Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

Detecting Code Evolution in Programming Learning

Thais Castro[1,2], Hugo Fuks[1], and Alberto Castro[2]

[1] Department of Informatics (PUC-Rio)
R. Marquês de S. Vicente, 225 RDC – Gávea – 22453-900 – Rio de Janeiro – RJ – BR
[2] Computer Science Department (UFAM)
Av. Gal. R. O. J. Ramos, 3000 – 69077-300 – Manaus – AM – BR
{tcastro,hugo}@inf.puc-rio.br, albertoc@dcc.ufam.edu.br

Abstract. This article describes a research in code evolution for programming learning that has as its main components the identification of the specific demands for detecting code evolution in the context of programming learning, and the identification of an initial set of programming learning supporting strategies. The findings within the knowledge represented in the codes allowed us to carry out a deeper analysis concerning the inferences internal to the code strategy, which is especially useful for exploring the supporting strategies and the techniques students normally use for their knowledge acquisition in programming.

Keywords: Supporting Strategies for Code Evolution, Programming Learning, Knowledge Representation.

1 Introduction

The search for knowledge about programming skills is recurrent in this research area, starting with Dijkstra [5] in his work "On the Teaching of Programming" and in Weinberg [13], which addresses the psychological aspect of programming learning. Although there is a lot of research on the subject, the milestones of programming learning are not fully elicited yet and that is why it is so difficult to understand which elements contribute to the knowledge acquisition in programming.

In the context of programming learning, in which the students are starting to acquire knowledge in programming, a possible way to be aware of this process is by tracking students' code evolution and categorizing the changes made from one version to the next.

This paper aims at finding out how to join simple techniques and tools from knowledge representation, in order to reduce teacher's work load, improving the feedback and support to programming learners, making the programming learning process more explicit.

In order to address the issues aforementioned, the following subsection contextualizes this work. Section 2 presents a bibliographical review on knowledge acquisition in programming. As a result of that review and its analysis in the context of programming learning, Section 3 presents a deeper view in the inferences internal to the code, concerning code evolution. Finally, Section 4 explorers the analysis made in the previous section, relating it to the development and use of a tool to support code evolution.

G. Zaverucha and A. Loureiro da Costa (Eds.): SBIA 2008, LNAI 5249, pp. 145–156, 2008.

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

1.1 Programming Learning Supporting Strategies

In the context of programming learning, we have been conducting a series of case studies, since 2002 [2], to find out what kind of knowledge first year undergraduate students need for acquiring programming skills. The introductory course where these studies were conducted is based on a methodology in which the focus is in problem solving and coding in Haskell.

Based on these previous results, literature (Section 2) and interviews with experts in programming learning, we have identified some supporting strategies tailored to help the teacher in her search for coding improvement evidences.

In order to find out what milestones are most relevant in programming learning, it is important to extract information from every activity proposed during the introductory course, inferring the knowledge acquired in each programming topic. The coding improvement evidences will depend on the methodology adopted. Following, the improvement evidences are presented based on the assumption that the introductory course where this study takes place proposes activities as coding and conversation records.

1. Code evolution
   a. A quantitative analysis for each solution
   b. Inferences internal to the code
   c. Inferences external to the code
2. Conversation follow-up
3. Conversation report
4. Use of search and inference resources to follow the transition from individual to group solution

This work is part of a research project, which aims at investigating and refining the coding improvement evidences mentioned above. This article deals with the code evolution supporting strategy (1b), specifically with the inferences internal to the code. The other supporting strategies will be discussed in future works, based mainly on [6] and [11].

2 Bibliographical Review

By understanding which cognitive processes are involved in knowledge acquisition in programming, the practices used by each learner become evident and the knowledge generated can be reused in programming education for beginners. There are some works, summarized in the paragraphs below, which use different techniques and methods to elicit elements that may indicate an increase in the knowledge acquisition in programming.

In the article described in [9], a survey on the cognitive aspects involved in the acquisition of the skills needed for programming is carried out. Based on this survey the authors discuss the sequencing of the curriculum in introductory courses as well as the interdependence among the concepts. Such reflection does a better job of organizing introductory courses but does not guarantee that the students will learn more, once the aspects found were not elicited by means of an experiment.

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

In another work, described in [10], a study is conducted which objective is to elicit the manner in which expert programmers organize their knowledge on programming in comparison with beginners. The results were reported in quantitative terms, the most experienced students getting higher marks using the card sorting technique to measure their programming skills. However, there is no evidence that this technique helps in the knowledge acquisition.

Most of the articles, such as [4] and [8], attempt to elicit the skills needed for learning programming either through the discussion on models for programming understanding by relating them to models in the domain of reading [4], or through the assessment of the cognitive capability of storing new information in memory [8].

Problem solving is related to some investigations, such as those described in [1] and [7], that use the clinical method proposed by Piaget and explained in [3] to elicit aspects relevant for knowledge acquisition in programming. This approach is closer to what we propose, but in our work collaboration is a means for learning.

3 Inferences Internal to the Code Strategy

Observing code, we could infer three categories of changes found in code evolution. Along with examples in Haskell, we provide Prolog fragments that capture the example's code changes.

3.1 Syntactic

E.g. indentation, insertion and characters deletion. It aims at capturing changes made to make it correctly interpreted by the Haskell interpreter Hugs, suggesting exhaustive trial and error attempts in order to find a solution.

3.1.1 Indentation

e.g. indentation of the function's second line, positioning the `+ x/y` code after the equality symbol, in order to be viewed as a single line by the Hugs. The following example shows a needed adjustment (from Program A to Program B) to Haskell's specificities, without changing program representation:

Program A

```
f x y = x*y
+ x/y
```

Program B

```
f x y = x*y
        + x/y
```

A way to automatically detect the extra spaces added to Program B is transforming terms (every character in the program) into list elements. Then, it is possible to compare each list element, and infer what type of change occurred. There is a Prolog code example below, which illustrates how to compare any two programs.

```
sameF(T1,T2) :-
        T1 =.. [FunctionName|BodyList],
        T2 =.. [FunctionName|BodyList].
```

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

3.1.2 Insertion, Change or Character Deletion

e.g. misuse of , instead of ; or vice versa and also the absence of some connective symbol, as an arithmetic operator. The following example shows a needed alteration in the solution as well as a needed adjustment to Haskell's specificities:

Program A

```
f (x,y) = (x;1) + (2,y)
```

Program B

```
f (x,y) = (x,1) + (2,y)
```

In Program B, it was necessary to change the symbol used in the tuple (x,1).

A way to automatically detect such changes involves the implementation of a DCG (Definite Clause Grammar), capable of understanding Haskell code. In the Prolog program presented below there is a fragment of a Haskell grammar, based on the predicate pattern/1, which detects the aforementioned syntactic errors. The other predicates namely, expr/1, variable/1 and pattern_seq/1, are an integral part of Programs A and B. The predicates sp/0 and optsp/0 match for necessary and optional spaces in the code.

```
decls(Z) --> pattern(X), optsp, "=", optsp, expr(Y),
{Z=..[f,head(X),body(Y)]}.

decls(Z) --> pattern(X), optsp, "=", optsp, decl_seq(Y),
{Z=..[f,head(X),Y]}.

decls(Z) --> variable(X1), sp, patternseq(X2), optsp, "=",
optsp, expr(Y),  {X=..[head,X1|X2], Z=..[f,X,body(Y)]}.

decls(Z) --> variable(X1), sp, pattern_seq(X2), optsp, "=",
optsp, decl_seq(Y), {X=..[head,X1|X2], Z=..[f,X,Y]}.

decl_seq(Z) --> declseq(X), {Z=..[body|X]}.

declseq(Z) --> expr(X), {Z=[X]}.

declseq(Z) --> expr(X), ";", declseq(Y), {Z=[X|Y]}.
```

3.1.3 Inclusion of a New Function

e.g. adding a new function to the program, in order to incrementally develop and test the whole solution. This is normally found in student's codes and indicates the use of a good programming practice, emphasized in the introductory course, based on the divide and conquer strategy.

A way to automatically detect such changes involves the implementation of the same type of rule as the one presented for the previous change, shown in 3.1.2.

3.2 Semantics

e.g. data structures modification; changing from tuples to lists; inclusion of a recursive function; and bugs correction. These changes directly affect function evaluation, resulting in a wrong output.

3.2.1 Changing from Independent Variables to Tuples

e.g. a second degree equation could be calculated in two different ways: using independent roots or using tuples. Both representation methods use the same arguments, although in the former there is a need for duplicating the function definition. Example:

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

Program A

```
r_x a b c = (-b) + e / 2*a
            where
              e = sqrt(b^2-4*a*c)
r_y a b c = (-b) -  e / 2*a
            where
              e = sqrt(b^2-4*a*c)
```

Program B

```
rs a b c = (x,y)
           where
            x = ((-b) + e)/2*a
            y = ((-b) - e)/2*a
            e =  sqrt(b^2-4*a*c)
```

In Program A two functions are used to solve the second degree equation. Besides the programmer extra effort in repeating the functions, if these functions were too large, code readability would be affected. Program B presents a more elegant and accurate solution to the problem. By directing the output to a tuple (x,y) and defining locally the formula it becomes more readable, saving the programmer from an extra effort.

A way to automatically detect such changes also involves the implementation of a DCG. This DCG consists of transforming the terms into lists and comparing each set in both programs. As illustrated in the code below, it is possible to identify what structure has changed between code versions. There is a Prolog fragment example below, which suits both the semantic changes presented in this subsection and in 3.2.2.

```
compare2(T1,T2,Subst) :-
          T1 =.. [f,H1,B1|[]],
          T2 =.. [f,H2,B2|[]],
          H1 =.. [head|Head1],
          B1 =.. [body|Body1],
          H2 =.. [head|Head2],
          B2 =.. [body|Body2],
          comp2(Head1,Head2,[],Subst1),
          comp2(Body1,Body2,Subst1,Subst).
```

In the above fragment, a substitution list has been constructed, making it possible the comparison between T2 and T1, T2 = subst(T1).

3.2.2 Changing from Tuples to Lists

e.g. if a solution could be represented using tuples or lists, it would be better to use lists instead of tuples for handling large collections. In the following example, it is shown the change from tuples (Program A) to lists (Program B), which is a desirable technique to improve data handling:

Program A

```
composed a b = (a, b, b+a)
```

Program B

```
composed a b = [a, b, b+a]
```

A way to automatically detect such changes also involves the implementation of a DCG. This directly identifies changes in structure, between two code versions.

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

Moreover, using the Prolog built-in predicate ':=/2', changes such as the exemplified above are easily detected, as in the Prolog fragment shown in 3.1.1.

3.2.3 Inclusion of a Recursive Function

e.g. using a recursive function instead of an iterative one. In most cases, in functional programming, recursion suits better and it is often more accurate. That is why usually the teacher would like to know whether recursion has been correctly understood. The following example shows the representation of the solution for the factorial function, where Program A presents an iterative solution and Program B a recursive one:

Program A

```
fat n = if n==0 then 1
        else product [1..n]
```

Program B

```
fat n = if n==0 then 1
        else n * fat(n - 1)
```

A way to automatically detect such changes involves the implementation of a rule for detecting whether a program has been written using a recursive function. This involves the identification of specific terms in a code version.

3.2.4 Bugs Correction

e.g. small changes made to formulae' or functions' definitions, consisting of simple bug correction in a trial and error fashion. When a function does not work, students usually make consecutive changes without stopping to reason, ignoring this way the process used in problem solving techniques. The following example shows the inclusion of an IF-THEN-ELSE conditional structure:

Program A

```
fat n = n * fat (n - 1)
```

Program B

```
fat n = if n==0 then 1
        else n * fat(n - 1)
```

A way to automatically detect such change involves the implementation of a pattern matching predicate, which checks whether certain desired structure, as the one shown above, is part of the next code version. This is especially useful for checking whether the students are taking some time reasoning about the problem, before start coding.

3.3 Refactoring

e.g. changes which objective is to improve the code, according to known software engineering quality metrics. The changes presented below were extracted from the Haskell refactoring catalog [12]. We have adapted the catalog in order to fit it into two categories: (i) data structure, which affects data representation, and consequently all functions involved by the refactoring (e.g. algebraic or existential type, concrete to abstract data type, constructor or constructive function, adding a constructor); and (ii) naming, which implies that the binding structure of the program remains unchanged after the refactoring (e.g. adding or removing an argument, deleting or adding a definition, renaming). Most refactoring are too complex for beginners, so when that is the case, it is not followed by an example.

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

3.3.1 Algebraic or Existential Type

e.g. a data type, that could be recursive, is transformed into an existential type, in the same way as an object oriented data representation. This transformation is unified applying binary functions to the data type.

3.3.2 Concrete to Abstract Data Type

e.g. it converts a concrete data type into an abstract one with hidden constructors. Concrete data types provide direct representations for a variety of data types, and pattern matching syntax in data types constitutes an intuitive style of programming. The disadvantage of it is the resulting rigidity: the data representation is visible to all type clients. On the other hand, that is especially useful in order to turn the representation into an abstract one, by a modification or data type extension.

3.3.3 Constructor or Constructor Function

e.g. a data type constructor must be defined in terms of its constructors. There are complementary advantages in keeping the constructor and in eliminating it for its representation.

3.3.4 Adding a Constructor

e.g. new patterns are aggregated to case analysis that recurs over the data type in question. The positioning of the new pattern is allocated, preserving the order in which the new data type was defined.

3.3.5 Adding or Remove an Argument

e.g. adding a new argument to a function or constant definition. The new argument default value is defined at the same level as the definition. The position where the argument is added is not accidental: inserting the argument at the beginning of the argument list, implies that it can only be added to partial function applications. The following example shows the inclusion of an undefined function:

Program A	**Program B**
`f x = x + 17`	`f y x = x + 17`
	`f_y = undefined`
`g z = z + f x`	`g z = z + f f_y x`

A way to automatically detect such change is by comparing two functions and checking whether the function names or function parameters have changed. The following Prolog fragment shows a simple change in the function parameters, suggesting a solution refinement. It exemplifies a pattern matching, keeping the data in a substitution list. The code presented below is similar to the one presented in 3.2.1, with a few simplifications.

```
compare1(T1,T2,Subst) :-
        T1 =.. [FunctionName|List1],
        T2 =.. [FunctionName|List2],
        comp1(List1,List2,[],Subst).
```

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

3.3.6 Deleting or Adding a Definition

e.g. deleting a definition that has not been used. The following example shows the deletion of the function `table`:

Program A	Program B
`showAll = ...`	`showAll = ...`
`format = ...`	`format = ...`
`table = ...`	

A way to automatically detect such change is by using the same Prolog fragment presented in 3.2.1, `compare2/3`, which also checks whether a function name has been modified or deleted.

3.3.7 Renaming

e.g. renaming a program identifier, which could be a valued variable, type variable, a data constructor, a type constructor, a field name, a class name or a class instance name. The following example shows the change in the function named `format` to `fmt`:

Program A	Program B
`format = ... format ...`	`Fmt = ... fmt ...`
`entry = ... format ...`	`entry = ... fmt ...`
`table = ...`	`table = ...`
`where`	`where`
`format = ...`	`format = ...`

A way to automatically detect such change is by using a predicate as `compare/2` (3.2.1). The potential of this change, besides readability, is especially useful for the teacher for detecting plagiarism. If the teacher is aware of such change and notices no improvement in the software quality metrics, she may look for a similar code in someone else's code versions and look for plagiarism.

4 Exploring the Supporting Strategies

The supporting strategies described above were implemented in a tool, named AcKnow, which aims at analyzing and categorizing student's code evolution. AcKnow takes as input codes from students that were indicated by a quantitative analysis made by a version control tool named AAEP [1]. These indications are based on the number of versions that each student did. When this number significantly differs from the normal distribution, i.e., the student is identified as a special case and her codes are submitted to AcKnow. Then, based on AcKnow's analysis of each pair of versions, the teacher may do a qualitative analysis, providing feedback relative to that student's current milestone. Figure 1 shows the AcKnow's architecture, including its subsidiary parts.

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

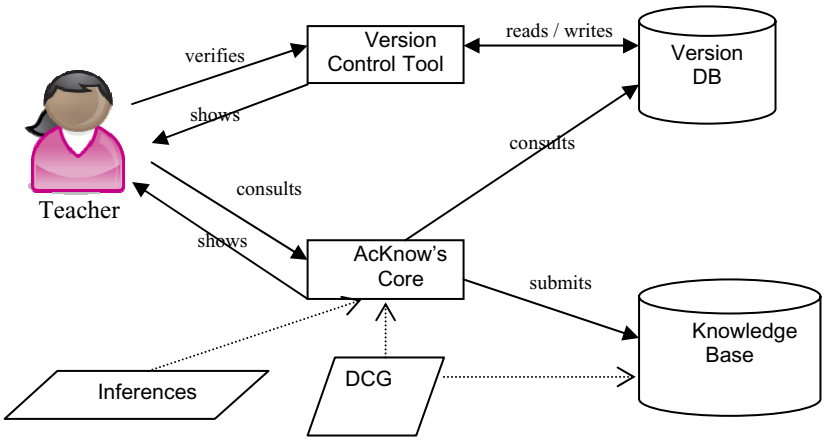

Fig. 1. AcKnow's Architecture

AcKnow classifies as syntactic changes modifications such as indentation, addition or removal of comma or semi-colon, spacing between characters and addition or removal of parentheses. The semantic changes are those related to function evaluation. For example, changes in data structures and operations in functions are classified in this category. Furthermore, refactoring, in this context, are special cases of semantic changes aimed at improving code quality according to software quality metrics. Currently, AcKnow partially uses the Haskell catalogue of refactoring [12].

Table 1. Jane Doe's History

Version	Interval	Category
1	0	Syntactic
2	Same minute	Syntactic
3	1 minute	Syntactic
4	1 minute	Syntactic
5	8 minutes	Refactoring
6	171 minutes	Semantic
7	44 hours	Refactoring

AcKnow has been used to analyze the development history of the students of the introductory programming course. That will be exemplified here by Jane Doe's code evolution. In Table 1 the categories of changes in Jane Doe's versions are presented, associated with the time intervals between them.

Table 2 shows an instantiation of the rules, applied to Jane Doe's code versions for the following problem: Given a line segment r, through two points, a and b, located in the Cartesian plane's first quarter, and any point p, determine whether p belongs to the segment or to the continuity of r or is located above or under the line. The texts inside the boxes indicate the changes made from one version to the next. After each version there is a description of Jane Doe did to the code.

Based on the code version's analysis presented next, the teacher observes that Jane Doe correctly uses the incremental development of solutions, using the divide and

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

conquer strategy visible on almost every code versions, which is an indicative of a sound understanding of the problem solving method presented in classroom. In addition, she uses refactoring (the renaming and the addition of an extra line at the end of the code), which is an indicative of a concern with software quality metrics. Following we present some change examples:

(1)
```
pertseg x1 y1 x2 y2 x3 y3 = if x1>0 && y1>0 && x2>0 && y2>0
                              then if seg x1 y1 x2 y2 x3 y3
                                    then "p belongs to AB"
                                    else if det==0
                                          then "p belongs to r"
                                          else if det>0
                                                then "p is above r"
                                                else "p is below r"
                              else "p is not in the 1st quarter"

  seg x1 y1 x2 y2 x3 y3 = (cresc x1 y1 x2 y2 x3 y3||decresc x1 y1 x2 y2
x3 y3||conth x1 y1 x2 y2 x3 y3||contv x1 y1 x2 y2 x3 y3)
```

(2)
```
det x1 y1 x2 y2 x3 y3 = x1*y2+x2*y3+x3*y1-(x3*y2)-(x2*y1)-(x1*y3)
```

As it is shown in the above codes (1) and (2), in the first version (1), she partially solves the problem, but it is not accurate enough. Then, in the second version (2), she keeps the previous code and adds a new line, describing another function.

(3)
```
cresc x1 y1 x2 y2 x3 y3 = x3>x1 && x3<x2 && y3>y1&& y3<y2
decresc x1 y1 x2 y2 x3 y3 = x3>x1 && x3<x2 && y3<y1 && y3>y2
conth x1 y1 x2 y2 x3 y3 = x3>x1 && x3<x2 && y3==y1 && y3==y2
contv x1 y1 x2 y2 x3 y3 = x3==x1 && x3==x2 && y3>y1 && y3<y2
```

As it is shown in the above code (3), in the third version, she adds the functions needed to establish the line's condition. In the forth version, she attempts to modify the last code line, but she gives up after adding and removing a space. In the fifth version, she renames `pertseg` with `segment`.

(4)
```
  segm (x1,y1) (x2,y2) (x3,y3) = (cresc (x1,y1) (x2,y2) (x3,y3)||decresc
(x1,y1) (x2,y2) (x3,y3)||conth (x1,y1) (x2,y2) (x3,y3)||contv (x1,y1)
(x2,y2) (x3,y3))
  det (x1,y1) (x2,y2) (x3,y3) = x1*y2+x2*y3+x3*y1-(x3*y2)-(x2*y1)-
(x1*y3)
```

As it is shown in the above code (4), in the sixth version, she has a cognitive jump, changing `seg x1 y1 x2 y2 x3 y3` for `segm (x1,y1) (x2,y2) (x3,y3)` and `det x1 y1 x2 y2 x3 y3` for `det (x1,y1) (x2,y2) (x3,y3)` in order to use tuples. In the seventh version, she adds a new line at the end of the code.

Another relevant piece of information related to Table 1 and the above described code's evolution is that between versions 5 and 6 Jane Doe solved another 5 problems from the same list of exercises. Some of these exercises asked for the use of tuples in their solution. After that, she returned to the problem in question and submitted version 6 using tuples in its solution. So, Jane Doe had the winning insight when solving another problem.

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

The course where AcKnow was used had 100 students enrolled. The statistical method used by AAEP indicated an average of 3 students per problem as special cases. Besides Jane Doe, the other students presented a similar pattern of changes, carrying out initially many syntactic changes within a short time interval followed by one or two semantic changes within a longer time interval; only few of them (4 in 2 different exercises) carried out refactoring. When refactoring occurred, it was only identified in the very last versions.

5 Conclusion

This article proposes an initial set of programming supporting strategies which inspects students' code evolution in order to detect the following situations: expertise levels, difficulties, cognitive jumps in programming; and bottlenecks in solution planning.

We understand that having access to more code from future case studies, it will be possible to identify other strategies being used by these students. This way we will be moving towards a more complete set of programming supporting strategies and, consequently, their formal representation for the sake of knowledge acquisition research in programming.

Finally, in order to provide the teacher with a tool to follow up the programming learning development of her students, AcKnow was implemented. At this moment of time it does not support all the supporting strategies. For the sake of proof of concept the programming learning development of Jane Doe, a fictitious name for a real student, was analyzed.

Acknowledgement

This project is partially sponsored by the Brazilian Ministry of Science and Technology through the CTInfo project grant nº 550865/2007-1. Hugo Fuks is sponsored by CNPq individual grant nº 301917/2005-1 and he also receives grant from the FAPERJ project "Cientistas do Nosso Estado". Thais Castro receives a CNPq PhD grant. This research also receives resources from project ColabWeb – Proc.553329/2005-7, CNPq/CT-Amazônia n.27/2005.

References

1. Almeida Neto, F.A., Castro, T., Castro, A.N.: Utilizando o Método Clínico Piagetiano para Acompanhar a Aprendizagem de Programação. In: XVII Simpósio Brasileiro de Informática na Educação, 2006, Brasília. Simpósio Brasileiro de Informática na Educação, vol. 17, pp. 184–193. Gráfica e Editora Positiva Ltda, Brasília (2006)
2. Castro, T., Castro, A., Menezes, C., Boeres, M., Rauber, M.: Utilizando Programação Funcional em Disciplinas Introdutórias de Computação. In: XXII Congresso da Sociedade Brasileira de Computação / X Workshop sobre Educação em Computação, 2002, Workshop sobre Educação em Computação, vol. 4, pp. 157–168. SBC, Porto Alegre (2002)
3. Delval, J.: Introdução à Prática do Método Clínico. Artmed Publisher (2002) ISBN 8536300132

Published in the Proceedings of the 19th Brazilian Symposium on Artificial Intelligence, Salvador, Brazil, October 26-30, 2008.
Series: Lecture Notes in Computer Science , Vol. 5249
Sublibrary: Lecture Notes in Artificial Intelligence. ISBN: 978-3-540-88189-6, pp. 145-156

4. Détienne, F.: What Model(s) for Program Understanding? In: The Proceedings of the Colloque Using Complex Information, UCIS 1996 (1996)
5. Dijkstra, E.: On the Teaching of Programming, i.e. on the Teaching of Thinking. In: Selected Writings on Computing: A Personal Perspective. Springer, Heidelberg (1982)
6. Gerosa, M.A., Pimentel, M., Fuks, H., Lucena, C.J.P.: Development of Groupware based on the 3C Collaboration Model and Component Technology. In: Dimitriadis, Y.A., Zigurs, I., Gómez-Sánchez, E. (eds.) CRIWG 2006. LNCS, vol. 4154, pp. 302–309. Springer, Heidelberg (2006)
7. Gomes, A., Mendes, A.J.: Problem Solving in Programming. In: The Proceedings of PPIG as a Work in Progress Report (2007)
8. Mancy, R., Reid, N.: Aspects of Cognitive Style and Programming. In: The Proceedings of the 16th Workshop of the Psychology of Programming Interest Group, Carlow, Ireland (2004)
9. Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., St. Clair, C., Thomas, L.: A Cognitive Approach to Identify Measurable Milestones for Programming Skill Acquisition. In: The Proceedings of ITiCSE. ACM, New York (2006)
10. Murphy, L., McCauley, R., Westbrook, S., Fossum, T., Haller, S., Morrison, B., Richards, B., Sanders, K., Zander, C., Anderson, R.E.: A Multi-Institutional Investigation of Computer Science Seniors' Knowledge of Programming Concepts. In: The Proceedings of SIGCSE. ACM, Missouri (2005)
11. Pimentel, M., Escovedo, T., Fuks, H., Lucena, C.J.P.: Investigating the assessment of learners' participation in asynchronous conference of an online course. In: 22nd ICDE - World Conference on Distance Education: Promoting Quality in On-line, Flexible and Distance Education (CD-ROM), September 3-6. ABED, Rio de Janeiro (2006)
12. Thompson, S., Reinke, C., Li, H.: Refactoring Functional Programs. Final Report GR/R75052/01 (2006), `http://www.cs.kent.ac.uk/projects/refactor-fp`
13. Weinberg, G.M.: The Psychology of Computer Programming. Computer Science Series, F9264-000-4. Litton Educational Publishing, USA (1971)