Building Better Programmers: An AI System for Guided Program Decomposition

Author: Arnav Chopra
Date: 17 June, 2025

TU Delft

Contents

1

Introduction

One of the longstanding goals in the evolution of programming languages has been to raise the level of abstraction, from machine code to assembly, to low-level and then high-level languages. Generative AI now introduces a new abstraction layer: natural language (Reeves et al., 2024; Halpern, 1966). This shift means students can approach programming problems at a more conceptual level, focusing on problem-solving rather than on language-specific syntax (Prather et al., 2023). As a result, they become more adaptable across languages and better equipped to translate ideas into working code.

The rise of generative AI is reshaping how computer science is taught, offering students more interactive, intuitive, and personalised ways to learn programming. AI-driven tools in education can provide contextual hints, instant feedback, offer corrections, and support learners in developing foundational software engineering skills (Woodrow et al., 2024). Among these, one particularly important skill is decomposition: the ability to break down a complex problem into simpler, independent parts. Decomposition supports modularity, where each unit of code has a well-defined role. It improves code clarity, encourages reuse, and simplifies both testing and debugging (Keen et al., 2015). For beginners, learning to decompose problems effectively is often the first step toward writing clean, maintainable software.

Planning decomposition before implementation, rather than doing it on the fly or even after, offers clear advantages for novice programmers. It helps learners define dependencies early, gain a deeper understanding of the problem, and allocate resources more effectively. This can reduce complexity, streamline development, and lower the

likelihood of major refactoring later on (Charitsis et al., 2023).

To support the development of decomposition skills in novice programmers, we propose a new method to teach decomposition, using an automatic decomposition feedback tool integrated into educational programming systems. Our aim is to investigate how such a tool might affect learners' cognitive processes and confidence in structuring code effectively. Specifically, we ask:

1. How does using a guided program decomposition system affect a learner's cognitive processes, as observed in a concurrent think-aloud study?
2. How does using a guided program decomposition system affect a learner's confidence in their own decomposition skills?

This system provides an extension to JetBrains' educational system, Cognifire (Potriasaeva et al., 2024), which supports structured code generation through natural language prompts. Cognifire is currently deployed within the in-IDE learning environment (Birillo et al., 2024) of the JetBrains Academy plugin (*JetBrains Academy Plugin* n.d.), allowing learners to engage with real-world tools while developing their skills. The system currently supports the Kotlin programming language.

As part of this work, a part of the implementation of the proposed method was contributed. In particular, this research contributes the following:

- A novel program decomposition approach in educational systems.
- Prototypes for the user interface design in the JetBrains Academy plugin.
- The artificial intelligence functionality of the system: crafting prompts and handling all interactions with the AI services.

The implementation of the user interface design prototypes into the plugin was done by JetBrains.

By embedding decomposition guidance directly into the programming process, we aim to help learners internalise this essential skill, improving both their technical proficiency and their confidence as they progress toward becoming capable software developers.

This thesis is structured in the following manner: Section 2 presents some introductory materials, which explain the concepts that are important to this research. Next,

Section 3 presents the core of this research in the form of a scientific article. Thus, the introductory materials aim to provide readers with some background knowledge of concepts used in the scientific article. Finally, Section 4 contains supplementary material corresponding to this research.

2

Introductory Materials

2.1 Artificial Intelligence

Artificial intelligence, or AI, is a greatly transformative area of computer science, touching everything from healthcare and finance to education, logistics, and creative work. At its core, AI is about building systems that can perform tasks we typically associate with human intelligence, such as recognising images, understanding language, making decisions, and learning from data.

The field itself is broad and interdisciplinary, drawing on ideas from computer science, statistics, cognitive science, and even philosophy. AI includes many subfields, such as machine learning (where systems improve through learning), computer vision (understanding images and video), natural language processing (interpreting and generating human language), and robotics (interacting with the physical world). While each of these areas has its own methods and goals, they are all connected by the central aim of creating systems that can adapt to uncertainty, respond intelligently to complex environments, and learn from new information.

AI offers enormous potential. In science and engineering, it accelerates research by automating pattern discovery and simulation. In medicine, it supports diagnosis, drug discovery, and personalised treatment. Most importantly for this research, AI in education can provide tailored learning experiences, identify gaps in understanding, and offer timely feedback (MacNeil et al., 2023; Balse et al., 2023); something that is hard to scale through traditional means.

2.2 Natural Language Processing

Natural language processing (NLP) is a field of artificial intelligence, concerned with the ability of a computer to process data presented in the form of natural language rather than traditional machine-readable formats. NLP is a critical intersection between computer science and linguistics, as it helps us to interact meaningfully with machines using human language. Unlike structured formats, such as code, natural language is extremely context-dependent and full of irregularities. These stark differences make NLP a rewarding but challenging field of development, with several everyday use cases, such as search engines and voice assistants.

An especially transformative area within NLP is text generation. The ability for machines not only to interpret language but also to produce coherent, contextually appropriate text opens the door to a range of applications, from creative content generation to conversational agents and educational tools. Generated text can be crafted to summarise information, translate between languages, or answer queries.

2.3 Large Language Models

Large language models, or LLMs, represent a significant leap in the capabilities of artificial intelligence, particularly in the realm of language understanding and generation. These models are designed to process and produce human language at scale, enabling a range of applications that were previously out of reach, from coherent essay writing and real-time dialogue to code generation and advanced question answering. Built on modern deep learning architectures and trained on massive corpora of text, LLMs are rapidly becoming the foundational engines behind a new wave of AI systems.

At their core, LLMs are statistical models that learn to predict the next word in a sequence, given the words that came before it. While this may seem like a simple task, scaling it up to billions of parameters and training on trillions of words allows the model to internalise a rich representation of grammar, semantics, knowledge, and even elements of reasoning. The result is a model that can produce fluent, contextually appropriate language and handle a wide variety of linguistic tasks without needing any task-specific programming or training.

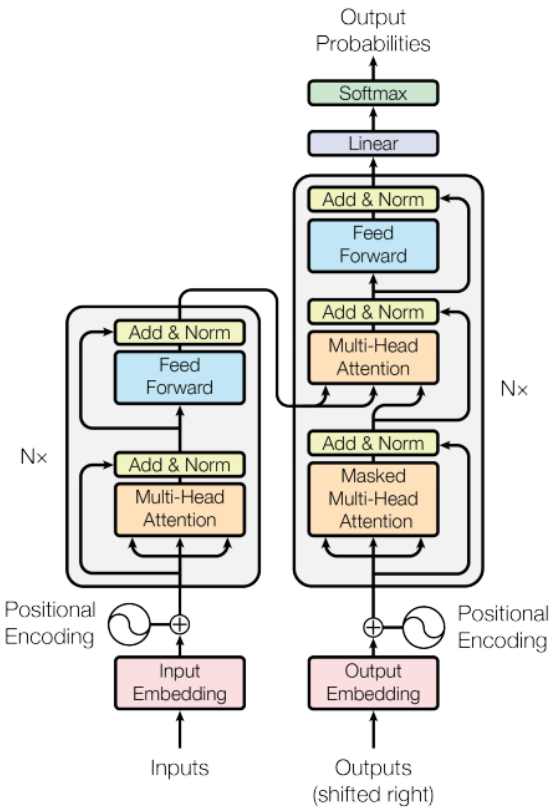

Figure 2.1: *Transformer architecture* (*Vaswani et al., 2023*).

Modern LLMs are based on the transformer architecture (Vaswani et al., 2023) (displayed in Figure 2.1), which was introduced in 2017 and has since become the standard for building scalable and efficient language models. Unlike earlier models, transformers rely on a mechanism called self-attention to process input data in parallel, making them well-suited for training on large datasets and for learning long-range dependencies in text. Transformers also make it possible to represent each word or token not in isolation, but in context, thus capturing the nuances of meaning that shift depending on surrounding words.

LLMs have led to a paradigm shift in how NLP systems are designed and deployed: instead of building specific models for each task, developers can use a single, general-purpose model that adapts to a wide range of use cases based on how it is prompted.

LLMs can explain complex concepts in simple terms, answer questions in real-time, and adapt responses to suit individual learning styles and levels. For students, LLMs serve as accessible study companions; helping with writing, problem-solving, and critical thinking (Pankiewicz et al., 2023). By combining language understanding with

subject knowledge, LLMs help make education more interactive, inclusive, and scalable.

2.4 Prompt Engineering

As large language models have grown in sophistication and versatility, a new practice has emerged at the centre of using them effectively: prompt engineering. While traditional computer interaction involves writing code to instruct a machine, prompt engineering involves crafting natural language inputs to produce desired outputs from a model.

Prompt engineering is, in essence, the science of communicating with language models. Given that these models are trained to predict and generate text based on patterns in human language, the phrasing, structure, and context of the input can dramatically influence the quality and relevance of the output. A poorly worded prompt can yield vague, off-topic, or misleading responses, while a well-structured prompt can render precise, informative, or creative results (White et al., 2023).

At first glance, prompt engineering might seem simple, just asking questions or giving instructions in plain English. However, when interacting with LLMs, it becomes quickly apparent that small variations in wording, format, and context can produce vastly different outcomes. For instance, framing a request as a question versus a command, providing examples within the prompt, or defining the desired output format can all steer the model in meaningful ways. Prompt design becomes a process of iterative refinement, where one experiments, evaluates, and adjusts to get closer to the intended result.

One common strategy in prompt engineering is the use of few-shot learning, where the prompt includes a few annotated examples of input-output pairs to demonstrate the desired behaviour. This allows the model to infer the pattern and apply it to new inputs. Even in the absence of examples, zero-shot prompting can work well if the task is described clearly and concisely, though results tend to improve with more contextual information. In more complex scenarios, chain-of-thought prompting can be used to encourage the model to reason step-by-step by explicitly instructing it to ”think out loud”, leading to better performance on logic-heavy or multi-step tasks.

With the new natural language abstraction level, learning to interact with LLMs and write prompts has become an essential skill for students to learn (Denny et al., 2023).

2.5 Problem Decomposition

Problem decomposition is a cognitive strategy fundamental to effective problem-solving across several domains. It involves dividing a complex problem into smaller, more manageable sub-problems, each of which can be addressed with greater clarity and precision (Boyd et al., 2008). This process reduces cognitive load and exposes the internal structure of a problem.

From a psychological standpoint, problem decomposition aligns with well-established theories of human cognition. Working memory has a limited capacity, and large, interdependent problems often exceed this capacity, leading to decision paralysis or superficial reasoning. By breaking the problem into discrete components, individuals can focus attention on specific elements, reducing mental complexity and allowing deeper analytical engagement with each part. This structured approach enhances comprehension and improves the accuracy and efficiency of problem-solving behaviour.

A key advantage of decomposition is that it reveals dependencies and constraints that may not be obvious in the initial, undivided formulation of the problem. Some subproblems may be independent and solvable in parallel, while others may be hierarchical or conditional, requiring sequential resolution. This exposure of structure is particularly important in project management, systems thinking, and policy development, where understanding the relationships between subcomponents can guide prioritisation, risk assessment, and resource allocation.

2.6 Program Decomposition in Software Engineering

In software engineering, program decomposition (Hsu et al., 2018) is not just a technique but a foundational design practice that shapes how systems are built, understood, and maintained over time. At its core, program decomposition is the process of taking a software problem and breaking it down into smaller, self-contained units

of logic: functions, modules, classes, components, and services. Each of these serves as a building block that encapsulates a specific aspect of the system's behaviour. When done well, decomposition enables clarity, modularity, and scalability; when done poorly, it results in tangled code, duplication, and fragility (Charitsis et al., 2023).

The need for decomposition arises naturally from the nature of software itself. Even modest programs can quickly become too complex to hold entirely in one's head. Rather than trying to write a large program from top to bottom, engineers rely on decomposition to break the problem into discrete parts, each with a clear purpose and well-defined interface, so they can work incrementally and reason locally.

At the most granular level, decomposition begins with functions or procedures. Each function is ideally responsible for a single task: one transformation, one decision, one calculation (Martin, 2017). This is the principle of single responsibility, which is central to having maintainable code. A well-decomposed function does exactly one thing, has a clear name, and abstracts away the details of how that thing is done. This simplicity enables reuse, testing, and composability; functions become small tools that can be rearranged to form higher-level behaviour. Separation of concerns means that different aspects of the program are handled in separate areas of the codebase, so changes in one area do not unnecessarily affect others.

Perhaps the most valuable aspect of program decomposition is how it facilitates reasoning. When a program is decomposed into well-defined, loosely coupled parts, developers can work on one part without needing to fully understand the whole. It enables testing in isolation, debugging with focus, and documentation that matches the structure.

2.7 In-IDE Learning

An Integrated Development Environment (IDE), is a software application that provides a comprehensive set of tools for writing, testing, and debugging code within a unified interface. It typically includes a code editor, compiler or interpreter, debugger, and build automation tools, all designed to streamline the software development process and improve productivity.

In-IDE learning is the practice of gaining knowledge and developing skills directly

within a programming environment, rather than stepping away to consult external resources (Birillo et al., 2024). This approach leverages the tools built into modern IDEs, such as inline documentation, code completion, usage hints, real-time feedback, and increasingly, AI-powered assistants, to deliver relevant information in context, exactly when and where needed. The key advantage is immediacy: developers can learn while staying focused on the task at hand, avoiding context switching that interrupts flow and slows down comprehension.

As development environments become more sophisticated, in-IDE learning is evolving from simple tooltips and syntax reminders into a powerful, interactive model of education. Developers can explore unfamiliar APIs, understand language idioms, learn best practices, and even receive real-time suggestions for code improvement, all without leaving the code editor. This embedded, task-driven approach to learning supports both novice programmers gaining confidence and experienced developers expanding into new tools or frameworks, making it an increasingly central part of the modern software development experience.

2.8 JetBrains Academy

JetBrains Academy (Birillo et al., 2024) is a platform developed by JetBrains that embeds interactive, project-based learning directly within professional development environments. Unlike traditional approaches that separate theory from practice, JetBrains Academy integrates both elements inside IDEs, allowing students to engage with theoretical content and apply it immediately through coding tasks and real-world projects. This method supports a "learn by doing" model, which cognitive research has long associated with improved retention, deeper conceptual understanding, and stronger problem-solving abilities (Lesgold, 2001).

At the core of JetBrains Academy is its IDE plugin[1], which transforms the development environment into a dynamic learning space. Through structured course materials, learners progress through topics by completing lessons that combine theory, quizzes, and coding exercises. The plugin provides immediate feedback via automated test systems and visual cues, helping students identify and correct errors in real-time.

[1] https://plugins.jetbrains.com/plugin/10081-jetbrains-academy

This continuous loop of instruction, implementation, and correction mirrors key principles of active learning, which are foundational in educational theory.

The platform is designed to bridge the often-cited gap between academic programming instruction and industry practice. By enabling students to learn within a full-featured IDE, JetBrains Academy fosters familiarity with professional tools, workflows, and debugging practices from the outset. This reduces the cognitive and technical friction typically experienced during the transition from education to professional software development. The inclusion of customisable tasks, support for multiple programming languages, and seamless integration of course content within the IDE allows for both flexibility and scalability, accommodating a wide range of learner needs and instructional designs.

2.8.1 Cognifire by JetBrains

Targeting Kotlin and built into the open-source JetBrains Academy plugin, Cognifire (Potriasaeva et al., 2024) is a tool that implements a new approach to teaching students to code in the low-code era of programming. It combines intelligent prompt engineering, code generation, and direct coding to teach algorithmic thinking and problem decomposition.

A dedicated DSL provides support for special descriptions and draft blocks, giving students a space for prompting. Meanwhile, static analysis is used to check whether students are only using defined variables and functions and to improve the quality of the model's output. An overview of the current working on Cognifire is presented in Figure 2.2.

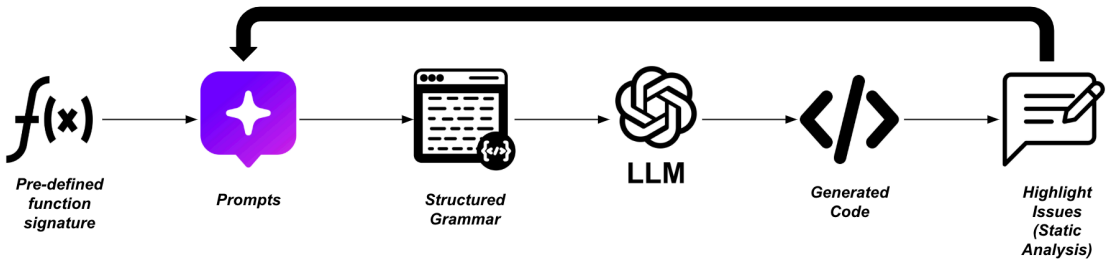

Figure 2.2: *A broad visualisation of the working of the Cognifire (Potriasaeva et al., 2024) system.*

The current version of Cognfire only supports function-level descriptions. The courses using Cognifire have pre-decomposed tasks, where the student simply has to

implement each sub-task. Therefore, while the students learn essential programming concepts, such as variables, loops, and conditional statements, they do not learn to decompose a given problem themselves.

2.9 Think-Aloud Protocol

Think-aloud protocols are a qualitative research method used to study cognitive processes by having individuals verbalise their thoughts in real-time as they perform a task (Ericsson et al., 1998). This technique provides direct insight into how people understand problems, make decisions, and reason through complex activities, making it a valuable tool in fields such as cognitive psychology, human-computer interaction, usability testing, and education.

In a typical think-aloud session, participants are asked to speak continuously about what they are thinking, noticing, or considering while engaging in a task, such as solving a maths problem, navigating a user interface, or writing a piece of code. The researcher observes and records these verbalisations, often alongside screen recordings or behavioural data, to later analyse patterns in reasoning, misconceptions, strategies, or decision-making sequences. The emphasis is not on performance, but on capturing the internal cognitive flow that would otherwise remain undetected.

Think-aloud data is typically analysed qualitatively through coding schemes that identify different types of statements, such as planning, evaluation, inference, confusion, or hypothesis generation. In usability research, for example, the method helps uncover not only where users encounter friction but why, by capturing hesitation, mental models, and expectations in the user's own words.

3

Scientific Article

Building Better Programmers: An AI System for Guided Program Decomposition

Analysing how guided program decomposition affects cognitive processes in computer science students

Arnav Chopra[1,3]
Supervisor(s): Gosia Migut[1]**, Anastasiia Birillo**[2]
[1]EEMCS, Delft University of Technology, The Netherlands
[2]JetBrains, Belgrade, Serbia
[3]JetBrains, Amsterdam, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Master of Computer Science
June 17, 2025

Name of the student: Arnav Chopra
Final project course: IN5000
Thesis committee: Marcus Specht[1], Arie van Deursen[1], Gosia Migut[1]

Abstract

Generative AI has opened up new possibilities in computer science education. Large language models have made it possible for learners to get instantaneous and customised feedback on different programming concepts, as well as the ability to use natural language to implement these concepts. One such concept is program decomposition, an essential skill in software engineering. This work presents a novel method for teaching program decomposition, using a three-stage guided AI decomposition system. We analyse how this method affects a learner's program decomposition cognitive processes via a concurrent think-aloud protocol where a student decomposes three simple programming tasks. Furthermore, we measure how using the system changes a student's confidence in their decomposition skills. We find that participants do not display any significant change in confidence levels after using the system. We observe that the students display a significant improvement in performance during the course of the study. The participants also display a significant decrease in metacognitive confusion and a clear emergence of reflection based on previous errors. We conclude that the proposed method and the implemented system lead to a level of internalisation of decomposition skills in the students. We recommend that a study of change in decomposition skills is conducted over a longer time period to observe the full effects of the method.

1 Introduction

Increasing the level of abstraction has historically been a key objective in the development of programming constructs, from machine code to low-level languages to high-level languages [1], [2]. The rise of Generative AI has now introduced a novel abstraction level, natural language. Learners who can solve programming problems at a high conceptual level are better able to translate their ideas into code in a way that is versatile and less constrained by the syntactic and grammatical differences between programming languages [3], [4].

Artificial intelligence tools in computer science education can provide students with hands-on experience supplemented with context-based tips and real-time feedback [5]. These systems can help novice programmers acquire essential software engineering skills, including testing, debugging, and code quality control. In this work, we focus on one such skill: decomposition. Program decomposition is a fundamental software development skill to learn for any aspiring programmer, as it enables them to break down complex problems into manageable sub-problems. This approach promotes modularity using modules with well-defined and independent purposes. Decomposition also encourages code legibility and minimising duplicate code fragments. Furthermore, by maintaining smaller components, each component becomes easier to test and debug [6].

Decomposing a problem in advance, rather than at the time of implementation, can provide several benefits for novice programmers. Upfront decomposition can help define dependencies, optimise resource allocation, and improve one's understanding of a problem. Identifying these characteristics in advance may lead to reduced complexity and make it less likely that the solution will have to be refactored later on [7].

In this research, we introduce a method for guided decomposition in educational programming systems, using natural language. In particular, we aim to answer the following questions:

> How does using a guided program decomposition system affect a learner's cognitive processes, as observed in a concurrent think-aloud study?
>
> How does using a guided program decomposition system affect a learner's confidence in their own decomposition skills?

The implementation of this proposed method serves as an extension to JetBrains' educational programming system, Cognifire [8]. This system enables students to produce structured code using natural language prompts and is currently integrated into the in-IDE learning format [9] of the JetBrains Academy plugin [10], with support for Kotlin [11].

As part of this work, a part of the implementation of the proposed method was contributed. In particular, this research contributes the following:

- A novel program decomposition approach.
- Prototypes for the user interface design in the JetBrains Academy plugin.
- The artificial intelligence functionality of the system: crafting prompts and handling all interactions with the AI services.

The implementation of the user interface prototypes into the plugin was done by JetBrains.

The rest of this paper is structured as follows, Section 2 describes related work and similar existing tools. The methodology is then described in Section 3, including how the system was implemented and improved. Section 4 describes how the user study was conducted and what was inferred from the study. The results of the user study are then highlighted and discussed in Sections 5 and 6. Finally, the paper is concluded and future improvements are described in Section 7.

2 Related Work

2.1 Prompt-Based Programming

Researchers have highlighted the value of prompt engineering in the context of code generation. Liang et al. interviewed 20 prompt engineers and found notable differences between traditional software engineering and prompt programming, suggesting that a translation between the two must be approached with caution [12]. In [13], Ma et al. underline the importance of requirement specification when prompting a Large Language Model (LLM), and introduce a new human-AI system with a focus on requirement specification from a human perspective.

Promptly is an interactive platform created by Denny et al. that teaches students how to prompt code LLMs, by providing an example of functionality that should be replicated, as well as real-time execution of the generated solution [14].

Reeves et al. argue that the historical development of programming constructs has had a clear focus on increasing the level of abstraction. They argue that programming has always been heading towards a natural language level of abstraction and that GenAI has finally enabled this possibility. This new level of abstraction can encourage beginners to focus on actual problem-solving, rather than getting stuck in the daunting syntactical subtleties that each programming language has to offer [1].

Therefore, prompt programming provides several advantages over classical coding for novice programmers. It helps learners learn to solve problems at a higher level, rather than focusing on the exact technical details of different programming languages.

2.2 AI in CS Education

Liu et al. [15] explore the use of generative AI in Harvard University's introductory CS50 course with 70 students, in the form of a chatbot. They found that 73% of students found the tool to be "helpful" or "very helpful" and 70% of students found the tool to be "effective" or "very effective". However, they also point out that AI assistants are susceptible to errors, just as humans are, but that they display unwavering confidence despite being inaccurate, something which humans do not always tend to do.

Woodrow et al. [5] developed a real-time feedback system in a course with over 8,000 students. They found that students who received feedback in real-time were five times more likely to view and engage with their feedback than students who received delayed feedback. Furthermore, they found that students who viewed the provided feedback were 79% more likely to make relevant changes based on this feedback. Despite these positives, they find that the tool displayed inconsistent behaviour and was unable to effectively provide feedback on certain areas.

In [16], Zastudil et al. interviewed 12 students and 6 instructors to investigate their preferences regarding AI in computer education. They find that students and instructors are concerned about potential over-reliance on AI systems, as well as their trustworthiness. They also find that both sets of stakeholders believe that AI will broaden access due to its constant availability and low cost. However, they find that students and instructors are not exactly aligned on how to adapt assessment methods to keep up with the rise of AI in education.

CodeAid [17] is an LLM-powered programming assistant, developed by Kazemitabaar et al., which aims to help students without explicitly revealing solutions. The system answers conceptual questions, generates pseudo-code with line-by-line explanations, and annotates students' incorrect code with fix suggestions. The developers deployed this system in a programming class of 700 students and analysed the resulting 8,000 usages of the system. This analysis was further supplemented with student and educator interviews. They find that students value its constant availability, contextual knowledge of a given task, and question-answering capabilities.

Overall, students generally perceive AI programming tools to be helpful due to their context-based and instantaneous feedback. However, there is still considerable doubt over their trustworthiness from students and educators alike.

2.3 Teaching Program Decomposition

Haldeman et al. present a framework for teaching decomposition [18] in introductory programming courses. They identify three main concerns for program decomposition: the single responsibility principle, reduction of artificial coupling, and reusability. Furthermore, they suggest strategies to teach students how to identify decomposition patterns and procedures to decompose them.

Keen and Mammen introduce a long-term project approach to teaching decomposition in computer science. In this project, the students must solve a computer graphics problem by implementing smaller individual components, while building towards the overarching solution [6].

Earthworm [19] is an automated decomposition suggestion tool implemented by Garg and Keen. This tool leverages metrics such as variable flow, program slicing, cyclomatic complexity, and consecutive statements to generate decomposition suggestions.

Charitsis et al. delve into the reasons for students to decompose functions during a task. They analyse 45,000 snapshots from 168 students during an introductory programming assignment [7]. They designate three reasons for creating a new function: adding new functionality, restructuring code, and removing duplicate code. They find that the students who are able to correctly decompose the program at once, without the need for any restructuring, tend to need less time to find a viable solution.

There are several tools aimed at teaching students the skill of program decomposition. However, these tools focus on improving the structure of an already implemented program. Therefore, none of them explore the possible benefits of upfront decomposition, without any code.

3 Methodology

This section describes the proposed decomposition framework, the methodology for implementing the AI-guided system, and the iterative process of developing the system.

3.1 Framework for decomposition

As mentioned earlier, using natural language can help novice programmers develop a more intrinsic understanding of the programming concept that they are learning, free from the syntactical and technical differences between programming languages. Furthermore, we discussed the several benefits of AI in computer education, such as instantaneous and context-based feedback. Lastly, we discussed the possible benefits that decomposing a problem before implementation can have for novice programmers. Defining dependencies upfront and developing a better understanding of a given problem can help reduce complexity and refactoring during implementation. As discussed in Section 2, current tools to teach pro-

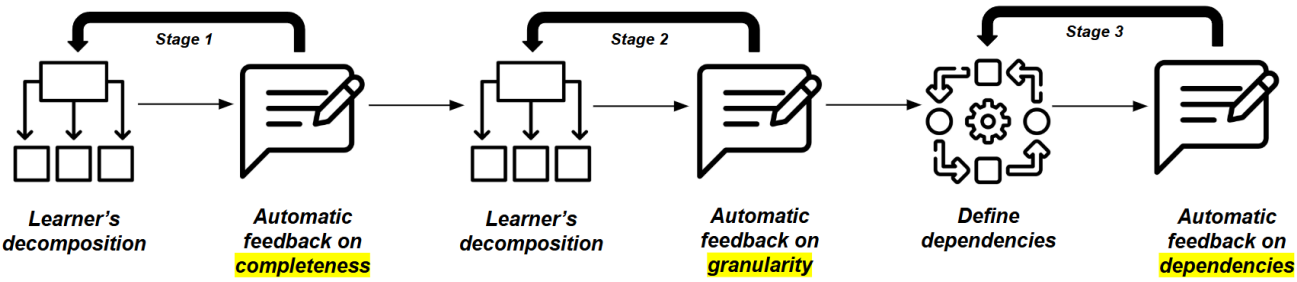

Figure 1: An overview of the AI program decomposition feedback system. The figure highlights how a user's decomposition is iteratively improved based on a three-stage feedback loop from the system. Once the user has a satisfactory solution for a stage, the feedback loop moves on to the next stage.

gram decomposition are focused on improving or refactoring already implemented programs, rather than developing an understanding of decomposition as a distinct step in the programming process. Combining all these points, we propose an AI feedback system for a student to learn program decomposition using natural language, before writing any code. This system follows a three-stage feedback loop, based on completeness, granularity, and defining dependencies. In this system, the user simply provides high-level descriptions of the functions needed to solve a given task.

Stage One: Completeness The first step checks the completeness of the user's proposed decomposition by verifying whether the provided set of functions satisfies the respective task. This stage aims to verify that the functions described by the user provide complete coverage of all the functionality needed to solve the provided problem.

Stage Two: Granularity The second step verifies whether the proposed solution follows good software architecture practices, specifically whether each function follows the Single Responsibility Principle. The Single Responsibility Principle (SRP) is a software architecture principle aimed at making a program robust and modular [20]. The principle states that a module must play a singular and distinct role in a system. In this case, a function described by the user must have a singular responsibility within their solution.

Stage Three: Dependencies The final stage verifies the relationships and dependencies between functions as defined by the user. Here, the user connects the functions on a high level. Specifically, the user defines which functions would be dependent on which other functions when the program is implemented. This stage aims to help the user better understand the flow of the program so they have a clearer idea of how the functions are connected when they implement their solution in code. By defining these dependencies, the user can map out a recomposition of their sub-tasks into their final solution.

The user iteratively improves their solution based on the feedback provided, and only once the solution is deemed to be satisfactory for the current stage, the feedback loop moves on to the next stage. An overview of this system is shown in Figure 1.

3.2 Implementing the system

The feedback system begins with a provided problem description which must be decomposed by the user. The user provides the set of functions that they believe will be sufficient to solve this problem. It is important to note that the user does not actually provide implementations of the functions that they propose, but only a high-level description of what the function does, described in natural language.

At each stage, the proposed solution is verified using a Large Language Model (LLM). The LLM is provided with the set of functions described by the user, a description of the task to be solved, and the specific criteria to be met at a given stage.

First, to verify the completeness of a set of functions with respect to a given problem, the LLM is prompted to check whether the combination of the functions as a whole would be adequate to solve the task. This approach is chosen over using a model solution and comparing the user's solution to the model solution due to the inherently subjective nature of decomposing a problem. As there is not just a singular ideal solution to a problem, the system instead verifies the proposed solution without a reference solution, thus allowing for individual design choices.

To verify whether the functions provided by the user follow the single responsibility principle, the LLM is prompted to analyse each function description separately. The LLM checks whether a function is described as performing multiple tasks, which should be split into separate functions to keep in line with the SRP. If a function does not adhere to this principle, it is highlighted, and the user is expected to rework their function description, potentially by creating a new function for one of the sub-tasks.

Finally, the user defines the expected dependencies between functions. Specifically, the user defines which function(s) would make a call to which other function(s) in a future implementation of the solution. This stage aims to help the user consider how the different modules in their solution will interact, thus setting the foundation for when they eventually implement their described functions. The LLM

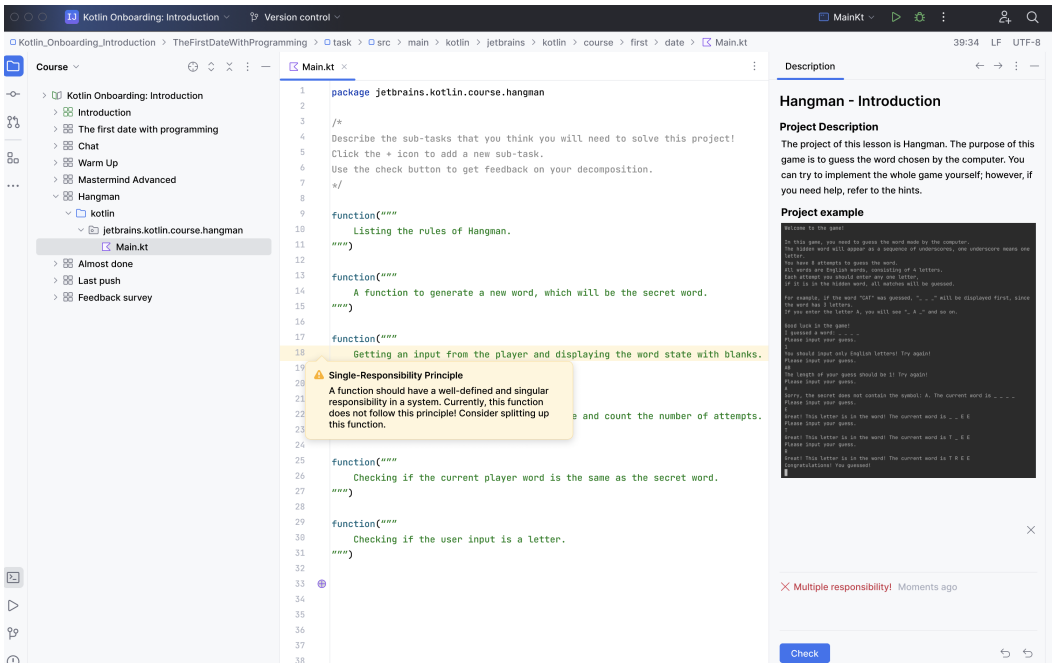

Figure 2: An example of how the decomposition system provides feedback in case a function does not follow the Single Responsibility Principle.

is prompted to verify that the flow of the program is logical using the user-defined dependencies and to ensure that all functions are connected within the larger program.

An example of the system prototype inside the JetBrains IntelliJ IDE is displayed in Figure 2.

3.3 Validating the system

As the LLM's ability to correctly verify a solution is at the core of the system's functionality, it is necessary to fine-tune the prompts used for each stage of the system's feedback loop. For stages one and three, this was done by generating simple programming tasks and 25 valid and 25 invalid solutions. For stage two, 50 function descriptions were generated, half of which followed the SRP and the other half did not. Several prompting methods were tested, such as adjusting the temperature of the LLM, using different models, and altering the structure of the prompt. In the end, the prompts were augmented with few-shot learning. This process was refined until the LLM was able to correctly verify the valid solutions in the generated dataset with 100% accuracy and correctly label the invalid solutions with a satisfactory accuracy of over 80%, depending on the corresponding stage.

4 Student Experiments

To evaluate the proposed method of teaching program decomposition and the developed system, student experiments were conducted. These experiments took the form of a concurrent think-aloud study. The study aims to observe how a student's cognitive process of decomposing a given programming problem changes while using the system. Furthermore, the study also measures the students' self-confidence in their decomposition skills before and after using the tool.

This study employs a mixed-methods analysis to examine how a feedback-enhanced decomposition system affects the cognitive processes of novice programmers. The setup of the experiment was as follows: the study consisted of 12 first- and second-year computer science students from the Delft University of Technology. Each student participated in an interview lasting approximately one hour. The student decomposed three simple programming tasks using the feedback system, each lasting around 20 minutes. The tasks are sourced from the *Kotlin Onboarding: Introduction* course from the JetBrains Academy plugin, these tasks are presented in Appendix A. While performing the decomposition, the students were asked to verbalise their thoughts about their experience with decomposing the task, the feedback provided by the system, and any other relevant thoughts that the students may have about the task. The students were also asked to complete a pre- and post-confidence survey on their decomposition skills.

4.1 Self-confidence assessment

The participants rated their self-confidence on decomposition-related skills, before and after performing the tasks and using the tool. This assessment is done on a five-point Likert scale of strongly agree, agree, neutral, disagree, and strongly disagree, adapted from validated sources [21]–[23]. The skills evaluated in this assessment are displayed in Table 1.

Item #	Statement
Q1	I am confident in my ability to break a programming task into smaller parts.
Q2	I feel comfortable identifying the exact steps needed to complete a programming sub-task.
Q3	I know how to decide which sub-tasks should become separate functions.
Q4	When I plan a program, I trust my ability to structure it well from the start.

Table 1: Self-Confidence Assessment Questions for Programming Task Decomposition Skills.

Category	Code	Description
Decomposition Reasoning	DEC-COMPLETE	Function descriptions cover all major responsibilities in the task
	DEC-INCOMPLETE	One or more key functions are missing from the decomposition
	DEC-SINGLE	Each function has a single, clear responsibility
	DEC-MULTI	One or more functions combine unrelated or multiple concerns
Feedback Interpretation	FB-UNDERSTAND	Student correctly interprets the feedback on decomposition
	FB-CONFUSED	Student misinterprets or is unsure about the feedback
	FB-DISAGREE	Student disagrees with the feedback
	FB-APPLY	Student revises their decomposition based on the feedback
Cognitive Strategy	PLAN-HIERARCHY	Student reasons about structure or relationships between functions
	PLAN-EXAMPLES	Student uses example inputs or outputs to inform decomposition
Metacognitive/Affective States	CONFIDENT	Expressing certainty or clarity about decomposition choices
	CONFUSED	Expressing uncertainty or hesitation about what to include
	REFLECT	Reflecting on mistakes, learning, or changes in thinking

Table 2: Coding scheme for the think-aloud protocol based on the decomposition feedback tool.

4.2 Think-Aloud Protocols in Computer Science Education

The think-aloud protocol was first developed by Ericsson and Simon [24]. This method involves participants thinking aloud while completing a set of specified tasks. The goal of this method is to understand the cognitive processes behind a participant's actions. While think-aloud protocols have been used in several different domains, their prevalence in computing education is sparse.

In [25], Whalley et al. use a think-aloud protocol to gain insight into the performance and behaviour of three students performing debugging tasks. In this study, the participants were asked to think aloud while solving the given debugging tasks. Furthermore, the participants were asked to rate their confidence in their programming and problem-solving abilities. Finally, a retrospective think-aloud interview was also conducted, where participants were asked to explain certain actions and revisit incoherent utterances.

Think-aloud protocols provide direct insight into how people understand problems, make decisions, and reason through complex activities, making them a valuable tool in fields such as cognitive psychology, human-computer interaction, usability testing, and education. However, there are also certain drawbacks to think-aloud protocols. Constant verbalisation while performing tasks may lead to a higher cognitive workload, potentially affecting performance. Furthermore, due to the unnatural requirement to verbalise all relevant thoughts, the participant can find it hard to articulate their mental models. Lastly, some participants may be naturally better at verbalising their thoughts, thus leading to high variability in the results.

Despite these drawbacks, a concurrent think-aloud protocol was chosen as the evaluation method for this study. Due to the limited timeframe of this research, it was not viable to measure how well a student can learn program decomposition while using the system. Furthermore, A/B testing, where one group decomposes a problem using the system and the other decomposes it without using it, is not logical as the system will eventually converge to an 'ideal' solution. Therefore, a think-aloud protocol was chosen due to its ability to extract trends in cognitive processes exhibited by the learner, which can be indicative of a broader process of learning and internalising concepts.

4.3 Think-aloud interview and processing

All think-aloud sessions were audio-recorded and transcribed verbatim. Non-verbal vocalisations (e.g., pauses, sighs, self-directed mutterings) were retained when they contributed to interpreting cognitive or affective states.

A theory-driven coding scheme was developed and informed by existing models of self-regulated learning, programming cognition, and feedback engagement [26]–[29]. The transcriptions were then annotated using this coding scheme.

The scheme comprises four major code families, each with

subcategories to capture nuances in students' behaviours and thought processes. These themes and their corresponding codes are shown in Table 2.

The coding was manually done concurrently with the interview. Later, the audio recordings and transcripts were revisited and the code occurrences were validated and corrected. Originally, the validation was planned to be conducted using a semi-automatic annotation tool. However, due to the nature of the data and the coding scheme, the tools were unable to correctly distinguish between and annotate the cognitive and feedback interpretation categories, and the annotations had to be corrected completely manually. Since the decomposition reasoning category is purely quantitative, and the cognitive and feedback interpretation categories were not well annotated using semi-automatic tools, the validation was done manually.

4.4 Interpretation of collected data

In order to interpret the data collected during the think-aloud interview, changes in coding patterns over the three tasks are analysed. We tracked the frequency, sequence, and co-occurrence of codes across phases to observe progression in behaviour and cognitive processes.

The first coding change analysed is the change in DEC-INCOMPLETE and DEC-MULTI codes. In particular, we observe how many DEC-INCOMPLETE codes are detected during each task before reaching a DEC-COMPLETE code. Similarly, we observe this pattern for DEC-MULTI to DEC-SINGLE codes.

Next, we analyse the emergence of feedback uptake and cognitive strategies. The presence of these codes signifies that the student understands how the feedback provided by the tool relates to their work and has a clear plan on how to tackle the given task.

Finally, we analyse the emergence of metacognitive codes. In particular, we observe whether there is an emergence of CONFIDENT and REFLECT codes, and a decrease in CONFUSED codes. These codes reflect a student's emotional state while performing a task.

In order to interpret the data collected using the self-confidence surveys, we measure the difference in reported confidence before using the system and after completing all tasks using the tool during the interview. We observe whether there are any statistically significant confidence gains in the post-confidence survey.

Patterns across these layers are used to infer both individual and group-level improvements.

5 Results

This section describes the results of the self-confidence assessment, the think-aloud protocol code trends, and general opinions on the method and system.

5.1 Self-confidence assessment

Figure 3 shows the average self-confidence reported by students before and after using the decomposition system. We observe that the average perceived self-confidence in the four skills, before and after the experiment, remains consistent. In fact, the average confidence for question 1 and question 4 remains equal before and after the experiment. Meanwhile, the average confidence for question 2 slightly increases and the average confidence for question 3 slightly decreases.

When asked why their confidence changed, students generally had two lines of thought. In case of an increase in confidence, the students said that they felt confident as they did not need much feedback from the system, either throughout all three tasks or due to a decrease in the feedback required after the first task.

In case of a decrease in confidence, some students felt like they had initially overestimated their decomposition abilities. This feeling of underconfidence was directly linked to needing several rounds of feedback from the system before reaching a valid solution. Interestingly, some students exhibited this underconfidence even if the number of feedback rounds needed decreased over the three tasks.

5.2 Think-aloud protocol

Table 3 shows the occurrences of certain codes for each participant in the three tasks.

For the decomposition reasoning codes, we see that 8 out of the 12 participants display a decrease in the number of incomplete iterations before reaching a complete set of functions, from the first to the third task. 2 of the 12 students had an increase in the number of incomplete iterations between the first and last task. The participants generally performed well on the single-responsibility checks, with a significantly lower amount of iterations required. 5 of the 12 students needed a lower number of feedback rounds for this step by the third task, and no students needed more rounds.

For the feedback uptake codes, the trend was less clear. Overall, the participants did not disagree with the feedback very often, 5 of the participants did not disagree with the feedback provided at any stage of the interview. The number of times the feedback was understood by participants, proportional to the number of times they received feedback, did not show any significant change. There was a clear decrease in the number of times the students were confused by the feedback, with 10 participants showing a decrease in this category by the final task.

There was no clear trend in the number of hierarchical planning codes compared to the number of example-based codes. In general, students tended to maintain the type of planning they used while completing the tasks.

Confidence codes exhibited a generally downward trend over the three tasks, with 5 students displaying this trend. Meanwhile, 2 students showed an increase in confidence codes. 7 out of the 12 students displayed a downward trend in confusion, while none of the participants had an increase in confusion. All participants showed some form of reflection over the course of the interview.

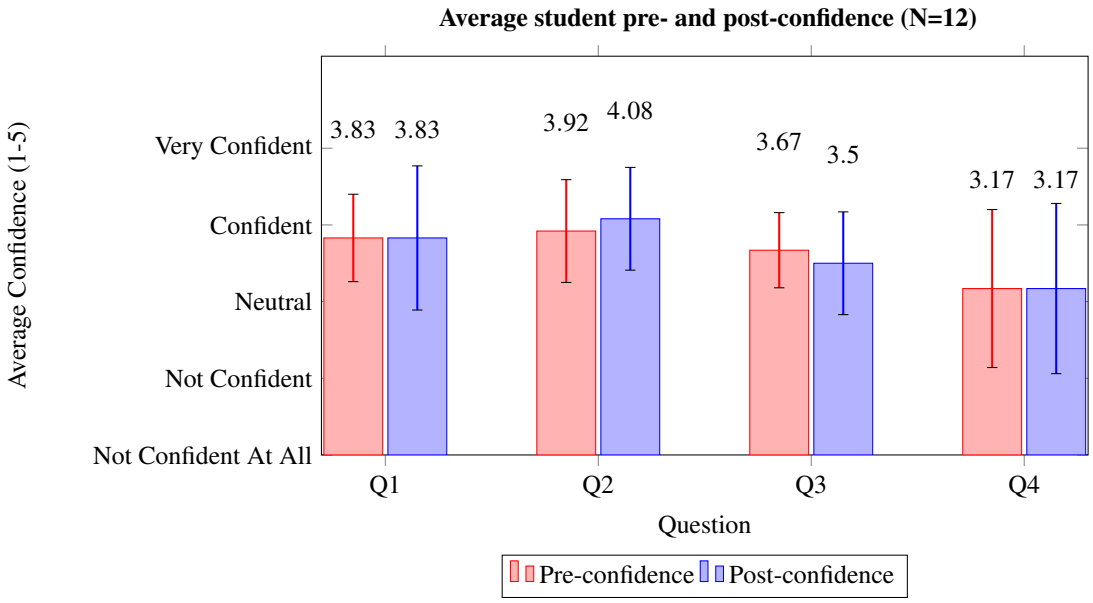

Figure 3: Comparison of mean pre- and post-confidence levels with error bars on a 5-point Likert scale, based on the questions stated in Table 1. The survey asks the students about their perceived abilities in: Breaking-down large problems [Q1], identifying exact functionalities needed in a sub-task [Q2], following the single responsibility principle [Q3], and structuring a program well from the beginning [Q4].

		Decomposition		Feedback			Planning		Metacognitive		
Participant	**Task**	**INCOMPLETE**	**MULTI**	**UNDERSTAND**	**DISAGREE**	**CONFUSED**	**HIERARCHY**	**EXAMPLE**	**CONFIDENT**	**CONFUSED**	**REFLECT**
P01	One	4	1	3	1	4	6	5	2	4	0
	Two	3	0	3	1	1	8	4	0	0	1
	Three	4	1	4	2	2	4	3	3	0	1
P02	One	4	0	3	0	1	8	2	3	9	1
	Two	2	0	2	0	0	6	4	1	0	5
	Three	0	0	0	0	0	4	2	2	1	3
P03	One	1	1	0	2	2	4	6	1	2	0
	Two	1	0	0	2	0	4	5	0	0	2
	Three	0	0	0	0	0	3	2	0	0	1
P04	One	3	1	4	0	4	0	6	2	6	0
	Two	4	0	1	1	0	0	8	0	0	2
	Three	2	1	2	1	1	0	5	0	1	1
P05	One	0	0	0	0	0	6	1	0	0	1
	Two	1	0	0	0	1	5	3	0	0	1
	Three	2	0	1	0	0	3	6	0	0	1
P06	One	6	0	2	1	1	1	5	0	0	1
	Two	4	0	2	1	0	2	3	0	0	1
	Three	1	0	0	1	0	1	2	2	0	0
P07	One	3	0	1	1	2	4	6	0	0	2
	Two	3	0	3	0	1	5	8	1	0	5
	Three	0	0	0	0	0	1	4	0	0	1
P08	One	1	1	1	0	0	6	5	0	1	2
	Two	2	0	2	0	0	7	4	0	0	2
	Three	1	0	1	0	0	3	5	0	0	1
P09	One	3	0	2	0	1	0	7	0	1	0
	Two	1	0	1	0	1	3	9	1	2	2
	Three	0	0	0	0	0	0	6	0	0	1
P10	One	2	1	1	1	2	2	3	1	0	1
	Two	7	1	4	0	2	2	3	0	0	0
	Three	3	0	1	0	0	1	2	0	0	0
P11	One	14	4	10	0	7	0	6	0	3	2
	Two	4	2	2	0	2	0	6	0	1	2
	Three	5	2	3	0	1	0	5	0	0	2
P12	One	8	3	1	2	5	1	3	1	0	1
	Two	6	2	3	2	1	0	4	0	0	2
	Three	0	0	0	0	0	1	4	0	0	1

Table 3: Coding scheme occurrences in the think-aloud study, by participant and task. Participants 2 and 11 are highlighted as representative examples of the observed general trend. We observe a clear decrease in incomplete decomposition occurrences. Furthermore, there is a decrease in the confused metacognitive state and an emergence of reflection based on past errors.

5.3 General opinions on method

The opinion on the decomposition method was generally positive. 2 students had some negative opinions on the method. One student said, "I would probably just use ChatGPT for this kind of thing." Another student said, "I don't think I would use it much at this point [in my programming jour-

ney]." The other participants had a generally positive opinion of the method. One of the students was also appreciative of the dependencies step of the system. They said, "I think the dependencies is a good way to model things. It makes sense in my brain... I think this is a good level of abstraction to decompose a problem."

Regarding the idea of decomposing before implementation, a student said, "I think it's really really useful as a planning tool because I often have trouble with planning stuff and when I just start coding and I realise something midway, I have to fix a lot of things and it's a lot of time wasted that way." One of the students who provided some negative feedback also said, "once you've made a small enough problem anyone can solve anything, right? I mean if the function is to print Hello World! then anyone can do it. There is definitely a lot of value to getting to that point." There were also a few students who stated that they would probably not decompose a problem before implementation in such a manner, but that they see the value in doing so, especially for programmers who are just starting out.

The idea of using natural language to do this decomposition was slightly more divisive. Most students found it to be more enjoyable as they did not have to "first think about the problem in natural language, then think about how to solve it in code." However, a few students had opposing views. One student said, "we as programmers are not as used to putting everything into human language, you are used to just coding stuff, and maybe you're not as good in describing it as you think you are." Furthermore, certain students also struggled with finding the right phrase for what they wanted to describe, this was mainly due to a slightly lower proficiency in the English language.

The opinions on the feedback provided by the system were generally positive. However, the negative opinions on the feedback were not exactly unanimous. Some students felt that the feedback was not specific enough, while others felt like it was pushing them too much towards a very specific direction. One of the students also suggested that they would have preferred if the feedback regarding the completeness became more obvious with every incorrect solution provided to the system.

6 Discussion

From the student experiments, we see that there is no statistically significant difference in the reported self-confidence of the participants before and after the experiment. We reason that this is due to the participants changing their perception of their decomposition skills in two ways, based on their statements while completing the survey after the interview. First, the students believe that, over the course of using the tool, they have improved their decomposition skills due to an improvement in performance (decrease in feedback needed). Second, the students realised that they had over-estimated their decomposition abilities, and they felt less confident due to the number of feedback rounds needed to solve the tasks. Furthermore, many students did not believe that their decomposition skills had been affected while using the system, thus there was no change in their confidence levels. Due to these reasons, the average results remain unchanged.

From the coding of the think-aloud protocol, we observed no obvious trend in the planning of the cognitive strategies of the participants. There was also no obvious change in understanding feedback by the system. However, there was a significant decrease in confusion caused by the feedback from the system, which shows that the participants were able to grasp what the system suggested, after a short initial learning curve.

The most interesting trends came in the metacognitive category. There was a small decline in confidence codes exhibited by the participants. However, there was an even sharper decline in the confusion displayed by the participants. This clearly shows that the participants started to develop a clearer picture of what was expected from them while decomposing a task, as several participants did not know where to start at the beginning of the first task. Lastly, each participant reflected on their past errors throughout the interview, thus showing that they had internalised feedback provided by the system and implemented those learnings in future tasks, or even later in the same task.

The participants displayed a clear trend in the reduction of the number of iterations of feedback needed to reach a final solution. Thus, their performance throughout the interview undoubtedly improved, in relation to what the system considered to be a valid solution. Once again, this shows a clear internalisation of what was expected of them from the system.

Most students maintained a positive opinion of the method used in the system, in particular, the upfront decomposition and defining dependencies. Even participants who said that they would not necessarily use the system themselves noted that it could be a valuable tool for novice programmers. The use of natural language was more divisive, as more students struggled with correctly phrasing their descriptions or finding the right words due to a lack of English proficiency. However, many students also noted the benefits of being able to describe functions in the same way as they think. Some of these issues can be rectified by extending support to more languages. Furthermore, with the increasing level of abstraction in programming tools, it is possible that future generations of programmers will not struggle as much with using natural language in programming.

7 Conclusions and Future Work

In this research, we proposed a new method of teaching program decomposition, involving natural language and upfront decomposition of a problem. We aimed to evaluate how the new method, as implemented in an AI system, affects the self-confidence of students in their decomposition skills and how their (meta)cognitive processes change while using the system.

We find that the reported self-confidence of the students

does not display any significant change. We note a significant increase in performance, in terms of the number of feedback rounds needed before reaching a valid solution, Finally, we note a clear decrease in confusion while decomposing a problem and an emergence of reflection based on past errors. Thus, we conclude that the method is able to somewhat internalise good decomposition practices, as determined by the AI system. We also find that students found the proposed method of decomposition to be beneficial overall. However, some students struggled with the natural language aspect of the system.

Limitations Due to the limited time span of this research, it was not possible to conduct a long-term observation of the development of decomposition skills in the participants. Therefore, it is not possible to conclude that the proposed decomposition method improves the decomposition skills of the participants, rather we can only observe trends in the cognitive and metacognitive processes during the short think-aloud study. Furthermore, the current decomposition method emphasises a function-level decomposition. Therefore, other essential aspects of software engineering, such as an object-oriented approach, are not covered.

Future work For the future, we recommend extending support for more languages and more decomposition paradigms in the system. For the evaluation of the method, we recommend a longer-term study of how the students' decomposition skills develop, to observe the full effects of the internalisation of decomposition practices from the method. Furthermore, conducting the study with participants who have minimal programming experience, such as high school students rather than university students, would be more accurate as that is the intended demographic for the system.

References

[1] B. N. Reeves, J. Prather, P. Denny, *et al.*, *Prompts first, finally*, 2024. arXiv: 2407.09231 [cs.CY]. [Online]. Available: https://arxiv.org/abs/2407.09231.

[2] M. Halpern, "Foundations of the case for natural-language programming," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall), San Francisco, California: Association for Computing Machinery, 1966, pp. 639–649, ISBN: 9781450378932. DOI: 10.1145/1464291.1464360. [Online]. Available: https://doi.org/10.1145/1464291.1464360.

[3] J. Prather, P. Denny, J. Leinonen, *et al.*, "The robots are here: Navigating the generative ai revolution in computing education," in *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '23, Turku, Finland: Association for Computing Machinery, 2023, pp. 108–159, ISBN: 9798400704055. DOI: 10.1145/3623762.3633499. [Online]. Available: https://doi.org/10.1145/3623762.3633499.

[4] A. Raman and V. Kumar, "Programming pedagogy and assessment in the era of ai/ml: A position paper," in *Proceedings of the 15th Annual ACM India Compute Conference*, ser. COMPUTE '22, Jaipur, India: Association for Computing Machinery, 2022, pp. 29–34, ISBN: 9781450397759. DOI: 10.1145/3561833.3561843. [Online]. Available: https://doi.org/10.1145/3561833.3561843.

[5] J. Woodrow, A. Malik, and C. Piech, "AI Teaches the Art of Elegant Coding: Timely, Fair, and Helpful Style Feedback in a Global Course," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2024, event-place: Portland, OR, USA, New York, NY, USA: Association for Computing Machinery, 2024, pp. 1442–1448, ISBN: 9798400704239. DOI: 10.1145/3626252.3630773. [Online]. Available: https://doi.org/10.1145/3626252.3630773.

[6] A. Keen and K. Mammen, "Program decomposition and complexity in cs1," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15, Kansas City, Missouri, USA: Association for Computing Machinery, 2015, pp. 48–53, ISBN: 9781450329668. DOI: 10.1145/2676723.2677219. [Online]. Available: https://doi.org/10.1145/2676723.2677219.

[7] C. Charitsis, C. Piech, and J. C. Mitchell, "Detecting the reasons for program decomposition in cs1 and evaluating their impact," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023, Toronto ON, Canada: Association for Computing Machinery, 2023, pp. 1014–1020, ISBN: 9781450394314. DOI: 10.1145/3545945.3569763. [Online]. Available: https://doi.org/10.1145/3545945.3569763.

[8] A. Potriasaeva, K. Dzialets, Y. Golubev, and A. Birillo, *Using a low-code environment to teach programming in the era of llms*, Jun. 2024.

[9] A. Birillo, M. Tigina, Z. Kurbatova, *et al.*, *Bridging education and development: Ides as interactive learning platforms*, 2024. arXiv: 2401.14284 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2401.14284.

[10] *Jetbrains academy plugin*. [Online]. Available: https://plugins.jetbrains.com/plugin/10081-jetbrains-academy.

[11] "Kotlin." (), [Online]. Available: https://kotlinlang.org/.

[12] J. T. Liang, M. Lin, N. Rao, and B. A. Myers, *Prompts are programs too! understanding how developers build software containing prompts*, 2024. arXiv: 2409.12447 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2409.12447.

[13] Q. Ma, W. Peng, H. Shen, K. Koedinger, and T. Wu, *What you say = what you want? teaching humans to articulate requirements for llms*, 2024. arXiv: 2409.08775 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2409.08775.

[14] P. Denny, J. Leinonen, J. Prather, *et al.*, *Promptly: Using prompt problems to teach learners how to effectively utilize ai code generators*, 2023. arXiv: 2307.16364 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2307.16364.

[15] R. Liu, C. Zenke, C. Liu, A. Holmes, P. Thornton, and D. J. Malan, "Teaching cs50 with ai: Leveraging generative artificial intelligence in computer science education," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2024, Portland, OR, USA: Association for Computing Machinery, 2024, pp. 750–756, ISBN: 9798400704239. DOI: 10.1145/3626252.3630938. [Online]. Available: https://doi.org/10.1145/3626252.3630938.

[16] C. Zastudil, M. Rogalska, C. Kapp, J. Vaughn, and S. MacNeil, *Generative ai in computing education: Perspectives of students and instructors*, 2023. arXiv: 2308.04309 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2308.04309.

[17] M. Kazemitabaar, R. Ye, X. Wang, *et al.*, "CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24, event-place: Honolulu, HI, USA, New York, NY, USA: Association for Computing Machinery, 2024, ISBN: 9798400703300. DOI: 10.1145/3613904.3642773. [Online]. Available: https://doi.org/10.1145/3613904.3642773.

[18] G. Haldeman, J. R. Bernal, A. Wydra, and P. Denny, *Teaching program decomposition in cs1: A conceptual framework for improved code quality*, 2024. arXiv: 2411.09463 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2411.09463.

[19] N. Garg and A. W. Keen, "Earthworm: Automated decomposition suggestions," in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '18, Koli, Finland: Association for Computing Machinery, 2018, ISBN: 9781450365352. DOI: 10.1145/3279720.3279736. [Online]. Available: https://doi.org/10.1145/3279720.3279736.

[20] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st. USA: Prentice Hall Press, 2017, ISBN: 0134494164.

[21] R. Schwarzer, M. Jerusalem, J. Weinman, S. Wright, and M. Johnston, "Generalized self-efficacy scale," *Measures in Health Psychology: A User's Portfolio. Causal and control beliefs Windsor*, Jan. 1995.

[22] V. Ramalingam, D. LaBelle, and S. Wiedenbeck, "Self-efficacy and mental models in learning to program," *SIGCSE Bull.*, vol. 36, no. 3, pp. 171–175, Jun. 2004, ISSN: 0097-8418. DOI: 10.1145/1026487.1008042. [Online]. Available: https://doi.org/10.1145/1026487.1008042.

[23] A. Bandura, "Social cognitive theory of self-regulation," *Organizational Behavior and Human Decision Processes*, vol. 50, no. 2, pp. 248–287, 1991, Theories of Cognitive Self-Regulation, ISSN: 0749-5978. DOI: https://doi.org/10.1016/0749-5978(91)90022-L. [Online]. Available: https://www.sciencedirect.com/science/article/pii/074959789190022L.

[24] K. A. Ericsson and H. A. S. and, "How to study thinking in everyday life: Contrasting think-aloud protocols with descriptions and explanations of thinking," *Mind, Culture, and Activity*, vol. 5, no. 3, pp. 178–186, 1998. DOI: 10.1207/s15327884mca0503_3. eprint: https://doi.org/10.1207/s15327884mca0503_3. [Online]. Available: https://doi.org/10.1207/s15327884mca0503_3.

[25] J. Whalley, A. Settle, and A. Luxton-Reilly, "A think-aloud study of novice debugging," *ACM Trans. Comput. Educ.*, vol. 23, no. 2, Jun. 2023. DOI: 10.1145/3589004. [Online]. Available: https://doi.org/10.1145/3589004.

[26] J. Hattie and H. Timperley, "The power of feedback," *Review of Educational Research*, vol. 77, no. 1, pp. 81–112, 2007. DOI: 10.3102/003465430298487. eprint: https://doi.org/10.3102/003465430298487. [Online]. Available: https://doi.org/10.3102/003465430298487.

[27] N. Kiesler, *An exploratory analysis of feedback types used in online coding exercises*, 2022. arXiv: 2206.03077 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2206.03077.

[28] M. Wu, N. Goodman, C. Piech, and C. Finn, *Prototransformer: A meta-learning approach to providing student feedback*, 2021. arXiv: 2107.14035 [cs.CY]. [Online]. Available: https://arxiv.org/abs/2107.14035.

[29] M. Someren, Y. Barnard, and J. Sandberg, *The Think Aloud Method - A Practical Guide to Modelling CognitiveProcesses*. Jan. 1994.

A Programming Tasks

The programming tasks used in the study, adapted from the Kotlin: Onboarding course of the JetBrains Academy Plugin, are listed below.

Mastermind The project of this lesson is Bulls and cows (Mastermind). This is a popular children's game of guessing the hidden word. With each attempt, the player receives the number of exact matches (correct letters in the right position) and partial matches (correct letters in the wrong position). For example, with ACEB as the hidden word, the BCDF guess will result in 1 full match (C) and 1 partial match (B).

Pattern generator The project of this lesson is a pattern generator. The purpose of this project is to create an application for automatically generating character images of a size and recurring pattern provided by the user, using a text input interface. The user can either provide their own pattern or choose a pre-defined pattern.

Hangman The project of this lesson is Hangman. The purpose of this game is to guess the word chosen by the computer, letter-by-letter. Each incorrect guess brings the player closer to "hanging" a stick figure. The game ends when the word is guessed or the figure is fully drawn.

4

Supplementary Materials

4.1 Student Experiment

Human Research Ethics

The experiment involves human participants, who are its primary focus. In accordance with TU Delft's policies, "all research involving Human Research Subjects – including Master's theses – requires approval from the Human Research Ethics Committee (HREC) before it can go ahead.[1]" Therefore, prior to conducting the experiment described earlier, an application was submitted to the HREC, and approval was granted to proceed with the study. This approval process is intended to minimise potential risks to participants. In this context, such risks could include the collection of sensitive information, improper handling of data, or participants feeling coerced into taking part. As specified in the HREC application, the experiment does not collect any personally identifiable information, as it is not relevant to the study. All data is stored in locations that comply with TU Delft's current data management practices. Participation in the experiment is entirely voluntary, and participants are free to decline or withdraw at any time.

4.2 System Prototypes

[1] https://www.tudelft.nl/over-tu-delft/strategie/integriteitsbeleid/human-research-ethics

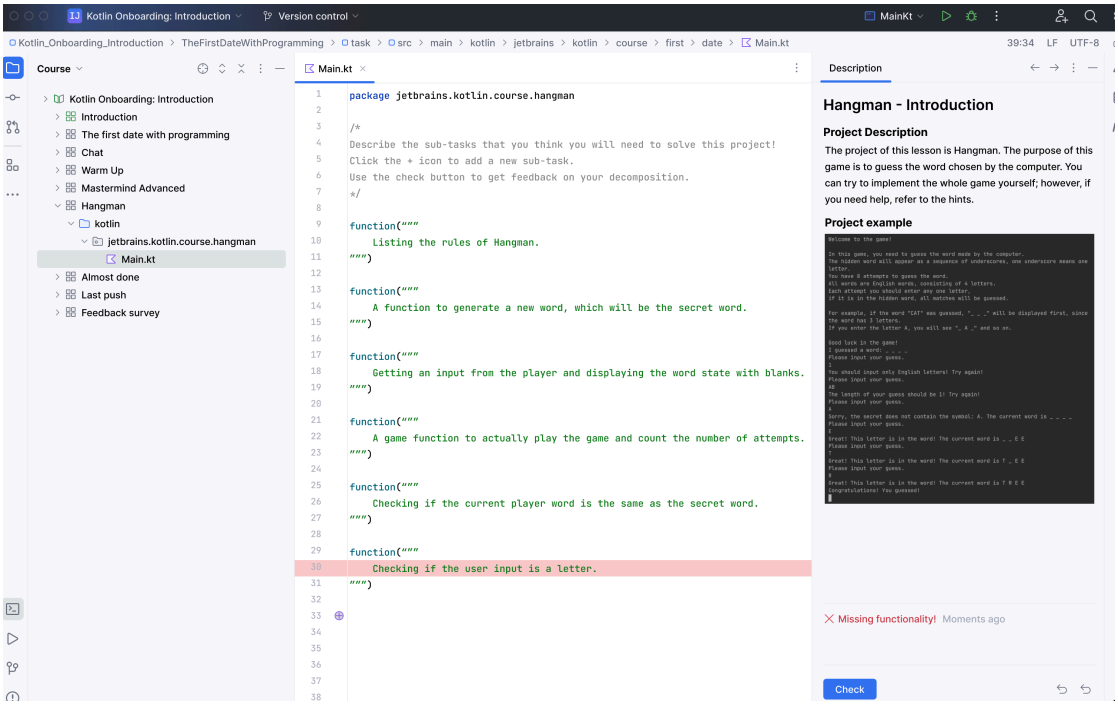

Figure 4.1: *An example of how the system provides feedback on an incomplete decomposition.*

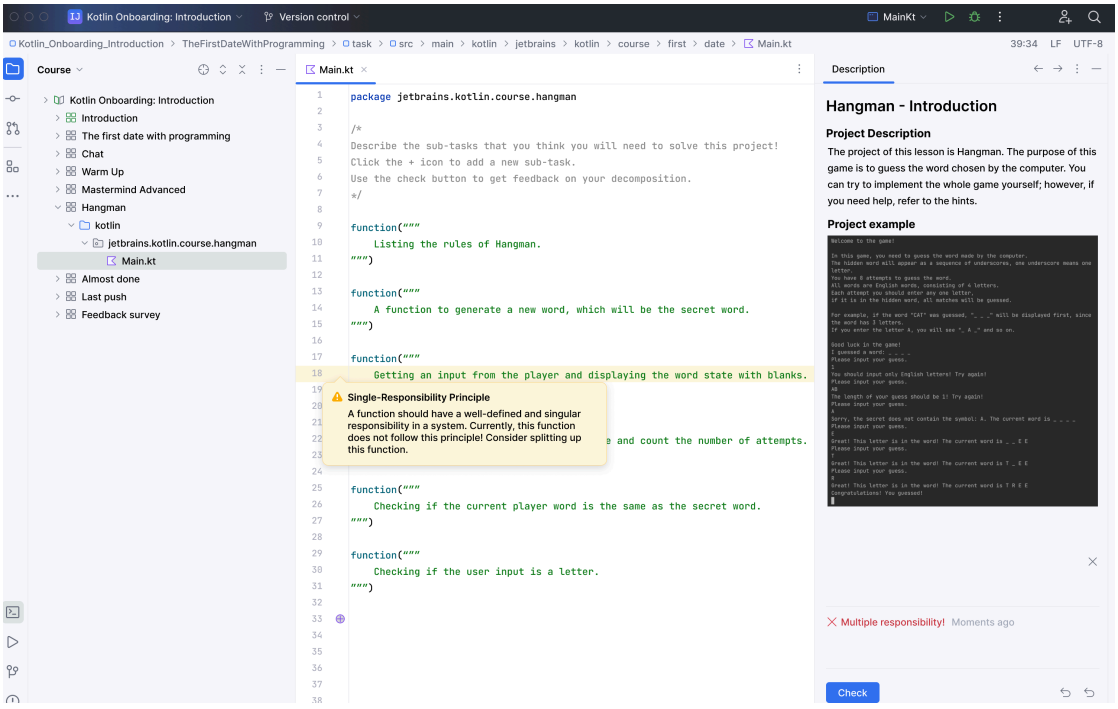

Figure 4.2: *An example of how the system provides feedback on a function which does not follow the Single Responsibility Principle.*

Bibliography for Introductory Materials

Balse, Rishabh et al. (2023). "Evaluating the Quality of LLM-Generated Explanations for Logical Errors in CS1 Student Programs". In: *Proceedings of the 16th Annual ACM India Compute Conference*. COMPUTE '23. Hyderabad, India: Association for Computing Machinery, pp. 49–54. ISBN: 9798400708404. DOI: 10.1145/3627217.3627233. URL: https://doi.org/10.1145/3627217.3627233.

Birillo, Anastasiia et al. (2024). *Bridging Education and Development: IDEs as Interactive Learning Platforms*. arXiv: 2401.14284 [cs.SE]. URL: https://arxiv.org/abs/2401.14284.

Boyd, Stephen P. et al. (2008). "Notes on Decomposition Methods". In: URL: https://api.semanticscholar.org/CorpusID:4539264.

Charitsis, Charis, Chris Piech, and John C. Mitchell (2023). "Detecting the Reasons for Program Decomposition in CS1 and Evaluating Their Impact". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2023. Toronto ON, Canada: Association for Computing Machinery, pp. 1014–1020. ISBN: 9781450394314. DOI: 10.1145/3545945.3569763. URL: https://doi.org/10.1145/3545945.3569763.

Denny, Paul et al. (2023). *Prompt Problems: A New Programming Exercise for the Generative AI Era*. arXiv: 2311.05943 [cs.HC]. URL: https://arxiv.org/abs/2311.05943.

Ericsson, K. Anders and Herbert A. Simon and (1998). "How to Study Thinking in Everyday Life: Contrasting Think-Aloud Protocols With Descriptions and Explanations of Thinking". In: *Mind, Culture, and Activity* 5.3, pp. 178–186. DOI: 10.1207/

s15327884mca0503_3. eprint: https://doi.org/10.1207/s15327884mca0503_3. URL: https://doi.org/10.1207/s15327884mca0503_3.

Halpern, Mark (1966). "Foundations of the case for natural-language programming". In: *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*. AFIPS '66 (Fall). San Francisco, California: Association for Computing Machinery, pp. 639–649. ISBN: 9781450378932. DOI: 10.1145/1464291.1464360. URL: https://doi.org/10.1145/1464291.1464360.

Hsu, Ting, Shao-Chen Chang, and Yu-Ting Hung (July 2018). "How to learn and how to teach computational thinking: Suggestions based on a review of the literature". In: *Computers Education* 126. DOI: 10.1016/j.compedu.2018.07.004.

JetBrains Academy Plugin (n.d.). URL: https://plugins.jetbrains.com/plugin/10081-jetbrains-academy.

Keen, Aaron and Kurt Mammen (2015). "Program Decomposition and Complexity in CS1". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: Association for Computing Machinery, pp. 48–53. ISBN: 9781450329668. DOI: 10.1145/2676723.2677219. URL: https://doi.org/10.1145/2676723.2677219.

Lesgold, Alan M. (2001). "The nature and methods of learning by doing." In: *The American psychologist* 56 11, pp. 964–73. URL: https://api.semanticscholar.org/CorpusID:10883999.

MacNeil, Stephen et al. (2023). "Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2023. Toronto ON, Canada: Association for Computing Machinery, pp. 931–937. ISBN: 9781450394314. DOI: 10.1145/3545945.3569785. URL: https://doi.org/10.1145/3545945.3569785.

Martin, Robert C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1st. USA: Prentice Hall Press. ISBN: 0134494164.

Pankiewicz, Maciej and Ryan S. Baker (2023). *Large Language Models (GPT) for automating feedback on programming assignments*. arXiv: 2307.00150 [cs.HC]. URL: https://arxiv.org/abs/2307.00150.

Potriasaeva, Anna et al. (June 2024). *Using a Low-Code Environment to Teach Programming in the Era of LLMs*.

Prather, James et al. (2023). "The Robots Are Here: Navigating the Generative AI Revolution in Computing Education". In: *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '23. Turku, Finland: Association for Computing Machinery, pp. 108–159. ISBN: 9798400704055. DOI: 10.1145/3623762.3633499. URL: https://doi.org/10.1145/3623762.3633499.

Reeves, Brent N. et al. (2024). *Prompts First, Finally*. arXiv: 2407.09231 [cs.CY]. URL: https://arxiv.org/abs/2407.09231.

Vaswani, Ashish et al. (2023). *Attention Is All You Need*. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

White, Jules et al. (2023). "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT". In: *ArXiv* abs/2302.11382. URL: https://api.semanticscholar.org/CorpusID:257079092.

Woodrow, Juliette, Ali Malik, and Chris Piech (2024). "AI Teaches the Art of Elegant Coding: Timely, Fair, and Helpful Style Feedback in a Global Course". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. event-place: Portland, OR, USA. New York, NY, USA: Association for Computing Machinery, pp. 1442–1448. ISBN: 9798400704239. DOI: 10.1145/3626252.3630773. URL: https://doi.org/10.1145/3626252.3630773.