An Incremental Hint System For Automated Programming Assignments

Paolo Antonucci
ETH Zurich
paolanto17@gmail.com

Christian Estler
ETH Zurich
christian.estler@inf.ethz.ch

Đurica Nikolić
ETH Zurich
durica.nikolic@inf.ethz.ch

Marco Piccioni
ETH Zurich
marco.piccioni@inf.ethz.ch

Bertrand Meyer*
ETH Zurich
bertrand.meyer@inf.ethz.ch

*Also Politecnico di Milano, Italy, and Innopolis University, Kazan, Russia

ABSTRACT

The advent of Massive Open Online Courses makes it essential to develop tools and techniques that automatically support computer science students in solving programming assignments. Complementing existing tools for automatically checking the correctness of students' programs, we have developed and evaluated an *incremental hint system* for programming exercises. The hint system displays, upon request from a student, a series of hints on how to approach a solution. The hints are created in advance from the source code of the exercise's reference solution using our hint generation tool. This tool can run in fully automatic mode, where hints reveal more and more parts of the solution code; in manual mode, where teachers can customize hints by annotating the input source code; and in a combination of the two modes. We evaluated the hint system throughout our Introduction to Programming course which provides a companion online course. The findings suggest that students who needed assistance with an exercise used the hint system and found it helpful to guide them through the process of building a solution.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

Keywords

CS1, MOOC, Pedagogy, SPOC

1. INTRODUCTION

A big hurdle for beginner students attempting to solve a programming exercise is to come up with an initial, albeit not perfect, solution. Once such a initial solution is found, the process of transforming it into a fully correct solution can be tackled with skill and some patience: refining the initial solution through a series of compilation, execution, and testing steps until the desired result is achieved. But what can a student do when unable to find such an initial solution? Asking peers or a teacher is an obvious thing to try, but sometimes this is not possible or practical, as for example in the context of online courses and MOOCs.

In the Autumn semester 2014 we offered a companion online course to our Introduction to Programming residential course, in which we used and evaluated an incremental hint system for programming exercises. The hint system displays, upon request from the student, a series of hints, incrementally unveiling new suggestions and details of the expected solution. The hint system also aims at supporting self-study and minimizing the need to ask for assistance to a teacher or to peers in a forum. A series of hints for each given exercise is created in advance by the teacher, using our tool either in automatic or manual mode (or a combination of both). The automatic mode obeys to some predefined (customizable) rules and generates all the needed files completely automatically. However, teachers can always tweak and personalize the hints for any given exercise by using an ad hoc mini-language embedded in the exercise's code comments. The tool was evaluated by making it available to the students, logging and observing how it was used, and administering a final questionnaire. The results are, in general, encouraging, and show that the students a) used the hint system and b) found it to be helpful.

The rest of this paper is organized as follows: Section 2 provides some background on our residential Introduction to Programming course and its structure; Section 3 describes the infrastructure we used for our online course; Section 4 describes the hint system in detail; Section 5 summarizes the data we collected; Section 6 details related work; and Section 7 concludes with final comments.

2. OUR INTRODUCTION TO PROGRAMMING COURSE

Our Introduction to Programming course is a fourteen week long residential course. It includes four hours of frontal lectures and two hours of exercise sessions each week. The live lectures are of two kinds: traditional (frontal) lectures, and Socratic lectures. There are ten home assignments and two mock exams that simulate the final exam setting. Exercise groups typically include twenty students each and are differentiated by skill level (beginners, intermediate, or advanced) and language (English or German). Differentiating

ITICSE'15, July 04–08, 2015, Vilnius, Lithuania.

ACM ACM 978-1-4503-3440-2/15/07 ...$15.00
http://dx.doi.org/10.1145/2729094.2742607.

the exercise groups according to students' self-assessed skill levels proved to be beneficial, as it lead to more homogeneous groups, which helped to keep students of different skill levels interested and motivated.

Throughout the rest of this paper we refer to the Introduction to Programming course we taught in the Autumn semester of 2014. Its students reported, similarly to previous years, the following distribution of preexisting programming experience: 64% of students came with some object-oriented programming experience, 25% had non-object-oriented programming experience, and only 11% had never programmed before. Additionally, 19% of our students attended the residential course while having a part-time job that required some programming.

The programming language of choice for our course was Eiffel [6]. Eiffel makes it easy to express and teach object-oriented concepts and methods, and allows for a gentle introduction to program verification through design by contract [5].

3. OUR ONLINE LEARNING PLATFORM

The residential course described in Section 2 was accompanied by an online course, which provided video lectures, quizzes and programming exercises. The infrastructure we used for the online course consisted of three parts: a Moodle[1] installation, enhanced with a plugin we developed for in-video quizzes, a web-based programming environment, Codeboard[2], for compiling and executing programs in the browser, and our incremental hint system.

The online course sequence is linear and consists of 14 lectures. Every online lecture is divided into one or more segments (of variable duration), one or more quizzes, and zero or more programming exercises. Topics' durations vary between 5 and 40 minutes, with an average of 17 minutes.

To tackle the known issues deriving from reduced attention span [7], we embedded quizzes within longer video lecture segments. The purpose of the quizzes is twofold: to allow attention span recovery by breaking the online lecture flow, and to test short-term topic comprehension. In general, we designed quizzes and exercises to be useful in the short term (after taking an online lecture) and in the long term (for reviewing material for the exam, that in our case takes place 8 months after the course end).

Our web-based programming environment, Codeboard, allows students to write, compile, test, and submit programs from within the browser. Submissions are automatically graded based on a set of unit tests provided by the teacher. Using this web-based tool removes the need to install any ad hoc software for the students and simplifies the distribution and grading of exercises for the teacher. Codeboard integrates seamlessly with Moodle (and other e-learning platforms as well), making it possible to exchange data about students' progress and grades between the two systems.

4. THE INCREMENTAL HINT SYSTEM

When solving in-browser programming exercises students have the option to request hints by clicking "hint" buttons. For each exercise there can be one or more hint buttons. At the beginning only the first button (corresponding to a level 1 hint) is active. Each other button becomes available after pushing the previous one. This section describes the hint system in detail.

4.1 Hint mechanism

AutoTeach [3], our hint generator, provides hints directly from within the source code. Hints are incremental, that is, they are organized in *hint levels*, each level containing more hints than the previous one. In practice, this means that for every exercise to be solved, which in most cases consists of a single class file, there are n versions of the class file that are generated in advance and served to students upon request, with the first file containing just the skeleton of the code (feature declarations, possibly routine contracts, etc.) and the following ones containing incrementally more information.

Hints are of two kinds:

- **Textual hints:** comments in the code manually written by the teacher which become visible from a specific hint level on.
- **Code-revealing hints:** a part of the solution the student is expected to implement. They are generated automatically by AutoTeach according to certain customizable rules.

These two kinds of hints can be combined at will for greater flexibility, allowing teachers, for example, to refer in a textual hint to a part of the code which they know it will become visible on the next hint level.[3]

4.2 Meta-commands and textual hints

Teachers write textual hints directly within the code by taking the solution files, annotating them with special processing directives called **meta-commands**, and passing them to AutoTeach. AutoTeach will then scan the file multiple times, once for each hint level, process the meta-commands, and generate the output files.

Meta-commands have the form of special comments within the code. They can either contain a textual hint, which should be printed to the output, or alter AutoTeach's default behavior in processing the code. In both cases, they may specify a range of hint levels, outside of which they are ignored. As an example:

```
--# [3] HINT: Start by iterating on 'a_numbers'
--# [0-4] HIDE_NEXT if
```

The first meta-command is a textual hint which must be printed to the output at hint level three and higher. The second meta-command shows one of the many supported processing directives, and indicates that between hint levels 0 and 4, the subsequent 'if' instruction should be hidden from the output.

4.3 Code processing

Although until now we have referred to "code-revealing hints", we have in fact developed a multi-level code processing model which determines the visibility of every part of the code. This model fits particularly well our purpose of generating hints based on revealing parts of the code, but nothing prevents it from being used for more general code processing tasks that need to distinguish between the visibility of different sections of the code.

[1] `https://moodle.org`, accessed April 22, 2015

[2] `https://codeboard.io`, accessed April 22, 2015

[3] For a code example, see `http://goo.gl/YBVQpp`, accessed April 22, 2015

The main design goal of this code visibility model and the related code processing system is to be as flexible and powerful as possible, and at the same time simple enough to be used in those cases where no particular customization is required. Ideally, we want our Hint Generator to be able to run on most exercises *without any kind of extra annotation* and yield a satisfactory result, which could be further refined with additional annotations only in cases where a high granularity of hints is important (mostly very short exercises).

4.4 Code blocks

An important choice in designing the code visibility model is the granularity, that is, the elementary units which the Hint Generator should be able to handle. With respect to this, we define a list of supported code syntactic elements which we call "blocks". Blocks are classified as **atomic blocks** and **complex blocks**. The difference between the two is that complex blocks may contain other blocks (either atomic or complex) nested in them, while atomic blocks cannot contain any other kind of block. Examples of *atomic* blocks are instructions (excluding compound statements such as 'if's and loops), assertions in contracts, and conditions within 'if' statements. Examples of *complex* blocks are routine preconditions, 'if' statements, and 'if' branches. Figure 1 shows the decomposition of a code sample into blocks.

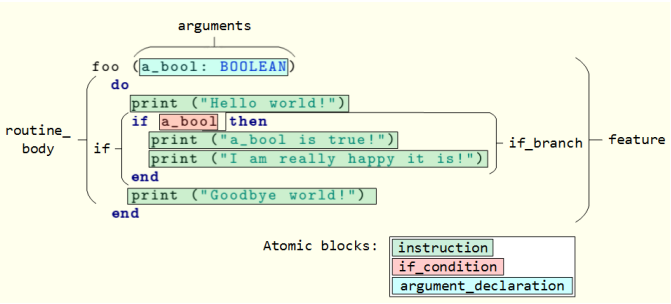

Figure 1: Decomposition of code into blocks.

4.5 Block basic visibility

AutoTeach works by scanning class files sequentially, writing their content to the output files on the fly and replacing the sections of the source code that should be hidden at the selected hint level with a placeholder. As AutoTeach abstracts the input code into blocks, the ultimate question for every occurrence of a code block is for us: "should we print this block or should we hide it?".

The question is answered by a lookup in the **basic visibility table** (see Table 1). The basic visibility table is a table associating block types (rows) with hint levels (columns). Every cell of the table contains a boolean value indicating whether or not the corresponding block type should be visible, and thus be printed to the output, at that level. Cells can also be undefined (allowing for three possible values: *true*, *false*, and *undefined*), but we will ignore this for now.

4.6 Block content visibility

The approach shown in the previous section is very simple and quite flexible, yet not flexible enough. For example, consider the code in Figure 1. Assume that, on a certain hint level, the teacher wants to show all the instructions appearing directly within the body of a routine, but wants to hide all those inside the body of an *if* statement, having therefore the output shown in Listing 1.

Table 1: Simplified basic visibility table. Rows correspond to block types, columns to hint levels.

	0	**1**	**2**
feature	True	True	True
if	False	True	True
instruction	False	False	True

Listing 1: Desired output processing the code of Figure 1.

```
foo (a_bool: BOOLEAN)
  do
    print ("Hello world!")
    if a_bool then

      -- Your code here!

    end
    print ("Goodbye world!")
  end
```

According to what we have said so far, there is no way to achieve this. Working with the visibility of *instruction* blocks will not help, as it will affect both instructions inside and outside the 'if' statement. To make this possible, we need a new paradigm, which we call **content visibility**. This paradigm is orthogonal to and independent from the paradigm of basic visibility.

Every point in the code, where "point" means any place in the text on which we could click and set the cursor for editing, has a **content visibility policy** in force in that place. The content visibility policy is a boolean value which, as an approximate definition, indicates whether or not the code appearing in that region should be visible (i.e. printed to the output). The policy may also be undefined, which effectively allows for three different values for the content visibility policy (*true*, *false*, and *undefined*). In our case, to achieve the desired goal, we need to ensure that the content visibility policy be *false* within the body of the *if* instruction and *true* outside of it.

The distinction between atomic and complex blocks, which we introduced in section 4.4, becomes relevant here, as complex blocks can specify a **content visibility policy**, which is valid within their body. The content visibility policy for complex block types is defined in the **content visibility table**. This table is totally analogous to the basic visibility table: every row represents a complex block type and every column a hint level. The value in every cell (which may be undefined) indicates the content visibility policy for that block type on that hint level. Complex blocks for which the policy is *undefined* inherit the content visibility of the parent block, if any.

We can now be more precise and state that the effective visibility of a code block is affected by the content visibility policy applicable at its position in the following way:

- **Atomic blocks**: their visibility is defined by the content visibility policy in force at their location. If the content visibility policy at their location is *true*, they will be printed to the output, if it is *false* they will be hidden.
- **Complex blocks**: **no effect**. Complex blocks are immune to the content visibility policy in force at their location. This design choice, which might be perplexing, is motivated by the thought that complex blocks build up the "boning" of the structure of the code. Setting the content visibility to *false* indicates the wish of hiding the "flesh" of the code, not the boning.

By now it should be clear that in our example we can obtain the output of Listing 1 by using the content visibility table shown in Table 2.

Table 2: Simplified example of a content visibility table. Rows correspond to block types, columns to hint levels.

	0	**1**	**2**
routine_body	False	True	True
if	False	False	True

4.7 Joining the concepts: hint tables

In the previous sections we have introduced the two orthogonal concepts of blocks' **basic visibility** and **content visibility**. In both cases, we left open the question of what happens if the visibility of a block is undefined. We can now join the two concepts and provide the final visibility rules for code blocks, which will answer this question.

- The visibility of **complex blocks** is simply equal to their basic visibility. However, they have the ability of *specifying* a content visibility policy valid within their body.
- The visibility of **atomic blocks** is equal to their basic visibility, *if it is defined*. If it is undefined, then their effective visibility *is equal to the content visibility in force at their location*, which in this case has to be defined (otherwise the table is malformed).
- In addition to the previous rule, no block can be printed if its parent block is not visible. This is in order to avoid having "detached" blocks in the output, which would be confusing to students and possibly break the syntactic validity of the generated output.

Clearly, these rules require the two visibility tables (basic and content) to be harmoniously used together, which is only possible if they are thought from the beginning as the two sides of the same coin. The combination of both tables takes the name of **hint table**.

In addition to these rules, AutoTeach supports overriding the processing policies defined in hint tables through the use of specific meta-commands. These make it possible, for example, to force all argument declarations to be always visible throughout a file, or a specific crucial instruction to remain hidden until a higher hint level than the one defined in the table is reached. The teacher can even decide to load a blank (all *undefined*) hint table and exclusively rely on visibility meta-commands to define how an exercise should be processed. This is thoroughly discussed in [3].

4.8 Flexibility of the mechanism

AutoTeach ships with a default hint table, for which we tried to find good general-purpose values which work well for the majority of exercises without requiring additional intervention. The table starts by only showing the bare feature[4] declarations at level one, and then incrementally revealing routine arguments, contracts, and local variable declarations. The *existence* of complex instructions is then revealed, without their content, so that students get an idea of how the code is structured (for example, how many nested loops are needed). On the following step, some more information about complex instructions is revealed (the condition of 'if' instructions and the termination condition of loops). At this point, basic instructions are not visible yet. Going up one more level will then reveal all the instructions which are not contained in any complex instruction, producing an output similar to Listing 1. At this point the student sees almost everything, but still can't see the instructions within 'if' statements and loops, which are often the core of the solution of many programming tasks. Only at the highest hint level all instructions are eventually shown. For our course we decided to never show the whole solution, to avoid the scenario in which a students just unveils all the hints and then copy-pastes the solution without too much thinking.

In our experience, this table works remarkably well for most medium-sized exercises, providing good results without the need of even touching the source code. It is not unrealistic to imagine of using AutoTeach in a setting where teachers writing the exercises have no knowledge at all of the hint mechanism, and where batches of existing exercises need to be processed in this fully automated way, which would not have been possible if the hint system had only consisted of manual annotations defining the visibility of single instances of blocks.

AutoTeach also comes with the ability of loading alternative custom hint tables. This enables teacher to define a custom processing policy which may be completely different from our approach. The flexibility of the hint table model give teachers ample freedom.

Finally, there are cases where greater accuracy is required and where teachers may want to define the visibility of *single instances* of blocks. This is especially true for very short exercises, where students are only required to write a handful of lines of code and where showing a crucial instruction too early can spoil the solution. For these exercises, and wherever else necessary, teachers can use manual visibility annotations, slightly tweaking or radically redefining the processing policy of AutoTeach.

With AutoTeach, we believe we are providing an extremely flexible code processing tool. Although the basic processing is entirely automatic, teachers can step in at any time and take full control. Besides the AutoTeach tool itself, this code processing model is general enough to be applicable

[4]In Eiffel, the term 'feature' indicates routines, functions, and class attributes.

Table 3: Hint system usage: NH is no. of students not using any hints, L_i is usage at hint level i, - indicates non-existent hint levels.

Exercise	NH	L_1	L_2	L_3	L_4	L_5	L_6
Object creation	34	40	32	29	-	-	-
Refs and assignment	39	17	17	14	12	11	10
Control structures	28	21	19	-	-	-	-
Palindrome	25	14	12	9	7	6	5
Queue inverter	21	19	18	16	16	15	-
Recursive gcd	25	11	9	8	8	-	-
Decimal to binary converter	16	21	20	19	18	18	-

to any general-purpose programming language with little or no changes, and we are looking forward to seeing alternative implementations of it which will make it available it to a broader audience.

5. DATA ANALYSIS AND RESULTS

We automatically collected usage data from the hint system, and asked students to answer a questionnaire to assess their experience using it. Table 3 shows which and how many hints the students used per programming exercise. These data suggest that a large number of students who submitted a solution did indeed use the hint system.

The ratio of students using hints (columns "L1" to "L6") to those not using them (column "NH") is non-negligible for all exercises. It also noteworthy that certain exercises like the one on object creation, which is perceived as more difficult, have a high hint usage ratio, while others, like the one on references and assignments, have a low hint usage ratio. Such information can be useful when assessing the exercises and possibly updating them for future iterations of the course.

Table 4 shows a more detailed view of the data from Table 3. For every exercise we show how many students submitted correct and incorrect solutions while using or not using the hint system. We observed that students who did not use the hints submitted almost all correct solutions (99.5%). In contrast, there are much fewer correct submissions by students who used the hints (63%). A possible explanation is that students with incorrect solutions (37%) would have needed further assistance. One way of achieving that could be by adding more hint levels and explanations within the hints.

Table 4: Number of correct and incorrect submissions, differentiated by usage or non-usage of the hint system.

Exercise	Submissions			
	not using hints		using hints	
	correct	incorr.	correct	incorr.
Object creation	33	1	27	13
Refs and assignment	39	0	11	6
Control structures	28	0	11	10
Palindrome	25	0	13	1
Queue inverter	21	0	6	13
Recursive gcd	25	0	9	2
Decimal to binary converter	16	0	13	8
Total	187 (99.5%)	1 (0.5%)	90 (63%)	53 (37%)

During our data collection students did not get penalties for using the hint system. Furthermore, students had unrestricted access to the hints. The only limitation we enforced was that students can access the hints only in sequential order from the least detailed to the most detailed, i.e hint level L_{i+1} only became accessible after accessing hint level L_i. We also controlled the students who solved an exercise more than six hours after using the hint system, as well as for students who solved an exercise without hints but subsequently accessed the hints out of curiosity. In both cases the numbers were negligible.

We had 38 students answering the final questionnaire, among which 16 (42%) actually used the hint system in at least one exercise. Among those, 5 students (33%) found the hint system *too difficult to use*, while for the rest (66%) it was *about right*, or *easy to use*. In general, 73% of the students who tried the hint system found it useful, and 80% stated that the level of granularity of the hints was appropriate. Among the students who answered the questionnaire and did not use the hint system (58%), 68% said they did not need hints while solving the programming exercises, while 32% claimed they did not know they could use a hint system. The latter finding is interesting, as apparently advertising the hint system via email, in the lecture, and in the exercise sessions was not sufficient. Additionally, the minimalistic GUI we designed (just a "hint" button close to the frame in which students are supposed to write the code) might have negatively impacted the ability to notice the tool for some students.

In conclusion, our data indicates that the hint system was used by a large number of students and found to be helpful when solving programming exercises.

6. RELATED WORK

Maximizing learning for students is an obvious goal for educators, both in a frontal and in an online lecture scenario. A programming task can become frustrating when a student does not know where to start.

To the best of our knowledge, this is the first paper introducing an automatic hint system assisting students of an introductory programming online course in solving programming tasks. In recent years researchers in software engineering for education have worked on different critical challenges, such as problem generation [2, 4, 1], intelligent tutoring [8], automatic grading [10, 11], and facilitating human interaction in an online programming course [12].

Tillmann et al. [11] introduced Pex4Fun, an environment for teaching and learning programming. Their basic idea is that, given a task and a sample solution invisible to the student, the latter tries to work towards the solution by iteratively supplying code. At each iteration, Pex4Fun returns some passing and failing test-cases for the submitted code. The test-cases are computed automatically by using symbolic execution and a theorem prover. Students keep submitting new versions of the code until the tool cannot find any more failing test-cases. The failing test-cases can be seen as a kind of automatically provided semantical hints that students use to iteratively construct a solution, ideally semantically equivalent to the hidden one. In contrast, our hints are both syntactic and semantic, they are inferred from the sample implementation or provided by the teacher, and are preprocessed, which means they do not depend on the students' solutions.Our hints guide the students to our

master solution, while alternative solutions are anyway evaluated by running them against a series of unit tests.

Singh et al. [10] introduced an error-model language used to model possible errors that students might make while trying to solve a programming assignment. Starting from these error models, from an implementation provided by a student, and from the one provided by a teacher, their approach produces hints suggesting students how to modify their code in order to remove the errors. In contrast our hints, rather than analyzing the student's solutions, guide the students through the construction of the master solution.

CodeSkulptor [12] is a system helping students to learn programming in Python, that also assists them during the construction of an assignment's solution. However, this assistance is not automatic, it is provided by other students via a forum or by a teaching assistant via email. Our hint system is totally automatic and hints are obtained immediately by a simple click.

Peddycord et al. [9] introduce a technique based on BOTS, an educational programming game teaching the basic concepts of programming through block-moving puzzles. The goal of their hints is to help a student solving a puzzle pass from the current state to another one, which is closer to the goal. The hints are generated by using successful solutions of other students who dealt with the same puzzle and who were in the same state. Although the approach is supposed to be used for teaching concepts of programming languages, it does not use any concrete programming language. The idea of hint generation starting from the correct solutions of other students is, however, interesting, and we plan to investigate it in the future as part of our hint generation.

7. CONCLUSIONS

Finding ways to assist introduction to programming students in an automatic fashion is, nowadays, with the advent of MOOCs, becoming even more relevant. We contribute a programming environment in the cloud with an associated hint system, which gradually unveils parts of the desired solution to students that require support to produce a solution. We designed our hint system to provide flexibility in how teachers can generate hints while requiring little to no input when using default settings.

Our hint system was able to effectively process most of our programming exercises without any extra annotation in the source files, yielding satisfactory hints. Wherever needed, teachers could easily step in and redefine the processing policy by defining a custom hint table.

The data we collected using questionnaires and usage logs suggest that the hint system is a beneficial addition to our teaching programming platform in the cloud. We are now running a larger experiment in the edX MOOC "Computing: Art, Magic, Science" in Spring 2015, and we are developing a Java version of our hint system.

Acknowledgments

We would like to thank Andre Macejko for his work on many technical aspects of our online course.

8. REFERENCES

[1] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically Generating Problems and Solutions for Natural Deduction. In *IJCAI*, 2013.

[2] E. Andersen, S. Gulwani, and Z. Popovic. A Trace-based Framework for Analyzing and Synthesizing Educational Progressions. In *SIGCHI*, pages 773–782, 2013.

[3] P. Antonucci. Autoteach: incremental hints for programming exercises. Master's thesis, ETH Zurich, sep 2014.

[4] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing Geometry Constructions. In *PLDI*, pages 50–61, 2011.

[5] B. Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

[6] B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, 1997.

[7] J. Middendorf and A. Kalish. The "change-up" in lectures. *The National Teaching & Learning Forum*, 5(2), 1996.

[8] T. Murray. Authoring Intelligent Tutoring Systems: an Analysis of the State of the Art. *Intern. Journal of Artificial Intelligence in Education*, (10):98–129, 1999.

[9] B. Peddycord III, A. Hicks, and T. Barnes. Generating hints for programming problems using intermediate output. In *EDM*, pages 92–98, 2014.

[10] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*, pages 15–26, 2013.

[11] N. Tillmann, J. De Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and Learning Programming and Software Engineering via Interactive Gaming. In *ICSE*, 2013.

[12] J. Warren, S. Rixner, J. Greiner, and S. Wong. Facilitating Human Interaction in an Online Programming Course. In *SIGCSE*, pages 665–670, 2014.