Problem Distributions in a CS1 Course

Anthony Robins[1], **Patricia Haden**[2], **Sandy Garner**[1]

[1] Computer Science Department, The University of Otago, Dunedin, New Zealand

{anthony,sandy}@cs.otago.ac.nz

[2] School of Information Technology, Otago Polytechnic, Dunedin, New Zealand

phaden@tekotago.ac.nz

Abstract

In this paper we describe an ongoing study of novice programmers in an introductory programming course. Building on previously published results from the study we explore the distributions of different kinds of language related (rather than general or design related) problems over the sequence of laboratory exercises in the course. Data collected from student cohorts in 2003 and 2004 are compared, and the consistency of the effects observed gives us considerably confidence in the reliability and validity of the mechanisms of the study. While great care must be taken in the interpretation of the problem distributions, we suggest that they are useful for diagnosing aspects of course design and delivery, and that they may contribute to our general understanding of the process of teaching and learning a first programming language.

Keywords: learning novice programming errors CS1

1 Introduction

This paper builds on a previous publication (Garner, Haden & Robins, 2005) describing a study which has been running at the University of Otago since 2001. Each year data has been collected on the kinds of problems that students encounter while working in the laboratory sessions of an introductory programming paper (of the kind often described as "CS1"). The aim of the study is to explore the process of learning a first programming language, with the long term goal of increasing our understanding of this process and creating a more effective learning environment.

Results arising from the study have already been used to adjust the amount (and kind) of attention devoted to various topics in lectures, laboratories and other resource materials, and to construct targeted help materials. Further analysis of problems encountered may allow us to address such topics as focusing demonstrator (teaching assistant) training on the most common and / or most difficult problems, or highlighting areas of particular difficulty to students to aid their meta-learning and study. It may be possible to recognise different kinds of "novice programming style" from the patterns of problems that students experience. It may even be possible to identify at risk students early, and provide specifically targeted help.

Garner, Haden & Robins (2005) presented the tools and methods employed in the study, in particular presenting the list of problem definitions which is used to classify students' problems. Data collected during 2003 were presented and discussed. The results described were consistent with trends noted in the literature, and highlight the significance of both fundamental design issues and the procedural aspects of programming. Different problem distributions were observed for high and low performing students. We suggested that an analysis of individual lab sessions was useful for refining course materials and teaching practice.

The purpose of the current paper is to build on the foundation of the earlier publication and pursue a more detailed examination of a specific topic: the distribution of the occurrence of certain problem types over the sequence of laboratory sessions. In the process data from both 2003 and 2004 will be presented. The implications of the general similarity of the results over the two years for the reliability and validity of the study will be briefly discussed.

2 Method

In this section we briefly describe the course on which our study is based, and summarise the tools and processes used to collect information about novice programming problems. Further details can be found in Garner, Haden & Robins (2005).

2.1 The Programming Labs

Data for the study is collected in the programming laboratory sessions of our introductory programming course COMP103 (with a typical enrolment of roughly 220 students). The course teaches Java, and consists of 26 fifty minute lectures, and 25 two hour laboratory sessions.

Each lab is run over several specific streams with up to 40 students working on the set task for the session. Tasks are well specified in a workbook. When a student

requests help they are visited by the next available demonstrator. There are 2 - 5 demonstrators per stream (drawn from a pool of roughly 15). A demonstrator works with a student to try to help them solve their problem, then moves on to the next help call. The aim of the study is to capture, for every help call, the best information that we can about the problem(s) that led to the call.

The course content, course materials and course delivery were almost identical in 2003 and 2004, but there was some change in the pool of demonstrators.

2.2 Problem List Design

The crucial tool of the study is the problem list which is used by demonstrators to classify the problems that they observe. The problem list used in 2004 is shown in the Appendix (and minor differences with the version used in 2003 are noted)[1]. The process by which the problem list was originally developed is described in Garner, Haden & Robins (2005).

The main tradeoff involved in designing such a list is between richness (implying a large number of problem types / codes that can support a detailed classification) and practicality (implying a small number of problem codes that different demonstrators can become familiar with and use reliably). Early versions of the list erred in the direction of far too many problem codes, but the list has been more or less static in its current form since 2003. Demonstrators on the whole find it practical, and report that they are able to classify most problems adequately (see also the discussion of inter-rater reliability below).

2.3 Data Collection

The major factor constraining the collection of data is that the process must be practical, and not impact on teaching and learning in the labs. In particular the intervention must be easily manageable for the demonstrators – they are often under a lot of pressure and don't have time to "waste" making notes.

The following process was used in both 2003 and 2004. At the end of a help session with a student, the demonstrator records on a checklist brief details about the session, in particular noting the relevant problem codes. In order to provide some structure for standardising these decisions, two guidelines were stressed:

> At the end of a session with a student record the codes that *best describe the problems about which you gave advice*. The student may be having many problems – do not try to guess them all! Record only the problems about which you actually gave advice.
>
> Record up to three codes, corresponding to the three most important topics that you helped with.

This methodology has the typical advantages and disadvantages of naturalistic observation – see for example Sheil (1981), Gilmore (1990). It lacks the formal rigor of an experimental study, but it is higher in ecological validity. A major issue, however, is inter-rater reliability. Measures taken to increase reliability included: using some demonstrators who were already experienced with the study from previous years; a short training session on the problem list and the data collection process; and a Senior Teaching Fellow who was present in most lab sessions to provide support and help adjudicate difficult cases.

The degree of inter-rater reliability was also explicitly monitored as follows. The Senior Teaching Fellow "shadowed" each of the other demonstrators for at least one lab session, listening to the discussions with students, and making her own independent checklist entries. These entries were then compared with the demonstrator's checklist. Most demonstrators had good agreement with the Senior Teaching Fellow. The less reliable demonstrators, however, also tended to be those with the lightest demonstrating commitments, and thus they contributed a small proportion of the total number of observations recorded. In short, we are confident that the data collected are generally reliable. The consistency between 2003 and 2004 data (discussed below) also helps to support this assumption.

3 Language Related Problem Distributions

The study to date has generated a rich and interesting data set. It captures as closely as reasonably possible a count of every problem experienced by every student in an introductory programming paper for two large student cohorts (2003 and 2004). As noted above there are many specific topics which can be explored with such a data set, such as for example the frequencies of problem types, the problems experienced by different types of students (e.g. based on grades), the problem distributions within each lab session, the problem distributions recorded by individual demonstrators, the problem distributions for individual students, and so on.

In Garner, Haden & Robins (2005) we explored two of the basic characterisations of the 2003 data set and a breakdown of the problem counts for selected example labs. The current 2004 data generally matches the results presented in the earlier paper well (see further comparisons of the two cohorts below), confirming the trends reported.

For the purposes of the current paper, however, we will briefly consider total counts by problem type, and then focus on individual problem types and their distribution (frequency of occurrence over the sequence of laboratory sessions). In particular we will consider the specific problem types that represent language related issues (see S1 to S21 in the Appendix) rather than background and general problems.

If three assumptions are satisfied, then we would expect to see a characteristic pattern in all problem type distributions. The three assumptions are:

> (1) the actual problems experienced by students are of similar difficulty, and

[1] Garner, Haden & Robins (2005) present data collected in 2003, the current paper presents data collected in 2003 and 2004.

(2) students make progress steadily and similarly for all topics and their associated problem types, and

(3) after the introduction of the underlying topic the subsequent lab sessions afford equal opportunities (of equal difficulty) to experience associated problem types.

If these assumptions are satisfied then the expected distribution for each problem type would be an initial peak in the problem count (when the underlying topic was introduced) followed by a steady decline in frequency over later lab sessions (as students made progress). To the extent that the actual distributions of problem types vary from this expected pattern, it suggests that one or more of the assumptions listed above are invalid.

4 Results

A total of 11240 problem codes were recorded in 2003 (roughly 250 students over 25 labs with a maximum of 3 problem numbers per help session), and 7768 in 2004 (roughly 220 students). The lower count in 2004 is in part due to a smaller class size, and in part due to a faster drop-off in the rate of lab attendance in that year.

4.1 Total Counts by Problem Type

The total counts for each problem type / code are shown in Figure 1 for 2003, and Figure 2 for 2004. There is a very high degree of similarity in the distributions of the problem counts for the two cohorts[2], and this gives us further confidence in the reliability of the data collection methodology.

The implications of this distribution were discussed in Garner, Haden & Robins (2005). We were very surprised at the high frequency (Figure 1) of problems relating to trivial mechanics (G4), and despite emphasising a stricter definition of this problem type in 2004 it continues to dominate (Figure 2). Note also the prominence of problems relating to the understanding of the task and very general matters of program design (B and G codes). This supports various claims in the literature (e.g. Spohrer & Soloway (1989), Winslow (1996)) that issues relating to basic design can be more significant than issues relating to specific language constructs. For these fundamental problem types there was considerable variation when broken down over individual labs, and also a clearly increasing trend in the incidence over the range of student abilities (from A+ to E grades).

In the current context these figures are presented as a useful reference when considering the following discussion. Although the patterns of errors are interesting, it is also important to consider the different absolute difficulties of the programming principles involved in each of the specific problem types (S codes).

[2] To aid comparson both figures are coded using the 2004 version of the problem list (as shown in the Appendix). Note that the codes G1 and G2 were – in the 2003 list – represented by a single problem code which is shown in Figure 1 as a count for G1 (hence G2 has a count of 0).

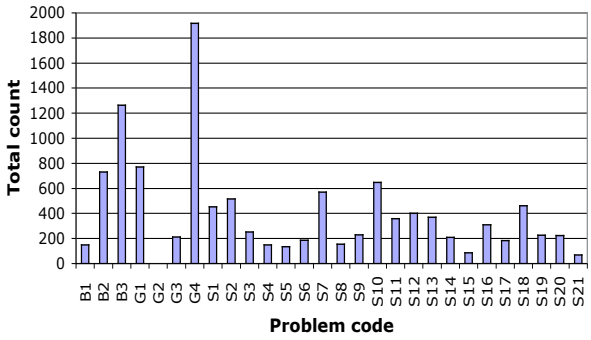

Figure 1: Total problem counts by problem type 2003

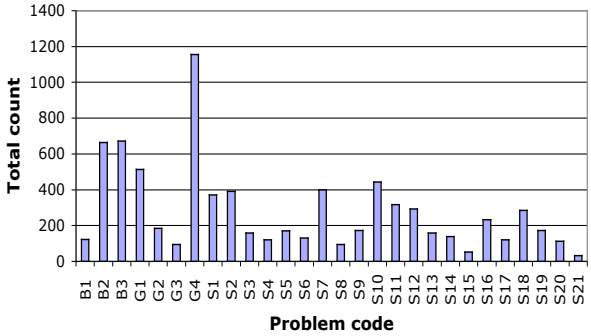

Figure 2: Total problem counts by problem type 2004

The incidence of such problems will be influenced both by the inherent difficulty of the problem, and the number of labs for which the problem type was possible, e.g. array errors will not (in general!) occur until after arrays are introduced.

4.2 Problem Distributions

Putting aside the issues of background and general design related (B and G) problem types, the main focus of this paper is to explore the distributions of "language related" problem types S1 to S18 over the sequence of lab sessions. (S19 to S21 are not considered as they relate to topics that are only introduced late in the course and result in too few data points to extract reliable trends). The trends for each problem type are summarised in Figure 3, and some example distributions are shown in Figures 4 to 9.

The summary presented in Figure 3 shows a measure of the slope of the distribution of each specific problem type. In each case the slope is measured using a regression coefficient (RC) computed from first peak to final lab, and shown in the figure with its associated 95% confidence interval (CI). The steepest slope, for example, is for problem type S10 (Arrays) in 2003 with an RC of –12.2 and a CI of –18.3 to –6.0. At the other end of the spectrum is problem S16 (Class vs. instance) in 2003 with a slope / RC of 0.3 and a CI of –0.9 to 1.5.

Note that CIs with upper bound less than 0 indicate a significant negative slope – i.e. the number of errors decreased over the sequence of lab sessions. CIs with upper bounds greater than 0, on the other hand, provide no evidence of negative slope. Fortunately, there were no significantly positive slopes!

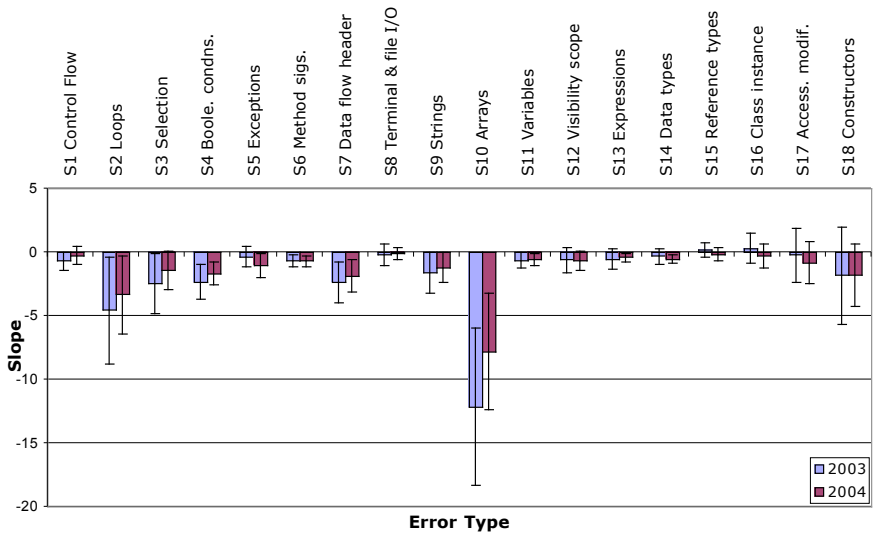

Figure 3: Slope (regression coefficients) and confidence intervals for problem types S1 to S18

We can divide problem types into those that had significant negative slopes in both 2003 and 2004, those with neutral (no significant slopes) in both 2003 and 2004, and those which are inconsistent over the two years:

Negative slopes (2003 and 2004)
- S2 Loops
- S4 Booleans and conditions
- S6 Method signatures and overloading
- S7 Data flow and header mechanics
- S10 Arrays
- S11 Variables

Neutral slopes (2003 and 2004)
- S1 Control flow
- S8 Terminal or file I/O
- S9 Strings
- S12 Visibility and scope
- S15 Reference types
- S16 Class vs. instance
- S17 Accessors and modifiers
- S18 Constructors

Inconsistent (2003 vs. 2004)
- S3 Selection
- S5 Exceptions, throw catch
- S13 Expressions and calculations
- S14 Data types and casting

Figures 4 to 9 show selected examples of distributions of problem types over the sequence of laboratory sessions. The proportions displayed (*y* axis) are the total count of problems of that type in that lab divided by the total count of problems for that year. The selected examples include four with negative slopes (S2, S4, S10, S11) and two with neutral slopes (S1, S12).

4.3 Problem Distributions by Grade

The distributions and slopes presented above are all calculated over the entire student population. It is interesting to consider whether students of different ability (as measured by final grade in the course) have significantly different distributions.

For the purposes of this analysis students were divided into three groups, "high" (final grade in the A range), "medium" (B and C range) and "low" (D and E range)[3]. Problem distributions were then calculated for each of these groups separately. Examples are shown for two problem types (S1 and S10) in Figures 10 and 11. Since there are different numbers of students in the groups, the values plotted (*y* axis) are mean errors per student.

As could be predicted from the analysis described in Garner, Haden & Robins (2005), the three groups had robustly different mean problem frequencies. More problem codes are recorded on average for students in the medium group than either the high or low groups. The obvious interpretation, proposed in the earlier paper, is that medium students dominate because high students in general require less help, and low students are (for whatever reason) not asking for it when they need it (or are simply absent from more lab sessions).

While the three groups are distinguished in terms of mean numbers of problems, the distributions of those problems over the sequence of laboratory sessions are remarkably similar. Consider for example the pattern of the distributions over labs shown in Figures 10 and 11. There is no clear distinction between the high, medium and low groups on the basis of the problem distributions or their slopes. Considering all problem types this effect is very consistent – the similarity between the three groups is remarkably robust. In many cases the distributions are effectively identical.

[3] Data for 2003 and 2004 were simply combined for this analysis.

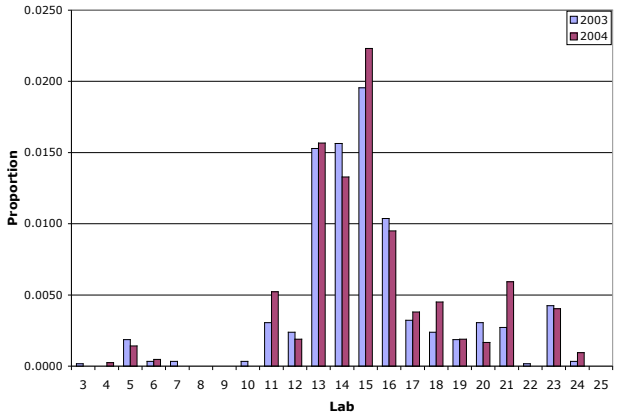

Figure 4: Distribution for S2 Loops (negative slope)

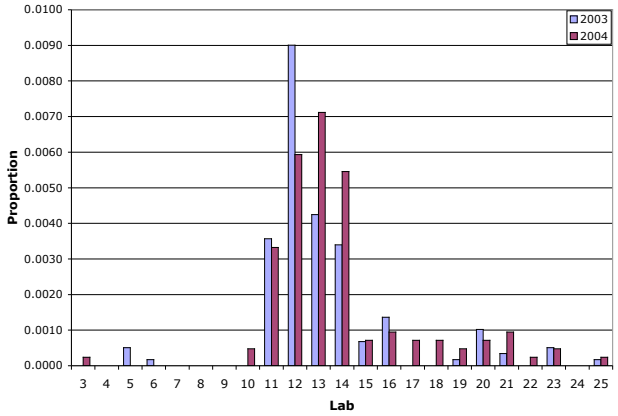

Figure 5: Distribution for S4 Booleans and conditions (negative slope)

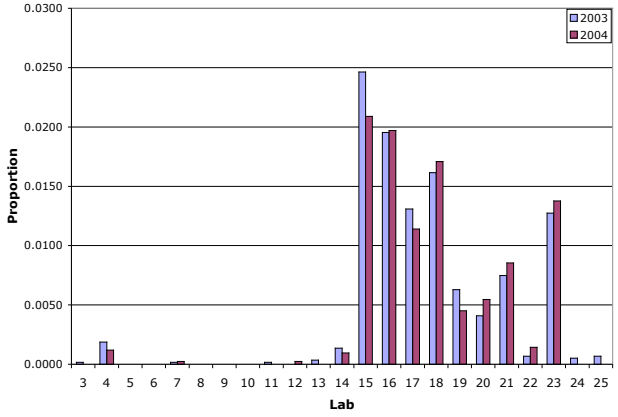

Figure 6: Distribution for S10 Arrays (negative slope)

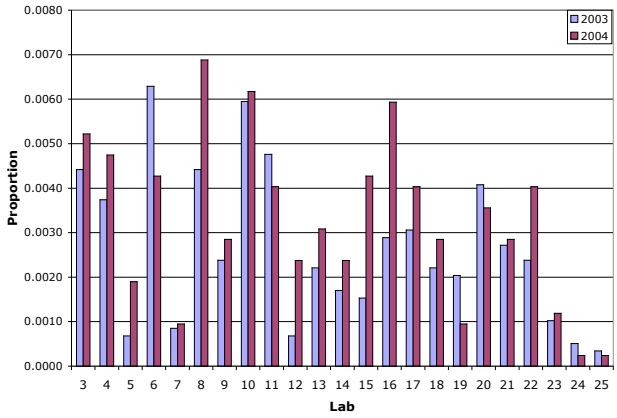

Figure 7: Distribution for S11 Variables (negative slope)

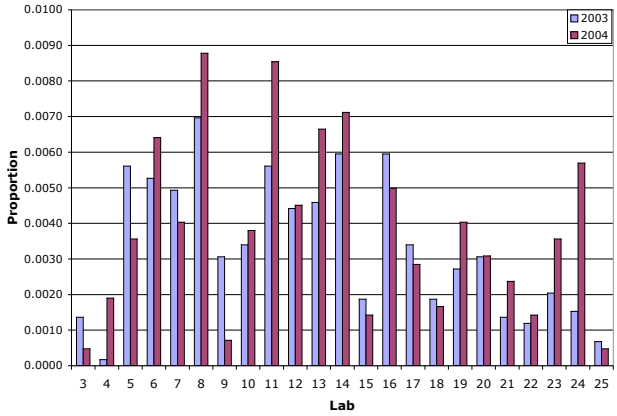

Figure 8: Distribution for S1 Control flow (neutral slope)

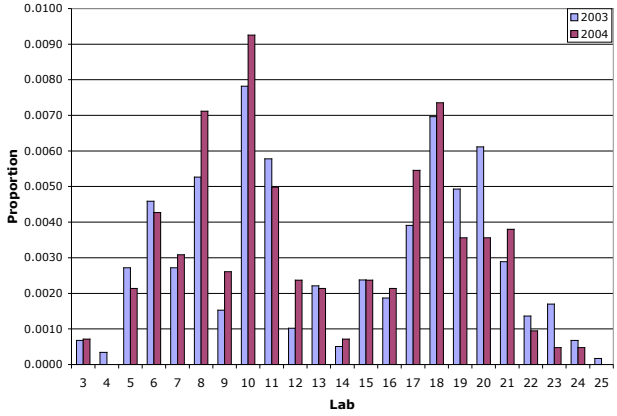

Figure 9: Distribution for S12 Visibility and scope (neutral slope)

5 Discussion

Individual problem distributions are useful for "diagnosing" aspects of course design and delivery, but great care must be taken in their interpretation.

A particular distribution may, for example, be an indicator of particularly difficult (or perhaps poorly explained) material in the lab sessions. The distribution shown in Figure 6 for Arrays (S10), for example, shows a significant spike at Lab 23. This lab involves an array of references to objects, which is content well covered in earlier labs. Hence on this occasion the high error spike suggests an unanticipated difficulty with the specific task or its description which warrants further investigation. In some cases, however, later spikes of problem counts may have a different interpretation. In Garner, Haden & Robins (2005) we noted that some topics may appear to be well learned when introduced because instructions are very explicit, when in fact the underlying concepts have been poorly understood and hence are difficult to apply to (cause problem spikes for) later more open ended tasks.

In some cases important aspects of individual problem distributions appear to be artefacts of the problem list design. The bimodal distribution in Figure 9 for Visibility and scope (S12), for example, is almost certainly a function of the definition of that problem type. Matters relating to scope arise early and account for the spike from Labs 6 to 11, whereas matters relating to visibility do not arise until later labs and account for the spike from Labs 17 to 21. Taken individually these topics may well have had negative slopes, but combined into one problem type the slope for the distribution as a whole is neutral.

The problem types with robustly negative slopes can generally be considered to indicate successful learning. Arguably in these cases the three assumptions outlined in Section 3 hold (at least approximately), and students make progress with the underlying topics. Considering the list of negative slopes outline in Section 4.2, however, nothing obvious characterises or distinguishes this group. The inclusion of some topics, such as arrays and loops, is surprising given that they are typically regarded as very difficult concepts in introductory programming (Robins, Rountree & Rountree, 2003). The pattern observed here may in this case be a result of the extra attention paid to these topics in COMP103[4]. But in other cases no obvious factor accounts for the negative slopes.

It would be tempting to suggest that, conversely, the problem types with neutral slopes indicate that the underlying topics have not been learned as quickly. Once again, nothing obvious characterises this group of topics. Note that in some cases the difference between negative (e.g. Figure 7) and neutral (e.g. Figure 8) slopes is not obvious to casual inspection, hence in these cases the difference (while statistically reliable) is not strong.

[4] Preliminary results in this ongoing study were used to motivate a restructuring of the course in 2002 to spend more time addressing these topics.

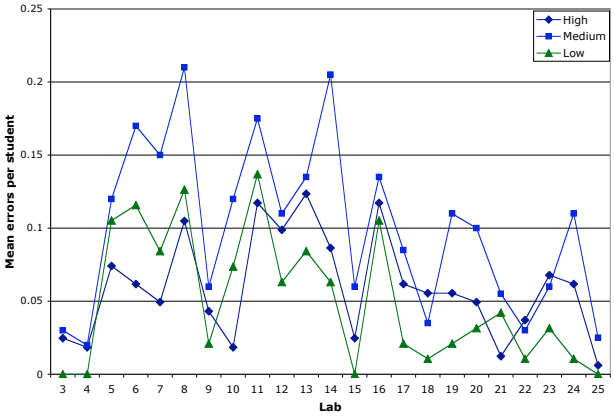

Figure 10: Distribution for S1 Control flow, high medium and low groups

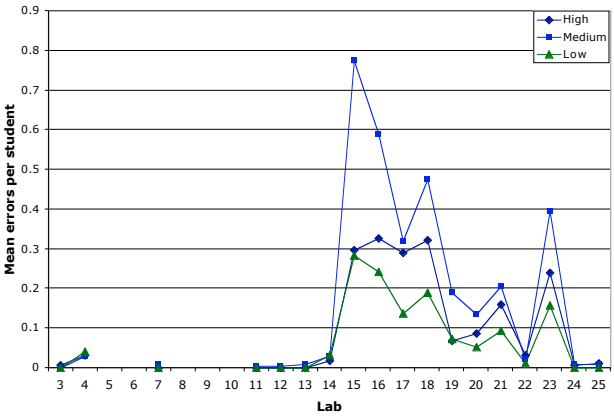

Figure 11: Distribution for S10 Arrays, high medium and low groups

What, for each group, might be an alternative explanation for observed negative or neutral slopes? An alternative explanation for the cases of negative slopes would be if the exercises relating to the underlying topics got steadily easier over the sequence of lab sessions. There is no reason to suppose that this is the case, however, and this explanation risks becoming trivial (if any improvement is automatically attributed to easier exercises). An alternative explanation for cases of neutral slopes would be if the third assumption outlined in Section 3 was false, i.e. material in later labs was inherently harder (or unevenly distributed), causing more errors, and thus balancing improvements due to learning. This seems entirely plausible, as the lab exercises are designed to build on and extend earlier topics. In this case neutral slopes may be seen as indications of areas where the lab exercises and course content need closer examination. Are exercises relating to these topics too difficult, or developing too quickly? Should more attention be paid to these topics in other course materials? Are there any systematic regularities in the kinds of topics which appear (when other factors have been taken into account) to be genuinely more difficult, and if so, what are the pedagogical implications? In future work we hope to explore some of these questions in more detail.

One further observation suggests a caveat on the interpretation of negative slopes. If these are indeed measures of learning it would be reasonable to expect that more able students learned more quickly, i.e. had steeper slopes, but as noted in Section 4.3 this is not the case. More able students (the "high" group) are characterised by a lower mean number of problems (cf. the "medium" group), but the distributions of the problem counts are very similar. It may be that the small number of labs over which the decline occurs precludes any delicate discriminations between the grade groups (a type of floor effect). But it is interesting to note that at present it appears that more able students are characterised by having fewer problems, but not by having significantly different distributions of problems.

6 Conclusions

Analysis and interpretation of the individual problem distributions presented in this paper is not straightforward. Care must be taken to consider the reliability and validity of the data collection process, artefacts of the problem list design, the clarity and difficulty of the laboratory exercises, and the predispositions of the different kinds of students engaging in the course. Bearing in mind these constraints, however, we suggest that this kind of analysis is useful in a practical sense for diagnosing aspects of course design and delivery. It may also be useful conceptually. If pedagogically interesting distinctions between different kinds of material or different kinds of learning effects can be observed, these should contribute to our understanding of the issues involved in teaching and learning a first programming language.

Acknowledgments

This work has been supported by University of Otago Research into Teaching grants. Thanks to Janet Rountree, Nathan Rountree, the demonstrators and students of COMP103, and the colleagues who have commented on earlier versions of the problem list or other aspects of the study.

References

Garner, S., Haden, P. and Robins, A. (2005): My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. *Proceedings of the Seventh Australasian Computing Education Conference (ACE2005) CRPIT 42*. Newcastle, Australia: ACS , 173-180

Gilmore D. J. (1990): Methodological issues in the study of programming. In Hoc J. M., Green T. R. G., Samurçay R. and Gillmore D. J. (Eds.): *Psychology of Programming*. London, Academic Press, 83-98

Robins, A., Rountree, J. and Rountree, N. (2003): Learning and teaching programming: A review and discussion. *Computer Science Education* 13(2): 137-172.

Sheil B. A. (1981): The psychological study of programming. *Computing Surveys* **13**:101-120.

Spohrer J. C. and Soloway E. (1989): Novice mistakes: Are the folk wisdoms correct? In Soloway E. and Spohrer J. C. (Eds.): *Studying the Novice Programmer*. Hillsdale NJ, Lawrence Erlbaum, 401-416.

Winslow L. E. (1996): Programming pedagogy -- A psychological overview. *SIGCSE Bulletin* 28(3):17-22.

Appendix

The descriptions of each problem type / code used in the study were as set out below. This list (used in 2004) is essentially the same as the old one (2003), except that:

- mnemonic problem codes such as "B1" have replaced the old problem numbers such as "1",
- problems have been explicitly divided into background problems (B1 to B3) general problems (G1 to G4) and specific problems (S1 to S21),
- problems G1 and G2 have expanded on and replaced the old problem 4, and
- the definition of G4 (old problem 6) has been refined.

B – Background Problems

B1 Tools

Problems with the Mac, OS X, directories (lost files), jEdit, Applet runner, or other basic tools. Includes being unable to find the resources described in the lab book, but not other kinds of general lab book / text book issues (do not record these). Does not include Java / file naming conventions (Problem G4).

B2 Understanding the task

Problems understanding the lab exercise / task or its "solution". In other words, whatever other problems they may be having, in this case they don't actually know what it is that the program is supposed to be doing. (Does not include minor clarifications of some detail, which do not need to be recorded).

B3 Stuck on program design

They understand the task / solution (its not Problem B2) but can't turn that understanding into an algorithm, or can't turn the algorithm into a program. Cases such as "I don't know how to get started" or "what classes should I have?" or "what should the classes do?".

G – General Problems

G1 Problems with basic structure

They have a general design and classes (its not Problem B3), but are getting basic structural details wrong. E.g. code outside methods, data fields outside the class, data fields inside a method / confused with local variables. Meant to capture problems at the class / major structural level – problems specifically with data fields or about or within methods (e.g. mixing up loops) will be some other problem code.

G2 Problems with basic object concepts

Covers very basic problems with creating and using instance objects, e.g. how many, what they are for (but not more specific problems e.g. with class vs instance Problem S16, or constructors Problem S18).

G3 Problem naming things

They have problems choosing names for things. Especially where this seems to suggest that they don't understand the function of the thing that they are trying to name.

G4 Trivial mechanics

Trivial problems with little mechanical details (where these are not better described by some other problem). Braces, brackets, semi–colons. Typos and spelling. Java and file naming conventions. Import statements (when forgotten, when misunderstood see Problem S12). Formatting output. Tidiness, indenting, comments.

This category only covers trivial problems (e.g. accidentally mismatched {}). When the underlying issue is actually a conceptual one (e.g. they don't understand the strucutre that the {} describe) use the best matching Specific Problem.

S – Specific Problems

S1 Control flow

Problem with basic sequential flow of control, the role of the main or init method. Especially problems with the idea of flow of control in method call and return (e.g. writing methods and not calling them, expecting methods to get called in the order they are written). (For issues with parameter passing and returned results see Problem S7). Does not include event driven model, Problem S21.

S2 Loops

Conceptual and practical problems relating to repetition, loops (including for loop headers, loop bodies as {blocks}).

S3 Selection

Conceptual and practical problems relating to selection, if else, switch (including the use of {blocks}).

S4 Booleans and conditions

Problems with booleans, truth values, boolean expressions (except boolean operator precedence, see Problem S13). Problems with loop or selection headers / conditions will have to be judged carefully – is this a problem formulating the boolean expression (Problem S4) or understanding how the expression / result is relevant to the loop or selection (Problems S2, S3)?

S5 Exceptions, throw catch

Problems with exceptions, throw catch.

S6 Method signatures and overloading

Problems related to overloading. Failure to understand how method signatures work / which version of a method gets called. (Includes problems with constructors that are really about the signatures of constructors – c.f. Problem S18).

S7 Data flow and method header mechanics

Especially conceptual problems with arguments / parameters and return types / values. Includes problems with method header mechanics (incorrect or mismatching parameter specifications, incorrect return types or use of void). Includes any other problems with "data flow" that are not better described by Problem S8.

S8 Terminal or file IO

Problems with terminal or file IO / data flow (not including exception handling Problem S5, or output formatting Problem G4).

S9 Strings

Strings and string functions. Does not include formatting output (Problem G4) or problems relating specifically to strings as reference types (Problem S15).

S10 Arrays

Problems relating to arrays as a data structure, including array subscripts, array contents, array declaration and initialisation (cf Problem S11). Does not include failing to understand that an array as a whole is itself a reference type or may contain references (Problem S15).

S11 Variables

Problems with the concept of or use of variables. Includes problems with initialisation and assignment. (Missing the distinction between a data field and a local / method variable is Problem G1). Does not include cases more accurately described as problems with reference types (Problem S15) or arrays (Problem S10) rather than the concept of a variable.

S12 Visibility & scope

Problems with data field visibility, local variable scope (e.g. defining a variable in one block and trying to use it in another, problems arising from unintended reuse of identifiers), and namespace / imported package problems (but not including forgotten "import" statement, Problem G4). Includes cases confusing data fields and variables of the same name, but not where this is better described as a failure to understand "this" (Problem S15).

S13 Expressions & calculations

Problems with arithmetic expressions, calculations, notation such as "++", and all forms of precedence (including boolean operator precedence, c.f. Problem S4).

S14 Data types & casting

Problems caused by failing to understand different data types and casting for primitive types (reference types are Problem S15).

S15 Reference types

Problems arising from a failure to understand the concept or use of reference types (references / pointers, "this", different references to the same object, etc), or that reference types behave differently from primitive types (when assigned, compared etc).

S16 Class versus instance

Problems understanding the class object vs. instance object distinction, including problems with class and instance data fields (and use of "static").

S17 Accessors / Modifiers

Specific problems (c.f. for example Problem S7) with the concepts of / purpose of an accessor or a modifier method.

S18 Constructors

Specific problems (c.f. for example Problems S6, S7, S16) with the concept of / purpose of a constructor.

S19 Hierarchies

Problems relating to hierarchical structure, inheritance (extends, overriding, shadowing, super), and issues relating to the use of abstract methods or interfaces.

S20 GUI mechanics

Problems with GUIs and the use of AWT, Swing etc. Includes problems with specific required methods such as actionPerfomed(), run(), implements actionListener and so on (but some issues might general problems with the concept of interfaces, Problem S19). Does not include the underlying concepts of event driven programming.

S21 Event driven programming

Problems with the underlying concepts of event driven programming, and general "flow of control" type issues that arise in the transition from application to applet.