Strategy-based feedback in a programming tutor

Hieke Keuning

Bastiaan Heeren

Johan Jeuring

Technical Report UU-CS-2014-026
November 2014

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Strategy-based feedback in a programming tutor

HIEKE KEUNING and BASTIAAN HEEREN and JOHAN JEURING
Open University of the Netherlands, Heerlen, The Netherlands

More and more people take up learning how to program: in schools and universities, in large open online courses or by learning it by themselves. A large number of tools have been developed over the years to support learners with the difficult task of building programs. Many of these tools focus on the resulting program and not on the process: they fail to help the student to take the necessary steps towards the final program.

We have developed a prototype of a programming tutor to help students with feedback and hints to progress towards a solution for an introductory imperative programming problem. We draw upon the ideas of a similar tutor for functional programming and translate these ideas to a different paradigm. Our tutor is based on model solutions from which a programming strategy is derived, capturing the different paths to these solutions. We allow for variation by expanding the strategy with alternatives and using program transformations. The instructor is able to adapt the behaviour of the tutor by annotating the model solutions.

We show a tutoring session to demonstrate that a student can arrive at a solution by following the generated hints. We have found that we can recognise between 33% and 75% of student solutions to three programming exercises that are similar to a model solution, which we can increase by incorporating more variations.

Categories and Subject Descriptors: K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction (CAI); K.3.2 [**Computer and Information Science Education**]: Computer science education

Additional Key Words and Phrases: programming tutor, feedback generation, programming strategies

ACM Reference Format:

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. CSERC '14: Computer Science Education Research Conference 2014 Proceedings.

1. INTRODUCTION

Learning how to program is becoming increasingly popular. Students learn programming in universities and colleges to become skilled software engineers. Many people, both young and old, teach themselves programming as a hobby or to pursue a new career path. In many countries, secondary schools offer programming classes.

Over the last few years different initiatives have been advocating the importance of providing people with the opportunity to learn how to program. Computer programming education is being promoted actively.[1] The underlying justification is both foundational ('it teaches you how to think') and vocational: the shortage of software engineers is only expected to grow in the near future. Whatever the underlying reasons are, how easy is it for anyone to learn how to program and how can students be supported in their learning process?

1.1 Learning programming

A small study examining the emotional state of students learning to program for the first time showed that after engaged (23%) the major emotions were confusion (22%), frustration (14%) and boredom (12%) [Bosch et al. 2013]. Another study [McCracken et al. 2001] shows that even after their first programming courses students do not know how to program. The failure rate of introductory programming courses is not disturbingly low [Bennedsen and Caspersen 2007; Watson and Li 2014], but can be improved significantly to increase the number of computer science graduates. Therefore, students need all the help they can get to acquire the necessary skills to become successful in the field of computer science, a field that constantly asks for highly educated people.

In an international survey from 2005 [Lahtinen et al. 2005] on the difficulties of novices learning to program, in which over 500 students and teachers were questioned, the following tasks are considered most difficult:

—Designing a program.
—Dividing functionality into procedures.
—Finding bugs in programs.

These findings are consistent with previous studies [Robins et al. 2003; Soloway and Spohrer 1989] that emphasise the importance of program design and applying the right programming constructs.

Teaching the actual programming *process* is considered important [Bennedsen and Caspersen 2008]. The programming process consists of a number of elements, among which the incremental development of a program by taking small steps and testing along the way. Another aspect is the refactoring of programs to improve their quality. Bennedsen and Caspersen note that traditional teaching methods such as textbooks and slide presentations do not cover this process. The authors state a long-term objective of programming education: '... that students learn strategies, principles, and techniques to support the process of inventing suitable solution structures for a given programming problem.'

At the same time learning has become more individual and is frequently done online. The traditional role of the teacher is changing.

Address: Faculty of Management, Science and Technology, Open University of the Netherlands. P.O. Box 2960, 6401 DL Heerlen, The Netherlands.
E-mail: hieke.keuning@ou.nl, bastiaan.heeren@ou.nl, J.T.Jeuring@uu.nl.

ACM 978-1-4503-3347-4/14/11 $15.00
DOI:http://dx.doi.org/10.1145/2691352.2691356

[1]For example, early 2013 the American non-profit organization code.org launched a campaign to promote computer programming education. The accompanying short film, featuring a large number of prominent people (politicians, business leaders, celebrities), has had millions of views.

Teachers have limited time to spend with their students. In online courses, teachers and students do not have face to face interaction. A recent trend in education is the Massive Open Online Course, or MOOC.[2] These large-scale courses are often offered by renowned universities and can be done entirely on the Internet. These developments depend heavily on tools to support the learning process.

1.2 Related work

There has been active research into intelligent programming tutors ever since programming courses were introduced in schools and universities. Many programming tutors have been developed over the years, that all have their own characteristics:

—The supported programming paradigm.

—The type of exercises that are offered. Le and Pinkwart have proposed a categorisation [Le and Pinkwart 2014] based on the degree of ill-definedness of the problem. Class 1 exercises have a single correct solution, class 2 exercises can be solved by different implementation variants and class 3 exercises can be solved by applying alternative solution strategies. Other exercise classes have not been found in programming tutors.

—The knowledge base that is used for the tutoring, such as model solutions, sets of constraints, test data, templates and production rules.

—The type of feedback the tutor provides: *feed up*, *feed back* or *feed forward* [Hattie and Timperley 2007].

We have found that many tutors [Sykes and Franek 2003; Singh et al. 2013] can only provide feedback for complete student solutions and thus cannot guide a student on her way to the right solution. Tutors that focus on the process of solving an exercise are still rare or have limitations: some are targeted at declarative programming [Hong 2004], are less flexible because they do not support exercises that can be solved by multiple algorithms [Anderson and Skwarecki 1986; Miller et al. 1994], or only support a static, pre-defined process [Jin et al. 2014]. Furthermore, it often requires substantial work to add new exercises (e.g. [Singh et al. 2013]) and tutors can be difficult to adapt by a teacher.

A recent trend in programming tutors is to generate hints based on historical student data [Rivers and Koedinger 2013; Jin et al. 2012]. An advantage of this data-driven approach is that the instructor does not have to provide any input. On the other hand, it requires the existence of a large data set. Furthermore, there might be unusual or unwanted solutions in this data set.

1.3 Contributions

We have developed a prototype of a tutor for imperative programming[3] that helps students with incrementally constructing programs. We draw upon the ideas of ASK-ELLE, a similar tutor for functional programming [Gerdes et al. 2012] and translate these ideas to a different paradigm. We make the following contributions:

—Our tutor supports the step by step development of solving class 3 problems in the categorisation by Le and Pinkwart [Le and Pinkwart 2014]: a student can practise with imperative programming exercises that can be solved by multiple algorithms. The tutor supports constructing a program by expanding it statement by statement and by using templates for refining expressions. The tutor can diagnose if a student is on the right track, and can give hints on the various ways on how to proceed.

—Feedback is derived from an exercise description together with a set of model solutions. We have adopted the annotations from the ASK-ELLE tutor, adjusting and expanding them for imperative programs. An instructor can annotate the model solutions to adapt the feedback, such as providing specific feedback messages, allowing alternative statements and enforcing the use of a specific language construct (enabling class 2 problems).

—We use a general purpose *strategy language* to define the process of creating a solution for a programming exercise. We derive strategies for the creation of a number of common language constructs in imperative programming. We encode alternatives to a solution path into the derived strategy so we are able to provide feedback on these alternatives. Minor variations on which we do not want to give feedback are dealt with by performing a number of program transformations.

1.4 Outline

In Section 2 we show a tutoring session to give an impression of the behaviour and capability of the prototype. In Section 3 we discuss the generation of a programming strategy from model solutions. Section 4 describes the feedback services that are offered and explains how the feedback can be adapted by an instructor. We show the results of the evaluation of the tutor in Section 5. We conclude in Section 6 by summarizing our work and discussing our plans for future research.

2. A TUTORING SESSION

Our programming tutor prototype consists of a *domain reasoner* for imperative programming and a user interface. A domain reasoner [Goguadze 2011] provides facilities to support solving exercises in a particular domain and to generate personalised feedback and guidance. These facilities are offered through feedback services, which the user interface calls through JSON or XML messages [Heeren and Jeuring 2014]. A web interface has been created to offer a simple learning environment in which students can solve exercises using the various feedback services. Figure 1 shows a screenshot of this web front-end, which has been created using HTML, JavaScript/JQuery and Ajax-calls with JSON messages to the services of the domain reasoner.

The prototype can be used for languages that support the imperative programming paradigm. We support a subset of frequently used language constructs including variables, loops, conditional statements and arrays in Java and (to a lesser extent) PHP. Currently, the prototype does not support more advanced constructs such as method declarations, object orientation, exception handling and object types.

2.1 A tutoring session

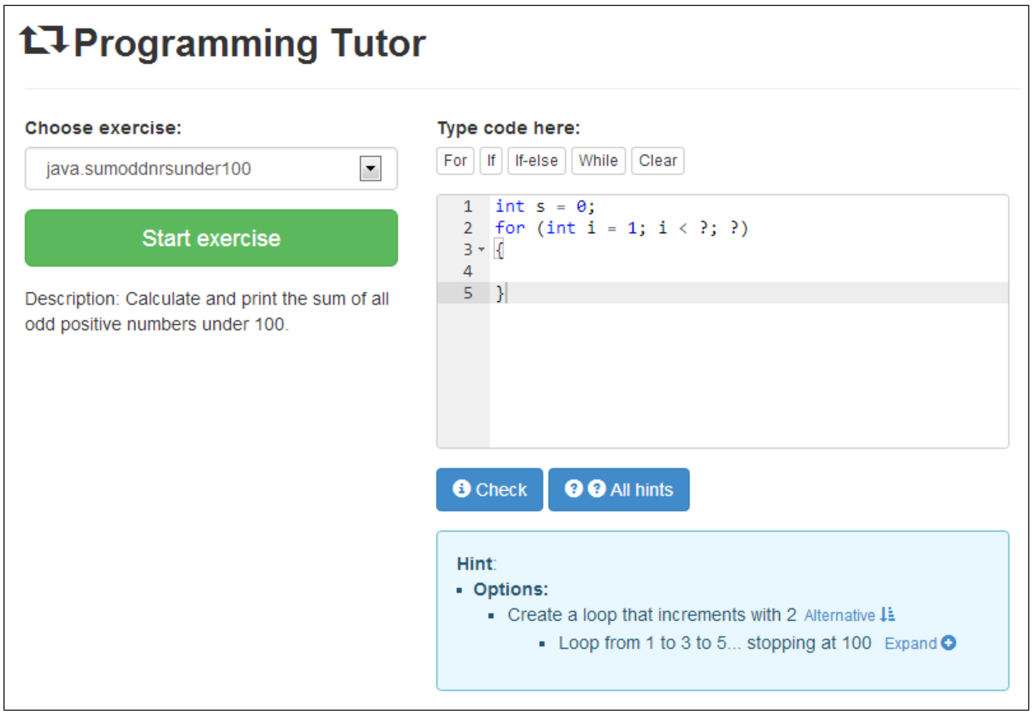
We show a simulated session with our tutor for a simple exercise with the following description:

> *Calculate and print the sum of all odd positive numbers under 100.*

We provide three model solutions:

[2]Laura Pappano. 2012. The Year of the MOOC. *The New York Times*. http://www.nytimes.com/2012/11/04/education/edlife/massive-open-online-courses-are-multiplying-at-a-rapid-pace.html

[3]Studies (such as [Davies et al. 2011]) show that the most popular languages taught in schools and universities are based on the imperative programming paradigm.



Description: Calculate and print the sum of all odd positive numbers under 100.

```
int s = 0;
for (int i = 1; i < ?; ?)
{

}
```

Hint:
- Options:
  - Create a loop that increments with 2 Alternative
    - Loop from 1 to 3 to 5... stopping at 100 Expand

Fig. 1. Web front-end

(1) Adding up the odd numbers in a loop that increments with two.
(2) Adding up the numbers in a loop if the loop counter is odd.
(3) Using a mathematical formula.

The models are annotated by the instructor. We show the annotated model for solution 2:

```
/* DESC "Create a loop and test for odd
     numbers with %"
   PREF 2 DIFF medium */
int sum = 0;
/* FEEDBACK Loop through all numbers
    from 1 stopping at 100 */
for (int i = 1; i < 100; i++) {
   /* FEEDBACK Use if and % to check if
        the counter is odd */
   if (i % 2 == 1) {
      /* FEEDBACK Add to total */
      sum = sum + i;
   }
}
print(sum);
```

In the web interface the student can select the exercise from a list of available exercises. The editor, in which the code can be typed, has syntax highlighting. There are buttons that generate templates for some statements at the top and buttons to ask for feedback (a diagnosis/check and hints) at the bottom.

The focus of our research is on the generation of relevant hints, and not so much on how we present these hints to the student. At the moment we show all hints in a tree structure. When the student first asks for a hint, the first option (branch) of the hint tree is shown. The student has the opportunity to 'expand' (denoted by the ⊕ symbol) on a specific path and view a hint with more detail. If other options are allowed, the 'alternative' link will provide a hint on a different solution path. In this session we show parts of the hint tree and fold certain branches for clarity.

Student. The student does not know how to start and asks for a hint.

Tutor. Multiple solution algorithms are suggested, using the descriptions in the model solutions. The hints are sorted by their preference number.

- Create a loop that increments with 2
  - Introduce a variable declaration ⊕
- Create a loop and test for odd numbers with % ⊕
- Perform a smart calculation ⊕

Student. The student declares a variable and uses the 'for' button to add a template of a for-statement.

```
int s = 0;
for (?; ?; ?)
{
}
```

Tutor. The step is recognised, even though the student used a different variable name. One model solution is no longer present in the hint tree (the mathematical formula) because it does not match the code.

- Create a loop that increments with 2 ⊕
- Create a loop and test for odd numbers with % ⊕

Student. The student starts implementing the for-statement, but leaves some expressions unfinished.

```
int s = 0;
for (int i = 1; i < ?; ?)
{
}
```

Tutor. Hints with increasing detail are available. One of the hints contains the text of the teacher annotation.

- Create a loop that increments with 2 ⊕
- Create a loop and test for odd numbers with %
  - Loop through all numbers from 1 stopping at 100
    - Expand the second part of the for-statement
      - Replace the ? on the right
        - Expand ? to the literal 100 ⊕

Student. The student completes the loop condition, using a different notation for the increment expression.

```
int s = 0;
for (int i = 1; i < 100; i+=1)
{
}
```

Tutor. Another teacher annotation that has become relevant for the current state of the program is shown in the hint tree.

- What to repeat?
  - Use if and % to check if the counter is odd
    - Introduce an if statement ⊕

Student. The student follows up on the hint and creates an if-statement, leaving the condition unfinished.

```
int s = 0;
for (int i = 1; i < 100; i+=1)
{
    if (? == 1)
    {
    }
}
```

Tutor. The hints help the student finish the expression.

- What do you want to check?
  - Expand ? to an expression with the operator % ⊕

Student. The student finishes the expression and also adds the odd counter to the sum, using a different expression than the model.

```
int s = 0;
for (int i = 1; i < 100; i+=1)
{
    if (i%2 == 1)
    {
        s += i;
    }
}
```

Tutor.

- Introduce a print-statement ⊕

Student. The student makes a mistake by printing the wrong variable.

```
int s = 0;
for (int i = 1; i < 100; i+=1)
{
    if (i%2 == 1)
    {
        s += i;
    }
}
print(i);
```

Tutor. The tutor notices that the output of this program does not match the output of the model solutions.

The output is incorrect.

Student. The student corrects the mistake.

```
int s = 0;
for (int i = 1; i < 100; i+=1)
{
    if (i%2 == 1)
    {
        s += i;
    }
}
print(s);
```

Tutor. Correct. You are done!

3. PROGRAMMING STRATEGIES

Our tutor is built on top of the IDEAS framework (Interactive domain-specific exercise assistants), which provides services to build exercise assistants to help students solve problems incrementally [Heeren and Jeuring 2014]. The framework is used for various domains and their corresponding exercises, such as solving equations, bringing a proposition in DNF, and functional programming in Haskell [Gerdes et al. 2012].

Skills such as programming or solving mathematical problems require learning *strategies* for solving these problems [Heeren et al. 2010]. A strategy describes how to make progress step by step and arrive at a good solution, and represents the various approaches that can be used to tackle a problem. In IDEAS, strategies are specified using a *strategy language*.

Building a tutor using IDEAS requires a number of components:

(1) *Domain specification.* A domain is described, among other things, by a grammar for its abstract syntax and an accompanying parser to process submitted work.
(2) *Steps.* A step is a transformation on values of the domain, such as refining or rewriting a student submission.
(3) *Strategies.* A strategy combines basic steps, and specifies which steps can be taken, in which order. Strategy *combinators* are used to compose strategies into more complex strategies. Examples of strategy combinators are the sequence combinator (`a <*> b`, first do strategy `a`, then do strategy `b`), the choice combinator (`a <|> b`, do either `a` or `b`), and the interleave combinator (`a <%> b`, interleave the steps of `a` and `b`).

In this section we elaborate on these three components. First, we describe how incomplete student submissions are supported. Next, we take a closer look at the programming process for imperative programming. Finally, we describe the steps and strategies we have implemented for our tutor prototype.

3.1 Support for incomplete programs

We want to give feedback on incomplete programs, to support students in creating a program step by step. We support incomplete programs in two ways. First, a student may only write the first few lines of a program, resulting in a syntactically correct but semantically incomplete program. Secondly, the question mark character ('?', hereafter referred to as *hole*) can be used inside a statement to represent an expression that is yet to be completed. For example:

```
int x = ? + ?;
for (?; ?; ?);
while (x < ?);
```

The hole symbol will cause error messages if the student uses an IDE for programming, because a standard compiler will not recognise the symbol. Currently, our tutor can only be used in a web environment in which the hole symbol is recognised and the student receives help to complete the statement before continuing with the remaining program.

3.2 The imperative programming process

To use the IDEAS framework for calculating feedback, we need to specify a strategy for each exercise. In an educational setting, the instructor serves as a guide to show students how to program. When an instructor is not present, we would like to stay close to what an instructor would have said when a student asks for help. Therefore, model solutions from an instructor are used as a basis from which we generate the strategy and the corresponding feedback. This approach is used in ASK-ELLE as well as in a number of other recent programming tutors [Dadic et al. 2008; Hong 2004; Singh et al. 2013] and provides a number of advantages:

—Instructors can easily add new exercises.
—Model solutions can be annotated, providing extra opportunities for didactic guidance.

Potential difficulties that should be looked into are the large solution space, which is discussed in the following sections, and the lack of clarity on what exactly distinguishes one solution from the other. It is the instructor's responsibility to provide model solutions that represent the solution space of an exercise. Every model solution should preferably represent a different algorithm that solves the problem.

The strategy to work towards a particular model solution should reflect how imperative programs can be constructed, such as:

—Quickly constructing a coarse solution and then improving it until there are no errors left. This approach might reflect the trial and error style students often adopt.
—Programming by contract: defining pre- and post-conditions prior to the actual implementation [Dijkstra 1975].
—The stepwise decomposition of a program using refinement steps [Wirth 1971]. In each step tasks are broken up into subtasks.
—Building up a program statement by statement, manipulating the program state in the meantime.

In recent tutors that support incomplete programs, we recognise the third option for logic programming [Hong 2004]. An imperative programming tutor [Jin et al. 2014] uses pre-defined subtasks, such as variable analysis, computation and output. In the ASK-ELLE tutor for functional programming refinement steps are used to gradually make a program more complete by replacing unknown parts (holes) by actual code. In two data-driven tutors for imperative programming [Jin et al. 2012; Rivers and Koedinger 2013] the steps are based on solutions paths that other students have taken in the past.

Currently, we support the statement by statement programming strategy. This strategy corresponds with the nature of imperative programming in which the program state is manipulated step by step. We also incorporate refinement of holes to complete an unfinished expression. This style can be used as a basis to expand the tutor with other strategies in the future.

To support this strategy in our tutor we need two components that are elaborated in the next sections:

—The definition of steps that a student can take to gradually build up a solution.
—A strategy generator that generates a strategy from model solutions using these steps.

3.3 Steps

A strategy to solve an exercise is made up of a number of steps. Two types of steps are used in the tutor for imperative programming: append steps and refinement steps.

Append steps. An append step appends a statement to the end of a block, which corresponds to updating the program state in steps. An example of four consecutive applications of an append step, starting with an empty program, is:

```
⇒
   x = 5;
⇒
   x = 5;
   y = 7;
⇒
   x = 5;
   y = 7;
   avg = (x+y)/2;
⇒
   x = 5;
   y = 7;
   avg = (x+y)/2;
   print(avg);
```

Refinement steps. A refinement step replaces a hole by an expression. An example of applying a sequence of three refinement steps is:

```
   avg = ?;
⇒
   avg = ? / ?;
⇒
   avg = sum / ?;
⇒
   avg = sum / 2;
```

3.4 Strategies

With the steps described in the previous section, we can now specify strategies for the stepwise development of a program. We have created a strategy generator that accepts a set of model solutions as input and produces a strategy as output. We generate a strategy for each statement and expression in a program. We illustrate this by showing the implementation of a selection of language constructs.

If-statement strategy. The strategy for an if-statement consists of three main steps, combined with the sequence (<*>) combinator. The first step is an append step that introduces an empty if-statement at the specified location, denoted by `@loc`, with a hole for the condition and an empty body. The next step is the sub strategy for building the condition to replace the hole. Note that a sub strategy may consist of multiple steps. The final step is the sub strategy to implement the body. The complete strategy is shown in the following functional pseudo code:

```
strategy (If cond body) loc =
       append (If hole emptyBody) @loc
   <*> strategy cond @hole
   <*> strategy body @emptyBody
```

The step-wise application of this strategy is shown in the following example:

```
   if (?) {}
⇒
   if (isOk) {}
⇒
   if (isOk) { call(); }
```

Infix expression strategy. The next fragment shows the strategy for an infix expression, such as `(a + b) < 2`. First, an infix expression with holes on both sides of the operator is introduced by refining a hole at the specified location, followed by the interleaving (<%>) of the sub strategies for the left and right operands of the expression. Interleaving implies that the sub strategies have to be applied but may be interleaved, meaning their order is not relevant. Refining the left hole first has a higher preference.

```
strategy (Infixed op l r) loc =
   refine (Infixed op holeL holeR) @loc
   <*> ( strategy l @holeL
         <%> strategy r @holeR )
```

The strategy expresses that we can arrive at an expression consisting of several subexpressions in multiple ways:

$$\begin{tikzcd}
 & \texttt{avg = ?;} \arrow[d, Rightarrow] & \\
 & \texttt{avg = ? / ?;} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
\texttt{avg = sum / ?;} \arrow[dr, Rightarrow] & & \texttt{avg = ? / 2;} \arrow[dl, Rightarrow] \\
 & \texttt{avg = sum / 2;} &
\end{tikzcd}$$

Loop strategy. A more complex situation arises when we encounter a for-statement in a model solution. A for-statement is easily transformed into a while-statement, which we want to support in our tutor. In the corresponding code we create a strategy for the for-statement together with an accompanying while-statement and combine their strategies with the choice (<|>) combinator. The while-statement is constructed by moving the initialisation of the for-statement to a new statement preceding the while-statement. The increment expression of the for-statement is appended to the end of the body of the loop. However, we only include the strategy for a while-statement if the for-statement has exactly one condition to avoid an empty while condition.

The resulting strategy allows the following sequence (skipping some steps):

```
   i = 0;
⇒
   i = 0; while(?) {}
⇒
   i = 0; while(i < 8) {}
⇒
   i = 0; while(i < 8) { call(); }
⇒
   i = 0; while(i < 8) { call(); i++; }
```

The next sequence is also allowed (skipping some steps):

```
   for(?; ?; ?) {}
⇒
   for(i = 0; ?; ?) {}
⇒
   for(i = 0; i < 8; ?) {}
⇒
   for(i = 0; i < 8; i++) {}
⇒
   for(i = 0; i < 8; i++) { call(); }
```

Block strategy. We are faced with a challenge when we want to define a strategy for a block, a list of statements. Every program is a list of statements and inside statements such as loops we find nested lists of statements. A program can be developed statement by statement, continuously manipulating the program state. However, the order of some statements can be changed with no consequences for the output of the program.

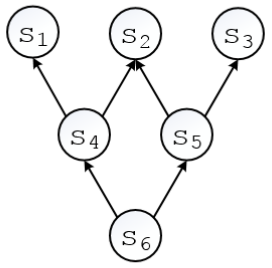
Consider the following code fragment:

```
1  a = 1;
2  b = 2;
3  c = 3;
4  d = a + b;
5  e = b + c;
6  f = d + e;
```

For example, lines 1, 2 and 3 can be swapped with no consequences for the resulting program, as well as lines 3 and 4. The possibility to change the order of statements depends on the type of the statements and the variables that are used. For example, lines 5 and 6 can never be switched because the value of e is needed for the calculation.

A relation has been defined to determine whether a statement depends on another preceding statement. Statement can be dependent because of variable use, the generation of side-effects such as printing output or because they control program flow, such as a break-statement. Using this relation, we construct a *directed acyclic graph* (DAG) for a list of statements. The arrows in the graph indicate that a statement depends on another prior statement. A strategy is generated for each node and is stored in the nodes.

The following graph represents the DAG for the previous code fragment, containing the sub strategies for lines 1 to 6 in the nodes:



We build a strategy from this graph by listing all *topological sorts* of the graph. A topological sort is a possible ordering of the vertices with the property that for every edge representing a dependency from a node a to a node b, b precedes a in the ordering. Incorporating *all* topological sorts into the strategy means that a student can put statements in any order, as long as the resulting program is valid. The graph is converted into a left-factored strategy using the strategy combinators for sequence (<*>) and choice (<|>).

Converting the graph from the example results into the following strategy that provides a student with multiple ways to build the program.

```
    (s1<*>(    (s2<*>(    (s3<*>(
                                   (s4<*>s5<*>s6)
                                <|>(s5<*>s4<*>s6)))
                           <|>(s4<*>s3<*>s5<*>s6)))
               <|>(s3<*>...)))
 <|>(s2<*>...)
 <|>(s3<*>...)
```

Exercise strategy. The final step in generating a strategy from a set of model solutions is combining the strategy for each model solution into one final strategy using the choice (<|>) combinator.

4. FEEDBACK GENERATION

A programming exercise can be specified by providing a set of model solutions and an exercise description in a text file. Students can do the exercises by creating a solution and asking for feedback. In this section we discuss the feedback services that are available in the tutor, after considering the issues related to recognising a wide variety of student programs. We conclude this section by looking at the different ways an instructor can adapt the feedback.

4.1 Recognising solutions

We have incorporated several variations to the model solutions in the derived strategy: multiple algorithms, variation in statement order and different language constructs. However, there are many more variations that should be taken into account. If a student works on a program creating a solution that closely matches a model solution but is not exactly the same, the student solution should be recognised. Furthermore, if the student deviates from a strategy, we would still like to provide some response.

We need to further establish when a student program and a model solution can be considered ‘equal’. For this reason, we define two relations on solutions that support the feedback generation: program similarity and output equivalence.

Program similarity. A model solution represents an algorithm to solve a particular programming problem. Therefore, we need to establish what exactly distinguishes one algorithm from another algorithm. The question of what defines an algorithm lacks a clear answer and is subject to various interpretations. Blass et al. [Blass et al. 2009] argue against the notion that algorithms are equivalence classes of programs, implying that there is no ‘precise equivalence relation capturing the intuitive notion of the same algorithm’. They provide several examples to illustrate their point, stating that opinions, subjective judgment and intended purpose influence this relation as well as a lack of clarity concerning the transitivity property.

We define our own relation on algorithms that indicates if two solutions are similar, to support the instructor in supplying a number of model solutions. The strategy to arrive at a particular model solution already incorporates a number of variations. Furthermore, the instructor is able to control which variations are allowed, and which are not allowed. Other than that, there are many more minor variations that we do not want to identify as different solutions. As an example, consider the following program fragment:

```
x = "*";
for(i = 0; i < 8; i++) { print(x); }
```

The next fragment shows an alternative implementation:

```
for(cnt=1;cnt<=8;cnt+=1) print("*");
```

In both solutions a looping structure is used to print a star symbol eight times. The differences (use of an extra variable, different loop counters, different variable names) do not change the algorithm used for this program and we would like to recognise the second program as a correct alternative for the first.

We define a *similarity* relation to determine if two programs are similar when matching the student solution with a program derived from the exercise strategy. We define that two programs are similar ($\approx$) if their representation in abstract syntax is equal, after *normalising* each program.

```
p1 ≈ p2 = normalise p1 == normalise p2
```

The properties for symmetry, reflexivity and transitivity all hold, therefore this relation is an equivalence relation. Normalisation continuously performs a series of transformations until no more transformations can be applied.

Xu and Chee [Xu and Chee 2003] have identified 13 types of *semantics-preserving variations* (SPVs) that occur in student programs. An SPV changes the computational behaviour (operational semantics) of a program while preserving the computational results

SPV	Description	Parser	Strategy	Normalisation
1	Different algorithms		●	
2	Different source code formats	●		
3	Different syntax forms	◐	◐	◐
4	Different variable declarations		◐	◐
5	Different algebraic expression forms			◐
6	Different control structures		◐	
7	Different Boolean expression forms			◐
8	Different temporary variables			
9	Different redundant statements			
10	Different statement orders		●	
11	Different variable names			●
12	Different program logical structures			
13	Different statements		●	

Table I. Support of SPV's

(computational semantics). We incorporate some of these variations in different components of the tutor (the parser, the strategy) as shown in Table I. Some differences that are not captured in the abstract syntax or in the strategy are implemented in a normalisation procedure.

We use the ●-symbol to indicate that an SPV is fully implemented in the prototype and the ◐-symbol for SPVs that are partly implemented. To increase the number of program variants that we can recognise, we would have to add several more normalisations. However, some normalisations have deliberately been omitted because the resulting program would deviate too much from the instructor solution. The normalisations that have been implemented in our prototype are elaborated in the first author's Master's thesis [Keuning 2014].

Normalisation regarding temporary variables (SPV 8) and redundant statements (SPV 9) poses several challenges if we apply them to incomplete solutions. Adding these variations to the tutor requires further research.

Output equivalence. When we encounter student programs in which no model solution can be recognised, we would still like to provide the student with some feedback. If we cannot recognise the program structure, we are left with looking at the output of the program. If the output of the student program is equal to the output of a model solution, we can inform the student that the solution produces correct results, although we cannot comment on the algorithm used. This algorithm may either be a potential addition to the set of model solutions, or an inefficient or inelegant solution.

Testing could be used to check if two programs produce the same output. Currently, the prototype does not support functions and focuses on writing output to a console. For this reason, an evaluator has been implemented that computes the output of a program based on print statements. We also have to take into account that incomplete programs may be submitted. We therefore do not define an equivalence relation but instead we use a relation to define if the output of a program is a prefix of the output of a model solution. This relation is not an equivalence relation because the symmetry property does not hold. We show a simplified version of this relation in which we have omitted dealing with evaluation errors.

```
program ⇝ sol =
   (eval program) `isPrefixOf` (eval sol)
```

This relation holds for the following programs that have equal output:

```
    x = "*";
    for(i = 0; i < 8; i++) { print(x); }
⇝
    print("********");
```

In the next example the output of the student program at the top is a prefix of the model solution on the bottom:

```
    print("a");
⇝
    print("abc");
```

If we would expand the prototype to support method declarations, we might be able to use (an expansion of) the evaluator for testing. We could use an existing testing tool, but such tools will probably not support incomplete programs with holes. Therefore, a custom made solution should be developed.

4.2 Feedback services

Hattie and Timperley [Hattie and Timperley 2007] stress that the powerful influence of feedback on the learning process can either be positive or negative. A model is proposed to clarify how feedback can be put into practice in the best way. The findings are mainly about feedback from actual human beings, but because we want to closely mimic this in intelligent tutoring systems, several conclusions are also of interest to our research. According to the model, the three questions that effective feedback should answer are:

—Where am I going? (*feed up*)

—How am I going? (*feed back*)

—Where to next? (*feed forward*)

These questions help learners to understand where they are right now and what has to be done to improve or finish a given task. The authors also claim that feedback is more effective when 'it builds on changes from previous trials'. If these characteristics are implemented in automated feedback systems, it will provide the student with a useful alternative to a human teacher.

Our tutor offers two main feedback services:

—The DIAGNOSE service for analysing a student submission (*feed back*).

—The HINTTREE service for providing a tree structure with hints at various levels. (*feed up* and *feed forward*)

Diagnosis	Explanation
Expected	The submitted program follows the strategy.
Correct	We cannot recognise the program, but the output of the program is correct so far.
Not equivalent	We cannot recognise the program and the output of the program is incorrect.
Similar	Currently unused, but can be used to detect that no significant changes have been made since the last submission.
Detour	A valid step is recognised but does not follow the strategy.

Table II. Diagnoses

Some meta services are also available, such as loading the list of available exercises.

Diagnosis. A student can submit a (partial) solution to a programming problem at any time. A student might even submit a finished solution straight away. We use the DIAGNOSE service from the ASK-ELLE tutor for diagnosing a student submission. We have made some adjustments and additions to enable the service for the imperative programming domain. We use our evaluator to inspect the output of a program if no strategy can be found. After submitting, the student receives a message indicating if the work was correct or if a mistake has been made. The available diagnoses are listed in Table II.

Hints. The HINTTREE service from the ASK-ELLE tutor is used to generate a tree structure with hints on how to proceed, as shown in the tutoring session in Section 2. The hints are based on the steps defined in the exercise strategy and the corresponding labels. The branching indicates the choice between different steps and the depth of a node indicates the level of detail of the feedback message.

During the generation of a strategy for an exercise, *labels* are attached to steps and sub strategies. These labels are used to provide textual hint messages. The following labels and other descriptions are inserted automatically:

—Steps have an identifier with a description, for example 'Introduce break-statement' for an append step and 'Expand ? to identifier' for a refine step.

—Sub strategies are labelled to provide more specific feedback.

We provide an example of a for-statement, which has the following format:

```
for (init; cond; incr) body;
```

We show the (simplified) strategy for a for-statement in which we label the sub strategies for the components with a corresponding label:

```
    append (For ...)
<*> label "loop-init" (strategy init)
<*> (    label "loop-cond" (strategy cond)
     <%> label "loop-incr" (strategy incr)
    )
<*> label "loop-body" (strategy body)
```

The corresponding textual descriptions for labels are stored in a text file. This file can be manually adjusted by an instructor. The IDEAS framework supports the parsing of these files and using them to generate textual feedback messages. We show a small fragment from this script file:

```
feedback loop-incr = {What should happen after
    each loop iteration?}
feedback assign = {What value should the
    variable get?}
feedback args = {What information should you
    pass to the function?}
```

In this section we have shown that the services developed for the functional programming tutor ASK-ELLE can also be used for imperative programming. Moreover, the services can be used for any domain: they support doing multiple steps at once regardless of the nature of the steps. We propose moving the services to the IDEAS framework to make them more widely available.

4.3 Adapting feedback

If an instructor wants to use our tutor for a particular exercise, the instructor only needs to provide a set of model solutions. Feedback will be calculated automatically based on these solutions. However, an instructor may sometimes want to provide additional information to further guide the process of solving an exercise. A number of instructor facilities are implemented in the prototype. The script that stores the textual representations for strategy labels and other feedback messages can easily be adjusted by an instructor. The model solution that the instructor provides can be customised with several annotations. These annotations enable instructors to create tailor-made exercises for their students.

The ASK-ELLE tutor introduced the concept of annotated instructor solutions. We have adopted a number of these annotations in our tutor for imperative programming. We also propose some adjustments. Annotations are added to the model code inside comments so they do not require the adaptation of compilers. Our parser is able to recognise these comments. We will describe the features that are currently available in the prototype.

General solution information. A model solution can be annotated with general information. At the top of the model solution the following annotation can be added:

```
/* DESC "Implement the Quicksort
    algorithm" PREF 2 DIFF Hard */
...
```

We label the strategy for a particular model with the solution description. The solution difficulty is currently unused, but could be used to exclude certain solution paths because they are either too difficult or too easy, when we have the student level at our disposal. We use the preference number to show the hints that lead to the most preferred solution path first.

Feedback messages. The FEEDBACK-annotation can be used to provide more information about the meaning of a statement in the context of a specific assignment, for example:

```
/* FEEDBACK Calculate the average of
     the two results */
double avg = (x + y) / 2;
```

Another example is:

```
/* FEEDBACK Create a loop through all
     even numbers below 100 */
for (i = 0; i < 100; i += 2) ... ;
```

The feedback text will be attached to the strategy for the statement that follows the annotation.

Mandatory language constructs. Occasionally, an instructor may create an exercise to train using a particular new language construct. For example, when introducing the for-statement, the students should practise with this statement and not revert to a while-statement that they already know. Because the strategy generator by default attempts to include as many variants as possible into the strategy, the generator should be instructed when this is not desirable. The instructor is able to do this by annotating a statement in a model solution using the MUSTUSE-annotation.

```
/* MUSTUSE */
for (i = 1; i < 10; i++) ...;
```

This annotation will instruct the strategy generator not to include the option to create a while-statement as an alternative. This annotation is currently only used to enforce the use of a for or while-statement, but can also be used for other language constructs in the future.

Alternatives. In some cases we want to allow an alternative for a single statement. Creating an entire new model solution for this is too much work and does not make sense for just one line of code. Using the ALT-annotation, we can provide an alternative for one specific statement that follows the annotation. As an example, we use the annotation to allow a library function instead of one's own implementation. Note that we currently restrict the implementation to one statement, but could expand this to multiple statements in the future.

```
/* ALT x = Math.max(a,b); */
if (a > b)
     x = a;
else
     x = b;
```

Feedback level. When generating a strategy for a set of models, a level is passed as a parameter to indicate the granularity of the steps in the strategy. For example, a particular exercise may be targeted at more advanced students who do not need feedback at a very low level. By default, all strategies are generated at the lowest level of one. Strategies for level two do not include refinement steps that help developing an expression step by step. Not including refinement steps implies that a student can no longer use the hole (?) symbol as a placeholder for unknown expressions.

Consider the following model solution as an example:

```
if (x > 10) f();
```

The feedback level can be set in a configuration file in the exercise folder. We show the first hint if the level is set to one:

- Introduce an if-statement.

The next hint will help the student complete the condition.

If the level is set to two, the hints target the entire statement.

- Introduce an if-statement with an expression with operator $>$ as condition.

The student will receive a new hint only after the condition has been implemented correctly.

5. EVALUATION

We have assessed the quality of our tutor prototype through various methods:

—The demonstration of several tutoring scenarios (one of which is shown in Section 2), showing that following the provided hints leads to a solution.

—A test suite for the automated testing of various cases.

—An analysis of student data, on which we elaborate in this section.

We collected data from first year IT-students (both full-time and part-time) from Windesheim University of Applied Sciences in The Netherlands during their Web programming course from September to November 2013 and their Java programming course from February to April 2014. We asked the students to solve a number of programming problems and submit their (either complete or incomplete) solutions.

The set of collected student programs provides us with information about different solutions. Although we do not know how an individual student arrived at his or her solution, it is still relevant to analyse the submissions to find out the diversity and to determine to what extent our tutor can handle this diversity. Because we do not want to recognise inefficient or inelegant solutions, we examine how many programs that closely match a model solution (manually assessed by a teacher) are recognised as such by our tutor. Assessing the code itself is a different approach from many assessment tools, that are often based on test execution. The results are summarised in Table III.

Exercise 1 and 2 are PHP exercises from the Web programming course. The PHP exercises are relatively simple; their solution consists of few lines of code containing basic constructs such as loops, variable assignments and conditional statements. Our tutor is capable of recognising 75% (24 out of 32, for the first exercise) and 33% (for the second exercise) of the solutions that we consider similar to a model if they would be manually assessed. Unfortunately, there were very few students with a decent solution for the second exercise. No false positives were identified. There were more correct

	Exercise 1	Exercise 2	Exercise 3
Submitted	60	49	80
Models	2	2	2
Similar to model by teacher	32	6	12
Similar to model by tutor	24	2	5
Correctly recognised	**75%**	**33%**	**42%**

Table III. Evaluation results

solutions, but they used different algorithms for which we did not add a model solution.

We have defined another exercise (exercise 3) for the Java programming course in which an array should be checked for containing the valid Fibonacci sequence. This exercise is more complex and has a larger solution than the PHP exercises. We have used this exercise to assess the possibilities and limitations of the tutor in its current form, but also to analyse the diversity in student solutions. Looking at the submissions, it was clear that the students had quite some difficulties solving this exercise with very limited to no help from an instructor.

After analysing a large number of student solutions it became clear that many students had understood the exercise differently than it had been intended. Therefore we decided to include another (suboptimal) model solution to extend our solution space. Also, the ALT-annotation is used in the models to allow for similar output, such as an output string starting with a capital.

We have performed a detailed analysis [Keuning 2014] on the variations we found in the solutions for this exercise, and which variations we do and do not allow, which we omit in this paper. Looking at the results, the number of recognised solutions is 42% because not all variants have been implemented in the prototype and we do not support all language constructs that the students used in their solutions. Many programs contain more than one variation, which explains why the percentage is not that high: with only one unrecognised variation (SPV) the entire program is discarded.

6. CONCLUSION

We have reported on our research into generating adaptable feedback to guide students step by step towards a solution for an introductory imperative programming problem. We have developed a prototype of a programming tutor using the IDEAS framework.

Our prototype supports a selection of basic imperative language constructs, such as loops, branching statements and variable assignments. We have developed a strategy generator that derives a programming strategy from a set of model solutions. The strategy describes the steps to arrive at one of these models. We incorporate alternative paths in the strategy for both the order of steps and some allowed variants of language constructs. To recognise more variations, we use a normalisation procedure. We have also implemented facilities for instructors to annotate a model solution to further control the feedback.

We have demonstrated the capabilities of the prototype in a tutoring session and we have found that we can recognise between 33% and 75% of the solutions, collected during two programming courses, that are similar to a model.

6.1 Future work

New features and improvements. We would like to recognise more student programs that are similar to a model by adding normalisations and by including more variants in the strategy. We also want to expand the annotation capabilities with, for example, a feedback message for multiple statements and using the difficulty of a solution combined with the skill level of a student to personalise the feedback.

Different strategies. Our tutor prescribes a fairly simple way of creating imperative programs: programming statement by statement with some variation in creating a more complex language construct step by step. We want to investigate how to support students in creating programs in different ways:

—Starting with a basic structure and gradually adding the difficult parts.

—Refactoring: restructuring code while maintaining its functionality, which is a valuable process for both novices and experienced programmers [Fowler 1999]. Refactoring teaches students about elegant and compact solutions. For example, a novice programmer could learn how to transform a for-loop into a while-loop, or eliminate an unnecessary variable. The model solutions could serve as examples to which we want to refactor.

—Including other elements of the programming process [Bennedsen and Caspersen 2008] in our tutoring, such as high-level program design and testing the program along the way.

We also need instructor annotations to direct and adapt these programming strategies.

Improved feedback. Performing transformations, such as the rewriting of an expression, may have consequences for the accuracy of the feedback. We should find a way to adjust the hints to take into account the normalisations that were performed and map back to the student's solution, as seen in the work of Rivers and Koedinger [Rivers and Koedinger 2013]. Another option is to move some variations to the strategy while keeping the size of the strategy manageable. We want to investigate if we can dynamically adjust the strategy based on what the student has done so far, and to what extent the IDEAS framework is able to support this.

Language expansion. Currently we support the basic constructs of imperative programming. On our wish list are constructs such as declaring methods, more data types, switch statements, enhanced for loops and do-while statements. Most contemporary imperative languages also support the object-oriented programming paradigm (OOP), with concepts such as (abstract) classes, objects, interfaces, inheritance and polymorphism. There are not many ITSs that support OOP and we would like to explore how to provide feedback for creating larger object-oriented programs. We would also have to investigate how to test incomplete programs that have methods and classes.

Support for multiple imperative languages. Many imperative languages are taught in programming courses for beginners, such as Java, C# and Python. These languages share the same foundation of imperative language constructs, but differ in syntax and specifics. So far we have been using the tutor for two imperative languages. Some issues with the shared abstract syntax have already been identified and we would prefer a different solution to embrace the differences between languages. We want to provide tutoring for multiple imperative languages without the need to create an entirely new tutor for each language. It should be easy to add new languages, possibly by generating parts of the tutor, which was proposed earlier by Jeuring et al. [Jeuring et al. 2012] but has not been explored yet.

Solution space. We base our tutor on model solutions provided by instructors, because they are experts in their field and their solutions serve as examples for students. However, variations to these model solutions are boundless. We want to limit the description of the solution space as far as possible by investigating if we can define a *canonical form* to describe similar solutions with respect to their algorithms. There are discussions on the existence of such categorisations [Blass et al. 2009], but we expect we can define one that is usable for programming problems for novices in tutoring systems.

Evaluation. After expanding our tutor we have planned several evaluation methods. We want to do experiments to investigate if the tutor corresponds with student behaviour and if students consider the feedback helpful. We are interested in how instructors value the tutor and if they are able to deal with creating exercises and annotating model solutions. We also want to design a *feedback benchmark* to assess the quality of generated feedback. Through a benchmark we can measure the evolution of a tutor and compare it to other tutors that are assessed using the same benchmark.

ACKNOWLEDGMENTS

We thank Alex Gerdes and Josje Lodder for commenting on this paper.

REFERENCES

John Anderson and Edward Skwarecki. 1986. The automated tutoring of introductory computer programming. *Commun. ACM* 29, 9 (1986), 842–849.

Jens Bennedsen and Michael Caspersen. 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin* 39, 2 (June 2007), 32–36.

Jens Bennedsen and Michael Caspersen. 2008. Exposing the Programming Process. In *Reflections on the Teaching of Programming*, Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.). LNCS, Vol. 4821. Springer, 6–16.

Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. 2009. When are two algorithms the same? *Bulletin of Symbolic Logic* 15, 2 (2009), 145–168.

Nigel Bosch, Sidney Mello, and Caitlin Mills. 2013. What Emotions Do Novices Experience during Their First Computer Programming Learning Session? In *Proceedings AIED 2013: Artificial Intelligence in Education*. LNCS, Vol. 7926. Springer, 11–20.

Tonci Dadic, Slavomir Stankov, and Marko Rosic. 2008. Meaningful learning in the tutoring system for programming. In *Proceedings ITI 2008: the 30th International Conference on Information Technology Interfaces*. 483–488.

Stephen Davies, Jennifer Polack-Wahl, and Karen Anewalt. 2011. A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings SIGCSE 2011: the 42nd ACM technical symposium on Computer science education*. ACM, 625–630.

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. 2012. Teachers and students in charge — Using annotated model solutions in a functional programming tutor. In *Proceedings EC-TEL 2012: the 7th European Conference of Technology Enhanced Learning*. LNCS, Vol. 7563. Springer, 383–388.

George Goguadze. 2011. *ActiveMath generation and reuse of interactive exercises using domain reasoners and automated tutorial strategies*. Ph.D. Dissertation. Universität des Saarlandes, Germany.

John Hattie and Helen Timperley. 2007. The power of feedback. *Review of Educational Research* 77, 1 (2007), 81–112.

Bastiaan Heeren and Johan Jeuring. 2014. Feedback services for stepwise exercises. *Science of Computer Programming* 88 (Aug. 2014), 110–129.

Bastiaan Heeren, Johan Jeuring, and Alex Gerdes. 2010. Specifying Rewrite Strategies for Interactive Exercises. *Mathematics in Computer Science* 3, 3 (March 2010), 349–370.

Jun Hong. 2004. Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies* 61, 4 (Oct. 2004), 505–534.

Johan Jeuring, Alex Gerdes, and Bastiaan Heeren. 2012. A programming tutor for Haskell. In *CEFP 2011: Central European Functional Programming School*. LNCS, Vol. 7241. Springer, 1–45.

Wei Jin, Tiffany Barnes, and John Stamper. 2012. Program representation for automatic hint generation for a data-driven novice programming tutor. In *Proceedings ITS 2012: Intelligent Tutoring Systems*. LNCS, Vol. 7315. Springer, 304–309.

Wei Jin, Albert Corbett, Will Lloyd, Lewis Baumstark, and Christine Rolka. 2014. Evaluation of Guided-Planning and Assisted-Coding with Task Relevant Dynamic Hinting. In *Proceedings ITS 2014: Intelligent Tutoring Systems*. Springer, 318–328.

Hieke Keuning. 2014. *Strategy-based feedback for imperative programming exercises*. Master's thesis. Open Universiteit Nederland. http://hdl.handle.net/1820/5388

Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In *Proceedings ITiCSE 2005: the 10th annual SIGCSE conference on Innovation and technology in computer science education*. ACM, 14–18.

Nguyen-thinh Le and Niels Pinkwart. 2014. Towards a Classification for Programming Exercises. In *Proceedings AIEDCS 2014: the Second Workshop on AI-supported Education for Computer Science*. 51–60.

Michael McCracken, Vicki Almstrum, and Danny Diaz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin* 33, 4 (2001), 125–180.

Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments* 4, 2 (Jan. 1994), 140–158.

Kelly Rivers and Kenneth Koedinger. 2013. Automatic Generation of Programming Feedback: A Data-Driven Approach. In *Proceedings AIEDCS 2013: the First Workshop on AI-supported Education for Computer Science*. 50–59.

Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13, 2 (2003), 137–172.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings PLDI 2013: the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 15–26.

Elliot Soloway and James Spohrer. 1989. *Studying the novice programmer*. Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Edward Sykes and Franya Franek. 2003. A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java (TM). In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education*. 78–83.

Christopher Watson and Frederick Li. 2014. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. ACM, 39–44.

Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (April 1971), 221–227.

Songwen Xu and Yam San Chee. 2003. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering* 29, 4 (April 2003), 360–384.