EMPIRICAL STUDY OF NOVICE ERRORS AND ERROR PATHS IN OBJECT-ORIENTED PROGRAMMING

Marie-Helene Ng Cheong Vee
SCSIS, Birkbeck,
University of London
Malet Street
London WC1E 7HX, UK
marie-helene@dcs.bbk.ac.uk
http://www.dcs.bbk.ac.uk/~gngch01

Bertrand Meyer
Chair of Software Engineering
ETH Zurich
ETH Zentrum, 8092 Zurich,
Switzerland
Bertrand.meyer@inf.ethz.ch
http://se.ethz.ch/~meyer/

Keith L. Mannock
SCSIS, Birkbeck,
University of London
Malet Street
London WC1E 7HX, UK
keith@dcs.bbk.ac.uk
http://www.dcs.bbk.ac.uk/~keith

ABSTRACT

What kind of errors do beginners make? Objective answers to this question are essential to the design and implementation of curricula that don't just reflect the educators' theories but succeed in conveying a course's topics and skills to the students. In the context of a new introductory programming course based on "inverted curriculum" ideas, and taking advantage of our ability to instrument the compiler, we performed automatic analysis of the — sometimes contorted — paths students actually take to solve programming exercises on their own. The results, collected from three different groups of students across two unrelated universities, included a number of surprises. These findings will help improve future sessions of the course, and are being used in the design and implementation of an Intelligent Tutoring System.

Keywords

Errors, Paths, Novices, Inverse Curriculum, Data.

1. INTRODUCTION

The best educational theories are only as good as the students' success with the subject matter. This is particularly true with an introductory programming course, whose goal is to make students comfortable with the basics of software development; the results are difficult to gauge objectively. Various methods used in the past involved interviews, "talk-alouds" and observing students while they solve problems in a "looking over the shoulder" manner. Although they provide some insight, these techniques are often tedious to apply and susceptible to observer bias. To obtain a more objective assessment, we automated data collection, with the help of the compiler, by storing "snapshots" of student programs at every compilation. The resulting interaction logs allow us to explore the behaviour of students while they solve programming tasks, usually outside of any human supervision. The analysis of the data gave us insights into helping students learn programming. These insights have already led to improvements to the next iteration of the course and will inform the design of the Intelligent Tutoring System under development.

Section 2 briefly presents related work. Section 3 describes the courses and the organisation of the study. Section 4 analyses some of the errors obtained from the interaction logs. Section 5 generalises this analysis to the concept of "error path" and proposes a *notion of behaviour* pattern. Section 6 concludes with a brief discussion of future work.

2. RELATED WORK

Studies similar in their scope to ours were carried out three decades years ago for imperative languages [4]. A more recent study [2] used Java and the BlueJ environment [3]. It focused on analyzing novice compilation behaviors by looking at features such as frequency of compilations, compilation times and others. Although the author's stated goal — to determine if novices have different characteristic compilation behaviors — is somewhat different from ours, he does provide a list of common errors, most of them syntactic.

3. THE STUDY

3.1 The Course

In October 2003, ten years after the first papers proposing an Inverted Curriculum for teaching introductory programming [5], ETH Zurich started applying these ideas to the *Introduction to Programming* course [7], part of the first year of the computer science program.

Instead of a bottom-up or top-down approach, the Inverted Curriculum, also known as “consumer-to-producer strategy” or “outside-in”, is the process of progressively opening “black boxes” to unveil the underlying principles of higher-level concepts gradually. The “black boxes” are libraries of reusable components. This approach enables beginning students to learn both how to re-use libraries as in real-life and how to build reliable software. In addition to the sense of achievement, motivation is improved from working with a real application: It is fun to play with something that works, is visible and non-trivial and there is greater opportunity for active learning.

In building such a course [1], the ETH group devised: (1) lectures slides and exercises; (2) A new online textbook called “Touch of class"[8]; (3) The software: *Traffic* library and *Flat-hunt* game.

In all the courses used for this study students learn programming using Eiffel, chosen since it is a pure OO language with clear syntax, support for *Design by Contract* and other mechanisms, making it a suitable choice as a teaching language.

3.2 Student Groups

The data came from two instances of the course, taught with minor variations to two groups of students across two unrelated universities:

- **ETH (Introduction to programming)**

In the 2004/2005 session, 22 out of 180[2] students from the *Introduction to Programming* course [8] at ETH voluntarily participated in the study. In the 2005/2006 session, an average of 64 out of 180[2] students voluntarily participated. The course lasts a semester (14 weeks) with, each week, two 2-hour full-class lectures and 3 hours of tutorials in groups of about 20. In addition to fundamental OOP and procedural concepts such as objects, classes, inheritance, control structures, recursion, students study more advanced topics such as event-driven and concurrent programming and fundamental concepts of software engineering.

- **Birkbeck (MSc part-time and full-time)**

52 out of 75[1] students taking the OOP course in the MSc program at Birkbeck[2] participated in the spring term 2004/2005. We required students to send their logs as part of the coursework submission although they were not penalised for not doing so. The course lasts a term (11 weeks). We taught OOP in Eiffel, including all the basic concepts and a few advanced ones (genericity with inheritance, exception handling) in the first part of the course; the remaining time was used to teach Java.

Most of the Birkbeck students are “mature” students, many already employed full-time in the IT industry (this explains their request for inclusion of some Java training). All of them did an *Introduction to Programming* module in C++ prior to the OOP module. By contrast, almost all ETH students are around 20 years old and fresh out of high school; they have varying exposure to IT and programming, with a fair number[3] being complete novices.

While teaching styles differed slightly between the two groups and instructors were obviously different in the two institutions, the teaching material was kept as similar as possible. The assignments were drawn from the same collection of exercises, but due to time constraints the Birkbeck students had fewer of them; the data analysis used the same seven exercises in all cases.

3.3 Data Collection

To collect interaction logs, we benefited from the “Melting Ice Technology" of the free EiffelStudio[4] environment used by the students.

This incremental compilation mechanism allows speedy and efficient development by only processing the classes changed since the latest compile step [6]. This feature meant we did not need to make any change to the compiler: participating students simply turned on the option and shared certain files with us. The data saved includes a copy of the program and some information relating to compilation. All such data was treated anonymously, allaying any privacy concerns.

The interaction logs contained a wealth of information. We obtained information such as the errors novices make, their frequency (enabling us to focus on the most acute problems), the amount of time taken to accomplish tasks, the number of compilations, and time between compilations.

[1] All group sizes are approximations because of dropouts and of some re-takes who do not need to submit coursework.

[2] In full-time mode, the degree lasts 1 year and in part-time mode, it lasts 2 years.

[3] In the 2004/2005 group, 17% describe themselves as complete beginners and 31% as having programmed a little bit. The percentages are 18% and 29% respectively in the following year.

[4] http://www.eiffel.com/products/studio

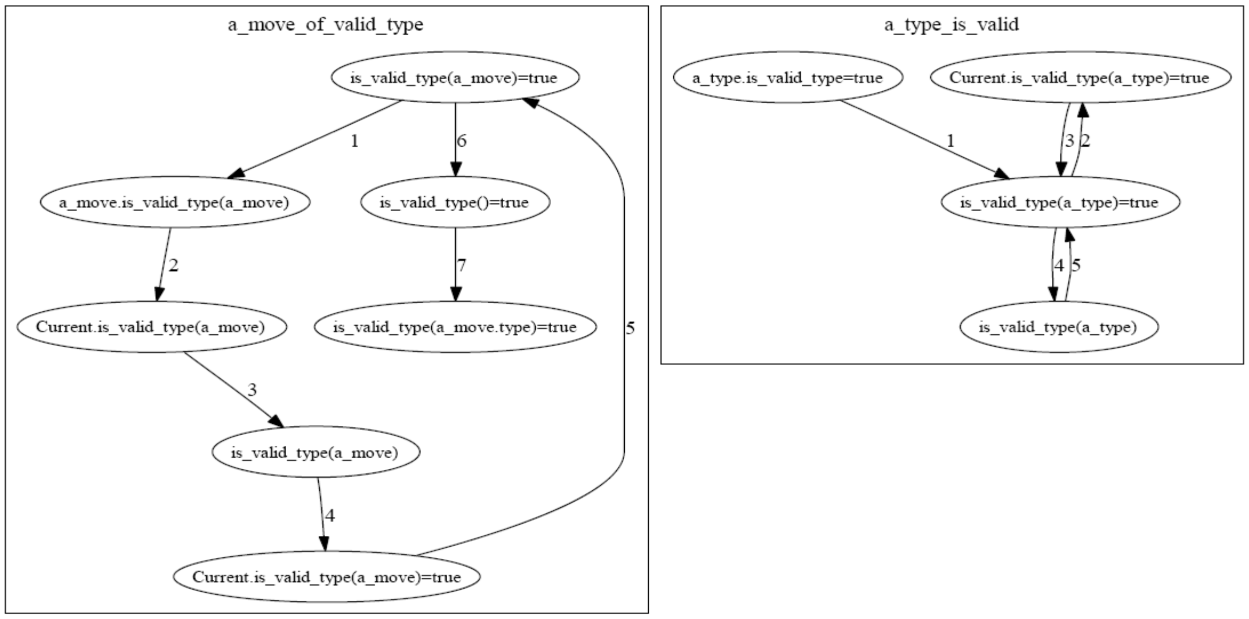
a_move_of_valid_type
is_valid_type(a_move)=true
a_move.is_valid_type(a_move)
is_valid_type()=true
Current.is_valid_type(a_move)
is_valid_type(a_move.type)=true
is_valid_type(a_move)
Current.is_valid_type(a_move)=true
1
2
3
4
5
6
7
a_type_is_valid
a_type.is_valid_type=true
Current.is_valid_type(a_type)=true
is_valid_type(a_type)=true
is_valid_type(a_type)
1
2
3
4
5

Figure 1: Student's path in solving *a_,move_of_valid_type* and *a_type_is_valid*

From the logs we were able to reconstruct scenarios of the student's problem-solving steps until he reaches the final solution. Examples of the reconstruction of such scenarios are shown in Figures 1 and 2 and discussed in Section 5.

4. REVIEW OF ERRORS

We will now examine some of the errors detected by the study, each selected because of some significant property; for example some occur in the work of many students, and some were particularly unexpected. Some of these errors occur repeatedly across the exercises, while others either disappear or occur less often as students progress through the exercises.

- **Extra variables**

Some students used more variables than necessary, in particular in Exercise 2. One subtask of this exercise was the conversion of a temperature provided in Celsius to its Fahrenheit equivalent. Some students wrote code similar to the one below:

```
v := 9/5 * value +32
create fahrenheit.make_with_fahrenheit(v)
Result := fahrenheit
```

They used two variables: one for storing the results of the conversion (*convalue*) and the other for the creation of a new object to represent the newly converted temperature (*fahrenheit*). The one-line solution which does not require declaring any variables is:

```
create  Result.make_with_fahrenheit(9/5  *
value + 32)
```

- **Feature call errors**

Various errors relate to feature calls: omitted target (*f (…)* instead of *x.f (…)*), superfluous target (*Current.f (…)*, where *Current* is redundant or wrong, as detailed in Section 5), wrong target, wrong type or number of actual arguments, calling a non-existent feature.

- **Rewrite instead of reuse**

In the first exercise, students were provided with a very simple feature *is_valid_type* which takes a type of transportation and returns a boolean value depending on whether the provided type of transportation is valid or not. This feature was meant to be used in two of the contracts they had to write. Some students rewrote most (if not all) of the body of *is_valid_type* instead of reusing the feature. In later exercises, some students, it seemed, still had not understood the concepts of modularity and reuse.

- **Inheritance**

The use of inheritance early on was interesting, prior to the concept being introduced in class and only mentioned briefly in an example from the *Traffic* software. We may attribute this to the use of libraries, where students have access to the source code. Many probably looked at them and did some research on more advanced topics. This is part of the reason for using libraries: to enable the more inquisitive and adventurous students to learn on their own, by study and imitation of carefully written software models.

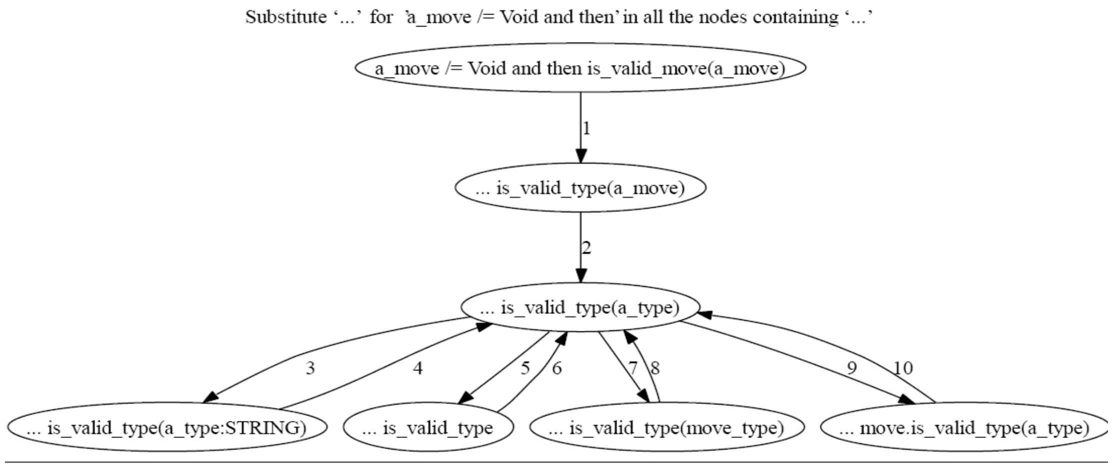
Substitute '...' for 'a_move /= Void and then' in all the nodes containing '...'
a_move /= Void and then is_valid_move(a_move)
1
... is_valid_type(a_move)
2
... is_valid_type(a_type)
3
4
5
6
7
8
9
10
... is_valid_type(a_type:STRING)
... is_valid_type
... is_valid_type(move_type)
... move.is_valid_type(a_type)

Figure 2: First part of paths followed by student solving a_move_of_valid_type

- **Syntactical issues**

The usual novice errors such as simple syntax errors occur, although less than in previous studies thanks to the use of Eiffel with its simple syntax (English keywords, optional semicolons, no "curly braces" and other cryptic symbols), which also makes it easier to analyse these errors. Many of these syntactical errors are simple mistypings. Other are: placing = before < or > in relational operators, forgetting the enclosing double quotes or single quotes for strings and characters respectively, and using semi-colons to separate arguments in a call. This last one may be due to the use of semicolons between formal arguments in feature *declarations*, whereas *calls* use commas for actuals.

- **Type errors**

The most common errors were type errors: wrong type in declaring a variable or argument, assigning to a variable etc.

- **Expressions used as instruction**

Some students did not differentiate between expressions and instructions. This was among the most common errors.

- **Assignment**

Problems with the notion of assignment were apparent when students assigned, for example, *i* to *j* when they meant assigning *j* to *i*. This was trivial to solve in the few cases where it happened. More serious were errors where an entity of some type was assigned to an entity of an unrelated type. Another error, syntactical by nature, is the confusion of assignment and comparison. Although Eiffel's syntax is clear – an equals sign means exactly what = is in mathematics – some students still compared when they meant to assign and vice-versa. This might have occurred because of the influence of other languages. What was unexpected was to find students assigning some value to a function. Additionally, in Eiffel, information hiding principles prohibit one assigning to a feature of a qualified call (as in *x.a := v*) even if the feature "a" is an attribute[5]. Many students made this mistake even though the point was stressed in class.

- **Language of instruction**

The use of English for the ETH course (where it is a foreign language for most students) may have affected the comprehension and completion of the task. One clear example is in Exercise 6 where students had to implement a class FRACTION. Many ETH students used variable names such as *dominator* for *denominator*. This is not an error but one particular student mistook *numerator* for *denominator* and consequently had the wrong algorithm and it took him/her quite some time (62 compilations) before realizing the mistake.

- **Language overlap**

It was obvious that some students in the ETH batch had studied another programming language, to varying degrees, prior to starting the course. The MSc group at Birkbeck had studied C++ and Java before, so it came as no surprise to see some language overlap, especially in terms of syntax. One MSc part-timer even wrote comments showing Java code that he was seemingly converting to Eiffel. Typically, some students would use the keyword *this* instead of *Current* or use logic operators used in languages other than Eiffel, such as != instead of /= for inequality.

[5] In the recent ECMA Eiffel standard (ECMA standard 367:http://se.ethz.ch/eiffel/standard.pdf),such constructs are allowed; they do not denote the direct assignment to an attribute but rather a call to the appropriate "setter"' feature.

5. EXAMPLE PATHS AND BEHAVIOUR PATTERNS

It was very interesting to observe the various strategies and patterns used by novices. Some students were consistent in their ways of solving problems. Some students seemed to use a particular strategy over and over again: for example, the use of backtracking: some students would try something, change it to something else to see how it affects output, then come back to the previous answer and so on; some would make many changes at one go, while others would change one thing at a time.

Exercise 1 provides a good example of this problem-solving style. In this exercise, students have to write two very similar assertions: *a_move_of_valid_type* and *a_type_is_valid*. Many students made similar mistakes in producing these two contracts. Figure 1 illustrates an example of a student using similar "strategies" and thus making similar mistakes in producing these two contracts. This student uses *Current* where it is not necessary, and compares the result of the query *is_valid_type* to true in both assertions.

One student had an interesting technique for solving problems. This student uses a lot of backtracking and was by far the most prolific producer of answers. Figure 2 shows the first part of the path he used to arrive at an answer to exercise 1, the assertion:

```
a_move.type    /=Void    and    then
is_valid_type (a_move.type)
```

The graph separates into two different problem-solving "strategies". The first part of the graph is enough to show how extensively this student explored the possibilities. At some point, he nearly has the answer but cannot find the correct argument to *is_valid_type*; then drops the first part of the answer. What is apparent here is that the student cannot determine the correct argument, and in trying to find it he introduces more mistakes. He seems to be trying various options without really understanding what the problem is and attending to that, which was the only obstacle to a correct answer.

6. CONCLUSION AND FUTURE WORK

The initial results of this study have provided valuable insights into the ways in which students learn to program: the errors they make and the ways in which they overcome them; in this paper we focused on the qualitative rather than quantitative results.

As it can be seen from [2], programming environments for other languages (e.g. Java) can be instrumented for automated data collection (the ease or difficulty of such a task depends on the environment). Although, the language used for a course will influence the results of the data analysis stage and the results will depend on the language's syntax, we believe the thought process of students will vary little. Using the approach highlighted in this paper can therefore help uncover and understand behaviours and error paths irrespective of programming language.

We are deriving formalisms for representing this information and are developing a prototypical intelligent tutoring system based upon the work described in this paper.

7. ACKNOWLEDGEMENTS

We are grateful to the assistants in the "Introduction to Programming course" at ETH for their help in setting up the experiment and for motivating students to participate in it. Many thanks also to Emmanuel Stapf for his help with EiffelStudio. And most of all many thanks to the participants of the study.

8. REFERENCES

[1] *Introduction to Programming*. Retrieved 27 Mar 06 from, http://se.inf.ethz.ch/teaching/ws2004/0001.

[2] Jadud, M. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, **15**(1):25-40,2005.

[3] Kölling, M, Quig, B., Patterson, A. and Rosenberg, J. The BlueJ System and its Pedagogy. Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology, 13(4), 2003.

[4] Litecky, C. and Davis, G. (1976) A Study of Errors, Error-Proneness, and Error Diagnosis in Cobol. Communications of the ACM, 19(1):33-38,1976.

[5] Meyer, B. Towards an OO Curriculum. *Journal of Object-Oriented Programming*, **6**(2): 76-81,1993.

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

[7] Meyer, B. The Outside-In Method of Teaching Introductory Programming. In *Manfred Broy and Alexandr Zamulin(Ed.), Perspective of System Informatics, Proceedings of Fifth Andrei Ershov Conference*, pages 66-78, Novosibirsk, July, 2003. Lecture Notes in Computer Science 2890, Springer-Verlag.

[8] Meyer, B. *Touch of Class – Learning to Program Well*. http://se.inf.ethz.ch/touch/, Online Edition, 2003.