Jinter: A Hint Generation System for Java Exercises

Jorge A. Gonçalves
jorgealexgoncalves@gmail.com
Instituto Universitário de Lisboa (ISCTE-IUL)
Lisboa, Portugal

André L. Santos
andre.santos@iscte-iul.pt
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR-IUL
Lisboa, Portugal

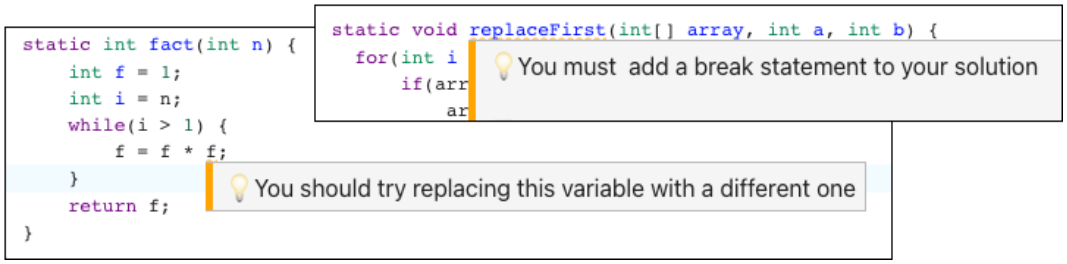

Figure 1: Jinter providing hints for progressing in programming exercises.

ABSTRACT

Programming novices often struggle when solving exercises, slowing down progress and causing a dependency on external aid such as a teacher, a more experienced person, or online resources. We present Jinter[1], a tool to generate hints to solve small exercises involving Java methods. The hints are produced taking into account the current state of an exercise and a backing model solution. The aid may refer to spotting errors or missing parts to achieve the desired outcome while taking into account behavioral equivalences of programming constructs (e.g., loop structures, forms of assignment, boolean expressions, etc). We evaluated the approach by surveying 8 programming instructors, finding that about two-thirds of the automated hints either match or are related to those given by instructors.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Applied computing** → **Computer-assisted instruction**.

KEYWORDS

introductory programming, scaffolding, hints, feedback

ACM Reference Format:
Jorge A. Gonçalves and André L. Santos. 2023. Jinter: A Hint Generation System for Java Exercises. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8–12, 2023, Turku, Finland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588820

[1]Prototype available for experimentation at https://jinter.org

ITiCSE 2023, July 8–12, 2023, Turku, Finland

ACM ISBN 979-8-4007-0138-2/23/07…$15.00
https://doi.org/10.1145/3587102.3588820

1 INTRODUCTION

Programming is an essential skill that all computer science students must master, as well as students from other fields, especially those related to STEM. In introductory programming courses, students exhibit various difficulties in syntactic knowledge, conceptual knowledge, and strategic knowledge [11]. Syntactic knowledge refers to the syntax of the programming language under usage. Conceptual knowledge relates to misconceptions of programming constructs or machine operation. Strategic knowledge of programming refers to expert-level knowledge about planning, writing, and debugging programs for solving novel problems using syntactic and conceptual knowledge.

Our work is focused on tool support for strategic knowledge, more concretely, aiding in the necessary steps to solve an introductory programming exercise. This aid consists of hints for helping learners to progress when they do not realize which steps are necessary to reach the desired goal, or when they do not understand what is causing the unexpected behavior (where are the bugs). According to the Cambridge Dictionary, a hint may refer to "a piece of advice that helps you to do something", or "something that you say or do that shows what you think or want, usually in a way that is not direct". This is the meaning that we aim for hints in the context of our work, as opposed to a tip with direct information such as "add statement i = 0". We aim to provide a learning aid that helps to reach a desired solution, but not in a prescriptive way through instructions that can be taken blindly without any thought. We envision a tool that mimics a human tutor that is not giving tips that spell out concrete code statements, but rather hints that require learners to think about how to proceed.

Automated hints are a central aspect in Intelligent Tutoring Systems (ITS), as they are a form of *scaffolding* [12]. Scaffolding refers to the learning support that is available to a student during the learning process, a core aspect of the 4C/ID model for complex learning [15]. The goal of scaffolding is to help students to acquire new skills and knowledge by breaking down the learning process into smaller, more manageable steps. Feedback on which aspects are correct is also a form of scaffolding, as it confirms that one is

on the right track, removing uncertainties and insecurities in the learning process. As a learner becomes more proficient, the amount of scaffolding provided can gradually reduce, allowing the student to be more secure about their learning.

This paper presents Jinter, a tool to generate hints to solve introductory exercises involving Java methods (see Figure 1). The usage scenario consists of a learner, in the context of a well-defined exercise (platform), requesting a hint to progress (next step) or to understand what can be improved (fix some part of the code). Jinter takes into account the current snapshot of the learner's code and computes a hint by contrasting it with reference solutions for the exercise being solved.

The analysis is performed using Behavior Trees (BTs), a more abstract representation than Abstract Syntax Trees (ASTs) that take into account behavioral equivalences among syntactical alternatives. The hints follow a priority directly related to the depth of the tree nodes where they are applicable. In addition, the similarities between the learner code and the reference solution are used to derive positive feedback that highlights which parts are correct. The more hints a learner requests, the more scaffolding a learner is having on a particular exercise. Hence, the same exercise given to different learners may have a wide range of scaffolding degrees, from none (no hints requested) to high (every step was assisted by a hint). Despite targeting Java, so far we did not approach object-oriented programming concepts such as polymorphism and inheritance, but instead, we focused on structured programming constructs.

Scaffolding is of utmost importance in an ITS, given that a human instructor is not available (at least in the short-run) to help learners to overcome difficulties. Nevertheless, even in traditional in-person or online classes with instructors, automated hints have the potential to make contact time more productive, especially in large classes where instructor availability for individual student assistance may be scarce. Even if a student is only able to take advantage of a fraction of the provided automated hints, that already alleviate the dependency on instructors, sparing some of their time to assist with other difficulties (possibly the more complex cases). Furthermore, unblocking students while they are autonomously doing homework may also contribute to more efficient usage of their time and alleviate frustration.

We evaluated the hints provided by Jinter by surveying 8 programming instructors, asking them to provide hints for 15 scenarios, organized into 3 sequences of steps to solve an exercise. Furthermore, participants had to contrast their hints with the automated hints, and further rate the appropriateness of the latter independently of their choice. Out of a total of 120 ratings, more than one-third (37%) of Jinter hints match those of instructors, and an additional third (33%) were considered similar.

These results are encouraging towards adopting a tool such as Jinter in teaching settings, which we have not done so far. Despite that the similarity with human instructor hints is satisfactory, it is necessary to carry out usability studies to investigate how well learners can understand and make use of the hint messages. Our contribution consists of demonstrating that it is possible to generate meaningful hints for small introductory programming problems involving Java methods.

2 RELATED WORK

The HINTS framework [8] describes techniques for the generation of programming hints. Our approach is situated in the category of tools where the learners' code is compared to existing solutions. We adopt the same definition of programming hint proposed in HINTS: *"... any type of feedback that improves a student's knowledge of how to complete a programming exercise. For example, it may help them to identify mistakes in their program, suggest potential ways to proceed, recommend concepts to revise or clarify the task requirements."* Feedback may have a major influence on learning processes, and is effective when it informs the learner about progress and how to proceed [4].

In the realm of programming, automated feedback may assume different forms [6, 10], such as expected result checking (unit test), source code quality, program efficiency, finding mistakes, or providing code hints – the focus of our work. Code hints may be provided by a human instructor, typically in a lab class. However, an instructor is not available in self-taught (online) or autonomous learning moments (homework). Hence, our approach aims at provisioning code hints at scale pertaining to strategic knowledge, and secondarily of syntactic and conceptual knowledge [11].

Keuning et. al, in their 2018 survey of automated feedback for programming exercises [6], concluded that very few tools provide feedback regarding knowledge on how to proceed (next step hints). AutoTeach [1] is one of those tools, supporting Eiffel, where exercise hints are predefined by teachers by annotating the reference solutions. Compared to our approach, Jinter hints are computed solely from the reference solution code — not requiring any other input. A more recent survey by Paiva et. al [10], focused on automated assessment, did not encounter tools that provide next-step hints and positive feedback in the style of our approach.

Knowledge about metacognition is a different type of feedback where a learner is evaluated concerning knowing *why* certain options were followed. Recent approaches have attempted to achieve this by posing questions about learners' code [7], fostering reflection over the written code, either in a just-in-time fashion when a potential misconception is detected [5] or in a post-submission to assess how well the learner understands the solution and the applied programming constructs [13].

Zimmerman and Rupakheti [17] developed a Java framework to recommend specific code edits relevant to students' problems when they are trying to solve a specific programming exercise. The framework offers recommendations based on the best match between the student's solution and the teacher's suggested solution. This work is the closest to our approach in terms of how we compute the hints, namely through computing tree edit distances to reference solutions. Their contribution focuses on the technical feasibility of computing source code edits, while no efforts on elaborating hints and evaluating scenarios were carried out.

CATNIP [3] is a hint generation tool for Scratch programming. Our approach, albeit targeting Java, shares the same strategy of matching learners' reference solutions by comparing their structure at the AST level. They have found that about half of the generated hints are considered useful, a result that roughly aligns with our results (about half of the Jinter hints were rated as very good, albeit in a different context).

3 JINTER

Jinter is a tool that materializes our approach in the realm of Java, though most of the ideas described here could apply to other languages with structured programming constructs. We implemented a Web-based service as a REST API for generating the hints and a Web browser client to demonstrate the approach (though other clients are possible, such as an IDE). This section describes the aspects about tool usage and user experience, illustrated with screenshots of the Web browser client.

3.1 User interface

Figure 2 illustrates the code editor while a user is writing the code to compute the maximum value in an array of integers, and is presented with a hint concerning the declaration of variables. The code marks holding the hints are inserted once the user has made an explicit request. When mouse-hovering the marks, the user sees a popup widget with the description of the hint, consisting of a small sentence that describes the hint, and further information that explains a relevant associated concept and provides an example (not visible in the figure due to space constraints).

The usage flow of Jinter is straightforward. Once stuck in the process of solving an exercise, a learner may request a hint. The current state of the code (snapshot) is sent along with the request. Having the code in a parseable state is a prerequisite for using the system. By parseable we mean without having lexical or syntax errors, but it may have semantic errors (e.g., type mismatches, missing returns, duplicate identifiers, etc).

We considered that a convenient way to present hints is to present them one at a time, prioritized according to their relevance. On the one hand, too many marks in the code may lead to excessive noise. On the other hand, we believe it would not make sense to provide a hint that requires elements that are not yet present. For instance, a hint related to an instruction to include in the method body would not make sense if the expected parameters are not yet declared (hence, the hint of declaring parameters has higher priority).

For the same snapshot, the system always outputs the same hint for improvement (if there are any). Once a learner modifies the code, either following a provided hint or not, a new request will lead to another hint based on the latest snapshot. Finally, if the learner feels no need to request more hints, he or she may submit the code and they will be checked against test cases. At this point, the learner may have a solution that works only for some cases and may request more hints that target missing or incorrect parts.

3.1.1 Next-step hints. Next-step hints are those that suggest some increment in the code (as in Figure 2). For instance, hints suggest (a) adding parameter or variable declarations, (b) including a return statement, or (c) declaring a loop. These hints are computed by checking the missing parts against the reference solution. Jinter prioritizes elements according to a depth-first traversal of the reference solution AST. In this way, hints concerning the return type and parameters of a method are suggested before the elements in the method body. In turn, the missing statements in the body are suggested according to their sequence in the reference solution. For example, if a return statement requires a variable, the hint to declare the latter will be suggested first.

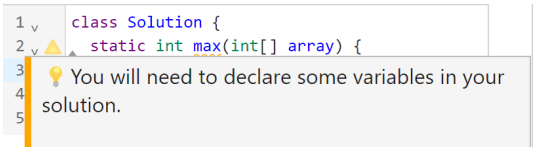

Figure 2: Jinter: next-step hint for declaring variables.

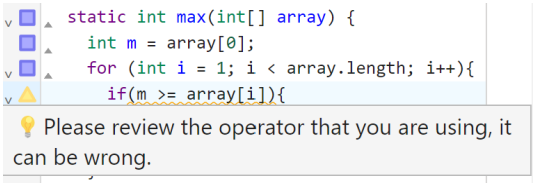

Figure 3: Jinter: fix hint for correcting operator.

3.1.2 Fix hints. Fix hints are those that suggest modifying or removing some part of the code (see Figure 3). For instance, hints suggest: (a) removing an extraneous parameter, (b) using a different expression for initializing a variable, or (c) using a different operator in a binary expression. These hints are computed by looking for differences in parts that partially match with the reference solution in terms of element types, but whose properties are different. In the example of Figure 3, there was a match in the if statement guards of both the code snapshot and the reference solution, both of which consisted of binary expressions, but with a different operator.

3.1.3 Positive feedback. Together with the improvement hints, Jinter also provides a list of positive feedback hints that emphasize which elements are correct (see Figure 4). During the comparison of learner code with the reference solution, the matching code elements will form the list of positive feedback. This list may be obtained even if the code is not executable due to semantic errors. The positive feedback is useful to reinforce to the learner that a viable path is being followed, and it was demonstrated to have a significant impact on learning in an ITS [9].

3.2 Behavioral equivalence

When checking the learner code against a reference solution, Jinter works with a representation that abstracts several syntactical aspects and identifiers of the learner and reference solution. This allows distinct exercise solutions, which may look substantially different on the surface, to be considered equivalent when computing the hints. Figure 5 illustrates this with two equivalent snippets that use different loop structures (while, for) and expression statements, yet, their behavior is the same.

Loop structures are treated as a uniform concept, and there is no distinct notion for a single statement or block statement in control structure bodies (these are all treated as a sequence of statements). Variable assignments are desugared so that incrementation/decrementation and compound assignment operators are normalized into a canonical form. Concerning relational and arithmetic expressions, we use specific comparators for taking operator semantics into account (see Table 1). These aspects are also illustrated in the examples of Figure 5.

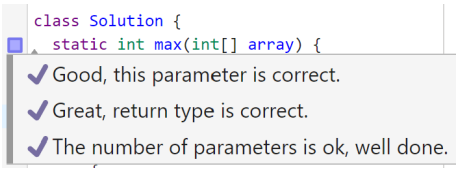

Figure 4: Jinter: positive feedback hints.

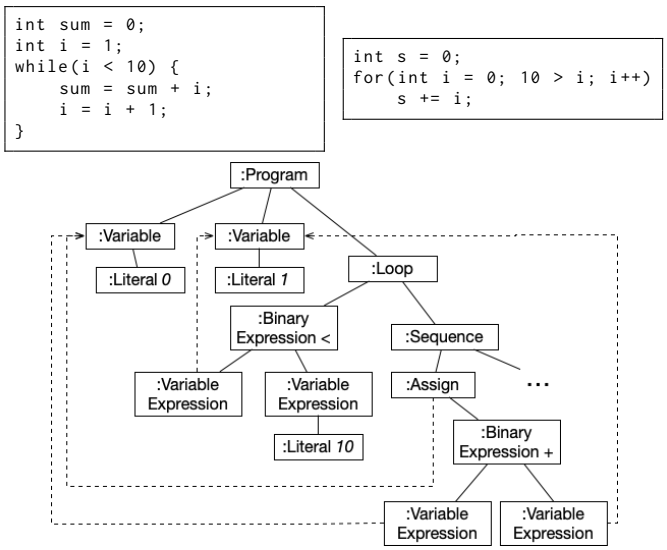

Figure 5: Two behaviorally equivalent loops for summing natural numbers and a common Behavior Tree that represents their behavior (when a normalisation of expressions is applied, see Table 1).

$a \geq b$	$b \leq a$	$\neg(a < b)$	$\neg(b > a)$
$a \leq b$	$b \geq a$	$\neg(a > b)$	$\neg(b < a)$
$a > b$	$b < a$	$\neg(a \leq b)$	$\neg(b \geq a)$
$a < b$	$b > a$	$\neg(a \geq b)$	$\neg(b \leq a)$
$a = b$	$b = a$	$\neg(a \neq b)$	$\neg(b \neq a)$
$a \neq b$	$b \neq a$	$\neg(a = b)$	$\neg(b = a)$
$a + b$	$b + a$		
$a * b$	$b * a$		
a++	a = a + 1	a = 1 + a	a += 1
a−−	a = a - 1	a -= 1	

Table 1: Semantic equivalence in expressions.

4 IMPLEMENTATION

As mentioned, our approach relies on comparing a snapshot of the learner code with reference solutions. For each one of these, we obtain an Abstract Syntax Tree (AST)[2] and further derive what we refer to as a Behavior Tree (BT), which allows for additional syntactic aspects to be abstracted and achieve the behavioral equivalence discussed in the previous section.

[2]for this purpose we use JavaParser (https://javaparser.org/), a popular open-source library for reengineering Java code

4.1 Behavior Trees (BT)

We use the notion of a Behavior Tree (BT) for representing a program. A BT is similar to an AST, but with fewer and more abstract concepts. The description provided by a BT focuses on the specification of behavior (content), rather than on syntax (form). An AST does abstract extraneous syntactic elements, such as curly braces and semicolons, but has distinct representations for equivalent constructs (e.g., while vs. for loop, compound assignments, incrementors, etc). Furthermore, a BT also discards identifiers (which are a matter of form), allowing declarations (such as variables) to have an identity and to be referenced from expressions. In a BT there is a single node type for each one of the elementary structured programming concepts. A BT comprises the control-flow concepts of sequence, selection, and repetition (loop), as well as the statement concepts of assignment, return, call, break, and continue.

Figure 5 presents a common BT for the two code snippets. We can see a tree overlay that links variable expressions to the variable declarations (dashed arrows).

4.2 Tree Edit Distance (TED)

Tree Edit Distance (TED) refers to the minimum number of node insertions, node deletions, and node reclassifications required to transform a given tree into a desired target tree. We compute the Tree Edit Distance between the BTs of the learner code and the reference solution using the Zhang-Shasha algorithm [16]. If several reference solutions with different ways of solving the exercise are available, the TED is used to select the one that is closer to the learner code and use it to derive the hints.

A TED between two BTs computes a matrix with the edit distances between all the subtrees, analogously to the Levenshtein string distance. The matrix values are used to match the nodes of the learner code BT with nodes of the reference solution BT, based on tree similarity (lower edit distances). As a precondition for the match, the root nodes have to be of the same type (e.g., loop, variable declaration), although they may have different properties. Figure 6 illustrates a matching of BTs, where grey nodes mark the matched nodes. On the left-hand side we can see a learner code snippet with an iteration skeleton and the respective BT, whereas, on the right-hand side, we have the BT of the previous example given in Figure 5.

The calculation of the edit cost directly relates to the number of node elements that differ. Variable declarations are a special case, where not only their properties are considered (type and initialization), but also their role in the program [2] by analyzing dependent statements using a similar technique as in [14]. This is relevant, because two variable declarations may have the same properties (type and initialization), but distinct roles in the program. Take the variable declarations of `sum` and `i` of the first code snippet of Figure 5 as an example. Although they have the same properties, the former has a Gatherer role [2] whereas the latter has a Stepper role [2].

Discerning variable roles allows improved matchings, as we illustrate in the case of Figure 6. Notice that the variable `n` matches the iteration variable of the reference solution, not the Gatherer one that accumulates the summation (despite that the latter is declared first). This match allows Jinter to suggest having a Gatherer

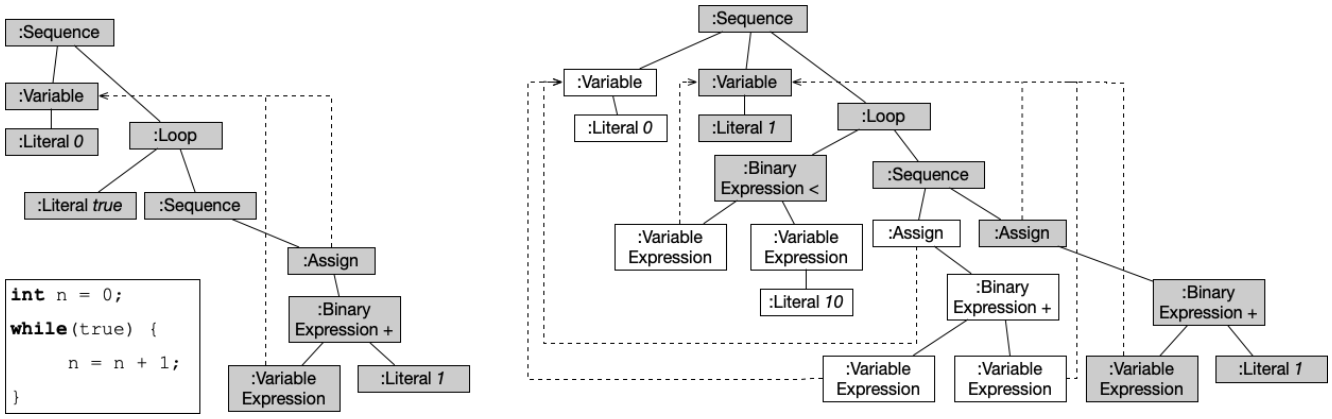

Figure 6: Matching of Behavior Trees: snippet in the left hand side matching the program of Figure 5 (right hand side).

variable since the variable for iteration (Stepper) is already present in the learner code. Follow-up hints would consist of modifying the initialization of the Stepper and the loop guard.

5 EVALUATION

Jinter was not yet used in teaching settings. As a first evaluation of our approach, we aimed at assessing the appropriateness of the hints provided by Jinter through comparison to those of experienced instructors. To achieve this, we asked programming instructors to fill in a questionnaire where they evaluated hints generated by Jinter and contrasted them with their own. The positive feedback items provided by Jinter were out of the scope of this evaluation. We focus on the following research questions:

RQ1. How do Jinter hints compare to those given by instructors?

RQ2. How do instructors rate the appropriateness of Jinter hints?

5.1 Methods

We recruited programming instructors that are currently working at our institution through direct invitation. The selection criteria were that the instructor had been involved in the lab classes of at least two offerings of the introductory programming course, while not being an author of this paper.

We handed in an anonymous online questionnaire in English and requested the answers to be in English as well. We estimated that the questionnaire would take about 30 minutes to complete. The questionnaire was answered by participants without any supervision or assistance, and was divided into three sections, each targeting one programming exercise. Each section consisted of a Jinter usage scenario description comprising:

(1) a reference Java solution to the exercise
(2) a sequence of 5 incremental Java code snapshots that are incomplete and/or incorrect towards solving (1)
(3) the generated Jinter hint for each code snapshot of (2)

We chose classic introductory programming exercises for the study, which are familiar to all instructors, namely:

(1) factorial (non-recursive)
(2) summation of an integer array
(3) replacing the first occurrence of an integer in an array

Each sequence starts with an empty method, possibly with an incorrect signature, and further stages progressively more closely to the reference solutions. The evolution of the sequence was artificially created, introducing errors that we frequently observe in our lab classes, so that a diverse set of scenarios could be evaluated, broadly covering the sorts of hints Jinter provides. We also used the survey to collect points of improvement to Jinter hints.

5.1.1 Instructions for participants. The following was the description of the task given in the questionnaire:

> *Your task is to describe a short hint (one sentence) that you find the most pertinent for progressing or correcting the code. Please consider that a hint is not a direct prescription of the next step, but rather an indirect clue of what might be missing or is incorrect. Please do not write straight tips in the form of "what to do next" or "change X to Y".*
>
> *After describing your hint you will be presented with an automated hint for the same scenario, for which you will be asked to (a) rate the similarity of the automated hint with yours (1 - 3 scale); (b) rate the appropriateness of the automated hint (independently of yours) based on your experience as a programming instructor (1 - 5 scale); and (c), explain how the hint could be improved (optional). Please do not edit your hints after seeing the respective automated hint.*

5.1.2 Threats to validity. The study participants are all instructors at the same institution. However, they have different backgrounds (CS, Electrical Engineering, Math), ages, and years of experience. Nevertheless, recruiting instructors from different institutions and countries would likely widen the range of perspectives.

The sequence of code snapshots for each exercise was artificially created, which raises the issue of the scenarios being too unrealistic. Collecting scenarios where actual learners request hints would strengthen the validity of the experiment. We believe that the most significant threat in this respect has to do with the artificially created incorrect parts, not the incomplete code snapshots. However, based on our teaching experience, we are confident that the cases of incorrect code which we considered are frequent in practice.

5.2 Results

A total of 8 programming instructors have accepted to participate in the study (6 men, and 2 women). Hence, we collected a total of 120 hint ratings (15 from each participant).

5.2.1 RQ1. Instructor/automated hints matching. For each hint scenario, participants were asked to rate the similarity of their hint with the one provided by Jinter. Figure 7 presents a stacked column chart with the percentage of matches grouped by the stage in the sequence of code snapshots (one to five). We obtained a global match of 37% between instructors and Jinter, with an additional 33% of cases where participants considered the hint to be related to theirs.

There is an apparent tendency for higher matching when the code snapshots approximate the reference solution. Recall that the higher the stages are closer to the reference solution. This tendency is not surprising given that code snapshots that are more completed will necessarily narrow down the scope for filling gaps through hints.

5.2.2 RQ2. Jinter hint rating. For each hint scenario, participants were asked to rate the appropriateness of the Jinter hints. Figure 8 presents a stacked column chart with the distribution of ratings, also grouped by the stage in the sequence of code snapshots (one to five). We can observe that almost at every stage, at least half of the hints were rated with the highest score. Furthermore, three-quarters of all the hints were rated with the first (very good) or second highest score (good), implying that instructors consider a great majority of the hints to be appropriate.

The hints with the lowest scores (neutral and below) are more prevalent in the first stages of the scenarios. We investigated these lower-rated hints through the textual explanations given by participants. We summarise two illustrative cases. In exercise 1, the first code snapshot consisted of a correct method signature and an empty body, and the Jinter hint was analogous to the one of Figure 2. Participants mentioned that a first hint should focus on using repetition (loop), not on declaring a variable (with no additional information). In exercise 2, the first code snapshot had an integer parameter (int), instead of a reference to an integer array (int[]), and the Jinter hint consisted of "the solution is not expecting a value type for this parameter". Participants considered the hint too indirect and involving a concept that beginners often do not master (distinguishing value and reference type).

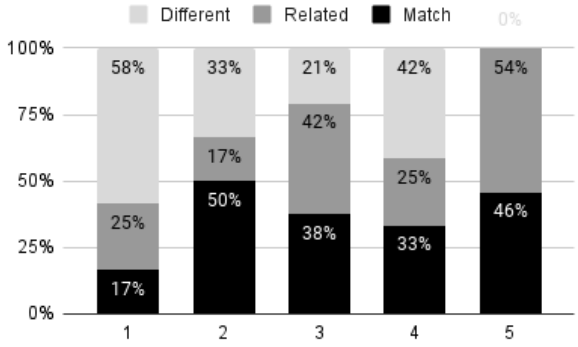

Figure 7: Instructor and Jinter hints match by sequence stage (1: very incomplete snapshot, 5: almost complete).

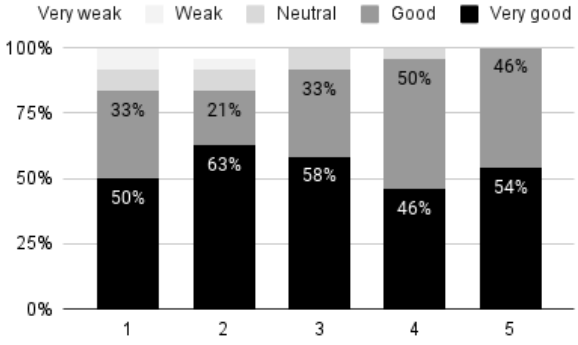

Figure 8: Instructor rating of Jinter hints by sequence stage (1: very incomplete snapshot, 5: almost complete).

6 CONCLUSIONS AND FUTURE WORK

Jinter stands as a proof of concept for hint generation for introductory Java exercises. We conclude that it is possible to generate meaningful hints, given that the majority of those are rated positively by programming instructors. Moreover, it is worth noting the considerable match of instructor and Jinter hints (over one-third of the total). However, the evaluation indicates that the initial hints, both concerning method signatures and empty bodies can be improved.

A limitation of our approach is when the learner code deviates from the available reference solutions while pursuing a valid path. In these situations, Jinter continues to provide hints, which may go against the learners' path. A point of improvement would be to have a form of deciding when not to provide hints if the learner code deviates considerably from the reference solutions. The analysis for deriving the hints could also benefit from more behavioral equivalences to those we currently support. For instance, concerning the cases where a break or a return statement are equivalent (applicable in our evaluation scenario 3), or inlining temporary variables that are merely used to break down statements.

To assess the usefulness of the Jinter hints one would need to carry out a user study with actual novice programming learners. Such a study would allow us to work with real scenarios of hint requests, and more importantly, to evaluate how learners can make use of the provided messages and positive feedback.

ACKNOWLEDGMENTS

We thank the anonymous programming instructors that accepted to participate in our study. This work was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020].

REFERENCES

[1] Paolo Antonucci, Christian Estler, Durica Nikolić, Marco Piccioni, and Bertrand Meyer. 2015. An Incremental Hint System For Automated Programming Assignments. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (Vilnius, Lithuania) *(ITiCSE '15)*. Association for Computing Machinery, New York, NY, USA, 320–325. https://doi.org/10.1145/2729094.2742607

[2] Pauli Byckling and Jorma Sajaniemi. 2007. A Study on Applying Roles of Variables in Introductory Programming. In *2007 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), 23-27 September 2007, Coeur d'Alene, Idaho, USA*. 61–68. https://doi.org/10.1109/VLHCC.2007.31

[3] Benedikt Fein, Florian Obermüller, and Gordon Fraser. 2022. CATNIP: An Automated Hint Generation Tool for Scratch. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 124–130. https://doi.org/10.1145/3502718.3524820

[4] John Hattie and Helen Timperley. 2007. The Power of Feedback. *Review of Educational Research* 77, 1 (2007), 81–112. https://doi.org/10.3102/003465430298487 arXiv:https://doi.org/10.3102/003465430298487

[5] Austin Z. Henley, Julian Ball, Benjamin Klein, Aiden Rutter, and Dylan Lee. 2021. An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 165–170. https://doi.org/10.1109/ICSE-SEET52601.2021.00026

[6] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (Sept. 2018), 43 pages. https://doi.org/10.1145/3231711

[7] Teemu Lehtinen, André L. Santos, and Juha Sorva. 2021. Let's Ask Students About Their Programs, Automatically. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 467–475. https://doi.org/10.1109/ICPC52881.2021.00054

[8] Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A Survey of Automated Programming Hint Generation: The HINTS Framework. *ACM Comput. Surv.* 54, 8, Article 172 (oct 2021), 27 pages. https://doi.org/10.1145/3469885

[9] Antonija Mitrovic, Stellan Ohlsson, and Devon K. Barrow. 2013. The effect of positive feedback in a constraint-based intelligent tutoring system. *Computers & Education* 60, 1 (2013), 264–272. https://doi.org/10.1016/j.compedu.2012.07.002

[10] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (jun 2022), 40 pages. https://doi.org/10.1145/3513140

[11] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

[12] Brian J. Reiser and Iris Tabak. 2014. *Scaffolding*. Cambridge University Press, United Kingdom, 44–62. https://doi.org/10.1017/CBO9781139519526.005 Publisher Copyright: © Cambridge University Press 2006, 2014..

[13] André Santos, Tiago Soares, Nuno Garrido, and Teemu Lehtinen. 2022. Jask: Generation of Questions About Learners' Code in Java. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 117–123. https://doi.org/10.1145/3502718.3524761

[14] André L. Santos and Hugo S. Sousa. 2017. PandionJ: a pedagogical debugger featuring illustrations of variable tracing and look-ahead. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research, Koli, Finland, November 16-19, 2017*. 195–196. https://doi.org/10.1145/3141880.3141911

[15] Jeroen J. G. van Merriënboer, Richard E. Clark, and Marcel B. M. de Croock. 2002. Blueprints for complex learning: The 4C/ID-model. *Educational Technology Research and Development* 50, 2 (2002), 39–61. https://doi.org/10.1007/BF02504993

[16] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262. https://doi.org/10.1137/0218082 arXiv:https://doi.org/10.1137/0218082

[17] Kurtis Zimmerman and Chandan R. Rupakheti. 2015. An Automated Framework for Recommending Program Elements to Novices (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 283–288. https://doi.org/10.1109/ASE.2015.54