Empirical study of novice errors and error paths

Marie-Hélène Ng Cheong Vee
SCSIS, Birkbeck, University of London
Malet Street, Bloomsbury
London, WC1E 7HX, UK
marie-helene@dcs.bbk.ac.uk

Bertrand Meyer
Chair of Software Engineering
ETH - Zurich
ETH Zentrum, 8092 Zurich, Switzerland
bertrand.meyer@inf.ethz.ch

Keith L. Mannock
SCSIS, Birkbeck, University of London
Malet Street, Bloomsbury
London, WC1E 7HX, UK
keith@dcs.bbk.ac.uk

ABSTRACT

What kind of errors do beginners make? Objective answers to this question are essential to the design and implementation of curricula that do not 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 an analysis of the - sometimes contorted - paths students actually take to solve programming exercises on their own. The results, collected from two different groups of students across two unrelated universities, include a number of surprises; they will help improve future sessions of the course, and are being used in the design and implementation of an Intelligent Tutoring System.

Categories and Subject Descriptors

K.3 [**Computers and Education**]: Miscellaneous; D.2.3 [**Software Engineering**]: Object-oriented programming—*Pedagogy, CS Ed Research, Curriculum Issues*

General Terms

Experimentation

Keywords

Errors, Paths, Novices, Inverse curriculum, Data collection

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 inefficient, tedious, time consuming and not immune 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 behavior of students while they solve programming tasks, usually outside of any human supervision. We particularly focus on compilation errors as a way to infer student behavior and interaction patterns. The analysis of the data gave us insights into helping students learn programming.

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

This article departs from local administrative terminology in the sake of consistency: what is called a "module" at Birkbeck appears here as a "course", etc. For object-oriented terminology, we follow the references used for the course: OOSC [8] and the Touch of class textbook [10].

2. RELATED WORK

Studies similar in their scope to ours were carried out three decades ago for imperative languages [5] [11] [13]. They highlighted and classified various common programming errors pertinent to the imperative paradigm [3].

A more recent study [2] used Java and the BlueJ environment [4]. 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. Moreover, he uses quantitative analysis while we consider qualitative analysis better suited for our purposes.

3. THE STUDY

This section is divided in two parts: a description of the courses that were used as the experimental test bed of our study and a description of the data collection procedure.

3.1 The course

In October 2003, ten years after the first papers proposing an Inverted Curriculum for teaching introductory programming [7], ETH Zurich started applying these ideas to the *Introduction to Programming* course [9], 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 (1) how to re-use libraries as in real-life, (2) 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. There is greater opportunity for active learning.

In building such a course [1], the ETH group devised: (1) Material for the course: lectures slides and exercises; (2) A new textbook called "Touch of class", available online [10]; (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 good choice for teaching, as well as the stamp of practicality provided by use for large industrial applications.

3.2 Student groups

Data for this study came from two instances of the course, taught with minor variations to two groups of students across two unrelated universities:

- **ETH (Introduction to programming)**
  22 out of a group of approximately 100[1] students from the Introduction to programming course [9] at ETH (first year of the Computer Science Bachelor's program) participated in the study. The module lasts a semester (14 weeks). There are two two-hour lectures per week. Tutorials are organized in small groups of 10-20 students twice a week (three hours in total per week). In addition to fundamental OOP and procedural concepts such as objects, classes, inheritance, control structures, recursion, etc., students also study more advanced topics such as event-driven and concurrent programming and fundamental concepts of software engineering. Data collection took place in October 2004.

- **Birkbeck (MS part-time and full-time)**
  52 out of a group of approximately 75[1] students taking the OOP course in the MS program at Birkbeck[2] participated. 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. Data was collected in the spring term 2004/2005.

[1]Group sizes are approximations because of drop-outs 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.

Table 1: Exercises and topics

Exercise	Topic	Additional Notes
1	Design by Contract	Skeleton code + hints provided
2	Object Creation	Code from scratch
3	Refactoring	Uses Traffic/Flat-hunt
4	Control Structures	Uses Traffic/Flat-hunt
5	Control Structures	Code from scratch
6	Inheritance	Code from scratch
7	Inheritance	Uses Traffic/Flat-hunt

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; table 1 shows these assignments and their themes.

The time given to the MS students to solve the exercises was adjusted to take into account the different mode of study. The Birkbeck exercises were graded and contribute towards the final grading of the degree. For the ETH group, these exercises are not graded but students are required to show they have made a reasonable attempt at solving them to be allowed to sit the exams.

3.3 Data collection

3.3.1 *How*

We benefited from the "Melting Ice Technology" of the free EiffelStudio environment used by the students (http://www.eiffel.com/products/studio). This incremental compilation mechanism allows speedy and efficient development by only processing the classes changed since the latest compile step [8].

To collect interaction logs, we were able to use an existing option of EiffelStudio enabling changes to be recorded from one incremental compilation ("Melting") to the next. The data saved includes a copy of the program and some information relating to compilation. Thanks to this feature we did not need to make any change to the compiler: we simply asked participating students to turn on the option and share certain files with us. All such data was treated anonymously, allaying any privacy concerns.

[3]17% describe themselves as complete beginners and 31% as having programmed a little bit.

3.3.2 What

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.

From the logs we were able to reconstruct scenarios of the student's problem-solving steps until he reaches the final solution. This was without recourse to tedious techniques such as talk-alouds, interviews, etc. 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.

- **Syntactical Issues**
  The usual novice errors such as petty 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 analyze these errors. Many of these syntactical errors are simple mistypings. A common one was to forget either or both of the colons in a feature declaration such as:
  divisible (other: **like** *Current): BOOLEAN is*
  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 of the wrong type, etc.

- **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.

  At the beginning of the course, many of the complete novices could not write a basic OO instruction of the form *x.f(a)*. They would write English-like short sentences (three or four words). These errors were however no longer present after the first few exercises.

- **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 - they duplicated code or did not properly modularize their solution.

- **Not following hints**
  Exercise 1 was designed with hints to help students devise contracts. Some contracts were provided fully or partially. A number of students did not use the hints at all.

- **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 MS 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 MS 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.

- **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:
  $convalue := 9/5 * value + 32$
  create fahrenheit.make_with_fahrenheit (convalue)
  Result := fahrenheit
  They used two variables: one for storing the results of the conversion (here *convalue*) and the other for the creation of a new object to represent the newly converted temperature (here *fahrenheit*). The one-line solution which does not require declaring any variables is:
  create Result.make_with_fahrenheit $(9/5 * value + 32)$

- **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[4]. Many

[4]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.

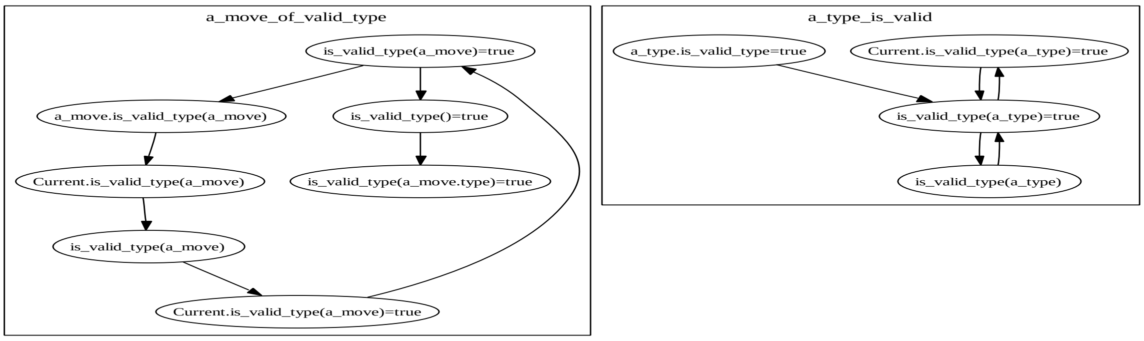
a_move_of_valid_type
is_valid_type(a_move)=true
a_move.is_valid_type(a_move)
is_valid_type()=true
Current.is_valid_type(a_move)
is_valid_type(a_move.type)=true
is_valid_type(a_move)
Current.is_valid_type(a_move)=true
a_type_is_valid
a_type.is_valid_type=true
Current.is_valid_type(a_type)=true
is_valid_type(a_type)=true
is_valid_type(a_type)

Figure 1: Student's path in solving *a_move_of_valid_type* and *a_type_is_valid*

students made this mistake even though the point was stressed in class.

- **Queries**
  Many students wrote code of the form:
  b=true or *is_valid_type(tram_type)=true*
  rather than
  b or *is_valid_type(tram_type)*
  This is not an error in itself but reflects insufficient mastery of boolean values. This point should be stressed more explicitly in class.
- **Expression used as instruction**
  Some students did not differentiate between expressions and instructions. This was among the most common errors.
- **Current**
  Some students consistently used *Current* unnecessarily. This indicates that they might not have understood the concept of unqualified call and applied *Current* "just in case". This error might also involve the influence of other languages, where *this* or *self* is more often necessary.
- **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. It took "her" sixty two compilations and quite some time before she realized the mistake.
- **Inheritance**
  It was interesting to note uses of inheritance early on, when the concept had not even been introduced in class, 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.

5. EXAMPLE PATHS AND BEHAVIOUR PATTERNS

We studied with particular care how students deal with the errors they come across. It was very interesting to observe the various strategies and patterns used by novices.

Some students were consistent in their ways of solving problems. If they adopted some particular methodology, they seemed to use it over and over again. For instance, some would use lots of backtracking: they 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 (see [1] for the exercise text) 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. "She" uses a lot of backtracking and was by far the most prolific producer of answers. Figure 2 shows the path she 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 of her paths is very large. It separates into two different problem-solving "strategies". At some point, she nearly has the answer but cannot find the correct argument to *is_valid_type*; then, she drops the first part of the answer. This graph shows how extensively she explored the possibilities. What is apparent in this example is that she cannot determine the correct argument, and in trying to find it she introduces more mistakes. She seems to be trying various options without really understanding what the problem is and attending to the error that she is getting, which was her 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. Therefore, we did not provide

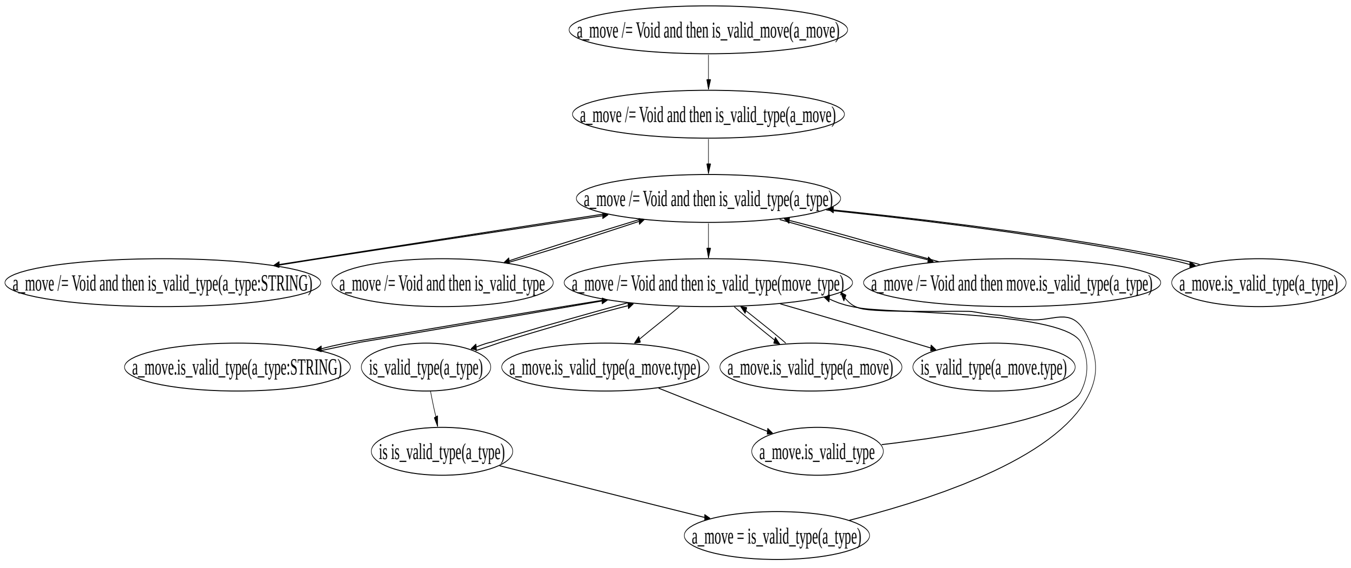
a_move /= Void and then is_valid_move(a_move)
a_move /= Void and then is_valid_type(a_move)
a_move /= Void and then is_valid_type(a_type)
a_move /= Void and then is_valid_type(a_type:STRING)
a_move /= Void and then is_valid_type
a_move /= Void and then is_valid_type(move_type)
a_move /= Void and then move.is_valid_type(a_type)
a_move.is_valid_type(a_type)
a_move.is_valid_type(a_type:STRING)
is_valid_type(a_type)
a_move.is_valid_type(a_move.type)
a_move.is_valid_type(a_move)
is_valid_type(a_move.type)
is is_valid_type(a_type)
a_move.is_valid_type
a_move = is_valid_type(a_type)

Figure 2: *a_move_of_valid_type*

detailed statistics for the different kinds of errors. In the analysis we try to understand why these errors occur and why students tackle them the way they do, with direct feedback into the design and teaching of the course, for which the results are full of lessons.

The data so far has been processed manually. We are now exploring ways of automating the processing of the data from the next iteration of this study. Various methods from compiler and debugging literature will be explored, for example program slicing [12], Advanced Object-oriented Program Dependence Graph (AOPDG) [6], Abstract Syntax Trees, etc.

We will continue to work on representing the information obtained in the study in a suitable format for provision of automated diagnosis and feedback in order to help students while they are learning using a prototypical intelligent tutoring system - the ultimate aim of this research. An initial attempt at automating the extraction of rules for the above was made with see5 (http://www.rulequest.com/see5-info.html). The results are nonconclusive at this stage and require reworking the derivation of attributes and classes. It might be necessary to explore other clustering and categorisation methods.

7. ACKNOWLEDGEMENTS

We are grateful to the assistants in the "Introduction to Programming course" at ETH for their support and help in setting up the experiment as well as for motivating their students to participate in the study. 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 7 July 2005 from, http://se.inf.ethz.ch/teaching/ws2004/0001.

[2] M. Jadud. A first look at novice compilation behavior using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[3] A. J. Ko and B. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16:41–84, 2005.

[4] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4), 2003.

[5] C. Litecky and G. Davis. A study of errors, error-proneness, and error diagnosis in cobol. *Communications of the ACM*, 19(1):33–38, 1976.

[6] J. McGregor, B. Malloy, and R. Siegmund. A comprehensive program representation of object-oriented software. *Annals of Software Engineering*, 2:51–91, 1996.

[7] B. Meyer. Towards an oo curriculum. *Journal of Object-Oriented Programming*, 6(2):76–81, 1993.

[8] B. Meyer. *Object-oriented software construction.* Prentice hall, 2nd edition, 1997.

[9] B. Meyer. 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, 2003.

[10] B. Meyer. *Touch of class - learning to program well.* http://se.inf.ethz.ch/touch/, online edition, 2003.

[11] P. Moulton and M. Muller. Ditran - a compiler emphasizing diagnostics. *Communications of the ACM*, 10(1):45–52, 1967.

[12] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3), 1995.

[13] M. Zelkowitz. Automatic program analysis and evaluation. In *ICSE'76: Proceedings of the 2nd inernational conference on software engineering*, pages 158–163. IEEE Computer Society Press, 1976.