so (open-ended question OE3, Table 3.4).

Table 3.4 – Open-ended questions posed to students in the specified target groups.

ID	Question	Target Student Group
OE1	If you did not use the DP Student Plugin in the ‘mini-test’: explain why.	Adopters that did not use the plugin during the ‘mini-test’.
OE2	Leave any comment that you find relevant.	Adopters.
OE3	If you do not use the plugin, explain why.	Non-adopters.

3.3.4 Results

The survey was conducted during the last two weeks of the second semester of the 2022/23 academic year. Out of a total of 154 students, 35 participated in the survey (22.7%). Among these respondents, 20 were identified as adopters of the plugin, while the remaining 15 were non-adopters. Furthermore, of the 20 adopters, half (10 students) used the plugin during the ‘mini-test’, whereas the other half did not. Figure 3.9 summarizes this breakdown.

**Figure 3.9** – Hierarchical breakdown of student participation and plugin adoption based on the survey responses.

Impact of the Plugin on Adopters

Table 3.5 shows the distribution and summary statistics for the replies given by adopters of the plugin. Replies to questions 1–6, which tend toward the highest possible evaluation, indicate that students found the plugin useful. Questions 1–4 have mean scores of at least 4.5, all with modes equal to 5. However, questions 5 and 6—regarding the assignment instructions and submission report displayed within the IDE—had slightly lower scores, with averages of 3.8 (mode 4) and 4.1 (mode 5), respectively. Students do not seem to greatly value having this information within the IDE, particularly the assignment instructions. We believe this is due to the richer formatting of these instructions in the web environment, which displays information such as required function prototypes in a colored format.

Question 7 asked if the plugin helps reduce students' anxiety and/or nervousness during the 'mini-test'. The mean score given by the participants was 3.6, indicating a generally positive response towards the plugin's efficacy. The standard deviation of 1.173 and the histogram distribution denote some variability in the participants' experiences and perceptions. Interestingly, the data were bimodal, with modes at both 3 and 5. This indicates that while a significant number of participants felt the plugin had a moderate effect (rating of 3), there was also a substantial group that found it highly effective (rating of 5) in reducing anxiety and nervousness. As for question 8, which focused on the plugin's effect on the students' performance, the mean response was 3.7, suggesting that many participants felt the plugin had a positive impact. The longer-tailed distribution (standard deviation of 1.159) indicates a moderate spread in the responses, meaning that while many participants agreed with the statement, there was still some variability in opinions. Notably, both options 4 and 5 were selected with the same frequency above all other options. This indicates that a significant number of participants either agreed or strongly agreed with the statement, highlighting the perceived effectiveness of the plugin during the 'mini-test'.

Table 3.5 – Histogram and summary statistics for the questions posed to plugin adopters. Questions 1–6 were answered by the 20 students who reported using the plugin, while questions 7–8 were only answered by the 10 students who reported using the plugin during the second 'mini-test'. All questions were answered using a 5-level Likert scale. Histograms are normalized to the maximum count value for visualization purposes.

ID	Histogram	Count	Mean	Std. Dev.	Mode
1		20	4.6	0.680	5
2		20	4.5	0.827	5
3		20	4.3	0.978	5
4		20	4.6	0.598	5
5		20	3.8	1.105	4
6		20	4.1	1.160	5
7		10	3.6	1.173	3; 5
8		10	3.7	1.159	4; 5

Reasons for Not Using the Plugin during the 'Mini-Test'

Nine out of ten adopter students who did not use the plugin during the 'mini-test' replied to question OE1 (Table 3.4). These replies were analyzed qualitatively to find common reasons for not using the plugin during the assessment.

The predominant topic found in the responses centered around issues arising from the compatibility between the version of IntelliJ IDEA that certain students were using and the plugin itself. This concern was raised by three students, leading them to opt against utilizing the plugin during the assessment. The remaining six students reported different issues that could not be grouped into any specific topic. For example, one student

mentioned, *“I didn’t think it was necessary and I wanted to be confident that I had sent it correctly and be able to see the results and errors immediately after the page loaded”*. Another student commented, *“I had to log in, and the instructions in the IDE are not as optimized and easy to read”*, while another student indicated, *“I changed computers and forgot to install it”*. Notably, one student commented on the instructions’ readability, which is consistent with the lower mean result of the respective qualitative question (i.e., question 5, mean = 3.8).

Open-Ended Comments about the Plugin

A total of 10 students (out of 20 adopters) answered question OE2 (Table 3.4). Again, students’ responses were analyzed qualitatively to find common topics. The following three recurring topics were found: (1) positive comments about the plugin (e.g., “useful”, “simplifies”, “saves time”) (four students), (2) suggestions for improving the formatting of the assignment instructions (two students), and (3) suggestions to avoid the need to log in again when changing assignments and/or when re-opening the IDE (two students). It should be noted that the suggestion about saving login data between uses has since been implemented.

Reasons for Not Adopting the Plugin

Thirteen out of 15 students replied to question OE3 (Table 3.4). A qualitative analysis highlighted the following two main topics: (1) “I’m not interested in it”/“I don’t feel the need” (six students) and (2) “I don’t know how to” (four students). An example reply that falls under the first topic is, “I never thought it was necessary, although I give credit and think it is useful”, while an example response for the second topic is, “I don’t understand very well how to use it. I think a video explanation would be useful”.

3.3.5 Discussion and Limitations

As the evaluation survey shows, the tool was well received by our students, who identified a number of positive effects, such as time savings. Students also provided interesting suggestions, some of which have already been implemented, such as the plugin remembering login details. Our findings suggest that an IDE plugin does improve the usability of a project-oriented AAT for small exercises, thus informing the design of future AATs suitable for mixed learning environments combining PB and EB assessment. The positive reception of the plugin highlights its potential to enhance the learning experience by streamlining workflows and reducing the time required for administrative tasks (e.g., ZIP file creation), allowing students to focus more on learning the material.

Results also show that there are two main types of non-adopters, namely (1) students who are happy and/or comfortable using DP’s web interface to the point that they are not open to trying new interaction formats and (2) students who need some help in order to be able to use the plugin. For the former, we believe they are acting out of habit,

since their initial exposure to DP was through the web interface. This will likely change in the future, as students also have the plugin available in the preceding Programming Fundamentals course. Regarding the latter group, who need more help, a video explaining the plugin’s installation and usage has since been made available [21].

Moreover, the integration of student suggestions into the plugin development process demonstrates the value of student feedback in creating more effective educational tools. The implementation of simple features such as remembering login details shows how even minor enhancements can improve the overall user experience, potentially leading to a more user-friendly and efficient learning environment. Additionally, student feedback raises awareness of the need for documentation (e.g., demonstration videos) to help learners use these educational tools effectively.

Given these results, we believe the proposed IDE plugin effectively extends DP to be a more practical AAT for small exercises, bridging the gap between its PB roots and its EB usage. Although other AATs, such as Mooshak and Web-CAT have proposed IDE plugins, Mooshak’s IDE plugin—Raccode—simply enhances the existing EB capabilities of Mooshak, itself an EB AAT. Web-CAT, a PB-oriented AAT, does provide an IDE plugin for improving its EB capabilities. However, the issues with Web-CAT—poorly organized code, out-of-date binaries, and missing documentation—make it a difficult choice as a modern and versatile AAT.

Nonetheless, our work presents several limitations. First, while the survey indicates that the majority of students find the plugin useful and believe it improves their workflows, the number of students who answered the survey was relatively small, which presents some threats to validity. Also, since participation was optional, there might have been some self-selection bias, as students who opted to participate in the survey might have been the ones more interested or motivated to discuss the plugin. However, we do not believe this to be the case, since approximately 42.5% of the responses were from students who did not use the plugin.

Furthermore, a student survey may not be sufficient to accurately measure the pedagogical impact of the proposed plugin. To gain a more comprehensive understanding and increase confidence in the plugin’s pedagogical value, it would be beneficial to complement the current survey data with other research methods, such as direct classroom observations and analysis of students’ academic results over time.

3.3.6 Conclusions

This section presented the DP Plugin for IntelliJ IDEA, a companion tool for the DP AAT, with the goal of making it more apt to handle small exercises used in EB learning. As the evaluation survey shows, the tool was well received by our students, who identified a number of positive effects, such as time savings. Students also provided interesting suggestions, some of which have already been implemented, such as the plugin remembering login details. Our findings suggest that an IDE plugin does improve the usability

of a project-oriented AAT for small exercises, thus informing the design of future AATs suitable for mixed learning environments combining PB and EB assessment.

Despite the promising feedback on the plugin's effectiveness, our study has some limitations. The small number of survey participants raises concerns about validity, and the optional nature of the survey could lead to self-selection bias. However, the significant number of non-adopters among respondents helps mitigate this concern. Additionally, a student survey alone may not fully capture the plugin's educational impact. Future research should include direct classroom observations and long-term academic performance analysis to provide a more thorough assessment of the plugin's pedagogical value.

3.3.7 Data Availability

A CSV file with the survey replies is available [46].

3.4 Lessons Learned in Automated Assessment

3.4.1 Introduction

We have been using AATs since 2017 for multiple types of assessment, from small formative exercises to larger scale projects which have a large weight on the students' grades. Furthermore, we have employed automatic assessment in multiple courses, ranging from Introduction to Programming, Data-Structures and Algorithms, and Object-Oriented Programming (OOP).

Our team employs a common practice in terms of automatic assessment. The starting point of the process is a text-based description of the exercise(s) or project to solve. This description is sometimes complemented with auxiliary diagrams containing mathematical formulas, UML diagrams, and so on. Besides presenting the description, the assignment also informs students about any mandatory files, classes and/or functions, thus defining an Application Programming Interface (API). Finally, when students submit a solution⁹, their code is validated by a set of teacher-defined unit tests. For example, here's the assignment text for a typical exercise used in introductory programming courses: *Implement the function `static long sum(int n1, int n2)` which should return the sum of its two arguments.* Listing 3.2 shows an unit test which could be used to validate this assignment.

⁹Assignments allow multiple submissions, promoting that students improve their work and learning outcomes.

Listing 3.2 – An example *JUnit* test for an exercise where students must implement a function which sums two integers. The feedback messages are only displayed when the respective assertion fails.

```
1 @Test
2 public void testSum() {
3     String feedback = "returned the wrong value";
4     assertEquals(5, Main.sum(1, 4), "sum(1, 4) " + feedback);
5     assertEquals(10, Main.sum(8, 2), "sum(8, 2) " + feedback);
6 }
```

A more complex OOP exercise could be the following: (...) *The GameManager class must have a static List<Player> getPlayers() function which returns a list with the players that are participating in the current game.* Listing 3.3 presents an example of an unit test for validating that requirement.

Listing 3.3 – An example *JUnit* test for an OOP project where students must implement a board game. The messages are only displayed when the respective assertion fails.

```
1 @Test
2 public void testGetPlayers01() {
3     GameManager manager = new GameManager("game-board-with-2-players.txt");
4     List<Player> players = manager.getPlayers();
5     assertNotNull(players, "getPlayers() should not return null");
6     String feedback = "getPlayers() returned a list of the wrong size";
7     assertEquals(2, players.size(), feedback);
8 }
```

We opted for this unit-tests-based approach because it is a more robust way of testing than the alternative of using ‘output matching’. Also, it is closer to current industry practices, with the added benefit of familiarising students with the types of tools that they will more likely use in their future careers.

Our experience over these last 7 years showed us that certain testing practices or inadequate feedback can lead students to request extra support from the teaching staff, meaning that their autonomy is reduced and that valuable teacher time is consumed. Consequently, this undermines two key benefits of employing AATs: fostering student autonomy and optimizing teacher resource allocation. Furthermore, certain testing patterns can result in assessments which are too strict and even unfair, either by letting students receive better grades than they should – for example, by considering hardcoded solutions as correct – or by penalizing them too much – for example, by not informing students when their code fails due to minor formatting issues.

From our experience, we have derived a set of recommendations which we share with new teaching assistants who join our courses and have to prepare new auto-graded assignments.

This section presents those lessons to the Computer Science Education (CSE) commu-

nity. We believe this information might be useful to other teachers and instructors who wish to use start using or already use AATs.

Finally, the recent advent of Large Language Models (LLMs) – artificial intelligence tools with capacity to generate computer code from natural language descriptions – such as OpenAI’s GPT and Google’s Bard has generated some controversy in the CSE community, due to the risk of producing graduates who are weak in fundamental computing skills such as computational thinking and program design [104]. While LLMs pose certain risks, they also offer new opportunities for the CSE community. We explore how integrating AATs with LLMs can generate beneficial possibilities.

This section makes the following contributions:

- Presents a list of recommendations for defining unit tests for automatic assessment, aimed at enhancing student autonomy and grading fairness;
- Presents adaptations of those recommendations for specific assessment types;
- Discusses how Large Language Models can play a role in the future development of AATs.

This section is organized as follows: first, in ‘Definitions’ (3.4.2) we present 1) a categorization of assessment types typically found in computer programming courses, 2) some AAT concepts, and, 3) some Unit Testing concepts. Our main contribution – recommendations for test-case creation and feedback definition – comes next: ‘Lessons learned: generic recommendations’ (3.4.3) presents generic recommendations related with test case creation and feedback definition, ‘Lessons learned: specific recommendations per assessment type’ (3.4.4) presents recommendations which take in consideration the type of assessment, and finally ‘Lessons learned: other recommendations’ (3.4.5) presents recommendations related with interacting with students when they fail to understand the tests’ feedback. Afterwards, ‘The Future of Automatic Assessment’ (3.4.6), discusses the future of AATs in light of the recent advent of LLMs. ‘Limitations’ (3.4.7) addresses some threats to the validity of the provided recommendations. Finally, in ‘Conclusions’ (3.4.8), we finish the section by drawing some conclusions.

3.4.2 Definitions

Assessment types

Our experience with automated assessment has led us to conclude that different assessment (or evaluation) types require different approaches, due to their specificities (e.g., whether the evaluation is time-bound or not).

This section presents the 4 evaluation types for which we recommend that different levels of feedback are considered and provided. These definitions should help teachers bridge our evaluation types with their own, and thus be able to more effectively apply our lessons and recommendations.

In-class/homework exercises: These are usually small exercises in which students have to implement code to solve small problems (e.g., 20-100 LOC) using specific techniques (e.g., iteration, recursion, etc). These are usually formative, although they might have some contribution to the students' grades. During said exercises, students can get support from both teachers as well as their colleagues.

Mini-test: Small exercises to be done individually and independently (e.g., without support). These exercises are time-bound (e.g., maximum one hour) and count towards the students' final grade.

Project: Mid-to-large scale exercises which take weeks or even months to solve. These usually have a larger weight in the students' final grades.

Defense: A proctored evaluation where students must make changes to their assignment's code, in order to demonstrate knowledge of it. This is usually done for projects and other evaluations with significant impact in the students' grades, in order for teachers to have more confidence that students actually authored the delivered code instead of copying or plagiarising. For group projects, it has the added benefit of adjusting the grades given to different group elements.

Test function types

Usually AATs inform students about their results. However, some AATs allow the definition of hidden tests, which are tests whose existence and result is not disclosed [51, 121, 124, 48]. As such, we make the following distinctions when defining test cases and the respective feedback:

Public tests: Test functions whose results are reported to the student (e.g., correct or incorrect) who possibly also receives messages with extra feedback.

Private or hidden tests: Test functions which are executed without the student receiving any feedback. Hidden tests are a common strategy to counter the usage of overfitted solutions: teachers prepare a public test for a function and then prepare a private test for the same function, but with different test cases. Since students will not even know that the test exists, they will not be able to derive an overfitted solution that passes both the public and the hidden tests. Hidden tests also have the potential for better distinguishing between students' grades, since students who test their code locally with more cases than the ones made available in the public test function might end up also passing the hidden tests. However, this will probably depend on the cases which are actually been tested and whether they are corner and/or otherwise complex cases.

Unit Testing frameworks

Unit Testing frameworks typically supply programmers with assertions: functions which verify if a certain condition is being respected. Some notable examples are: '`assertEquals()`', '`assertTrue()`', and '`assertFalse()`'. They also typically have a '`fail()`' function, which can be used when the programmer wants to test a condition without using one of

the default assertion functions.

These assertion functions usually receive an optional ‘message’ parameter, which can be used to customize the message that is displayed when the assertion fails. In professional software development practice, this parameter is usually not very relevant, because programmers have access to the test code and can observe the assertion’s implementation details. However, it gains importance when developing tests for AATs, since students will usually not have access to the assignment’s test code.

Several Unit Testing frameworks exist for a variety of programming languages. The most well-known example for Java is *JUnit* [74]. Another example is Python’s *unittest*¹⁰ library.

3.4.3 Lessons learned: generic recommendations

This section presents our recommendations for test construction and feedback definition. The examples are given in Java, using *JUnit* 5, but they can easily be adapted to other languages and frameworks, such as Python.

Tackling problem evasion

In order to detect cases where students try to evade the prescribed problem (e.g., recursion), we recommend that static analysis tools are employed in the assessment process. An example tool which can be used to do this for Java code is the Checkstyle plugin¹¹ which, among other functionality, allows teachers to define forbidden keywords (e.g., **for**, **while**) and signals their usage. This enables teachers to inspect and/or penalize solutions which use those keywords.

Another useful technique is Reflection [112]. This technique allows certain non-functional requirements, such as the need for a class to be abstract, to be validated using unit tests.

Test construction

Before we delve into our recommendations for test case construction, it should be noted that, in some scenarios, it might be necessary for the exercises to be changed in order for them to be properly evaluated in an automatic fashion using unit tests. This means that test functions influence API design, similarly to what happens in test driven development. In order to apply some of our recommendations, the tested functions should always receive at least one parameter, and the more parameters, the easier it will be to create rich test-cases. Consider this when evaluating our first 2 recommendations: they might seem somewhat contradictory, but they are achievable by adjusting the exercises’ APIs.

At least 2 independent test functions for each API function

We recommend creating at least two independent test functions for each of the exercise’s API. As an example, consider this exercise: *Implement a recursive function that*

¹⁰<https://docs.python.org/3/library/unittest.html>

¹¹<https://checkstyle.org/>

compares two char arrays through an extra argument representing the initial position (assume *a1* and *a2* have equal length): `static boolean equalArrays(char[] a1, char[] a2, int initialPos)`

You can break this problem down into different test functions, considering different input cases, such as 1) empty or null arrays, 2) initial position greater than the length of the array, 3) arrays with just one element, and, 4) arrays with 2 or more elements. By having multiple small test methods for each function, you allow gradual exercise completion with tests of increasing difficulty (an essential ingredient of gamification). Not only does this increase the student's motivation, but it also allows for more fine-grained grading (i.e., a student who is able to implement the base case but not the recursive case doesn't have zero). Also note that each of these test-methods can further help the student by having a name which hints at the case being tested (e.g., `testEqualArraysSingleElement()`).

At least 2 sub-cases in each test function

Each test should have at least 2 sub-cases for the same function, with different expected return types, in order to prevent hardcoded/overfitted solutions from being considered correct, even if only partially. This should be implemented by having 2 different assertions with different return values. The most obvious example for this appears when evaluating boolean functions: if the test function only has cases for when the expected value is 'true' (as in Listing 3.4), then a solution which always returns 'true' is going to pass that test.

Listing 3.4 – Test function which validates a boolean function using a single assertion. This test will pass if the student's `isOpen()` function returns an hardcoded 'true'.

```
1 @Test
2 public void testDoorOpening() {
3     Door door = new Door();
4     door.open();
5     assertTrue(door.isOpen(), "isOpen() incorrectly returned false");
6 }
```

On Feedback

Since we want to help students to solve the exercises, assertions without a specific message should be avoided (consistent with previous recommendations on feedback generation [172]). As such, the minimum feedback should, at least, indicate the name of the function returning the incorrect result (e.g., '`getSalary()` returned the wrong value' or, at least, '`getSalary()`'). This recommendation might be slightly relaxed if we are testing a function with a very specific return format which will make it obvious for the students which is the tested function. For example, when assessing an object's '`toString()`' method, if its output format significantly differs from that of other objects, this approach may be appropriate.

However, just mentioning the function name might not be sufficiently helpful. For example, consider the assertion in Listing 3.5:

Listing 3.5 – In certain cases, just referring the function name is insufficient and might cause confusion.

```
1 @Test
2     ...
3     List<String> movieTitles = Main.getTitles();
4     boolean found = movieTitles.contains("The Matrix");
5     assertTrue(found, "getTitles()");
6     ...
```

If the title “The Matrix” is not included in the list, the respective assertion will simply yield ‘`AssertionFailedError: getTitles() => Expected: true Actual: false`’. This will let the student know that something is wrong with the function ‘`getTitles()`’, but, unless the problem is very obvious, debugging will be hard since due to lack of clues. Also, we have seen situations where this pattern has led some students to confusion, due to the expected/actual result indicating ‘true’ and ‘false’ when the ‘`getTitles()`’ function is not supposed to return a boolean. It would be more helpful to let students know which value is missing so that they can direct their debugging efforts. Refer to Listing 3.6 for an alternative implementation which contains more actionable feedback:

Listing 3.6 – A test with more actionable feedback than the one in Listing 3.5.

```
1 @Test
2     ...
3     List<String> movieTitles = Main.getTitles();
4     String title = "The Matrix";
5     boolean found = movieTitles.contains(title);
6     assertTrue(found, "getTitles() missing title: " + title);
7     ...
```

Of course, giving students this extra piece of information could lead to partial or full overfitting, since the student could insert the “The Matrix” String in the returned list to circumvent their original error. In fact, students’ ability to produce overfitted solutions is highly dependent on the information which is available to them, both via the assignment’s description as well as via the employed test cases. To better understand this idea, consider the example unit test presented in Listing 3.2. That test can be easily bypassed using the solution shown in Listing 3.7:

Listing 3.7 – Overfitted solution that passes the test defined in Listing 3.2. It would be easy for a student to hardcode a solution for that test after seeing its feedback.

```
1  static long sum(int n1, int 2) {  
2      if(n1 == 1) {  
3          return 5; // first assertion indicates 1 + 4  
4      }  
5      else if(n1 == 8) {  
6          return 10; // second assertion indicates 8 + 2  
7      }  
8      return 0;  
9  }
```

As such, it is necessary to find a balance between the level of feedback which is given to students: on one hand, too much feedback might lead to overfitted solutions, since, if students know all expected input/output pairs, then they can prepare a function which returns the correct result but is not generic. On the other hand, too little feedback will leave students lost and unable to progress. This is particularly important when dealing with API functions such as *JUnit*'s 'assertTrue()' and 'assertFalse()', which provide no clue other than 'AssertionFailedError: Expected: true Actual: false'. This problem is made even worse in courses where the tests might be calling multiple functions, which commonly happens in OOP exercises. In those cases, such reduced feedback will likely make students wonder which of their functions has the problem.

However, with a varied number of test cases per test function (as previously recommended), it will be hard for students to implement an overfitted version which passes all the sub-cases. Furthermore, teachers can make testing functions which give a lot of detail in the first few cases, and little to no detail in the final cases (consistent with [172], which recommends employing provisions to prevent abuse of progressive hints). See Listing 3.8 for an example.

Listing 3.8 – Test function with a progressive reduction of feedback level.

```

1  @Test
2  public void testSumArray() {
3      String suffix = "returned an incorrect value.";
4
5      // feedback includes the arguments (vulnerable to overfitting)
6      int[] array1 = new int[]{1, 4};
7      assertEquals(5, Main.sum(array1), "sum({1, 4}) " + suffix);
8
9      // feedback includes the arguments (vulnerable to overfitting)
10     int[] array2 = new int[]{2, 2, 2};
11     assertEquals(6, Main.sum(array2), "sum({2, 2, 2}) " + suffix);
12
13     // feedback doesn't include the arguments, only a hint
14     String suffix2 = suffix + "Consider arrays w/ negative numbers.";
15     int[] array3 = new int[]{-1, -2, -3};
16     assertEquals(-6, Main.sum(array3), "sum(...) " + suffix2);
17
18     // feedback only includes the function name
19     int[] array4 = new int[]{1, 2, 3, 4, -1};
20     assertEquals(9, Main.sum(array4), "sum(...) " + suffix);
21 }

```

In some situations, besides the name of the function and some hints ([172]) with regards to the expected values, the minimum feedback should also indicate relevant particularities of what is being tested. For example, imagine that we are testing a function called ‘`int[] getStatistics()`’ which returns an integer array where each element has a different semantics: the first element is the input array’s minimum value, while the second element is the input array’s maximum value. If we provide students with simply:

```
assertEquals(42, Main.getStatistics()[1], "getStatistics()");
```

debugging will be hard due to lack of information. It can also cause interpretation issues, since the student might be confused when seeing a message similar to ‘`getStatistics()` expected: 42 but was: 99’ since ‘`getStatistics()`’ returns an ‘`int[]`’ and not a single ‘`int`’. In this example, it would be preferable to define the test’s feedback in order to let the student know which of the array’s positions has the wrong value, as follows:

```
assertEquals(42, Main.getStatistics()[1], "getStatistics()[1]");
```

This will give the student extra clues with regards to where to start analysing the bug.

Finally, the amount of feedback also depends on the exercise type. This will be further expanded later in this section.

On hidden tests

Avoid using hidden tests to validate things such as String contents. It is easy for students to make minor mistakes, such as having an incorrect capitalization or spelling mistake. Validating such behaviours only in a hidden test might be too penalizing. Fur-

thermore, although hidden tests can be an interesting tool, they can also be the source of problems: since their results are not visible to the students, eventual errors are likely to go unnoticed. An option to mitigate this issue is to use an AAT like Drop Project, which is able to indicate how many students are passing each test [48], and, if no student passes a test or only a small minority of students pass it, then maybe the teaching staff should review it, since it might be incorrect.

Assert-early and often

Consider performing one or more assertions after each action in the test, instead of performing multiple assertions after executing a number of actions. This is particularly helpful when validating OOP assignments, where an object's state changes with each action, meaning that an error on the first action might cause problems in the following actions. If the first assert is done after multiple actions, the resulting feedback might induce the student in error. Listing 3.9 presents an example of this strategy.

Listing 3.9 – Example of the assert-early and often strategy. After each action, a number of assertions is performed, in order to point to the bug as early as possible.

```
1 Door door = new Door();
2
3 // perform an action
4 door.open();
5 // assert that the object's state is 'open' after the first action
6 assertTrue(door.isOpen(), "isOpen() incorrectly returned false");
7 // also validate the object's toString()
8 assertEquals("This door is open!", door.toString());
9
10 // perform another action
11 door.close();
12 // assert that the object's state is 'closed' after the second action
13 assertFalse(door.isOpen(), "isOpen() incorrectly returned true");
14 // also validate the object's toString()
15 assertEquals("This door is closed!", door.toString());
```

Null references

Be careful when creating test cases which might result in hard to debug errors such as `NullPointerException`. Consider using `assertNotNull()` to let students know where their code is returning unexpected null references. See Listing 3.10 for an example.

Listing 3.10 – Gracefully handling potential null references to provide useful feedback.

```
1 // unprotected approach, which will result in a hard to debug
2 // NullPointerException
3 Actor actor1 = Main.getActorByID(1000);
4 assertEquals("Keanu Reeves", actor1.getName(), "Actor getName()");
5
6 // protected approach, which will let the student know that
7 // an unexpected null reference was returned
8 Actor actor2 = Main.getActorByID(1000);
9 assertNotNull(actor2, "getObject(1000) returned null");
10 assertEquals("Keanu Reeves", actor2.getName(), "Actor getName()");
```

Test-function names

Although the industry norms and conventions for test code do not recommend prefixing test functions with “`test_`” nor having a number for each test, we believe that these two practices are actually helpful in educational contexts. Including said prefix has the advantage of facilitating the interpretation of the tests’ stack-traces (when available) since it makes it obvious which of the functions in the stack is the test. Furthermore, numbering the test makes it somewhat easier to communicate with the student.

The order of the tests matters

Even though, according to the best practices, the tests should be independent of each

other, students tend to tackle each failed test by the order it shows up in the report provided by the AAT. As such, consider ordering the tests by difficulty level.

Progressive disclosure

Consider activating certain advanced tests only after students pass simpler tests. This is not directly accomplishable by *JUnit* (since it goes against the philosophy of independent tests) but it is easily implemented using static variables (see Listing 3.11). Important: Students should know that these tests exist and what is needed to “unlock” them (e.g., “This test will only be executed after you pass the tests based on the small input files”). This rule also interacts with the previous one: the feedback for more complex tests should appear after the feedback for the simpler tests.

Listing 3.11 – Progressive disclosure: this test is only executed after the student passes at least 4 simple tests.

```
1 @Test
2 public void advancedTest() {
3     if (SimpleTests.testsOK < 4) {
4         fail("This test will be unlocked after passing 4+ simple tests");
5     }
6     // ... actual test
7 }
```

Tests involving collections

Be careful with “OutOfBounds” runtime errors: check the length of the collection before accessing individual elements. However, don’t just check the total length of the collection upfront, since it will be difficult for students to understand the problem when they have insufficient or excessive data. Instead, first validate a few elements in order to allow students to debug missing values. See Listing 3.12 for examples of both approaches.

Listing 3.12 – Testing collections’ contents: its better to validate a few examples before asserting the full list size.

```
1 List ids = Main.calculateIds();
2
3 // this is difficult to debug
4 assertEquals(347, ids.size(), "calculateIds() returned wrong number of
   elements");
5
6 // this is better
7 int size = ids.size();
8 assertTrue(size >= 2, "calculateIds() should have returned at least 2 elements,
   but returned " + size);
9 assertEquals(3, ids[0], "calculateIds()[0]");
10 assertEquals(5, ids[1], "calculateIds()[1]");
11 assertTrue(size >= 5, "calculateIds() should have returned at least 5 elements");
12 assertEquals(6, ids[4], "calculateIds()[4]");
```

Include a timeout on each test

In order to detect infinite loops and/or implementations with inappropriate time complexity (i.e., big-O notation), teachers should consider defining timeouts for some or all of the test functions.

3.4.4 Lessons learned: specific recommendations per assessment type

As we've seen before, in order to allow students to act and solve a problem reported by the AAT, the respective tests should supply the student with some level of information. We argue that the ideal level of information will be different depending on the assessment type (e.g., homework, project).

As such, we hereby present adaptations of the aforementioned generic recommendations, making them more apt for specific assessment types.

In-class/Homework exercises

Due to the formative goals of these exercises, we consider it important to provide students with feedback which allows them to fully master the material. As such, all test functions should be public and all, or at least the majority, of assertions should have significant feedback.

Mini-tests

In such assignments, we recommend that some feedback is given, but at least one of the asserts should give very little feedback, in order to reduce the likelihood of *overfitted* solutions. For example, at least the last assertion of each test function should not give any hints in terms of input and expected output. All test functions should be public, since it is important for students to have a real notion of how they are doing.

Projects

Use hidden tests to make it harder for *overfitted* solutions to get good grades. Define a subset of the public tests as 'mandatory'. These tests will define the minimum mandatory functionality which a student must implement in order to get a passing grade, also promoting that the approved projects have a minimum level of quality. Furthermore, if you employ project defenses, having mandatory tests will aid in avoiding certain issues (refer to the next paragraph for details). Hidden tests should not be mandatory, as students are unaware of their existence and cannot take action to address or rectify any related issues.

Defenses

The rules presented for mini-tests also apply to project defenses, since both are time-bound evaluations where students tend to become stressed. However, a few additional recommendations make sense when defining tests for project defenses. If possible, build the defenses around the project's 'mandatory functionality' in order to prevent penalizing students whose projects have bugs that were not detected originally. If 'mandatory' tests were not defined, teachers should avoid testing scenarios which were only tested in the project's hidden tests. These two recommendations aim at reducing the chance of penalizing students whose projects have pre-existing bugs that might condition their per-

formance in the defense. Furthermore, for similar reasons, we recommend using adapted versions of the project’s public tests in the defense’s tests (e.g., same input file, similar actions, and so on). Teachers should also be careful with interdependence between the test cases. For example, if the defense’s assignment asks students to change the `toString()` function, then it might be dangerous to use the `toString()` as a criteria to determine the correctness of other defense tests. Finally, common pitfalls should be identified and avoided. For example, in projects involving people’s names, some students only support two-named individuals and overlook those with multiple names, like ‘Samuel L. Jackson’ versus ‘Samuel Jackson.’ It’s important to recognize and steer clear of these issues in project defenses, as students often fail to address them despite detailed feedback.

3.4.5 Lessons learned: other recommendations

Sometimes students complain that a test is incorrectly failing, justifying their position with something along the lines of ‘in my computer, it works’. As such, we find it important to educate students on how their code will be tested.

One way of doing this is to teach them how to programmatically test their own code. Even so, some students might still struggle with transferring this skill to the interpretation of teachers’ tests, due to lack of testing experience and/or other issues. To help with this, consider giving students access to the code of one of the assignment’s tests, in order to allow them to reflect on how testing is being performed. This might have the added benefit of students making their own adaptations of the sample test, thus better testing their code.

Finally, we also suggest a strategy of promoting that students write their own tests when they ask for help. In response to help requests, we often prompt students to demonstrate a unit test that evaluates a specific aspect, akin to our AAT’s test. Although this usually requires some interactions until the student produces a test which mimics the actual problem, it has the advantage of forcing students to enter a testing-oriented mindset.

3.4.6 The Future of Automatic Assessment

Although LLMs have been received somewhat controversially by the CSE community [104], we expect CS educators to adopt these technologies in their teaching practices sooner or later. More concretely, we believe LLMs will be used to improve AATs and/or assessment practices [40].

Some possible research avenues are: 1) using LLMs to automate the generation of tests ([104]) that respect the recommendations presented in this section; 2) integrating LLMs and AATs to simplify the process of providing hints for failed tests, similarly to [141], but with guardrail mechanisms to prevent LLM over-reliance; 3) employing LLMs to generate customized feedback that considers both the student’s code as well as the issues identified by the AAT; 4) integrating LLMs into AAT’s evaluation schemes ([75]) to detect issues such as overfitting and problem evasion; 5) replacing AATs with LLMs entirely. Note

that developments in terms of improving and/or complementing AAT feedback (i.e., #2 and #3) might reduce the need for educators to follow recommendations such as the ones presented in this section. With regards to #5, we find it unlikely, at least in the near future, since current state-of-the-art LLMs have limitations (namely, hallucinations¹²), and still require human supervision [75]. The need for said supervision would potentially negate one of the biggest advantages of automated assessment: reduction of teacher workload.

3.4.7 Limitations

The recommendations in this section are based on our multi-year experience across various courses but have not been statistically validated. As such, it is possible that they are somewhat biased. To mitigate this issue, we have reviewed our recommendations in relation to earlier studies on feedback, such as [172], and confirmed their alignment with prior recommendations.

3.4.8 Conclusions

This section presented a set of recommendations for the definition of unit tests to automatically verify students' work and provide students with formative feedback which allows them to learn and/or act on their mistakes. We believe that following these recommendations has helped us help our students in their learning processes. They also helped us avoid certain mistakes which would have otherwise consumed our time or penalized students excessively.

¹²Occasions in which LLMs produce outputs which are factually incorrect.

Chapter 4

Large Language Models and Object-Oriented Programming

4.1 Chapter summary

This chapter addresses the impact of Large Language Models in Object-Oriented Programming courses as well as some potential strategies to prevent students from over-relying on these tools.

This chapter is composed of two parts.

The first part (section 4.2) is strongly based on the publication titled **LLMs Still Can't Avoid Instanceof: An Investigation Into GPT-3.5, GPT-4 and Bard's Capacity to Handle Object-Oriented Programming Assignments** (presented in the 46th International Conference on Software Engineering, 2024, level A*). It explores the OOP capabilities of 3 state-of-the-art LLMs: GPT-3.5, GPT-4 and Bard. To evaluate the models, we used a set of real-world programming assignments focused on inheritance and polymorphism which had been previously used in our courses to evaluate students. Furthermore, the evaluation was done using the Drop Project AAT. Drop Project's OO-features were instrumental in understanding these models OO capabilities. Note that this was our second publication based on evaluating LLMs in the context of OOP education. Our initial work on this topic was titled **GPT-3 vs Object Oriented Programming Assignments** (presented in the 28th Conference on Innovation and Technology in Computer Science Education 2023, level B). We did not include the original paper [41] in this document, since the most relevant information is also presented in the most recent publication.

The second part (section 4.3) is strongly based on the publication titled **A Picture Is Worth a Thousand Words: Exploring Diagram and Video-Based OOP Exercises to Counter LLM Over-Reliance** (ECTEL 2024, B ranked). It resulted from our attempts to prevent students from solving their assignments by mere 'copy-and-prompting'. We devised a new visual notation to prescribe OOP exercises. This notation was crafted considering the paradigm's particularities. The proposed notation was used

during a semester of an OOP course, after which we performed a student survey to evaluate it, finding that it was well received by our students, who agreed that they are an effective way of preventing LLM-abuse and also reported other benefits.

4.2 An Investigation Into GPT-3.5, GPT-4 and Bard’s Capacity to Handle Object-Oriented Programming Assignments

4.2.1 Introduction

In the last few years, researchers introduced us to Large Language Models (LLMs), which are tools that can predict the next word in a sequence, after being trained on large amounts of data, and taking into consideration a large number of parameters. Popular implementations of LLMs are ChatGPT, a chatbot developed by OpenAI on top of their Generative Pre-trained Model (GPT), and GitHub Copilot¹, a developer support tool that can generate code from annotations and is also based on GPT.

As CS educators, we are interested in determining how these tools can be used to support students in their programming learning process, both inside and outside of the classroom. We are specifically interested in the possibilities of using these tools for teaching and learning object-oriented design and programming.

Previous research has shown that LLMs have the capacity to generate computer program code from natural language descriptions [202]. Furthermore, GPT-based models, such as Codex, have been shown to be able to solve most types of exercises that are used in introductory programming classes [69]. Additionally, researchers have reported that GPT-3.5 can handle object-oriented programming assignments, managing to reach decent to good scores, with the caveat that the generated code is not up-to-par with the industry’s best practices of that paradigm [41].

However, both the availability and the capabilities of these tools are increasing rapidly. Updated models, such as GPT-4², promise even better performance than their older versions [137, 34], while new tools, such as Bard³, a chatbot developed by Google on top of their Language Model for Dialogue Applications (LaMDA) [179, 186], are becoming available to the general public. These new models justify performing further research on these topics, to confirm and/or update the findings that resulted from the previous works.

Thus, we decided to explore these newer tools to learn their capabilities and limitations, as well as identify opportunities to integrate them into our courses, similarly to what is being done by other instructors [104, 52, 50, 108, 114].

As such, we performed an evaluation of some of the most recently available LLMs, GPT-4, and Bard, following previous research focused on object-oriented design, program-

¹<https://github.com/features/copilot>

²Launched on March 14, 2023

³Launched in the US & UK on Mar 21, 2023; in EU & Brazil on Jul 13, 2023.

ming and best-practices using GPT-3.5 [41]. This evaluation was based on assignments that are automatically graded by an Automatic Assessment Tool (AAT).

This section makes the following contributions:

- Presents and compares the performance (as measured by an AAT) of GPT-3.5, GPT-4, and Bard using 6 real-world assignments that have been used to grade students in an Object-Oriented Programming university course;
- Identifies and categorizes the errors found in the code generated by the tested LLMs;
- Provides full logs of the interactions used to get each model to solve one of the assignments, in an annotated prompt/reply format;
- Presents a set of recommendations for CS Educators wishing to adapt their classes to the availability of these technologies.

This section is organized as follows: first, ‘The assignments’ (4.2.2) briefly describes our course, as well as the 6 assignments used to evaluate the models. Afterwards, ‘Methodology’ (4.2.3) details how we interacted with the 3 models, as well as the technologies used to interface with them. A high level description of our interactions with the 3 models when trying to solve one of the assignments, as well as the most relevant failures of each model is presented in ‘It works but it’s not OO - how LLM(s) tried to solve the “IT Company” assignment’ (4.2.4). ‘Overall Findings’ (4.2.5) provides an overview of the models’ performance over all the assignments. Then, in ‘Recommendations for CS Educators’ (4.2.6), we present some ideas that can be explored by CS educators in response to this new reality. Finally, we present the ‘Limitations’ (4.2.8) of our study, after which we end the paper by drawing some ‘Conclusions and Future Work’ (4.2.9) and presenting some ideas for future work.

4.2.2 The assignments

This research was based on assignments used for student evaluation in a Computer Engineering degree. The course occurs on the 2nd curricular year of the degree and focus mostly on teaching Object-Oriented Design and Programming using the Java programming language. In this course, students are expected to learn how to implement object-oriented software solutions following the paradigm’s best practices [196], with a strong focus on issues like code readability, modularity and extensibility. For example, students are expected to understand the drawbacks of using type testing, such as those permitted by ‘`instanceof`’, to decide program behaviors, since that technique tends to make programs harder to modify.

The course’s assignments (e.g., tests and projects, among others) are evaluated using an open-source AAT, called Drop Project [48]. It evaluates if the student’s code is respecting the assignment’s requirements using teacher-defined unit tests. It is also capable of verifying if the student’s code follows the expected code quality rules and guidelines.

We want to implement a software to help an IT company that provides ITConsultant services with the management of their employees. In this challenge, there exist two types of employees: those who pertain to the human resources management area (HRWorker), and the IT (information technology) experts, who pertain to ITConsultant.

The classes Employee, HRWorker and ITConsultant must be created (...) however it should not be possible to instantiate an object using the super class's constructor.

A company is characterized by its name. A company can have multiple employees, but each employee only belongs to a single company.

A employee is identified by its id (int), name (String) and monthly salary (int).
(...)

Add, to the appropriate classes, the following methods:

- A public int getId() method, which returns the employee's ID.
 - A public int getSalary() method, which returns the employee's salary.
 - A public int getValue() method, which returns the IT consultant's hourly rate.
- (...)

(Within the Company class) add a public String toString() method, which must be implemented to return the following:

- "The company <name> does not have employees." - if the company does not have any employee;
- "The company <name> has <number_employees> employees:" - if the company has at least one employee. Besides that, the information for each employee should be display, considering the respective type.

Listing 4.1 – Partial instructions for the “IT Company” assignment

In this course, some assignments are configured to warn the teacher about the use of certain keywords that allow the program to work while disregarding some of the aforementioned best practices. An example of this is the usage of the ‘instanceof’ keyword or the ‘getClass()’ function. Due to the limitations of the plugin used by the AAT for the code quality validations (Checkstyle [89]), some quality validations are implemented using unit tests. A case in point is verifying whether a certain class has been declared as abstract.

To evaluate the LLMs, a set of assignments that have been used in this course as mini-tests focused on inheritance and polymorphism was used. Each assignment has a different business domain, as well as different requirements. All the assignments have the following goals: identify and implement entity relationships (both composition and inheritance), implement some getters and setters, implement a non-trivial ‘toString()’ function, and, implement some functions that have to create, query and/or manipulate objects of several classes. Note that the relationships aren’t directly provided to students, who must infer which classes are the super and sub-classes. This approach sets our study apart from others where assignments give more explicit directions [138, 162].

Moreover, each assignment has behaviors based on the object type to assess whether students can devise solutions without resorting to explicit type checks like those allowed by the ‘instanceof’ keyword and ‘getClass()’ function. Finally, in some cases, we ask for the student ID to be used as the value of an object’s attribute, to guarantee unique solutions among students.

Listing 4.1 presents a partial example of an assignment’s text.

Six such assignments were used in this study. The selected assignments were originally

used to grade students in different school years (2018/19, 2021/22, and 2022/23).

4.2.3 Methodology

To interface with GPT-3.5, openAI's Playground⁴ (free account) was used. The selected model was "text-davinci-003". The "temperature" parameter, which controls randomness, was left at the default value of 0.7, and the "maximum length" parameter was left at the default of 256. GPT-4 was studied using OpenAI's ChatGPT Plus⁵ (paid account). Finally, to study Bard, we used the respective chat interface⁶. Note that neither ChatGPT Plus nor Bard allows changing the models' parameters. The experiments were done in different time periods, since we did them when the models became available to us: the original experiment with GPT-3.5 was done in December 2022, the experiment with GPT-4 was done in May 2023, and finally, the experiment with Bard was done in July 2023. To evaluate the models' output, version 0.9.7 of the Drop Project AAT was used.

For each model and assignment, the following algorithm was performed:

1. paste the assignment's text into the LLM's text input area
2. submit it
3. examine the output to determine whether all the mandatory classes and functions are present
 - (a) if they are, move to step 4
 - (b) otherwise, prompt for the missing code and repeat step 3
4. inspect the code for syntactic, logic, and output format errors
5. manually fix any syntactic / compilation errors
6. submit the LLM's generated code to the AAT
7. analyze the AAT's output

4.2.4 It works but it's not OO - how LLM(s) tried to solve the "IT Company" assignment

This section presents a high level overview of our attempts to use each model to solve one of the 6 assignments: the "IT Company" assignment. In this assignment, students are expected to design the object model of an Information Technology Company with the following concepts: 'Company', 'Employee', 'ITConsultant', and 'HRWorker'. The concepts and their attributes are explained to the students by text. However, the text does not explicitly indicate the relations that exist between the classes: it is up to the

⁴available at <https://platform.openai.com/playground>

⁵available at <https://chat.openai.com/>

⁶available at <https://bard.google.com/>

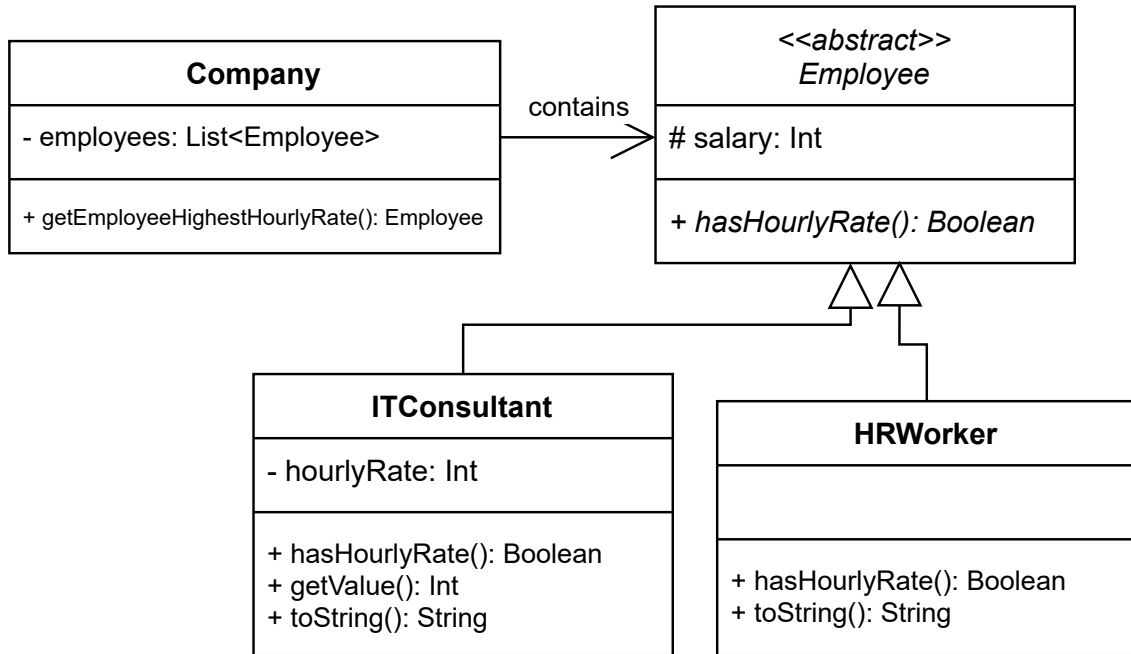


Figure 4.1 – Simplified UML class diagram for the “IT Company” assignment, excluding non-essential attributes, methods, and constructors. Students only receive textual instructions, not this diagram.

student to infer that there exists a composition relationship — between the ‘Company’ and the ‘Employee’ classes —, and an inheritance relationship where the ‘Employee’ class is the super-class and the ‘ITConsultant’ and ‘HRWorker’ classes are the sub-classes. Students are also expected to understand that the ‘Employee’ class should be abstract, since the text mentions that it must not be possible to instantiate that class. Finally, students must implement several mandatory functions, identifying the class of the hierarchy where each function belongs. For example, there is an attribute that only makes sense for the ‘ITConsultant’ class: the hourly rate. As such, only that class should have that attribute and the respective getter, which must be called ‘getValue()’. Also mandatory is the creation of a function called ‘getEmployeeHighestHourlyRate()’ in the ‘Company’ class. This function must return the ‘Employee’ with the highest hourly rate. However, since the hourly rate concept only applies to the ‘ITConsultant’ class, the student must find the proper Object-Oriented design to implement this challenge. As such, if a student uses ‘instanceof’ or ‘getClass()’, a penalty will be applied to the respective grade. Figure 4.1 presents an overview of this assignment’s required classes, as well as the most relevant design expectations. Listing 4.1 presents part of this assignment’s instructions. A full version of the assignment, including both instructions as well as unit tests is available online⁷. Refer to [42] for full logs of these interactions.

⁷<https://github.com/drop-project-edu/itCompanyAssignment>

Main analysis

GPT-3.5 required **18** interactions to generate a solution that included all mandatory elements. Issues found in that solution:

- Function `getEmployeeHighestHourlyRate()` uses `instanceof`.
- Fails 1 test because the `Employee` class is not abstract.
- Fails 2 tests because the package name was incorrectly included in `ITConsultant.toString()` function's return value.
- Fails the test for the `Main.myCompany()` function due to using an uninformed number in the solution.⁸
- Failed to import the `ArrayList` class.

Bard required **11** interactions to generate a solution that included all mandatory elements. Issues found in that solution:

- Function `getEmployeeHighestHourlyRate()` uses `instanceof`.
- Fails 1 test because the `Employee` class is not abstract.
- Fails 2 tests due to incorrect usage of the downcast operator in the `getEmployeeHighestHourlyRate()` function.
- Fails 2 tests due to not respecting the expected format of the `HRWorker.toString()` and `Company.toString()` functions.
- Fails the same test related to the `Main.myCompany()` function that the other models failed, with a difference in the implementation: instead of guessing a number, Bard's code attempts to obtain the student ID by using `System.getenv()`, resulting in a `NullPointerException`.⁹
- Failed to import the `ArrayList` class.
- Declared 3 attributes of the `Employee` class as private, and tried to access them directly in the `ITConsultant` sub-class.

GPT-4 required **4** interactions to generate a solution that included all mandatory elements. Issues found in that solution:

- Function `getEmployeeHighestHourlyRate()` uses `instanceof`.
- Fails 1 unit test because it did not fully respect the format of the `Company.toString()` function (a new line was missing).

⁸This problem was expected, since we did not input the student ID to the models.

⁹See footnote 8

- Fails the test related to the `Main.myCompany()` function due to using an uninformed number in the code¹⁰.

In summary... These experiments show that these 3 LLMs were able to partially solve the “IT Company” assignment, although with some errors, both in terms of basic syntax (e.g., missing library importations), as well as more complex object-oriented programming errors (e.g., not declaring a class as abstract, or using explicit type testing to decide program behaviors). This is similar to what happened in the other 5 assignments, as described in Section 4.2.5.

Attempts to improve the LLM(s)’ solutions

After obtaining each model’s solutions for the “IT Company” assignment and verifying that all of them were breaking one of our code quality rules — the restriction not to use `instanceof` —, we attempted to guide the models toward better solutions, similar to how we expect a student would do after seeing the AAT’s feedback, and to get a better grade. Note that these extra prompts are not part of our main methodology and are not represented in Table 4.1.

These attempts to improve the models’ solutions were done using an ad-hoc methodology where we identified problems with each model’s solution, and attempted to give it a direct indication of the problem, to see what it would do. The first extra prompt was the same for all models: ‘Change the code to not use instanceof’. Afterward, we re-prompted based on what each model gave us. The prompts for GPT-3.5 and Bard ended up being similar because the respective intermediate solutions had similar problems from the object-oriented perspective: they both were assuming that all sub-classes of ‘Employee’ have the `getValue()` function, which is not coherent with the rest of the respective solutions.

Besides the initial extra prompt, **GPT-3.5** required two more prompts to reach an acceptable solution: as a reply to the third prompt, it suggested adding the `getValue()` function to both the super-class and the sub-class with an implementation that would return 0. In this case, 0 is being used as a flag to indicate that the object should be ignored, which is coherent with the model’s implementation of `getEmployeeHighestHourlyRate()`. Although that solution is more OO, it is somewhat risky: such a design would be problematic if a function `getEmployeeLowestHourlyRate` is needed at some point in the future.

Bard had a behavior similar to that of GPT-3.5, although it required a fourth prompt to reach a solution acceptable from the OOP perspective. Bard’s fourth solution was equivalent to GPT-3.5’s third solution, thus having the equivalent problems. Note that, although the OO solution is acceptable, Bard implemented the `getValue()` function incorrectly: it is returning the value of the salary instead of the value of the hourly rate. Since we were focused on getting Bard to solve the OO issue, we ignored this implementation detail.

¹⁰See footnote 8.

Finally, **GPT-4** only needed the extra initial prompt to reach an acceptable OO solution. The model suggested adding an abstract version of `getHourlyRate()` to the `Employee` class, as well as adding a concrete version of that function (returning -1, which is much safer than returning 0) to the `HRWorker` class. It also gave us the code for those changes, as well as a version of the `getEmployeeHighestHourlyRate()` function that only considers objects for which `getValue()` returns more than 0.

In summary... Our experiments show that it was possible and easy to guide GPT-4 towards a fully working solution that respects the object-oriented best practice of avoiding explicit type testing. Also, while the other models also managed to reach acceptable, although risky, OO solutions, they required more effort (GPT-3.5: 3 prompts; Bard: 4 prompts) and generated some minor errors (e.g., Bard’s incorrect return of the salary attribute in the `getValue()` function). This extra effort is coherent with the findings from other researchers [199].

4.2.5 Overall Findings

This section presents the 3 models’ performances across the 6 assignments. As Table 4.1 shows, all models needed multiple prompts and no model passed all the tests.

GPT-4 generally had the best performance, requiring fewer interactions than the other models, generating fewer compilation errors and, in general, having better results in the unit tests. Bard always had equal or worse unit test results than GPT-3.5, except in the “Home Banking” assignment, where it surpassed both GPT models.

In terms of the Code Quality validation, we observed that GPT-4 yields the best results: while GPT-3.5 and Bard respected the code quality rules in 3 out of the 6 assignments, GPT-4 did it in 4 out of 6 assignments. Listing 4.2 shows an example of the type of code that these models tend to generate: it (mostly) works, but tends to decide behaviors using explicit type testing, via `instanceof` and/or `getClass()`. In summary, similarly to what previous research found for GPT-3.5 [41], the two newer models still can’t avoid using `instanceof`.

It should be noted that, in the “Condominium Mgt.” assignment, GPT-4 managed to pass the code quality, as well as 13/14 tests. Considering that the single test failure was expected¹¹, we can consider that GPT successfully solved one of the OOP assignments.

¹¹See footnote 8

Table 4.1 – Evaluation of the 3 models’ solutions for each assignment. G-3.5 is GPT-3.5. The values are related to the first solution that contained all mandatory classes and functions. For example, GPT-3.5 needed 5 prompts to generate all classes and functions for the first assignment, that solution had 4 compilation errors, the code quality failed and it passed 8/13 unit tests.

Assignment	Nr. of prompts			Nr. of Compilation errors			Code quality Ok?			Tests passed		
	G-3.5	GPT-4	Bard	G-3.5	GPT-4	Bard	G-3.5	GPT-4	Bard	G-3.5	GPT-4	Bard
Realtor Agency	5	2	2	4	0	8	No	Yes	Yes	8/13 (61.54%)	12/13 (92.31%)	7/13 (53.85%)
IT Company	18	4	11	1	0	4	No	No	No	9/13 (69.23%)	11/13 (84.62%)	7/13 (53.85%)
Home Banking	7	2	10	1	0	14	Yes	Yes	Yes	7/13 (53.85%)	7/13 (53.85%)	9/13 (69.23%)
Condominium Mgt.	5	2	5	12	0	3	Yes	Yes	No	11/14 (78.57%)	13/14 (92.86%)	10/14 (71.43%)
Home Cinema A/V	5	2	6	3	0	2	Yes	Yes	No	8/10 (80%)	9/10 (90%)	8/10 (80%)
Railway Co. Vehicles	11	6	8	4	3	10	No	No	No	11/16 (68.75%)	12/16 (75%)	10/16 (62.5%)
Average	8.5	3	7	4.17	0.5	6.83	50%	66.67%	33.33%	68.66%	81.44%	65.14%

Listing 4.2 – A function generated by Bard for the “Condominium Mgt.” assignment that fails our Code Quality validations. Besides using explicit type-testing (i.e., it uses `instanceof`) Bard also failed to implement the proper business rules since the formula for the ‘Garage’ should multiply the area by 3 instead of 4 (see line 3).

```

1 public int calculateCondominiumPayment() {
2     int value = baseValue;
3     value += area * 4;
4     if (this instanceof Apartment) {
5         value += ((Apartment) this).getFloorNumber() * 3;
6         value += ((Apartment) this).getNrRooms() * 2;
7     }
8     else if (this instanceof Store) {
9         value += ((Store) this).getNrFronts() * 2;
10        value += ((Store) this).getNrDoors() * 2;
11    }
12    else if (this instanceof Garage) {
13        value += Math.abs(((Garage) this).getNrFloor()) * 2;
14    }
15    return value;
16 }
```

Problem categorization

This section presents a brief categorization of the problems observed across assignments and models. After each problem, we indicate which models displayed it at least in one assignment.

Issues related with syntax

- Failed to include library imports. [GPT-3.5, Bard]

- Created minor compilation errors (e.g., missing “}”). [GPT-3.5, Bard]
- Failed to declare a required package. [GPT-3.5]
- Generated incoherent code (e.g., called a function with fewer arguments than required). [Bard]

Issues related with OOP concepts

- Used `instanceof` or `getClass()` to decide program behaviours. [all]
- Failed to declare a class as abstract. [all]
- Failed to correctly identify and implement business rules. [all]
- Did not respect the ‘`toString()`’ function’s required format. [all]
- Failed to apply inheritance best practices. [all]
- Suggested solutions with code duplication. [all]
- Accessed a super-class’s private fields from its sub-classes. [GPT-3.5, Bard]
- Declared constructors with problems (e.g., added extra arguments). [GPT-3.5]
- Failed to properly apply the downcast operator. [Bard]
- Declared abstract methods in the super-class but failed to implement them in the sub-classes. [GPT-4]

4.2.6 Recommendations for CS Educators

Students are likely to misuse tools like GPT-3.5 and Bard, which offer decent-to-good OOP code generation for free. Superior models like GPT-4 are currently available at a cost. With the evident progress from GPT-3.5 to GPT-4, observed both in other research [162] as well as in our study, we believe that educators should adapt to this emerging landscape.

Give more weight to code quality evaluations

For educators focusing on object-oriented programming, given our observations, as well as previous research [97], we recommend putting more weight on evaluating items such as code quality, design patterns and other similar aspects. The course’s focus should change from just producing “functional code” to producing “functional and high-quality code”. The assessment work can be scaled using an AAT with the capacity to evaluate functional and quality requirements.

Use LLMs in your classes

Consider embracing LLMs in your classes. One option is the adoption of in-class exercises where students have to 1) interact with LLMs to generate code, and then, 2) evaluate the respective solutions. This process would be supervised by the teacher, who would help the students analyze and critique the LLMs' output. This has the advantage of showing students that these models should not be trusted blindly, and that they should inspect and test the generated code, possibly improving their critical thinking, similarly to other mistake-finding exercises [133]. This exercise can be enhanced by having students validate their findings through an AAT using teacher-defined tests or by creating their own unit tests.

Adopt project-based learning

If you currently rely solely on using small assignments for assessments, we recommend that you consider including also project-based evaluations. Projects will require more complex code and interactions to be implemented, which will possibly require more complex interactions with the LLMs to obtain a fully working solution that respects the object-oriented best practices. The projects can be incremental, in order to allow students to grow and apply their knowledge incrementally. For example: consider having a first project delivery where the students do not need to use inheritance, followed by another delivery where inheritance is needed, followed by another delivery where some of the requirements change, and so on. Although LLMs' performance over larger assignments has not yet been measured, there are known benefits to project-based learning [132, 170].

4.2.7 On the importance of citing your sources

One of the interesting features of Bard is its ability to display sources for parts of the generated content. In the case of the “IT Company” assignment, Bard displayed a single source: a GitHub repository with a Java course.¹²

We performed a brief analysis of the repository and found some OOP examples that are incorrect. For example, the repository contains a class called `Product` with a `getPrice()` method, which is extended by the sub-class `ImportedProduct` that redefines the price calculation formula to take into account a customs fee. Instead of overriding the `getPrice()`, `ImportedProduct` declares a new method `totalPrice()`. This is clearly wrong, in light of OOP best practices.

Although we are unsure of how much of this repository actually contributed to Bard's reply, we suspect that it is the use of data sources that have not been curated that results in the generation of code that does not follow the OOP best practices.

Note that GPT-3.5 and GPT-4 do not disclose any information regarding their sources. Since the only publicly available information are high-level descriptions of the training

¹²<https://github.com/camilaabrantest/CursoJava>

datasets (e.g., a filtered version of Common Crawl ¹³, English Wikipedia, among others) [33], it was not possible to perform this analysis for those models.

4.2.8 Limitations

The 3 experiments were done using different interaction interfaces. This may have some kind of impact on the model’s replies, namely in terms of the number of prompts required to obtain a solution.

These models’ output is not deterministic: repeating the experiments for each assignment might have led us to different results.

We did not employ any Prompt Engineering (PE) techniques. The models were initially inputted with the original assignments’ text. The prompts used to obtain missing elements or to guide the models’ toward better solutions consisted of very literal and direct requests. It is possible that the models would have given better answers if some PE techniques were applied.

4.2.9 Conclusions and Future Work

In this work, we present the performance of 3 LLMs when solving several OOP assignments.

From our observations, GPT-4 demonstrated better performance than GPT-3.5, needing fewer interactions and producing code with fewer issues. Additionally, GPT-4 consistently passed more unit tests than GPT-3.5. In comparison, Bard lagged behind both GPT versions in number of issues, passed tests, and the number of interactions needed to guide it toward a reasonable solution.

If the 3 models were students, we would say that GPT-3.5 is a student that has learned the basics of programming but struggles with object-oriented design and programming concepts; GPT-4 is a more experienced student who, with minimal guidance, is able to achieve good solutions; and Bard is a student of a level slightly below GPT-3.5, with less autonomy than the others and more issues with basic programming elements. In the end, all these students struggled somehow and made some mistakes.

But can we blame students for making mistakes when they had a bad teacher? Perhaps the challenges encountered by LLMs in adhering to OOP best practices are not due to inherent reasoning limitations often associated with this technology but rather attributed to suboptimal choices in their information resources. We have verified the poor quality of the source referenced by Bard, and while GPT-3.5 and GPT-4 do not disclose their sources, it is plausible that they too might have relied on inadequate references. The significance of LLMs disclosing their sources, and the more general issue of explaining their reasoning has been a topic of repeated discussion [15]. We not only endorse such explanations but also emphasize the necessity of curating high quality sources for training these models.

¹³<https://commoncrawl.org/the-data>

4.2.10 Data Availability

Since this work is based on real assignments that we use for student assessment in our course, publishing the full assignment and/or interactions dataset would require us to fully recreate all the assignments, ensuring we use unpublished materials when evaluating our students. As such, we have decided to release just one of the assignments: the IT Company Assignment. This was achieved by creating the respective GitHub repository¹⁴, as well as publishing the respective interaction logs, for each LLM, in Zenodo [42].

4.3 Exploring Diagram and Video-Based OOP Exercises to Counter LLM Over-Reliance

4.3.1 Introduction

The advent of large language models (LLM) and their ability to generate computer code from natural language descriptions has led to robust discussion in the computing education community around the opportunities and challenges they present to educators and students [26, 54]. In fact, there is differing opinion amongst educators regarding whether to resist and fight the usage of these tools, or to try and find ways to embrace them [104, 169]. While there have been some initial attempts to integrate LLMs into teaching practice at the introductory level [150, 95], very little is known about the efficacy of these approaches and there is little discussion or consensus about how to adapt higher level courses [152].

Given that LLMs are becoming an essential part of industry practice [24], it is necessary for educators to explore approaches that promote the acquisition of core computing knowledge and skills alongside authentic use of code-generation tools. Tasks which are solvable through pure “copy-and-prompt” approaches may not be engaging or motivating for students. Very recent research suggests using image-based exercises that illustrate expected behaviors through diagrammatic problem descriptions [53]. The goals of this approach, which has only been explored at the introductory programming level, are twofold: first, students have to make an effort to understand and describe the underlying problem, and, second, they can’t just copy the assignment into ChatGPT, Bard or similar tool.

In this research we propose extending the idea of image-based problem specifications to more complex design-oriented tasks suitable for Object-Oriented Programming (OOP) courses. In such courses, the focus is not on implementing stand-alone functions but rather on tasking students with designing and implementing multiple classes that maintain mutable states and collaborate with each other to achieve specific objectives¹⁵. Our aim is also to develop an approach where problems can be presented in such a way that their solution is not uniquely specified. Instead, rather than guiding students towards a single model solution, we want students to have the flexibility to infer their own object-oriented

¹⁴<https://github.com/drop-project-edu/itCompanyAssignment>

¹⁵For example, in a racing simulator, when the ‘accelerate()’ method of a ‘Car’ object is called, the state of the respective ‘Engine’ and ‘Wheel’ objects is also changed.

design by performing a domain analysis¹⁶. To achieve this, we propose a novel notation for expressing program requirements. Problems are then presented to students using this notation, rather than as highly descriptive plain text specifications, which we hypothesize will help mitigate probable LLM-abuse as motivated in prior work [52].

We conducted an experiment with diagram-based OOP exercises and video-based OOP projects, both utilizing a custom notation, and evaluated them in a classroom setting over the course of a semester. Our investigation is guided by the following four research questions (RQs)¹⁷:

RQ1: How do students perceive the new diagram-based exercises compared to more traditional text-based specifications?

RQ2: Are diagram-based exercises perceived by students as being more suitable for certain OOP topics than for others?

RQ3: For larger-scale projects, are video-based specifications motivating for students and do they find them easier for interpreting required behavior?

RQ4: To what extent does the proposed notation for specifying programming tasks discourage students from inappropriate use of LLMs for code-generation?

Recently, both ChatGPT and Bard were updated with the capacity to interpret image content (i.e., ‘vision’)¹⁸. We performed some experiments to determine if these new capabilities could jeopardize our diagramming efforts. Despite being ad-hoc experiments, their relevance to our study led us to include a brief section on them in this section.

This section makes the following contributions:

1. Presents a novel notation to represent OOP exercises;
2. Presents the results of a student survey evaluating diagram-based and video-based exercises, as well as the impact of these new exercise formats on their LLM usage;
3. Presents the results of several ad-hoc experiences using GPT-4 and Bard’s ‘vision’ capabilities to solve diagram-based exercises.

This section is organized as follows. We begin with ‘Notation: Diagrams’ (4.3.2), which presents the proposed format for prescribing LLM-resistant OOP exercises. Afterwards, in ‘Notation: Videos’ (4.3.3), we explore the usage of videos for prescribing more complex assignments (projects). ‘Evaluation’ (4.3.5) describes the research methodology, which consisted of a student survey to assess the impact of the proposed formats in the students’ learning experiences during one semester of an OOP course. We then present the survey’s ‘Results’ (4.3.6) and further discuss them in ‘Discussion’ (4.3.7). ‘Limitations’ (4.3.8)

¹⁶We combine this with automated assessment by defining a frontier class which acts as an API that students must implement. In exercises, we sometimes define mandatory classes, but in the project students are free to design the object-model which supports the API.

¹⁷This section of the thesis presents four research questions (RQ1 to RQ4). In the published version of this research, RQ2 was omitted due to page constraints, and the remaining RQs were renumbered accordingly. However, the original RQ2 is reinstated here for completeness.

¹⁸ChatGPT received ‘vision’ support on September 25, 2023 (only available for paying subscribers) [136]. Google Bard started supporting image input on July 13, 2023 [102].

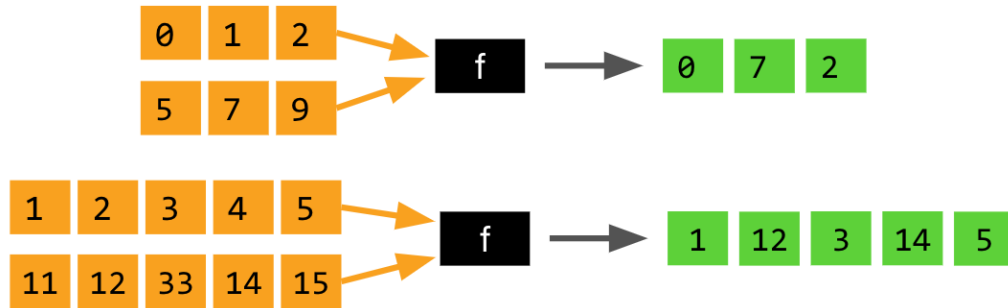


Figure 4.2 – Example algorithmic exercise. Students should understand that the function must create an array with elements from the two input arrays, in alternating fashion.

presents the the potential limitations of our work. Finally, we end the section by presenting some ‘Conclusions’ (4.3.9).

4.3.2 Notation: Diagrams

Our proposed notation aims to address the variety of scenarios that typically occur in OOP, such as functions that change the state of an object, functions that receive multiple objects and associate them with each-other, concepts with common attributes and/or behaviours that should take advantage of the inheritance and polymorphism mechanisms, and so on.

We propose 5 types of diagrams, and explain the relevant notation in the following subsections. A given assignment may combine several diagram types.

Algorithmic function diagram

These diagrams are used to present students with single non-instance (i.e., static) functions that transform input into output, similar to prior research adapting introductory programming courses [52]. See Figure 4.2 for an example.

These diagrams depict the function, with a black box denoting its name, input elements represented using inbound orange arrows, and output elements in green (with outbound black arrows). In cases where inputs or outputs are arrays, lines of small boxes should be utilized, as illustrated in Figure 4.2. The function name within the black box should be “obfuscated,” so as not to provide clues to its purpose (e.g., ‘f()’). This is recommended because some LLMs can generate working solutions from only meaningful function names [18, 204].

A minimum of two input/output examples should be provided to assist in clarifying the expected behavior. In certain instances, more than two examples may be necessary, particularly to illustrate how the function should behave with invalid inputs or boundary cases.

State-change function diagram

These diagrams are similar to the algorithmic function diagrams described in the previous section but are used for non-instance (static) functions that **change the state of one or more objects** and possibly also return a value. While this may be considered bad practice in imperative programming, where students should avoid implementing functions with side-effects, it is common practice in OOP. Since these functions can have two simultaneous effects, a different notation must be used for changing the objects and returning a value. The former is represented by a dashed arrow while the latter is represented by a solid arrow (in concordance with the algorithmic diagrams that only return values). Given that the distinction between solid and dashed arrows may not be immediately evident, a caption is provided to highlight this difference for students. The remaining notation is expected to be inferred by students.

The business rules can be directly written in the diagram as a side note but we consider it more interesting to just provide several examples and allow students to infer those rules. For more complex rules, it may not be feasible to describe them through examples – consider using a state transition rules diagram (see Section 4.3.2) in those cases.

To illustrate this concept, imagine an exercise where students must implement a function which associates ‘Employees’ and ‘Tasks’ within a ‘Company’ if certain conditions (or rules) are met. Figure 4.3 presents such an exercise. The presented function must receive 3 input arguments, as indicated by the orange arrows. Each input is described with an example object, represented by an ellipse annotated with the expected class name (e.g., ‘Company’). Relevant attributes and values (e.g., ‘description’) are drawn inside the ellipse. Relevant notes might also be added (e.g., ‘Employee obj w/ N tasks’). The diagram also conveys that this function has two distinct sub-behaviours: 1) the dashed arrow indicates that the function is expected to change some of its input arguments, and, 2) the solid arrow indicates it should return a boolean. Below the dashed arrow, we see a pink arrow suggesting that some kind of link should be created between the ‘Task’ and ‘Employee’ objects. That association is further explained with a textual hint which reads ‘Employee obj w/ N + 1 Tasks’. Finally, an asterisk near the expected return value presents rules that must be respected.

Notice that students are expected to implement not only the function but also the classes and methods that support its behavior.

Class declaration diagram

These diagrams provide students with guidelines for implementing predefined classes, attributes, and relationships (e.g., composition). Each class should be depicted within a box, enumerating its attributes (optionally with their types), specifying the necessary constructors, and outlining the expected behavior of fundamental methods like ‘toString()’. The behavior can be described similarly to algorithmic diagrams, with examples of input/output and corresponding arrows. It is important to note that the direct representation of

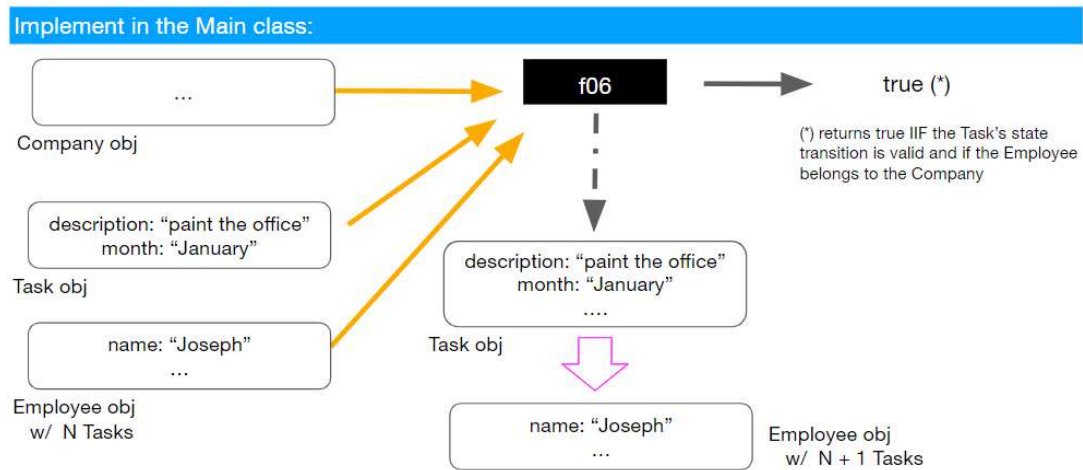


Figure 4.3 – Example state-change function exercise. Students should understand that the function must receive 3 objects (a ‘Company’, an ‘Employee’, and a ‘Task’) and associate them if some rules were respected. The rules are detailed in Figure 4.6’s diagram.

relationships between classes is omitted. Specific attributes are linked with an accompanying image, indicating that they are objects themselves.

See Figure 4.4 for an example where students had to implement two related classes. In that example, students had to infer that the small house-like figure within the ‘Person’ class diagram represented a composition relationship between the ‘Person’ and ‘Apartment’ classes. Note also the reference to the ‘toString()’ function, with a black arrow which indicates the expected return value, similar to the previously discussed diagrams for algorithmic exercises.

Although this information could be partially represented by UML class diagrams, this proposal has two advantages: (1) it presents both structure (e.g., the class’s attributes) and behaviour (e.g., ‘toString()’s expected return-value); and, (2) the composition relationship is less explicit than it would be in UML.

Inheritance diagram

These diagrams are an extension (or adaptation) of the previously mentioned class declaration diagrams, specifically tailored for exercising inheritance relationships between classes (i.e., generalization via inheritance). The notation is exactly the same, however the goal is different. Instead of describing all the classes that the students must implement (as in class diagrams), they only describe the child classes and students must determine the structure and behaviour of the parent class (which is absent from the diagram). After this, they must implement all classes (structure and behavior), both those explicitly outlined in the diagram and classes that are not present in the diagram but have been identified by the students.

Figure 4.5 presents one of these diagrams, in which two similar concepts are represented: ‘Managers’ and ‘IT Technicians’. Students are expected to infer that these two concepts should share a common super-class since they have common attributes (e.g.,

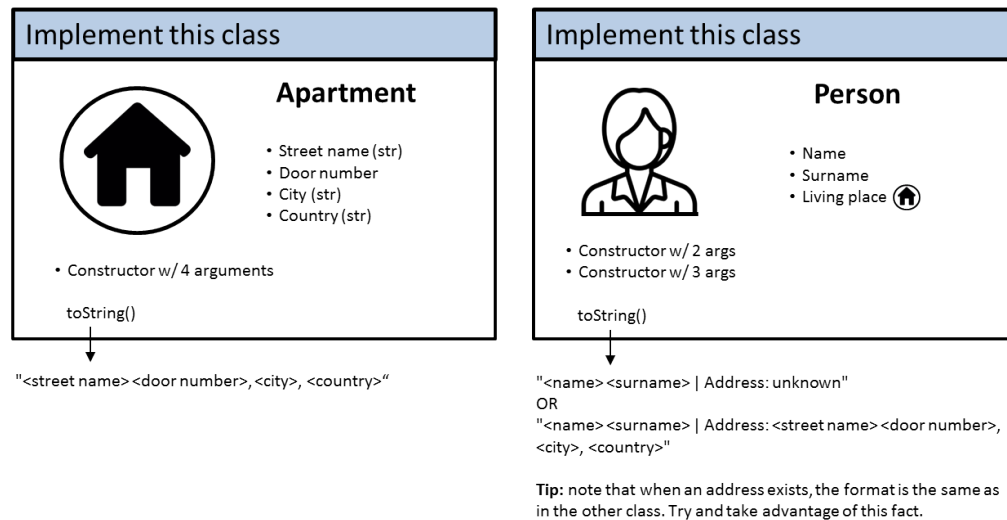


Figure 4.4 – Class declaration exercises. Students should infer that the ‘Person’ class has a composition relationship with the ‘Apartment’ class. Some data-types were omitted.

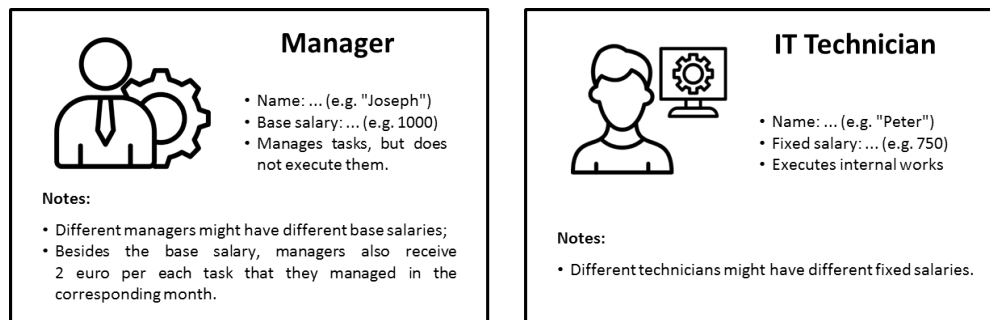


Figure 4.5 – Example inheritance exercise. Students were expected to understand that some common elements exist between the ‘Manager’ and the ‘IT Technician’, and create an inheritance relationship with a super-class above those.

name, salary) and similar behaviours (e.g., salary calculation).

State transition rules diagram

These diagrams should be used when the exercise has certain classes which can transition between different internal states, with rules which guide those transitions. In practical terms, these diagrams are simplified state transition diagrams which only display valid transitions. See Figure 4.6 for an example. That figure’s diagram explains the name of the action which changes the state (e.g., ‘plan’, ‘start’), as well as what information should exist inside the object in each state. For example, a newly created task only has value for the ‘description’.

These diagrams are useful for providing complementary information to the other exercises. For example, the state transition diagram of Figure 4.6 and the state-change function diagram of Figure 4.3 were part of the same assignment and are complementary: for students to correctly implement the ‘f06()’ function (i.e., to assign a ‘Task’ to an ‘Em-

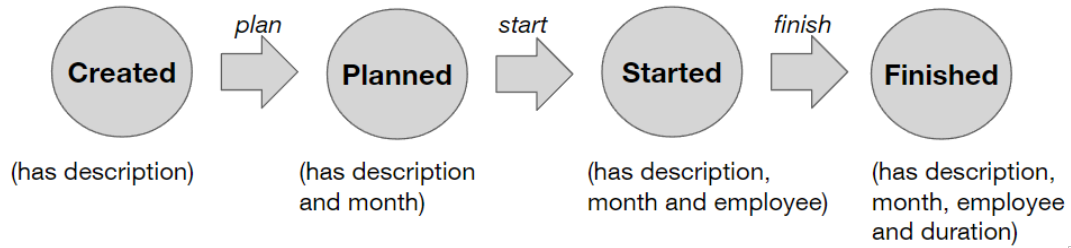


Figure 4.6 – State transition rules. This is a simplified state transition diagram which only displays valid transitions.

ployee’), they must understand that the association can only be done if the ‘Task’ is in the ‘Planned’ state.

4.3.3 Notation: Videos

While some exercises are easy to describe with a diagram containing 2 or 3 example inputs, or even with 2 or 3 diagrams, in more complex assignments, where there is a significant amount of user interaction or input validation, or projects which span multiple weeks or months, creating diagrams to represent all relevant behaviours and interactions is complex and time consuming.

In such cases, it is more practical to create video demonstrations of the expected behaviours. The videos can include a mixture of behavioural demonstrations and static diagrams. More concretely, we recommend using demonstrations for explaining user interactions, and diagrams for presenting any mandatory protocols which the students must follow. The videos should start with the demonstration of the expected functionality from the user’s perspective, and only afterwards go into required implementation details.

A video¹⁹ with an example OOP course project is available online²⁰.

4.3.4 Method

Experimental context

To evaluate the proposed diagrams and videos as formats to present students with OOP tasks, we applied them in the 2023/24 offering of a course focused on Object-Oriented Design and Programming taught at *Lusófona University*. In this course, students are exposed to OO analysis and design, as well as Java implementation details. They are also challenged to solve a number of practical programming activities, with the course following a mixed approach with both exercise-based and project-based learning.

The diagrammatic notation and videos proposed in this section were applied to the majority of these practical activities, which were originally described in natural language. Diagrams were used to describe the exercises, while the project was described using the video format.

¹⁹Narrated in Portuguese, but auto-translated captions are available.

²⁰https://www.youtube.com/watch?v=_WWpLkIc6S8

Table 4.2 – Survey questions categorized by Research Question.

RQ	Q. #	Question
1	1	“It is easier to understand the objective of the exercise when presented in the form of a diagram, than when it is presented textually.”
1	2	“When I come across a diagram-based exercise, I tend to think more carefully before writing code, compared to what I do with exercises described textually.”
1	3	“In general, I prefer exercises based on diagrams over exercises described textually.”
1	4	“Considering the interpretation of the exercise, the absence of the function name in the diagrams ...”
2	1 .. 4	“The use of diagrams to represent exercises of this kind is appropriate.” (kind=algorithmic $\vee$ state-change function $\vee$ state transition rules $\vee$ inheritance)
3	1	“It is easier to interpret the project when presented in video format, due to the combination of text, images and audio.”
3	2	“Video statements fall short when compared to statements in natural language, because it is more difficult to take notes on paper”
3	3	“I feel more motivated to do the project when the statement is in video than when it is in natural language.”
3	4	“I find it easier to develop the project with the video statement than with the traditional model”.
4	1	“I am more likely to use GPT/Bard with textual exercise descriptions than with diagrams and/or videos”.
4	2	“Diagram and/or video exercises effectively prevent abuse of GPT/Bard, as they force me to prepare a prompt instead of simply copying the description into GPT/Bard.”
4	3	“I consider diagram and/or video exercises a good step towards making me more prepared for a professional future where I have to interact with GPT/Bard.”

4.3.5 Evaluation

To assess the effectiveness of this novel notation, students were administered a structured anonymous questionnaire during the 12th week of the course. At this juncture, students had been exposed to multiple diagrams of all types presented in this section: this includes 10 assignments, each composed of multiple diagrams. On total, 72 unique diagrams were made available. Also, students were actively engaged in the project.

The questionnaire was composed of 15 quantitative questions, each associated with one RQ. Fourteen of the questions – presented in Table 4.2 – used a standard 5-point Likert scale (strongly disagree to strongly agree). One question was categorical, with three possible options. Finally, the questionnaire included a qualitative question for students to provide open-response comments.

4.3.6 Results

The course had 115 enrolled students. Among them, 84 students had some level of participation, submitting at least one diagram-based assignment. For those 84 students, the average number of submitted assignments was 6.25 and the mean was 7 (out of the 10 available assignments). The minimum number of submitted assignments was 1, while the maximum and mode were both 10 (12 students submitted all 10 assignments). A total of 56 students participated in the survey.

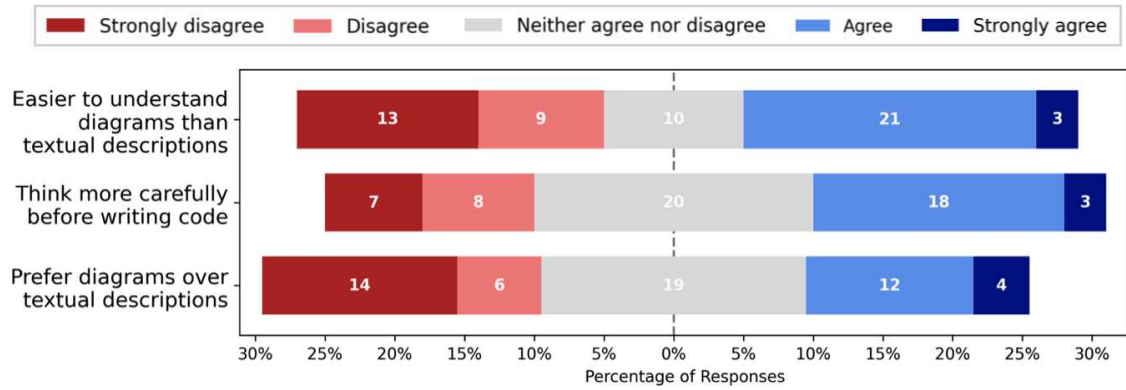


Figure 4.7 – Students’ opinions with regards to diagram-based exercises when compared to natural language exercises (RQ1).

RQ1: Perception of diagram-based exercises

Response distributions to the three quantitative questions associated with RQ1 are presented in Figure 4.7. Although the results are somewhat balanced between the agreement and disagreement sides, in the first two questions, more students agree with the benefits of diagrammatic exercises. However, in the third question, results were skewed to the disagreement side, indicating that most students do not prefer diagram-based exercises over text-based ones.

Finally, we asked students to evaluate the impact of the absence of the function name from the diagrams (Table 4.2: Question 4, RQ1). This categorical question had 3 possible options: “It causes me some difficulties” (selected by 37 students or 66.07%), “It doesn’t affect me” (14 students or 25.00%), and, “It causes me a lot of difficulties” (5 students or 8.93%). We were somewhat surprised by the results, as we expected most students to find the absence of function names more challenging.

RQ2: Suitability for various OOP topics

Figure 4.8 presents the survey’s results for each exercise or diagram type. In all diagram types, more students indicated some level of agreement than the students who indicated some level of disagreement. However, it is possible to see that some diagrams were better received by the majority of the students than others. For example, 65.31% of students who used the ‘Inheritance’ diagrams indicated either agreement or strong agreement with their suitability for the respective exercises, closely followed by the ‘Algorithmic’ diagrams, which were considered appropriate by 52.73% of the students who used them. On the other hand, both the ‘state-change function’ and the ‘state transition’ diagrams were only considered appropriate by 45.28% and 47.06% of students, respectively. As such, these aspects of the notation can still be improved to better enhance their effectiveness.

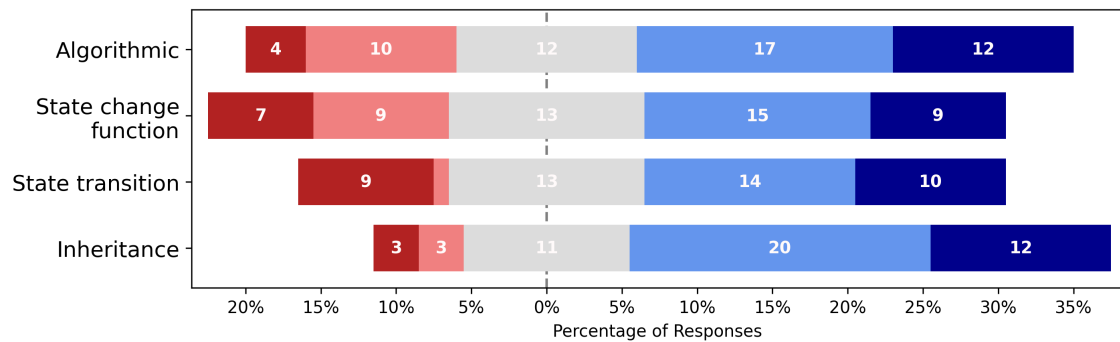


Figure 4.8 – Students' opinions about each of the diagram types (RQ2). In all cases, the mode was 4 (Agreement), with positive agreement values consistently exceeding negative ones. 'Inheritance' and 'algorithmic' diagrams were better received than the 'state-change' and 'state transition' diagrams.

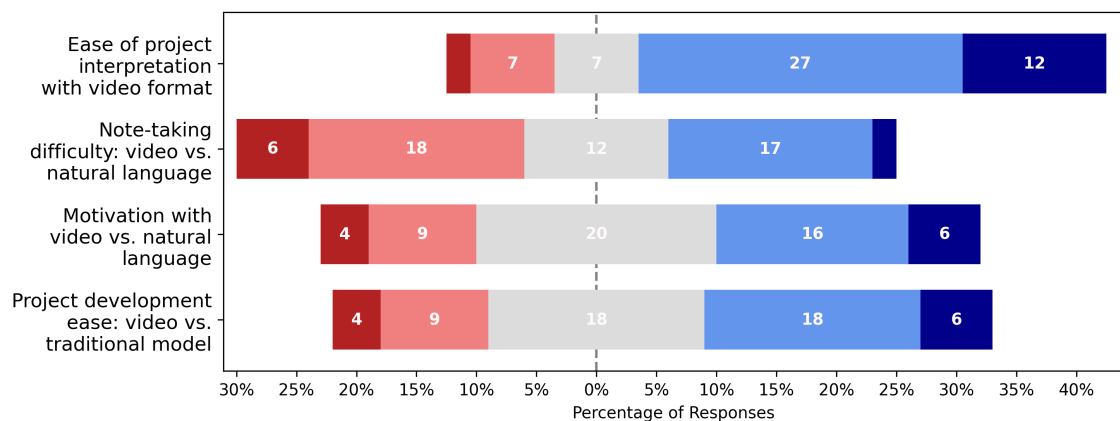


Figure 4.9 – Distribution of student replies to questions about video-based exercises (RQ3).

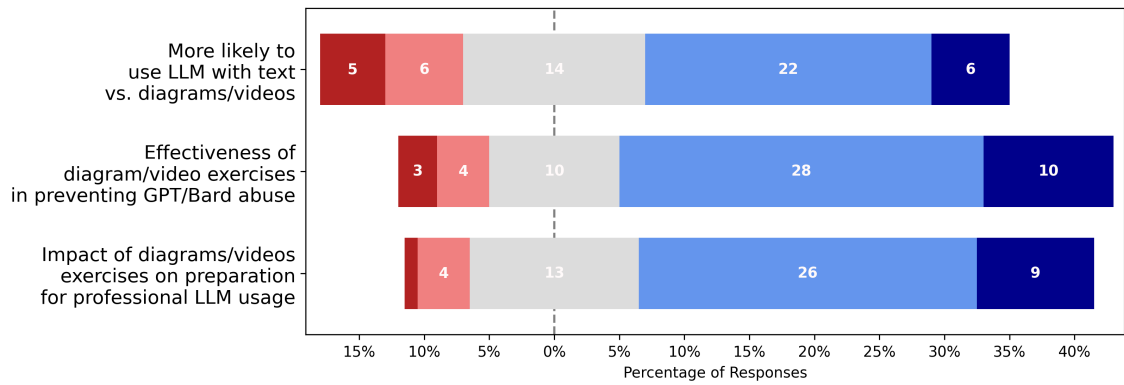


Figure 4.10 – Students’ opinions on diagram/video exercises’ effect on LLM usage (RQ4). Students generally find that these new formats effectively reduce LLM over-reliance and potentially prepare them for their future careers.

RQ3: Interpretability and motivation of video-based projects

Figure 4.9 presents the results for each question related with RQ3. The vast majority of students indicated that videos make projects easier to interpret. However, with regards to the ease of note taking, most students feel that videos are not as good as textual descriptions. Videos also seem to have a positive impact on students’ motivation (Table 4.2: Question 3, RQ3). Finally, the majority of students reported that developing the project with the video statement is easier than with the traditional text-based assignments.

RQ4: Use and Reliance on LLMs

Figure 4.10 presents results for questions related with RQ4. The majority of students indicate that they would be more likely to resort to an LLM when solving textual exercises than with diagrammatic or video exercises. The vast majority of students also agree that diagrams and videos are an effective way of preventing LLM abuse. Lastly, students agree that these formats will better prepare them for potential future careers where they have to use LLMs to produce code [24].

4.3.7 Discussion

The proposed notation and diagrams present a new visual language relevant for the specificities of teaching and learning OO design and programming. These diagrams require that students first understand the problem by analysing the diagrammatic sample sets, and then proceed to define an object model and/or algorithm which solves it, with or without the help of a LLM.

Diagrams and videos can prescribe more or less directed tasks, according to educators’ goals, by using clues such as ‘Implement in class Main’, ‘Represent these concepts’, and so on. However, educators should avoid exposing too much information textually or verbally, since some textual instructions can be parsed from images [84] and audio interpretation tools are also emerging [136].

We believe this model has three advantages: (1) it forces students to interpret test cases and infer a problem description; (2), it should prevent direct ‘copy-and-prompting’ from the assignment description to the LLM; and, (3) it requires students that wish to use an LLM to create prompts that guide it towards the goal. Although these last two ideas might seem contradictory, we argue that both have pedagogical value, since it is important that students gain some experience with using LLMs in an authentic way, as helpers for solving coding problems, due to the likelihood that they will use these tools professionally [24].

During the experiment, we observed some interesting student behaviours which we believe should guide future work on this topic. One interesting aspect was, at least in some cases, students’ difficulties in interpreting the diagrams prompted interesting discussions between students and teachers. When the doubts were not obvious, we would engage with the student and help them reach the expected interpretation by themselves. For example, when a student asked “What does this ‘N + 1’ tasks mean?”, the teacher pointed out that, before the function call, the object indicated ‘N tasks’, and then asked the student what they thought was happening that resulted in the change from ‘N tasks’ to ‘N + 1 tasks’. As for the videos, at least one student transcribed the video to a text document. We informally asked the student for their reasons, and they indicated doing it to support offline work because they didn’t have an internet connection at all times. One other student used a tool to download Youtube’s automatic captions for the video assignment. Although these techniques could be used to exploit the video assignment, since not all information is obtainable from the narration, the videos should remain LLM-resistant for the time being.

Besides countering LLM over-reliance, the survey shows that these approaches also seem to have other benefits, such as the apparent positive impact on students’ motivation levels. Moreover, the qualitative question also allowed us to identify another added benefit, since multiple students reported the need to analyse and reflect on the diagram’s contents, in order to fully understand it. For example, participant S7 indicated that “*Not knowing specifically what the function should do [...], I had to ‘spend’ some time trying to understand that*”. Another participant, S39, commented that “*Since I don’t have a [function] name, I don’t immediately know what I’m supposed to do, I have to think and reflect on the diagram*”. Finally, S56 indicated that “*Sometimes, just the diagram without a brief contextual explanation made the exercise more difficult to understand than solving it.*”. As such, it appears that diagrams might also promote some development of analytical skills, and this would be an interesting avenue to explore more rigorously in future work.

GPT-4 and Bard’s vision capabilities

Recently, both GPT-4 and Bard have become capable of interpreting images. Given the potential impact to our proposed diagram-based approach, we conducted initial explorations of the vision capabilities of these tools. As such, GPT-4 and Bard were supplied

Table 4.3 – Results of the ‘vision’ experiments. CS (compilation score) covers function location and protocol; LS (logic score) covers function behaviour. The first three columns show diagram-based outcomes, while the fourth column details text-based results.

	Experiment 1 (diagram)		Experiment 2 (diagram)		Experiment 3 (diagram)		Experiment 4 (textual)	
	CS	LS	CS	LS	CS	LS	CS	LS
GPT-4	0/3	1/3	0/3	1/3	3/3	0/3	2/3	3/3
Bard	0/3	0/3	0/3	1/3	0/3	0/3	3/3	1.5/3

with the same information that students had: a diagram containing some introductory text and Figure 4.6, and then a second diagram which was equal to Figure 4.3. This experiment was repeated 3 times with the same images, but with slightly different prompts and the generated code was evaluated considering 3 compilation and 3 logical items. None of the attempts yielded a correct implementation of the ‘f06()’ function, with both models generating code with compilation errors and/or logical errors. We also supplied both models with an equivalent text-based description of the exercise. With the textual input, GPT-4 was able to generate an almost correct solution, failing only to declare the function as ‘static’. Bard’s solution to the text-based variant was also better than all its diagram-based attempts. See Table 4.3 for an overview of our results. The logs of our experiments are available online in [45].

This experiment reveals that LLMs perform much better with text-based OOP exercises than with diagram-based ones. As such, diagrams seem a good approach to limit students’ over-reliance on LLMs, at least for the time being²¹.

4.3.8 Limitations

One limitation is the significant proportion of students opting for the neutral option in most quantitative questions. This could potentially have skewed some of our interpretations.

In relation to the questions about the impact of diagrams and videos on the utilization of LLMs, it is conceivable that some students may not have provided honest responses to this question given concerns around academic integrity.

4.3.9 Conclusions

We believe that the biggest challenge in adapting courses and classes to this brave new world where LLMs with code generation capacity are easily accessible is creating exercises that are hard for an LLM to solve, but are still accessible and can be solved by students. In this section we present a novel pedagogical approach to describe OO programming exercises and projects. This notation allowed us to replace the previously used natural language descriptions which could be solved, to some degree, by GPT-3.5, GPT-4 and Bard [41, 43]. The proposed notation was well received by our students, who also agree that it helps mitigate LLM over-reliance.

²¹This experiment was performed in the last week of December, 2023.

Chapter 5

Conclusions and Future Work

This thesis focuses on two CSE topics of technological nature: Automatic Assessment Tools and Large Language Models, specifically when applied to Object-Oriented Programming Education.

We believe that the pedagogical and technical tools used for teaching OOP should continuously evolve over time, taking advantage of technology as possible. This requires that educators pay attention to technological advancements and determine if and how they should be integrated in the respective courses. This is what we tried to do with both AATs and LLMs.

In the **AAT** side of things, we proposed two new tools: the Drop Project (DP) AAT and a complementary IDE plugin. Drop Project’s characteristics (multi-file, unit-tests-based and supporting code quality evaluations) make it adequate for validating OO projects. Furthermore, DP is an Open-Source, documented and unit-tested tool, which is under continuous development with new features being implemented and bugs being fixed. This allows other educators to adopt it in their courses if they see fit.

With the complementary IDE plugin, we have made DP more apt at handling small exercises, which also has benefits in terms of OO education since it facilitates the creation of mixed learning environments.

We also documented our knowledge in terms of test-case construction feedback definition. This knowledge resulted from our experience of working with automatic assessment for the last seven years. This resulted in a publication in which we try to raise awareness to the need to define, promote and use best practices in terms of 1) defining unit-tests for AAT-based assessments, 2) define useful feedback for better supporting students.

With these 3 publications, we believe we have contributed to improving AATs and their usage in OOP education, which was our first research goal.

On the **LLM** side of things, we started by evaluating the impact of some of these tools – GPT-3.5, GPT-4 and Bard – in OOP education. This evaluation was done using Drop Project¹. Drop Project’s capacity to evaluate functional as well as code quality requirements allowed us to evaluate the code generated by the LLMs and capture it’s

¹The Open-Source AAT which we proposed in our first AAT-related publication

strengths and weaknesses. To perform the evaluation, we resorted to multiple real-world OOP assignments which had been previously used for grading students. This evaluation allowed us to inform the CSE community about some particularities of LLM-generated OOP code. First, we reported that the tested tools are able to achieve what can be described as ‘passing scores’. This leads to the conclusion that some students might abuse these tools to obtain undeserved passing grades. Second, we reported that the tested LLMs do not properly use some of the paradigm’s techniques, such as inheritance and polymorphism. This means that even if students use these tools responsibly – for example, by referring to them to get help – they are at risk of learning from bad examples.

After researching and learning about the potential impacts of LLMs in OOP education, we decided to investigate ways of mitigating students’ over-reliance on these tools. In this scope, we devised a new visual notation for prescribing OOP exercises and evaluated it over the course of a semester, both in small exercises described as diagrams, as well as in a larger project, described in video. This visual notation requires that students first make an effort to understand the problem, and then proceed to define an object model and/or algorithm which solves it, with or without the help of an LLM. As such, it should prevent direct ‘copy-and-prompting’. These new formats were well received by our students, who reported other effects, such as the need to spend more time thinking on interpreting the problem (which potentially promotes the development of analysis skills), and being more motivated to work on video-based projects.

With our 3 LLM-oriented publications, we believe we have informed the CSE community about the particularities of LLM-generated OO code and we also believe that we have contributed to the responsible integration of LLMs in OOP courses, which was our second research goal.

Looking beyond the scope of the current thesis, we would like to further develop these two areas of knowledge, finding ways of integrating AATs and LLMs. We have done some preliminary work on this topic, by feeding the AAT feedback and assignment context into OpenAI’s API, allowing students to request GPT’s help for their assignments [147]. We plan to evaluate the impact of said integration in terms of providing helpful feedback to students.

Finally, we believe that it is of the utmost importance that educators and researchers keep making efforts in terms of responsibly integrating LLMs in Object-Oriented Programming courses. This work is an important step in that direction.

Bibliography

- [1] Codeforces. <https://codeforces.com/>. [Online; last accessed 18-July-2024].
- [2] Drop Project. <https://plugins.jetbrains.com/plugin/21870-drop-project>. [Online; last accessed 28-June-2024].
- [3] Drop Project API. <https://deisi.ulusofona.pt/drop-project/swagger-ui/>. [Online; last accessed 28-June-2024].
- [4] Drop Project for IntelliJ IDEA. <https://github.com/drop-project-edu/Drop-Project-for-IntelliJ-Idea>. [Online; last accessed 28-June-2024].
- [5] Introducing Claude. <https://www.anthropic.com/index/introducing-claude>. [Online; last accessed 02-May-2024].
- [6] Kattis Problem Archive. <https://open.kattis.com/>. [Online; last accessed 18-July-2024].
- [7] Use of Generative Artificial Intelligence in Courses. <https://ai.northwestern.edu/education/use-of-generative-artificial-intelligence-in-courses.html>, August 2023. [Online; last accessed 07-February-2024].
- [8] ACHIAM, J., ADLER, S., AGARWAL, S., AHMAD, L., AKKAYA, I., ALEMAN, F. L., ALMEIDA, D., ALTENSCHMIDT, J., ALTMAN, S., ANADKAT, S., ET AL. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
- [9] ALLAL, L. B. Open Large Language Models for Code. <https://llm4code.github.io/2024/slides/BigCode-LLM4Code.pdf>, 2024. Keynote address at the LLMs4Code Workshop, International Conference on Software Engineering.
- [10] ALLAL, L. B., LI, R., KOCETKOV, D., MOU, C., AKIKI, C., FERRANDIS, C. M., MUENNIGHOFF, N., MISHRA, M., GU, A., DEY, M., ET AL. SantaCoder: Don't Reach For The Stars! *arXiv preprint arXiv:2301.03988* (2023).
- [11] ALLEN, J. M., DOWNEY, K., MILLER, K., ET AL. Many Small Programs in CS1: Usage Analysis from Multiple Universities. In *2019 ASEE Annual Conf. & Exposition* (Tampa, Florida, June 2019), no. 10.18260/1-2-33084, ASEE Conferences, p. 1–13. <https://peer.asee.org/33084>.
- [12] AMELUNG, M., FORBRIG, P., AND RÖSNER, D. Towards Generic and Flexible Web Services for E-Assessment. *ACM SIGCSE Bulletin* 40 (2008), 5.
- [13] ANTAL, G., VOZÁR, R., AND FERENC, R. Toward a New Era of Rapid Development: Assessing GPT-4-Vision's Capabilities in UML-Based Code Generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code* (2024), pp. 84–87.

- [14] APACHE SOFTWARE FOUNDATION. Apache Maven Project. <https://maven.apache.org/>, 2021. [Online; last accessed 17-February-2021].
- [15] ARRIETA, A. B., DÍAZ-RODRÍGUEZ, N., DEL SER, J., BENNETOT, A., TABIK, S., BARBADO, A., GARCÍA, S., GIL-LÓPEZ, S., MOLINA, D., BENJAMINS, R., ET AL. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information fusion* 58 (2020), 82–115.
- [16] ASARE, O., NAGAPPAN, M., AND ASOKAN, N. Is GitHub’s Copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 129.
- [17] ATHIWARATKUN, B., GOUDA, S. K., WANG, Z., LI, X., TIAN, Y., TAN, M., AHMAD, W. U., WANG, S., SUN, Q., SHANG, M., ET AL. Multi-lingual Evaluation of Code Generation Models. *arXiv preprint arXiv:2210.14868* (2022).
- [18] BABE, H. M., NGUYEN, S., ZI, Y., GUHA, A., FELDMAN, M. Q., AND ANDERSON, C. J. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. *arXiv preprint arXiv:2306.04556* (2023).
- [19] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv preprint arXiv:1409.0473* (2014).
- [20] BAILLIE, C., AND FITZGERALD, G. Motivation and attrition in engineering students. *European Journal of Engineering Education* 25, 2 (2000), 145–155.
- [21] BALTAZAR, B. Drop Project Student Plugin. <https://www.youtube.com/watch?v=N2YUvU3Lxm4>, 2023.
- [22] BANDY, J., AND VINCENT, N. Addressing “Documentation Debt” in Machine Learning Research: A Retrospective Datasheet for BookCorpus. *arXiv preprint arXiv:2105.05241* (2021).
- [23] BANIASSAD, E., ZAMPROGNO, L., HALL, B., AND HOLMES, R. Stop the (auto-grader) insanity: Regression penalties to deter autograder overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (2021), pp. 1062–1068.
- [24] BARKE, S., JAMES, M. B., AND POLIKARPOVA, N. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (Apr 2023).
- [25] BECKER, B. A. A survey of Introductory Programming Courses in Ireland. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (2019), pp. 58–64.
- [26] BECKER, B. A., DENNY, P., FINNIE-ANSLEY, J., LUXTON-REILLY, A., PRATHER, J., AND SANTOS, E. A. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2023), SIGCSE 2023, ACM, p. 500–506.

- [27] BENDER, E. M., GEBRU, T., McMILLAN-MAJOR, A., AND SHMITCHELL, S. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency* (2021), pp. 610–623.
- [28] BI, X., CHEN, D., CHEN, G., CHEN, S., DAI, D., DENG, C., DING, H., DONG, K., DU, Q., FU, Z., ET AL. Deepseek LLM: Scaling Open-Source Language Models with Longtermism. *arXiv preprint arXiv:2401.02954* (2024).
- [29] BIDERMAN, S., AND RAFF, E. Fooling MOSS Detection with Pretrained Language Models. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (2022), pp. 2933–2943.
- [30] BOETTIGER, C. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 71–79.
- [31] BRIDSON, K., ATKINSON, J., AND FLEMING, S. D. Delivering Round-the-Clock Help to Software Engineering Students Using Discord: An Experience Report. In *53rd ACM Technical Symposium on Computer Science Education-Volume 1* (2022), pp. 759–765.
- [32] BROWN, P. F., COCKE, J., DELLA PIETRA, S. A., DELLA PIETRA, V. J., JELINEK, F., LAFFERTY, J., MERCER, R. L., AND ROOSSIN, P. S. A statistical approach to machine translation. *Computational linguistics* 16, 2 (1990), 79–85.
- [33] BROWN, T., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J. D., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., AGARWAL, S., HERBERT-VOSS, A., KRUEGER, G., HENIGHAN, T., CHILD, R., RAMESH, A., ZIEGLER, D., WU, J., WINTER, C., HESSE, C., CHEN, M., SIGLER, E., LITWIN, M., GRAY, S., CHESSE, B., CLARK, J., BERNER, C., MCCANDLISH, S., RADFORD, A., SUTSKEVER, I., AND AMODEI, D. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems* (2020), vol. 33, Curran Associates, Inc., pp. 1877–1901.
- [34] BUBECK, S., CHANDRASEKARAN, V., ELDAN, R., GEHRKE, J., HORVITZ, E., KAMAR, E., LEE, P., LEE, Y. T., LI, Y., LUNDBERG, S., ET AL. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712* (2023).
- [35] CARTER, T. Some universities are ditching AI detection software amid fears students could be falsely accused of cheating by using ChatGPT. <https://www.businessinsider.com/universities-ditch-ai-detectors-over-fears-students-falsely-accused-cheating-2023> September 2023. [Online; last accessed 07-February-2024].
- [36] CHANG, K. K., CRAMER, M., SONI, S., AND BAMMAN, D. Speak, Memory: An Archaeology of Books Known to ChatGPT/GPT-4. *arXiv preprint arXiv:2305.00118* (2023).
- [37] CHEN, L., ZAHARIA, M., AND ZOU, J. How is ChatGPT’s Behavior Changing over Time? *arXiv preprint arXiv:2307.09009* (2023).

- [38] CHEN, M., TWOREK, J., JUN, H., YUAN, Q., PINTO, H. P. D. O., KAPLAN, J., EDWARDS, H., BURDA, Y., JOSEPH, N., BROCKMAN, G., ET AL. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [39] CHOMSKY, N. *Syntactic Structures*. De Gruyter Mouton, Berlin, Boston, 1957.
- [40] CIPRIANO, B. P. Towards the Integration of Large Language Models in an Object-Oriented Programming Course. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 2*. 2024, pp. 832–833.
- [41] CIPRIANO, B. P., AND ALVES, P. GPT-3 vs Object Oriented Programming Assignments: An Experience Report. In *Proceedings of the 2023 Conf. on Innovation and Technology in Computer Science Education V. 1* (New York, NY, USA, 2023), ITiCSE 2023, ACM, p. 61–67.
- [42] CIPRIANO, B. P., AND ALVES, P. How GPT-3.5, GPT-4 and Bard handled an Object Oriented Programming Assignment - Full Interaction Logs. <https://doi.org/10.5281/zenodo.8246165>, August 2023.
- [43] CIPRIANO, B. P., AND ALVES, P. LLMs Still Can't Avoid Instanceof: An Investigation Into GPT-3.5, GPT-4 and Bard's Capacity to Handle Object-Oriented Programming Assignments. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training* (New York, NY, USA, 2024), ICSE-SEET '24, Association for Computing Machinery, p. 162–169.
- [44] CIPRIANO, B. P., ALVES, P., AND DENNY, P. A Picture Is Worth a Thousand Words: Exploring Diagram and Video-Based OOP Exercises to Counter LLM Over-Reliance. In *Proceedings of the 19th European Conference on Technology Enhanced Learning* (2024), Springer Nature Switzerland.
- [45] CIPRIANO, B. P., ALVES, P., AND DENNY, P. Exploring Diagram and Video-Based OOP Exercises to Counter LLM Over-Reliance - Vision Experiments . <https://doi.org/10.5281/zenodo.11552463>, June 2024.
- [46] CIPRIANO, B. P., BALTAZAR, B., ALVES, P., AND FACHADA, N. Drop Project Student Plugin for IntelliJ IDEA - Evaluation Survey. <https://doi.org/10.5281/zenodo.8432997>, Oct 2023.
- [47] CIPRIANO, B. P., BALTAZAR, B., FACHADA, N., VOURVOPOULOS, A., AND ALVES, P. Bridging the Gap between Project-Oriented and Exercise-Oriented Automatic Assessment Tools. *Computers* 13, 7 (2024).
- [48] CIPRIANO, B. P., FACHADA, N., AND ALVES, P. Drop project: An automatic assessment tool for programming assignments. *SoftwareX* 18 (2022), 101079.
- [49] COLEY, M. Guidance on AI Detection and Why We're Disabling Turnitin's AI Detector. <https://www.vanderbilt.edu/brightspace/2023/08/16/guidance-on-ai-detection-and-why-were-disabling-turnitins-ai-detector>, August 2023. [Online; last accessed 07-February-2024].
- [50] DAUN, M., AND BRINGS, J. How chatgpt will change software engineering education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (2023), pp. 110–116.

- [51] DE SOUZA, D. M., OLIVEIRA, B. H., MALDONADO, J. C., SOUZA, S. R., AND BARBOSA, E. F. Towards the use of an automatic assessment system in the teaching of software testing. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings* (2014), IEEE, pp. 1–8.
- [52] DENNY, P., LEINONEN, J., PRATHER, J., LUXTON-REILLY, A., AMAROUCHE, T., BECKER, B. A., AND REEVES, B. N. Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators. *arXiv preprint arXiv:2307.16364* (2023).
- [53] DENNY, P., LEINONEN, J., PRATHER, J., LUXTON-REILLY, A., AMAROUCHE, T., BECKER, B. A., AND REEVES, B. N. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2024), SIGCSE 2024, ACM, p. 296–302.
- [54] DENNY, P., PRATHER, J., BECKER, B. A., FINNIE-ANSLEY, J., HELLAS, A., LEINONEN, J., LUXTON-REILLY, A., REEVES, B. N., SANTOS, E. A., AND SARSA, S. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan 2024), 56–67.
- [55] DENNY, P., SMITH, D. H., FOWLER, M., PRATHER, J., BECKER, B. A., AND LEINONEN, J. Explaining code with a purpose: An integrated approach for developing code comprehension and prompting skills, 2024.
- [56] DESTEFANIS, G., BARTOLUCCI, S., AND ORTU, M. A preliminary analysis on the code generation capabilities of gpt-3.5 and bard ai models for java functions. *arXiv preprint arXiv:2305.09402* (2023).
- [57] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [58] DIANA, N., EAGLE, M., STAMPER, J., GROVER, S., BIENKOWSKI, M., AND BASU, S. An Instructor Dashboard for Real-Time Analytics in Interactive Programming Assignments. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference* (2017), pp. 272–279.
- [59] DOUCE, C., LIVINGSTONE, D., AND ORWELL, J. Automatic Test-Based Assessment of Programming: A Review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (Sept. 2005), 4–es.
- [60] DOUGHTY, J., WAN, Z., BOMPELLI, A., QAYUM, J., WANG, T., ZHANG, J., ZHENG, Y., DOYLE, A., SRIDHAR, P., AGARWAL, A., ET AL. A Comparative Study of AI-Generated (GPT-4) and Human-crafted MCQs in Programming Education. In *Proceedings of the 26th Australasian Computing Education Conference* (2024), pp. 114–123.
- [61] ECHTERHOFF, J., LIU, Y., ALESSA, A., MCAULEY, J., AND HE, Z. Cognitive Bias in High-Stakes Decision-making with LLMs. *arXiv preprint arXiv:2403.00811* (2024).

- [62] EDWARDS, S. H., BÖRSTLER, J., CASSEL, L. N., HALL, M. S., AND HOLLINGSWORTH, J. Developing a common format for sharing programming assignments. *ACM SIGCSE Bulletin* 40, 4 (2008), 167–182.
- [63] EDWARDS, S. H., AND MURALI, K. P. CodeWorkout: Short Programming Exercises with Built-in Data Collection. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education* (2017), pp. 188–193.
- [64] EDWARDS, S. H., AND PEREZ-QUINONES, M. A. Web-cat: Automatically grading programming assignments. *SIGCSE Bull.* 40, 3 (June 2008), 328.
- [65] ELENA ALSTON. ChatGPT vs. GPT: What’s the difference? <https://zapier.com/blog/chatgpt-vs-gpt/>. [Online; last accessed 17-July-2024].
- [66] ENSTRÖM, E., KREITZ, G., NIEMELÄ, F., SÖDERMAN, P., AND KANN, V. Five Years with Kattis — Using an Automated Assessment System in Teaching. In *2011 Frontiers in education conference (FIE)* (2011), IEEE, pp. T3J–1.
- [67] ESPINHA GASIBA, T., IOSIF, A.-C., KESSBA, I., AMBURI, S., LECHNER, U., AND PINTO-ALBUQUERQUE, M. May the source be with you: On chatgpt, cybersecurity, and secure coding. *Information* 15, 9 (2024).
- [68] FARRELL, W. C., BOGODISTOV, Y., AND MÖSSENLECHNER, C. Is Academic Integrity at Risk? Perceived Ethics and Technology Acceptance of ChatGPT.
- [69] FINNIE-ANSLEY, J., DENNY, P., BECKER, B. A., LUXTON-REILLY, A., AND PRATHER, J. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proceedings of the 24th Australasian Computing Education Conference* (2022), pp. 10–19.
- [70] FINNIE-ANSLEY, J., DENNY, P., LUXTON-REILLY, A., SANTOS, E. A., PRATHER, J., AND BECKER, B. A. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI’s Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference* (2023), pp. 97–104.
- [71] FOWLER, G. A. Detecting AI may be impossible. That’s a big problem for teachers. <https://www.washingtonpost.com/technology/2023/06/02/turnitin-ai-cheating-detector-accuracy/>, June 2023. [Online; last accessed 07-February-2024].
- [72] GAO, L., BIDERMAN, S., BLACK, S., GOLDING, L., HOPPE, T., FOSTER, C., PHANG, J., HE, H., THITE, A., NABESHIMA, N., PRESSER, S., AND LEAHY, C. The Pile - An 800GB Dataset of Diverse Text for Language Modelling. <https://pile.eleuther.ai/>. [Online; last accessed 29-March-2024].
- [73] GAO, L., BIDERMAN, S., BLACK, S., GOLDING, L., HOPPE, T., FOSTER, C., PHANG, J., HE, H., THITE, A., NABESHIMA, N., PRESSER, S., AND LEAHY, C. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027* (2020).
- [74] GARCIA, B. *Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. Packt Publishing Ltd, 2017.

- [75] GRANDEL, S., SCHMIDT, D. C., AND LEACH, K. Applying large language models to enhance the assessment of parallel functional programming assignments. In *Proceedings of the 2024 International Workshop on Large Language Models for Code* (2024), pp. 1–9.
- [76] GROENEVELD, D., BELTAGY, I., WALSH, P., BHAGIA, A., KINNEY, R., TAFJORD, O., JHA, A. H., IVISON, H., MAGNUSSON, I., WANG, Y., ET AL. Olmo: Accelerating the Science of Language Models. *arXiv preprint arXiv:2402.00838* (2024).
- [77] GROTTA, A., AND PRADO, E. P. V. Benefits of the project-based learning to cope with computer programming education: A systematic literature review. In *Conference Proceedings of Pbl2019 Immersive Virtual International Conference Hello Future* (2019), pp. 12–13.
- [78] GUERREIRO, P., AND GEORGOULI, K. Combating Anonymousness in Populous CS1 and CS2 Courses. *ACM SIGCSE Bulletin* 38, 3 (2006), 5.
- [79] HECKMAN, S., AND KING, J. Developing Software Engineering Skills using Real Tools for Automated Grading. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE '18* (Baltimore, Maryland, USA, 2018), ACM Press, pp. 794–799.
- [80] HENDRYCKS, D., BASART, S., KADAVATH, S., MAZEIKA, M., ARORA, A., GUO, E., BURNS, C., PURANIK, S., HE, H., SONG, D., ET AL. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).
- [81] HOODA, A., CHRISTODORESCU, M., ALLAMANIS, M., WILSON, A., FAWAZ, K., AND JHA, S. Do Large Code Models Understand Programming Concepts? A Black-box Approach. *arXiv preprint arXiv:2402.05980* (2024).
- [82] HOQ, M., SHI, Y., LEINONEN, J., BABALOLA, D., LYNCH, C., PRICE, T., AND AKRAM, B. Detecting chatgpt-generated code submissions in a cs1 course using machine learning models.
- [83] HORSTMANN, C. Codecheck. <https://horstmann.com/codecheck/index.html>. [Online; last accessed 15-July-2024].
- [84] HOU, I., MAN, O., METTILLE, S., GUTIERREZ, S., ANGELIKAS, K., AND MACNEIL, S. More Robots are Coming: Large Multimodal Models (ChatGPT) can Solve Visually Diverse Images of Parsons Problems.
- [85] HSIAO, SISSIE. Bard becomes Gemini: Try Ultra 1.0 and a new mobile app today. <https://blog.google/products/gemini/bard-gemini-advanced-app/>, February 2024. [Online; last accessed 08-February-2024].
- [86] IHANTOLA, P., AHONIEMI, T., KARAVIRTA, V., AND SEPPÄLÄ, O. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research* (2010), pp. 86–93.
- [87] INSA, D., AND SILVA, J. Semi-automatic assessment of unrestrained Java code: a library, a DSL, and a workbench to assess exams and exercises. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (2015), pp. 39–44.

- [88] IVANOV, R., ET AL. Checkstyle. <https://checkstyle.org/>, 2021. [Online; last accessed 17-February-2021].
- [89] IVANOV, R., ET AL. Checkstyle. <https://checkstyle.org/>, 2023. [Online; last accessed 20-January-2023].
- [90] JAN HENDRIK KIRCHNER, LAMA AHMAD, S. A. J. L. New ai classifier for indicating ai-written text. <https://openai.com/blog/new-ai-classifier-for-indicating-ai-written-text>, July 2023. [Online; last accessed 07-February-2024].
- [91] JELINEK, F., MERCER, R. L., BAHL, L. R., AND BAKER, J. K. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America* 62, S1 (1977), S63–S63.
- [92] JIANG, A. Q., SABLAYROLLES, A., MENSCH, A., BAMFORD, C., CHAPLOT, D. S., DE LAS CASAS, D., BRESSAND, F., LENGUEL, G., LAMPLE, G., SAULNIER, L., LAVAUD, L. R., LACHAUX, M.-A., STOCK, P., SCAO, T. L., LAVRIL, T., WANG, T., LACROIX, T., AND SAYED, W. E. Mistral 7b, 2023.
- [93] JURAFSKY, D., AND MARTIN, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, 2024.
- [94] KARAMOLEGKOU, A., LI, J., ZHOU, L., AND SØGAARD, A. Copyright violations and large language models. *arXiv preprint arXiv:2310.13771* (2023).
- [95] KAZEMITABAAR, M., CHOW, J., MA, C. K. T., ERICSON, B. J., WEINTROP, D., AND GROSSMAN, T. Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conf. on Human Factors in Computing Systems* (New York, NY, USA, 2023), CHI ’23, ACM.
- [96] KAZEMITABAAR, M., HOU, X., HENLEY, A., ERICSON, B. J., WEINTROP, D., AND GROSSMAN, T. How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment, 2023.
- [97] KEUNING, H., JEURING, J., AND HEEREN, B. A Systematic Mapping Study of Code Quality in Education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (2023), pp. 5–11.
- [98] KLEE, M. She was falsely accused of cheating with ai — and she won’t be the last. <https://www.rollingstone.com/culture/culture-features/student-accused-ai-cheating-turnitin-1234747351>, June 2023. [Online; last accessed 07-February-2024].
- [99] KOCETKOV, D., LI, R., ALLAL, L. B., LI, J., MOU, C., FERRANDIS, C. M., JERNITE, Y., MITCHELL, M., HUGHES, S., WOLF, T., BAHDANAU, D., VON WERRA, L., AND DE VRIES, H. The stack: 3 tb of permissively licensed source code, 2022.
- [100] KONG, A., ZHAO, S., CHEN, H., LI, Q., QIN, Y., SUN, R., AND ZHOU, X. Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702* (2023).

- [101] KRATZKE, N. Smart Like a Fox: How Clever Students Trick Dumb Automated Programming Assignment Assessment Systems. In *Proceedings of the 11th International Conference on Computer Supported Education, CSEDU 2019, Heraklion, Crete, Greece, May 2-4, 2019, Volume 2* (2019), H. C. Lane, S. Zvacek, and J. Uhoimoibhi, Eds., SciTePress, pp. 15–26.
- [102] KRAWCZYK, JACK. Bard’s latest update: more features, languages and countries. <https://blog.google/products/bard/google-bard-new-features-update-july-2023/>, 2023. [Online; last accessed 16-December-2023].
- [103] LAFFERTY, J., MCCALLUM, A., PEREIRA, F., ET AL. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Icml* (2001), vol. 1, Williamstown, MA, p. 3.
- [104] LAU, S., AND GUO, P. From “Ban it till we understand it” to “Resistance is futile”: How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools Such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1* (2023), pp. 106–121.
- [105] LE SCAO, T., FAN, A., AKIKI, C., PAVLICK, E., ILIĆ, S., HESSLOW, D., CASTAGNÉ, R., LUCCIONI, A. S., YVON, F., GALLÉ, M., ET AL. Bloom: A 176b-parameter open-access multilingual language model.
- [106] LEAL, J. P., AND SILVA, F. Mooshak: A Web-based Multi-site Programming Contest System. *Software: Practice and Experience* 33, 6 (May 2003), 567–581.
- [107] LEHTONEN, T. Javala-addictive e-learning of the Java programming language. In *Proceedings of Kolin Kolistelut/Koli Calling-Fifth Annual Baltic Conference on Computer Science Education*. (Finland, 2005), Koli, pp. 41–48.
- [108] LEINONEN, J., DENNY, P., MACNEIL, S., SARSA, S., BERNSTEIN, S., KIM, J., TRAN, A., AND HELLAS, A. Comparing Code Explanations Created by Students and Large Language Models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2023, Turku, Finland, July 7-12, 2023* (2023), M. Laakso, M. Monga, Simon, and J. Sheard, Eds., ACM, pp. 124–130.
- [109] LENFANT, R., WANNER, A., HOTT, J. R., AND PETTIT, R. Project-Based and Assignment-Based Courses: A Study of Piazza Engagement and Gender in Online Courses. In *Proceedings of the 2023 Conf. on Innovation and Technology in Computer Science Education V. 1* (2023), pp. 138–144.
- [110] LI, C., WANG, J., ZHANG, Y., ZHU, K., HOU, W., LIAN, J., LUO, F., YANG, Q., AND XIE, X. Large language models understand and can be enhanced by emotional stimuli. *arXiv preprint arXiv:2307.11760* (2023).
- [111] LI, R., ALLAL, L. B., ZI, Y., MUENNIGHOFF, N., KOCETKOV, D., MOU, C., MARONE, M., AKIKI, C., LI, J., CHIM, J., ET AL. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

- [112] LI, Y., TAN, T., AND XUE, J. Understanding and Analyzing Java Reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 1–50.
- [113] LI, Z. The dark side of chatgpt: Legal and ethical challenges from stochastic parrots and hallucination. *arXiv preprint arXiv:2304.14347* (2023).
- [114] LIFFITON, M., SHEESE, B., SAVELKA, J., AND DENNY, P. Codehelp: Using large language models with guardrails for scalable support in programming classes. *arXiv preprint arXiv:2308.06921* (2023).
- [115] LJUBOVIC, V., AND PAJIC, E. Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories. *IEEE Access* 8 (2020), 96505–96514.
- [116] LOBB, R., AND HARLOW, J. Coderunner: A tool for assessing computer programming skills. *ACM Inroads* 7, 1 (2016), 47–51.
- [117] LOZHKOVA, A., LI, R., ALLAL, L. B., CASSANO, F., LAMY-POIRIER, J., TAZI, N., TANG, A., PYKHOTAR, D., LIU, J., WEI, Y., ET AL. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).
- [118] LUKEFAHR, H., WATTERSON, C., ROBERTS, A., CARNEGIE, D., AND ATKINS, M. Project-based learning to improve core first-year engineering courses. In *2019 IEEE Frontiers in Education Conference (FIE)* (2019), IEEE, pp. 1–7.
- [119] LUXTON-REILLY, A., TEMPERO, E., ARACHCHILAGE, N., CHANG, A., DENNY, P., FOWLER, A., GIACAMAN, N., KONTOROVICH, I., LOTTRIDGE, D., MANOHARAN, S., ET AL. Automated assessment: Experiences from the trenches. In *Proceedings of the 25th Australasian Computing Education Conference* (2023), pp. 1–10.
- [120] MAGUIRE, P., MAGUIRE, R., AND KELLY, R. Using automatic machine assessment to teach computer programming. *Computer Science Education* 27, 3-4 (2017), 197–214.
- [121] MAGUIRE, P., MAGUIRE, R., AND KELLY, R. Using automatic machine assessment to teach computer programming. *Computer Science Education* 27, 3-4 (2017), 197–214.
- [122] MALAN, D. J., SHARP, C., VAN ASSEMA, J., YU, B., AND ZIDANE, K. Cs50’s github-based tools for teaching and learning. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2021), SIGCSE ’21, Association for Computing Machinery, p. 1354.
- [123] MALINKA, K., PERESÍNI, M., FIRC, A., HUJNÁK, O., AND JANUS, F. On the educational impact of chatgpt: Is artificial intelligence ready to obtain a university degree? In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (2023), pp. 47–53.
- [124] MANSUR, R. S., KAZEROUNI, A. M., EDWARDS, S. H., AND SHAFFER, C. A. Exploring the bug investigation techniques of intermediate student programmers. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (2020), pp. 1–10.

- [125] MARCUS, M., SANTORINI, B., AND MARCINKIEWICZ, M. A. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational linguistics* 19, 2 (1993), 313–330.
- [126] MARTÍN-BLAS, T., AND SERRANO-FERNÁNDEZ, A. The role of new technologies in the learning process: Moodle as a teaching tool in physics. *Computers & Education* 52, 1 (2009), 35–44.
- [127] MCCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* 3, 4 (1960), 184–195.
- [128] MCCARTHY, J., MINSKY, M. L., ROCHESTER, N., AND SHANNON, C. E. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence.
- [129] METZ, S. *Practical Object-Oriented Design in Ruby: An Agile Primer*. Addison-Wesley, 2012.
- [130] MEYER, B. *Object-Oriented Software Construction*, vol. 2. Prentice hall Englewood Cliffs, 1997.
- [131] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).
- [132] MILLS, J. E., TREAGUST, D. F., ET AL. Engineering education—is problem-based or project-based learning the answer. *Australasian journal of engineering education* 3, 2 (2003), 2–16.
- [133] NAUMOVA, E. N. A mistake-find exercise: a teacher’s tool to engage with information innovations, chatgpt, and their analogs. *Journal of Public Health Policy* 44, 2 (2023), 173–178.
- [134] OFFUTT, J., AMMANN, P., DOBOLYI, K., KAUFFMANN, C., LESTER, J., PRAPHAMONTRIPONG, U., RANGWALA, H., SETIA, S., WANG, P., AND WHITE, L. A novel self-paced model for teaching programming. In *Fourth (2017) ACM Conference on Learning@Scale* (2017), pp. 177–180.
- [135] OPENAI. OpenAI API Reference - Introduction. <https://platform.openai.com/docs/api-reference/introduction/>. [Online; last accessed 17-July-2024].
- [136] OPENAI. Chatgpt can now see, hear, and speak. <https://openai.com/blog/chatgpt-can-now-see-hear-and-speak>, 2023. [Online; last accessed 12-December-2023].
- [137] OPENAI. GPT-4 Technical Report, Mar. 2023. arXiv:2303.08774 [cs].
- [138] OUH, E. L., GAN, B. K. S., JIN SHIM, K., AND WLODKOWSKI, S. ChatGPT, Can You Generate Solutions for my Coding Exercises? An Evaluation on its Effectiveness in an undergraduate Java Programming Course. 54–60.
- [139] PAIVA, J. C., LEAL, J. P., AND FIGUEIRA, Á. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education* 22, 3 (Sept. 2022), 1–40.

- [140] PAIVA, J. C., LEAL, J. P., AND FIGUEIRA, Á. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–40.
- [141] PANKIEWICZ, M., AND BAKER, R. S. Large Language Models (GPT) for automating feedback on programming assignments. *arXiv preprint arXiv:2307.00150* (2023).
- [142] PAPINENI, K., ROUKOS, S., WARD, T., AND ZHU, W.-J. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (2002), pp. 311–318.
- [143] PARSONS, D., AND HADEN, P. Parson’s programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52* (2006), pp. 157–163.
- [144] PEARCE, H., AHMAD, B., TAN, B., DOLAN-GAVITT, B., AND KARRI, R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 754–768.
- [145] PEIXOTO, A. R., GLÓRIA, A., SILVA, J. L., PINTO-ALBUQUERQUE, M., BRANDÃO, T., AND NUNES, L. Use of programming aids in undergraduate courses. In *5th International Computer Programming Education Conference (ICPEC 2024)* (2024), Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- [146] PENG, Q., CHAI, Y., AND LI, X. HumanEval-XL: A Multilingual Code Generation Benchmark for Cross-lingual Natural Language Generalization. *arXiv preprint arXiv:2402.16694* (2024).
- [147] PEREIRA CIPRIANO, B., SILVA, M., CORREIA, R., AND ALVES, P. Towards the Integration of Large Language Models and Automatic Assessment Tools: Enhancing Student Support in Programming Assignments. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2024), Koli Calling ’24, Association for Computing Machinery.
- [148] PICCOLO, S. R., DENNY, P., LUXTON-REILLY, A., PAYNE, S., AND RIDGE, P. G. Many bioinformatics programming tasks can be automated with chatgpt. *arXiv preprint arXiv:2303.13528* (2023).
- [149] POPOVICI, M.-D. ChatGPT in the Classroom. Exploring Its Potential and Limitations in a Functional Programming Course. *International Journal of Human–Computer Interaction* (2023), 1–12.
- [150] PORTER, L., AND ZINGARO, D. *Learn AI-Assisted Python Programming With GitHub Copilot and ChatGPT*. Manning, Shelter Island, NY, USA, 2023.
- [151] PRASAD, S., GREENMAN, B., NELSON, T., AND KRISHNAMURTHI, S. Generating Programs Trivially: Student Use of Large Language Models. In *Proceedings of the ACM Conference on Global Computing Education Vol 1* (Hyderabad India, Dec. 2023), ACM, pp. 126–132.
- [152] PRATHER, J., DENNY, P., LEINONEN, J., BECKER, B. A., ALBLUWI, I., CRAIG, M., KEUNING, H., KIESLER, N., KOHN, T., LUXTON-REILLY, A., MACNEIL, S., PETERSON, A., PETTIT, R., REEVES, B. N., AND SAVELKA, J. The Robots are Here: Navigating the Generative AI Revolution in Computing Education, 2023.

- [153] PRATHER, J., REEVES, B. N., DENNY, P., BECKER, B. A., LEINONEN, J., LUXTON-REILLY, A., POWELL, G., FINNIE-ANSLEY, J., AND SANTOS, E. A. "it's weird that it knows what i want": Usability and interactions with copilot for novice programmers. *arXiv preprint arXiv:2304.02491* (2023).
- [154] RADFORD, A., KIM, J. W., HALLACY, C., RAMESH, A., GOH, G., AGARWAL, S., SASTRY, G., ASKELL, A., MISHKIN, P., CLARK, J., ET AL. Learning transferable visual models from natural language supervision. In *International conference on machine learning* (2021), PMLR, pp. 8748–8763.
- [155] RADFORD, A., NARASIMHAN, K., SALIMANS, T., SUTSKEVER, I., ET AL. Improving language understanding by generative pre-training.
- [156] RAHMAN, M. M., AND WATANOBE, Y. ChatGPT for Education and Research: Opportunities, Threats, and Strategies. *Applied Sciences* 13, 9 (2023), 5783.
- [157] RAMESH, A., PAVLOV, M., GOH, G., GRAY, S., VOSS, C., RADFORD, A., CHEN, M., AND SUTSKEVER, I. Zero-shot text-to-image generation. In *International Conference on Machine Learning* (2021), PMLR, pp. 8821–8831.
- [158] REEVES, B., SARSA, S., PRATHER, J., DENNY, P., BECKER, B. A., HELLAS, A., KIMMEL, B., POWELL, G., AND LEINONEN, J. Evaluating the performance of code generation models for solving parsons problems with small prompt variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (2023), pp. 299–305.
- [159] ROZIERE, B., GEHRING, J., GLOECKLE, F., SOOTLA, S., GAT, I., TAN, X. E., ADI, Y., LIU, J., REMEZ, T., RAPIN, J., ET AL. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [160] SALLOU, J., DURIEUX, T., AND PANICHELLA, A. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (New York, NY, USA, 2024), ICSE-NIER'24, Association for Computing Machinery, p. 102–106.
- [161] SANTOS, A. L., PRENDI, G., SOUSA, H., AND RIBEIRO, R. Stepwise API usage assistance using n-gram language models. *Journal of Systems and Software* 131 (2017), 461–474.
- [162] SAVELKA, J., AGARWAL, A., AN, M., BOGART, C., AND SAKR, M. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. In *Proceedings of the 2023 ACM Conf. on International Computing Education Research V.1* (August 2023), ICER 2023, ACM.
- [163] SAVELKA, J., AGARWAL, A., BOGART, C., AND SAKR, M. Large Language Models (GPT) Struggle to Answer Multiple-Choice Questions about Code. *arXiv preprint arXiv:2303.08033* (2023).
- [164] SAVELKA, J., AGARWAL, A., BOGART, C., SONG, Y., AND SAKR, M. Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses? In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (jun 2023), ACM.

- [165] SCHNEIDER, G. M. The introductory programming course in computer science: ten principles. In *Papers of the SIGCSE/CSA technical symposium on Computer science education* (1978), pp. 107–114.
- [166] SCHOLTE, R., ET AL. Apache Maven Invoker. <https://maven.apache.org/shared/maven-invoker/>, 2022. [Online; last accessed 09-March-2022].
- [167] SCHWALM, J., AND TYLEK, K. S. Systemwide Implementation of Project-Based Learning: The Philadelphia Approach. *Afterschool Matters* 15 (2012), 1–8.
- [168] SHARP, C., VAN ASSEMA, J., YU, B., ZIDANE, K., AND MALAN, D. J. An Open-Source, API-Based Framework for Assessing the Correctness of Code in CS50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2020), ITiCSE '20, Association for Computing Machinery, p. 487–492.
- [169] SHEARD, J., DENNY, P., HELLAS, A., LEINONEN, J., MALMI, L., AND SIMON. Instructor Perceptions of AI Code Generation Tools - A Multi-Institutional Interview Study. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2024), SIGCSE 2024, ACM, p. 1223–1229.
- [170] SHIN, M.-H. Effects of project-based learning on students' motivation and self-efficacy. *English Teaching* 73, 1 (2018), 95–114.
- [171] SHOKRI, R., STRONATI, M., SONG, C., AND SHMATIKOV, V. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE symposium on security and privacy (SP)* (2017), IEEE, pp. 3–18.
- [172] SHUTE, V. J. Focus on Formative Feedback. *Review of educational research* 78, 1 (2008), 153–189.
- [173] SILVA, A., LEAL, J. P., AND PAIVA, J. C. Raccocode: An eclipse plugin for assessment of programming exercises (short paper). In *7th Symposium on Languages, Applications and Technologies (SLATE 2018)* (2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [174] SIMON, MASON, R., CRICK, T., DAVENPORT, J. H., AND MURPHY, E. Language Choice in Introductory Programming Courses at Australasian and UK universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (2018), pp. 852–857.
- [175] SINGH, H., TAYARANI-NAJARAN, M.-H., AND YAQOOB, M. Exploring Computer Science students' Perception of ChatGPT in Higher Education: A Descriptive and Correlation Study. *Education Sciences* 13, 9 (2023), 924.
- [176] ŠKORIĆ, I., OREHOVAČKI, T., AND IVAŠIĆ-KOS, M. Exploring the acceptance of the web-based coding tool in an introductory programming course: A pilot study. In *International Conference on Human Interaction and Emerging Technologies* (2020), Springer, pp. 42–48.
- [177] SPACCO, J., DENNY, P., RICHARDS, B., BABCOCK, D., HOVEMEYER, D., MOSCOLA, J., AND DUVAL, R. Analyzing Student Work Patterns Using Programming Exercise Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2015), SIGCSE '15, Association for Computing Machinery, p. 18–23.

- [178] STRZELECKI, A. To use or not to use ChatGPT in higher education? a study of students' acceptance and use of technology. *Interactive Learning Environments* (2023), 1–14.
- [179] SUNDAR, P. An important next step on our AI journey. <https://blog.google/technology/ai/bard-google-ai-search-updates/>, Feb 2023. [Online; last accessed 10-August-2023].
- [180] TEAM, G., ANIL, R., BORGEAUD, S., WU, Y., ALAYRAC, J.-B., YU, J., SORICUT, R., SCHALKWYK, J., DAI, A. M., HAUTH, A., ET AL. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).
- [181] THAKUR, V. Unveiling Gender Bias in Terms of Profession Across LLMs: Analyzing and Addressing Sociological Implications. *arXiv preprint arXiv:2307.09162* (2023).
- [182] THANGARAJ, J., WARD, M., AND O'RIORDAN, F. A Systematic Review of Formative Assessment to Support Students Learning Computer Programming. In *4th International Computer Programming Education Conference (ICPEC 2023)* (2023), Schloss-Dagstuhl-Leibniz Zentrum für Informatik.
- [183] THE OPEN SOURCE INITIATIVE. The Open Source AI Definition – draft v. 0.0.9. <https://opensource.org/ai/drafts/open-source-ai-definition-draft-v-0-0-9>. [Online; last accessed 09-September-2024].
- [184] THE SPRING TEAM. Sprint Boot. <https://spring.io/projects/spring-boot>, 2021. [Online; last accessed 17-February-2021].
- [185] THIÉBAUT, D. Automatic evaluation of computer programs using Moodle's virtual programming lab (VPL) plug-in. *Journal of Computing Sciences in Colleges* 30, 6 (2015), 145–151.
- [186] THOPPILAN, R., DE FREITAS, D., HALL, J., SHAZEER, N., KULSHRESHTHA, A., CHENG, H.-T., JIN, A., BOS, T., BAKER, L., DU, Y., ET AL. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* (2022).
- [187] TOUVRON, H., LAVRIL, T., IZACARD, G., MARTINET, X., LACHAUX, M.-A., LACROIX, T., ROZIÈRE, B., GOYAL, N., HAMBRO, E., AZHAR, F., ET AL. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [188] TUNSTALL, L., VON WERRA, L., AND WOLF, T. *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media, 2022.
- [189] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [190] VENKATESH, V., MORRIS, M. G., DAVIS, G. B., AND DAVIS, F. D. User acceptance of information technology: Toward a unified view. *MIS quarterly* (2003), 425–478.

- [191] VENKATESH, V., VENKATESH, V., AND KUMAR, V. Evaluating Copilot on CS1 Code Writing Problems with Suppressed Specifications. In *Proceedings of the 16th Annual ACM India Compute Conference* (2023), pp. 104–107.
- [192] VENKATESH, V., AND ZHANG, X. Unified theory of acceptance and use of technology: Us vs. china. *Journal of global information technology management* 13, 1 (2010), 5–27.
- [193] WANG, B. Mesh-Transformer-JAX: Model-Parallel Implementation of Transformer Language Model with JAX. <https://github.com/kingoflolz/mesh-transformer-jax>, May 2021.
- [194] WANG, B., AND KOMATSUZAKI, A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. <https://github.com/kingoflolz/mesh-transformer-jax>, May 2021.
- [195] WANG, X., WEI, J., SCHUURMANS, D., LE, Q., CHI, E., NARANG, S., CHOWDHURY, A., AND ZHOU, D. Self-Consistency Improves Chain of Thought Reasoning in Language Models, 2023.
- [196] WEGNER, P. Concepts and paradigms of object-oriented programming. *ACM Sigplan OOps Messenger* 1, 1 (1990), 7–87.
- [197] WEI, J., WANG, X., SCHUURMANS, D., BOSMA, M., XIA, F., CHI, E., LE, Q. V., ZHOU, D., ET AL. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
- [198] WEIZENBAUM, J. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM* 9, 1 (1966), 36–45.
- [199] WERMELINGER, M. Using GitHub Copilot to Solve Simple Programming Problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (2023), pp. 172–178.
- [200] WINOGRAD, T. Procedures as a representation for data in a computer program for understanding natural language.
- [201] WÜNSCHE, B. C., CHEN, Z., SHAW, L., SUSELO, T., LEUNG, K.-C., DIMALEN, D., MARK, W. V. D., LUXTON-REILLY, A., AND LOBB, R. Automatic assessment of opengl computer graphics assignments. In *Proceedings of the 23rd annual ACM conference on innovation and technology in computer science education* (2018), pp. 81–86.
- [202] XU, F. F., ALON, U., NEUBIG, G., AND HELLENDORF, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (2022), pp. 1–10.
- [203] XUE, Y., CHEN, H., BAI, G. R., TAIRAS, R., AND HUANG, Y. Does ChatGPT Help With Introductory Programming? An Experiment of Students Using ChatGPT in CS1. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training* (2024), pp. 331–341.

- [204] YETİŞTİREN, B., ÖZSOY, I., AYERDEM, M., AND TÜZÜN, E. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT, 2023.
- [205] YILMAZ, H., MAXUTOV, S., BAITEKOV, A., AND BALTA, N. Student Attitudes towards Chat GPT: A Technology Acceptance Model Survey. *International Educational Review* 1, 1 (2023), 57–83.
- [206] YILMAZ, R., AND YILMAZ, F. G. K. Augmented intelligence in programming learning: Examining student views on the use of ChatGPT for programming learning. *Computers in Human Behavior: Artificial Humans* 1, 2 (2023), 100005.
- [207] ZANDER, C., THOMAS, L., MOSTRÖM, J. E., MCCARTNEY, R., AND ECKERDAL, A. Pseudocode: scaffolding student object-oriented software design. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (2023), pp. 1–2.
- [208] ZASTUDIL, C., ROGALSKA, M., KAPP, C., VAUGHN, J., AND MACNEIL, S. Generative AI in Computing Education: Perspectives of Students and Instructors. In *2023 IEEE Frontiers in Education Conference (FIE)* (2023), IEEE, pp. 1–9.