Understanding novice errors and error paths in Object-oriented programming through log analysis

M-Helene Ng Cheong Vee
SCSIS, Birkbeck,
University of London
marie-helene@dcs.bbk.ac.uk

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

Keith L. Mannock
SCSIS, Birkbeck,
University of London
keith@dcs.bbk.ac.uk

Abstract. What difficulties do beginners face when learning object-oriented programming? How are these reflected in the code they write? 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 obtained interaction logs from two different groups of students across two unrelated universities. We developed a methodology for analysing these logs to identify students' intentions, with the goal of providing contextual feedback in an Intelligent Tutoring System.

Keywords: Data collection, Data analysis, Intelligent tutoring system, Inverted curriculum

INTRODUCTION

Collecting student's interaction logs provides data for educational research in the relevant domain. Transforming this data into useful information usually requires deriving appropriate methodologies suited to the data and the aims of the study being carried out.

We wanted to determine the typical errors students make and understand their behaviours while learning to program in order to help them better in our courses and ultimately in an intelligent tutoring system (ITS) that will act as additional support in a one-to-one tutoring fashion. Various methods used in the past have involved interviews, "talk-alouds", and observing students while they solve problems in a "looking over the shoulder" manner (Jackson, Cobb & Carver, 2005). 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, using both statistical and non-statistical techniques, 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 ITS under development.

We first present the context of this work. We then describe the courses and the organization of the study. The methods used to analyse the collected data are detailed in the next section. We finally show the initial findings on the patterns discovered from this work and test results before concluding and a brief discussion of future work.

This article departs from local administrative terminology for the sake of consistency: what is called a "module" at Birkbeck appears here as a "course", etc. For object-oriented terminology, we follow the definitions used for the course: *OOSC* (Meyer, 1997) and the *Touch of Class* textbook (Meyer, 2003b).

OVERVIEW OF THE SYSTEM FRAMEWORK

The goal of this research is to build an ITS that will provide support to beginners in object-oriented programming (OOP). This paper describes the work done to implement the engine behind part of the pedagogical agent of the ITS (Intention Matching), whose architecture is shown on Figure 1. The figure illustrates communication between agents: For example, when the pedagogical agent needs information related to the exercise presented to the student, it communicates with the Tutor agent which has this information embedded in its exercises "module"; or the Interface agent which manages the user interface (UI) needs to pass information related to the actions performed by the student to the pedagogical agent which then processes the information by storing it in the student model, providing feedback, etc. The results of the intention matching process will be used to provide feedback to the students while they solve exercises.

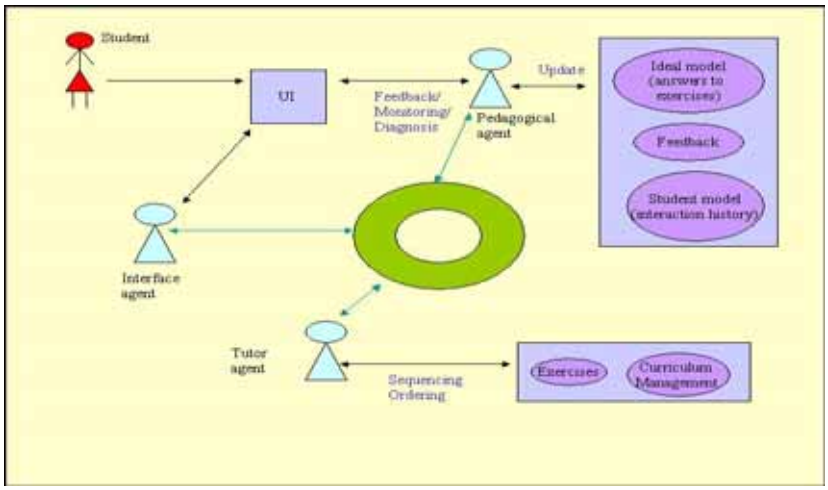

Figure 1: Prototype's architecture

THE STUDY

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

Rather than 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 students who are "beginner programmers" to learn both how to re-use libraries of real-world complexity and size 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 all the courses used for this study students learn programming using Eiffel (Eiffel Software, 2006)

Student Groups

Data for the first iteration of this study came from two instances of the course, taught with minor variations to two groups of students across two unrelated universities. The second iteration was done later in only one of the universities:

- **ETH (Introduction to Programming)**

In the 2004/2005 session, 22 out 180[1] students from the *Introduction to Programming* course ("Introduction to programming") at ETH (first year of the Computer Science Bachelor's program) voluntarily participated in the study. In the 2005/2006 session, an average of 64 out of 180[1] 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 students. In addition to fundamental OOP and procedural concepts such as objects, classes, inheritance, control structures, recursion, etc., students learn more advanced topics such as event-driven and concurrent programming and fundamental concepts of software engineering.

- **Birkbeck (Object-oriented programming part-time and full-time)**

52 out of a group of approximately 75[1] students taking the OOP course in the MSc program at Birkbeck[2] participated. Students were required 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. Data was collected in the spring term 2004/2005.

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

[1] All 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.

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 for all the groups.

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.

The Data Collection Mechanism

We benefited from the "Melting Ice Technology" of the free EiffelStudio environment used by the students (EiffelStudio, 2006). This incremental compilation mechanism allows speedy and efficient development by only processing the classes changed since the latest compile step (Meyer, 1997).

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. The information relating to compilation includes the version of EiffelStudio used, whether compilation was successful, etc. All such data was treated anonymously, allaying any privacy concerns.

The Structure of the Collected Data

Collecting the data meant processing a large number of files: total number of participants, multiplied by number of exercises, multiplied by number of compilations for each exercise.

Figure 2 shows the structure of the data: A student submits a number of compilation folders (zipped) for each exercise via an upload page. Each compilation folder in turn contains various files and folders. The relevant documents in the compilation folders are the source code files (Temperature.e in Figure 3) and the file compilation_info.txt containing information about the compilation. This last file contains meta-data associated with the compilation: compiler version used, compilation outcome (successful or not), program files changed, etc.

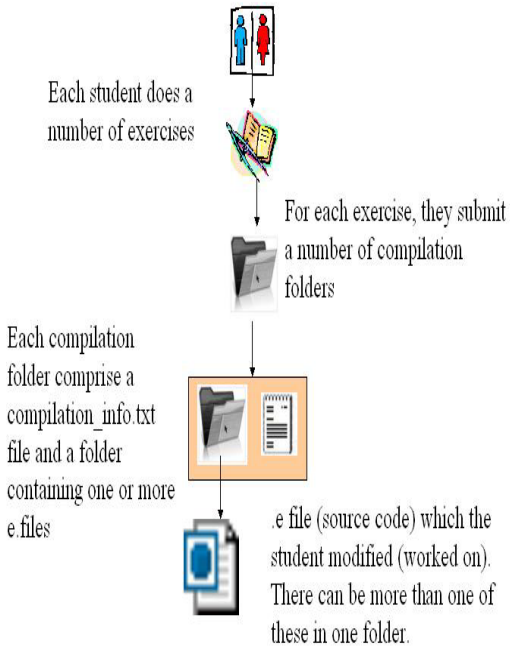

Figure 2: Structure of the data collected.

```
indexing
        description: "Objects that ..."
        author: ""
        date: "$Date$"
        revision: "$Revision$"

class
        TEMPERATURE

create

        make_with_celsius,
        make_with_fahrenheit,
        make_with_kelvin

feature -- Initialization

        make_with_celsius (a_value: DOUBLE) is
                        -- Create with 'a_value' of unit Celsius.

        require
                not_to_small: value >= -272

                do
                        value := a_value
                        unit := "Celsius"
```

Figure 3: Temperature.e

[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.

MINING THE COLLECTED DATA

Statistical Information

The interaction logs contained a wealth of information. The first step was to obtain statistical information such as the amount of time taken to accomplish tasks, the number of compilations, and time between compilations with the help of shell scripts working with the logs and compilation information files. This information was useful in getting a rough idea of the data we would be dealing with.

Student Errors

The statistical information described previously is not enough to reach any meaningful conclusions. More important and relevant is to understand where difficulties arise and provide help in these areas. The Unix "diff" utility was used to extract the differences in program files between successive compilations. To extract novice errors, the output of the "diff" scripts was manually examined. Individual errors were noted as well as their frequency (enabling us to focus on the most acute problems) and other verbose information such as why the error happened and who diagnosed the error (compiler, instructor, runtime?).

Figure 4 illustrates the database structure used for storing the information we extracted from the log analysis: Each exercise given to the student is decomposed into subproblems (PSE). Errors occur within PSEs. We tracked frequencies within groups of participants. Moreover, errors can occur for multiple reasons (speculative at this stage since we cannot gauge it in a better way).

From the logs we were able to reconstruct scenarios of the student's problem-solving steps from initial attempt to final solution. Using a small tool that tracks the evolution of certain portions of code using the output of the "diff" scripts, we specify what portion of code we want to track starting from a certain compilation (number). Examples of the scenarios/error paths produced are discussed in greater detail in (Ng Cheong Vee, Meyer & Mannock, 2006).

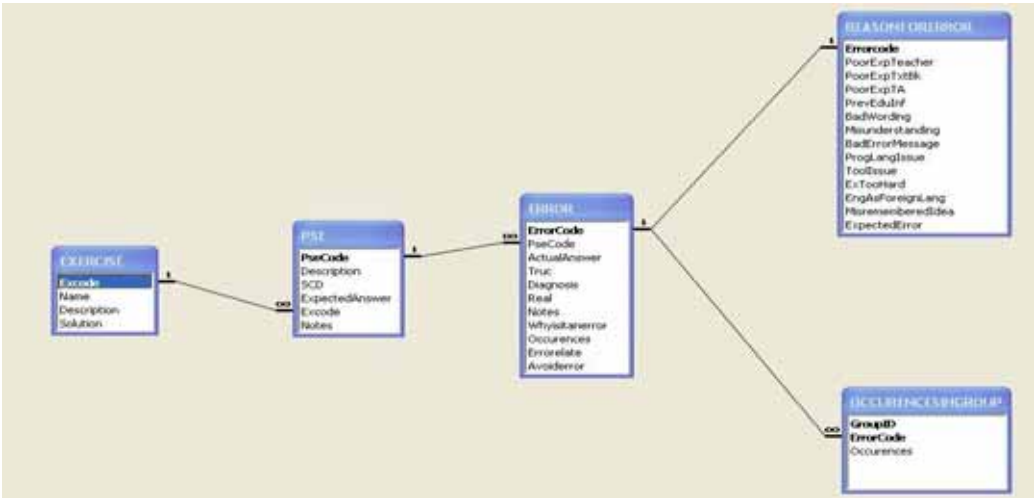

Figure 4: Database design for the information extracted from the log analysis

Intention Matching for the Provision of Feedback

It is not always possible to identify errors (whether syntactic or semantic) present in a student answer without placing it into context. In a few cases, identification of errors might be possible using pattern matching, but the manual analysis has shown that in many cases this is insufficient. We therefore relied on the concept of an "intention" (Johnson, 1990). Intentions, or reference programs, are correct solutions to the given problem. We estimate the intention the student was attempting to use and build a discrepancy list by applying an algorithm based on the Levenshtein or Edit-Distance algorithm (Levenshtein, 1965). Other possible approaches we considered for finding the best match for one piece of code to a number of model solutions are program slicing (Tip, 1995), Advanced Object-oriented Program Dependence Graph (AOPDG) (McGregor, Malloy & Siegmund, 1996), and Abstract Syntax Trees. These are, however, coarse-grained and would need to be instrumented for our purposes since we are working with small modular sections of code rather than complete programs. The Levenshtein distance algorithm yields the minimum number of character insertions, deletions and substitutions necessary to make two strings equal, allows us to estimate with simple computations a best match. We modified the original algorithm to take into account multi-line solutions.

Example of exercise and intentions

In the second of the seven exercises examined in the data analysis, students were asked to write a program to convert temperatures amongst three different units (Celsius, Fahrenheit and Kelvin). They were asked to use a given interface for the class *TEMPERATURE*. We decompose the problem into subtasks (PSE). Here we concentrate on the subtask of writing the body of the function *celsius_to_fahrenheit*. Two (out of the seven) intentions are:

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

```
local
        temperature: TEMPERATURE
do
        create temperature.make_with_fahrenheit (9/5 * value + 32)
        Result := temperature
end
```

These two intentions correspond to two different ways of creating and initializing a *TEMPERATURE* object. The second one uses a local variable for the creation, then assigns it to the *Result* of the function. The second one achieves conciseness by using *Result* directly.

Cost criteria

Several experiments were run to determine which criteria and cost combinations yielded a more accurate intention matching. For example, costs such as *too many lines*, *too few lines*, *threshold* (to discard very high scores) and *unmatched tokens* proved redundant and were discarded, being implicitly included in adjustment costs (see the " Adjustment ratios" in the next subsection). We tried placing more weight on some criteria rather than others but the cost combination yielding the best results was a uniform *add*, *delete*, *replace* set to 1.

Algorithms and steps for finding a best match

Finding which intention matches a given student code requires several steps. The "word-by-word" matrix shown on Figure 5 depicts a comparison of one of the lines of the student code to one of the lines of the model solutions. A "line-by-line" matrix must next be built to show the score obtained for each line of the student code compared to each line of the intention. The cells of the latter are populated by placing the value of the last cell (lower right cell) of each relevant "word-by-word" matrix into the appropriate cell of the "line-by-line" matrix.

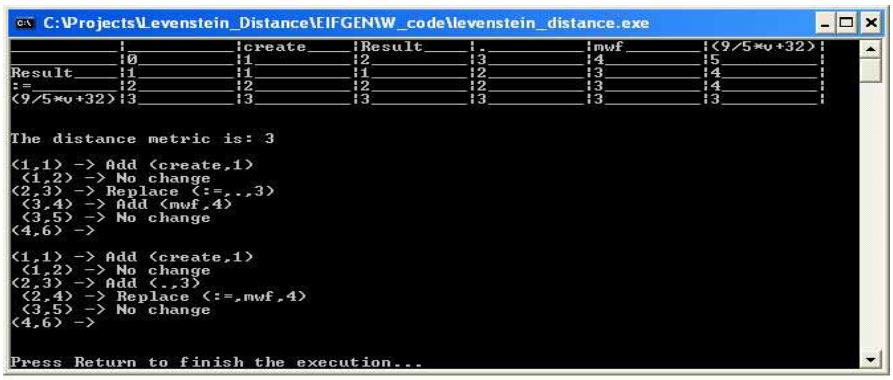

Figure 5: Output for the application of the Levenshtein distance to compare the student's answer and the intention

The "line-by-line" matrix having the lowest overall score indicates the best match. The algorithm used to calculate its overall score is as follows:

Case 1: When dealing with a matrix where the number of lines in the intention exceeds that of the student code: (1) Select a minimum per row (mark the row as being "taken"). (2) If there is more than one minimum in a column; starting from the column where a "clash" occurs (its row had already been marked previously), search for the next minimum in the row, checking that the new minimum's row is not marked as "taken" - otherwise repeat search in the same manner. (3) Add values of the marked cells.

Case 2: When dealing with a matrix where the number of lines in the student code exceeds that of the intention: (1) Select a minimum per column (mark the cell). (2) If there is more than one minimum in a row; starting from the row that "clashes", search for the next minimum in the column, checking that the new minimum's column is not marked - otherwise repeat search in the same manner. (3) Add values of the marked cells.

Adjustment ratios

To normalise the overall score of the matrices (and take account of the disparateness in the number of lines in the two pieces of code being compared), an adjustment ratio is applied. The adjustment ratio is a coefficient $\frac{no-of-lines-in-student-code}{no-of-lines-in-\text{int}ention}$ in the case where the number of lines in student code exceeds that of the intention. The reverse is applied in the opposite case.

The discrepancy list

A by-product of the Levenshtein distance algorithm are the series of transformations (add, replace, delete) needed to transform one string into another. In this context, we can also obtain the transformations which we have called the *discrepancy-list* (See Figure 5). This will be used to generate feedback. At the time of writing, the method for feedback generation from discrepancy-lists is incomplete. In Figure 5, the first discrepancy-list indicates the following transformations are necessary to achieve compliance between the student answer and the reference solution: (1) Add the *create* keyword at place-holder 1. (2) Replace '*:=*' by '.' at place-holder 3. (3) Add *make_with_fahrenheit* at place-holder 4.

PATTERN DISCOVERY OF INTENTIONS

To understand student behaviours from a different perspective than the one presented in (Ng Cheong Vee, et al., 2006), we considered how intentions cluster in the data, whether the same clusters appears in all three groups used for testing, and which were the most popular intentions. This analysis, summarised in Figure 6, revealed that: (1) Intentions in the data group according to how we manually grouped them. (2) A substantial number of students (4->5 grouping is the second highest point in the graph) made use of extra variables: an example of such behaviour was given in Section Example of exercise and intentions. This gives us an indication that students think in "small steps" rather than holistically. Teaching can be improved by pointing out such redundancies in code and helping students refactor their code in more elegant and efficient ways. (3) The comparison of results between the ETH group and MSc group in the Temperature exercise shows that in general, they had the same patterns – the distribution is similar.

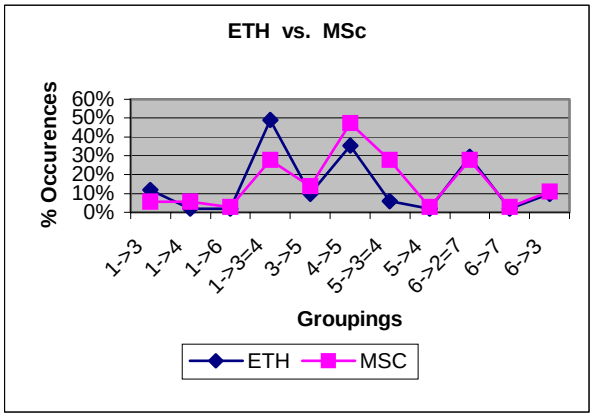

Figure 6: Intention grouping for ETH and MSc students

Number of …	MSC45T	MSC45F	ETH56T
Students	36	37	51
Answers	178	165	163
Syntax Errors	99	26	68
Not allowed answers	0	83	9
Successful predictions	79	53	84
Unclassifiable (syntax Correct)	0	3	1
Mispredictions	0	0	1

Table 1: Summary of test results

The groupings on the graph (x-axis), are represented as: the "first choice" intention is on the left hand side of the arrow and the intention that comes out as second choice in the data is on the right-hand side.

RELATED WORK

Studies similar in their scope to ours were carried out three decades ago for imperative languages (Litecky & Davis, 1976) (Moulton & Muller, 1967). They highlighted and classified various common programming errors pertinent to the imperative paradigm. Some of these studies used compiler instrumentation to gather data, while others used other methods such as interviews mentioned above.

A more recent study (Jadud, 2005) used Java and the BlueJ environment (Kölling, Quig, Patterson and Rosenberg, 2003). This study is more relevant, being situated in the object-oriented paradigm and using a very similar data collection method. It focuses on analysing novice compilation behaviours 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 behaviours – is somewhat different from ours, he also provides a list of common errors. Nevertheless most considered errors are syntactic, whereas we are interested in semantic

errors. Moreover, he uses quantitative analysis while we consider qualitative analysis better suited for our purposes.

It is often difficult to clearly distinguish syntactic from semantic errors. Methods for identifying bugs in program code in the area of "program analysis'" have been proposed for example in (Hovemeyer & Pugh, 2004), (Hangal & Lam, 2002). However, these are usually useful for intermediate to advanced programmers. Novices make very basic errors that are reflected syntactically but may carry more meaning and significance at the cognitive level. Moreover, their way of solving tasks (even simple ones) is different. They seem to often use trial and error and not think of the real problem.

(Sykes & Franek, 2004a) used a methodology based on the Levenshtein distance in their work. They use an "Intention Recognition" module (IR) to establish the student's intention and deliver feedback. Their IR module is composed of the A-type and B-type functionalities as they call it. In later work (Sykes & Franek, 2004b), they seem to have adopted a slightly different algorithm (JECA) based on the Burke-Fisher Error Recovery algorithm (Burke & Fisher, 1987) to get rid of the A and B-Type functionalities. Our definition of "Intention Matching" (what they call "Intention Recognition") is different from theirs. By intention matching, we mean to discover what conceptual plan or algorithm the student has been following, as we assume our intention matching engine received syntactically correct code (thereby dealing with logic or semantic errors). We compare two pieces of code on a word-by-word basis followed by a line-by-line comparison. In both their approaches (using the A and B-Type functionalities and the JECA algorithm), their comparison is done character-by-character for all the keywords that exist in Java. This is obviously computationally expensive and explains why they restrict the programming concepts covered in their system. Moreover they have decided to cover procedural aspects of programming (variables, operators and looping structures). We aim to cover basic OO concepts taught in our courses (e.g. classes, objects, message, passing, etc…). The main difference between their work and ours lies in the emphasis they place on syntax as illustrated by this statement:

"So even though the IR algorithm and tutoring process results in source code that is syntactically correct, there is no guarantee that it will satisfy the program requirements."(Sykes & Franek, 2004a).

We assume that our parser, based on the Gobo library (Gobo parser for Eiffel, 2006), will deliver syntactically correct and successfully compiled code to our intention matching engine. Moreover, because we use Eiffel, a language with a small number of keyword and syntactic constructs, the number of trivial syntax errors such as balancing brackets, missing semicolons, etc. are considerably reduced. Additionally, the straightforward correspondence between the language's constructs and the concepts of OO (each syntactic construct has one precise meaning and therefore are not overloaded) allows us to focus on more important conceptual issues and helps us better in finding intentions in student's solutions.

TEST RESULTS FOR INTENTION MATCHING

The data arising from ETH in the 2004/2005 session was used to derive the described methodology. Three other batches of data were used to test the methodology: The MSc data collected from Birkbeck in the 04/05 data collection for two of the problems set to the student: the Temperature and Fraction problem (Introduction to programming, 2006). The last batch is the ETH batch in the 05/06 data collection for the Temperature problem. The Fraction problem is a more complex exercise to solve than Temperature, involving the use of more concepts such as inheritance. We identified 17 intentions for this problem. The results are summarised in Table 1.

Those answers that would not occur in the prototype (since the prototype applies certain constraints on what students can do – for instance, the prototype imposes an order in which subproblems (PSEs) are solved) were pre-filtered (Not allowed answers row in Table 1[4]). There are also syntactically correct answers that would be allowed to be formed in the prototype but are difficult to map onto an intention due to the student programming in a non-standard way that can still yield correct output: three in MSC45F and one in ETH56T. To deal with such cases, an error margin metric tolerance was introduced to compute the real difference between the system's 1st and 2nd predictions. If the delta is below 10%[5] we cannot have confidence in the system's prediction. Otherwise, we accept the prediction. For three of those four answers, the margin was below 10%. When testing correct predictions in all three batches, one answer had a margin below 10%. We might, however, not have to use the margin metric as it is unlikely that basing feedback on the system's first prediction will confuse the student.

In twenty one of the MSC45T answers for the Temperature exercise, students used intentions that would not work in the given context but would work in others. These intentions were included in the intention matching process as they are useful for feedback generation.

[4] As can be seen this number is much higher in the MSC45F batch. This is because of the nature of the problem and the constraints that are therefore necessary to put in place.

[5] The 10% metric was decided based upon our experiments – these showed that the difference was below 10% in general in these specific cases. For the rest, the difference was above 10%.

These results show that the method can be used for various exercises and does intention matching with very high accuracy as only one misprediction was detected in ETH56T.

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 presented a methodology that combines pattern matching and an augmented version of the Levenshtein distance algorithm to analyse students' interaction logs and guess their programming intentions. This analysis has helped us 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 results of the tests demonstrated that the methodology can be successfully applied to problems of considerable complexity and size.

In our work on feedback generation, we are exploring patterns in the use of intentions by students and we are using the discrepancy-lists (by-product of the algorithm/method described in the paper) to extract rules to incorporate in the system. From these we will be able to design a feedback mechanism to guide students using the prototype.

REFERENCES

Burke, M.G. & Fisher, G.A. (1987) A practical method for LR and LI syntactic error diagnosis and recovery, ACM Transactions on Programming Languages and Systems, **9**, 2, 164-197.

Eiffel Software (2006): Retrieved 2 April 2006 from http://www.eiffel.com

EiffelStudio (2006): Retrieved 2 April 2006 from http://www.eiffel.com/products/studio

Gobo Parser for Eifel (2006): Retrieved 2 April 2006 from http://www.gobosoft.com/eiffel/gobo/geyacc/

Hangal, S. and Lam, M.S. (2002) Tracking down software bugs using automatic anomaly detection. *In Proceedings of the 24th International Conference on Software Engineering*.

Hovemeyer, D. and Pugh, W. (2004) Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications (OOPSLA*). ACM Press.

Introduction to Programming (2006). Retrieved 27 March 2006 from http://se.inf.ethz.ch/teaching/ws2004/0001

Jackson, J., Cobb, M. and Carver, C. (2005) Identifying top java errors for novice programmers. *35th ASEE/IEEE Frontiers in Education Conference*, Indianapolis, October.

Jadud, M. (2005) A first look at novice compilation behaviour using bluej. *Computer Science Education*, **15**, 1,25-40.

Johnson, W. (1990) Understanding and debugging novice programs. *Artificial Intelligence*, **42**, 1, 51-97.

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

Levenshtein, V.I. (1965) Binary codes capable of correcting spurious insertions and deletions of ones (original in Russian). *Russian Problemy Peredachi Informatsii*, **1**, 12-25.

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.

McGregor, J, Malloy, B. and Siegmund, R. (1996) A comprehensive program representation of object-oriented software. *Annals of Software Engineering*, **2**, 51-91.

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

Meyer, B. (1997) *Object-oriented software construction*. Prentice Hall, 2nd edition.

Meyer, B. (2003a) 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. Lecture Notes in Computer Science 2890, Springer-Verlag.

Meyer, B. (2003b) *Touch of class – Learning to program well*. http://se.inf.ethz.ch/touch/, online edition.

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

Ng Cheong Vee, M.H., Meyer, B. and Mannock, K.L. (2006) Empirical study of novice errors and error paths in object-oriented programming. Submitted to 7th Annual HEA-ICS conference, Dublin, April 2006.

Sykes, E.R. and Franek, F. (2004a) A prototype for an intelligent tutoring system for students learning to program in Java™. *Advance Technology for Learning*, 1.

Sykes, E.R. and Franek, F. (2004b) Presenting JECA: a Java error correcting algorithm for the Java Intelligent Tutoring System. *Proceedings of the IASTED Conference on Advances in Computer science and Technology*. St. Thomas, US Virgin Islands.

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