My Program is Correct But it Doesn't Run: A Preliminary Investigation of Novice Programmers' Problems

Sandy Garner[1], Patricia Haden[2], Anthony Robins[3]

[1,3]Computer Science Department, The University of Otago, Dunedin, New Zealand

{sandy,anthony}@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. The aim is to record (as close as possible to) all of the problems encountered by students during the laboratory sessions of our introductory Java programming class. We discuss the tools and methods employed, in particular presenting the list of problem definitions which is used to classify students' problems. Data collected during 2003 are presented and discussed. The results are 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 are observed for high and low performing students. An analysis of individual lab sessions can be useful for refining course materials and teaching practice.

Keywords: novice programming errors CS1

1 Introduction

If we understand the process of learning a first programming language we can create more effective learning environments. This paper describes a study which has been running at the University of Otago since 2001. The aim of the study is to analyse the problems encountered by students in an introductory programming paper (of the kind often described as "CS1").

A valid, reliable analysis of programming students' problems would have many potential applications. On the basis of such an analysis we can adjust the amount (and kind) of attention devoted to various topics in lectures, laboratories and other resource materials. We can focus demonstrator (teaching assistant) training on the most common and / or most difficult problems. We can highlight 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. Ultimately, such interventions should help us to create a supportive and effective learning environment.

The main focus of the study is an attempt to note and classify every problem for which students seek assistance during the laboratory work for our introductory programming paper. Data has been collected, as described below, every year since 2001[1]. The experience over several years has allowed us to significantly refine the tools and processes of the study. An analysis of early data has already been useful in motivating a moderate restructuring of the paper and creating targeted resource materials. We hope to soon be able to address some of the other possibilities noted above.

This paper describes and evaluates the tools and processes of the study. Data collected in 2003 are briefly summarised and discussed in the context of the novice programming literature. One of the purposes of the paper is to invite comments and suggestions for improvements, and these can be sent to the first author.

2 Novice Programmers

The literature relating to novice programmers is large and varied. A common theme is the comparison of novices and experts, and emphasise either the sophisticated knowledge representations and problem solving strategies that expert programmers can employ (see for example Détienne (1990), Gilmore (1990a), Visser & Hoc (1990), von Mayrhauser & Vans (1994)), or the specific "deficits" of novices (see for example many of the studies presented in Soloway & Spohrer (1989), and studies reviewed by Winslow (1996)). These and other topics are reviewed in Robins, Rountree & Rountree (2003).

Novices, by definition, do not have many of the strengths of experts. Of more relevance to the teaching of a first programming language, perhaps, is the distinction between effective and ineffective novices. Some novices learn a first programming language without undue difficulty. Some novices struggle, and require a huge

[1] The title of this paper begins with a statement made by a student during the first year of the study.

amount of assistance and support, and some of course abandon the attempt to learn programming. What characterises effective and ineffective novices? Robins, Rountree & Rountree (2003) suggest that answering such questions requires an explicit focus on the topics of novice's knowledge, their strategies, and their mental models of programs and the underlying "notional machine".

Perkins, Hancock, Hobbs, Martin & Simmons (1989) suggest that there are different kinds of "novice programming style". *Movers* are students who keep trying, experimenting, and modifying their code. Movers can use feedback about errors effectively, and have the potential to solve their problems and progress. *Stoppers* are those who are unable to proceed when confronted with a problem or a lack of a clear direction. "They appear to abandon all hope of solving the problem on their own" (Perkins *et al.*, 1989, p. 265). Student's attitudes to problems are important. Those who are frustrated by or have a negative emotional reaction to setbacks are likely to become stoppers. While movers can be effective, extreme movers – *tinkerers* – are not able to trace / track their program, and can be making changes more or less at random. Tinkerers, like stoppers, have little effective chance of progressing. Although not specific to programming, there are interesting discussions of the nature of "intelligent" (effective) novices in Bruer (1993) and Ertmer & Newby (1996).

While many studies have observed problems with novice's perceptions of specific language constructs, some authors suggest that a major underlying issue is more significant. Spohrer & Soloway (1989), for example, claim that:

> "Our empirical study leads us to argue that (1) yes, a few bug types account for a large percentage of program bugs, and (2) no, misconceptions about language constructs do not seem to be as widespread or as troublesome as is generally believed. Rather, many bugs arise as a result of plan composition problems – difficulties in putting the pieces of the program together [...] – and not as a result of construct–based problems, which are misconceptions about language constructs." (p. 401).

A similar conclusion is reached by Winslow (1996):

> "[An important point] is the large number of studies concluding that novice programmers know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs." (p. 17).

With respect to object oriented (OO) vs. procedural languages. Détienne (1997) reviews claims regarding the "naturalness, ease of use, and power" of the OO approach, and concludes that such claims are not supported by the evidence. Similarly Wiedenbeck, Ramalingam, Sarasamma & Corritore (1999) note that their results:

> "…suggest that the OO novices were focusing on program model information, in opposition to claims that the OO paradigm focuses the programmer on the problem domain by modelling it explicitly in the program text." (p. 274).

The authors conclude that:

> "The distributed nature of control flow and function in an OO program may make it more difficult for novices to form a mental representation of the function and control flow of an OO program than of a corresponding procedural program…" (p. 276).

Rist (1995) describes the relationship between plans (procedural elements) and objects as follows: "Plans and objects are orthogonal, because one plan can use many objects and one object can take part in many plans" (pp. 555 – 556). Rist (1996) suggests that OO programming is not different, "it is more", because OO design adds the overheads of class structure to a procedural system. We return to these points below as a context for discussing the results arising from our study.

3 Method

In this section we describe the course on which our study is based, and summarise the tools and processes used to collect information about novice programming problems.

3.1 An Introductory Paper

The study is based on COMP103, our introductory programming course (with a typical enrolment of roughly 250 students). The course teaches Java, and consists of 26 fifty minute lectures, and 25 two hour laboratory sessions. We believe that the students who take COMP103 are typical of CS1 students at other universities in similar countries. Based on various forms of student feedback, we can safely say that COMP103 polarises student opinion. A minority hate the course, but it is well regarded by the majority. It is seen as very difficult, but also challenging, well organised, and well presented. The pass rate is roughly 70%.

As the labs form the context for our study of the problems observed, the remainder of this section briefly sketches the range of topics covered and describes the way in which lab sessions are run.

Early labs introduce Mac OS X, the jEdit IDE used to develop Java programs, and very basic programs (using statements in the main method). The 5th lab introduces other methods in the application class. The 7th lab introduces the first use of a "worker" class and an explicitly created instance object, and the following labs explore the use of constructors, accessors and mutators. Labs 11 to 14 continue to use instance objects, but focus mainly on booleans, selection ("if"), and repetition (loops). Labs 15 and 16 introduce arrays and sorting, and Lab 17 is based on arrays of objects. Labs 18 to 21 sequentially develop a project which involves writing a reasonably complex program. The OO topics covered include hierarchies, visibility, class members and abstract classes. Labs 22 and 23 introduce applets and develop a simple graphical animation. Lab 24 involves replacing a text based interface for a calculator class with a GUI interface, and the final lab involves extending any of the previously developed applets.

Each lab involves several specific sessions with up to 40 students working on the set task for the lab, which is well

specified in a workbook. When a student requests help they are visited by the next available demonstrator. There are 2 - 5 demonstrators per session (drawn from a pool of roughly 20). 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.

3.2 Problem List Design

The study is built around a problem list (see Appendix) which is used by demonstrators to classify the problems that they observe. This methodology has the typical advantages and disadvantages of naturalistic observation – see for example Sheil (1981), Gilmore (1990b). It lacks the formal rigor of an experimental study, but it is higher in ecological validity. Clearly the quality of the data collected depends on the validity of the list (a set of descriptions of possible problems), and the reliability with which it is applied.

We are not aware of any predefined criteria for validating any particular taxonomy of problem descriptions, hence the development of our problem list was an empirical process. The design of first version (2001) was based on a survey of the relevant literature and of the topics covered in the labs. It had too many problem types, demonstrators found it difficult and impractical to use. Successive refinements were based on extensive feedback from the demonstrators and from a range of computer science educators surveyed via email discussion lists and in person at workshops.

The data described in this paper were collected during 2003 using the version of the problem list shown in the Appendix. Demonstrators found this version practical, and reported that they were able to classify most problems adequately (see also the discussion of inter-rater reliability below). The list is still evolving, however, and as a result of demonstrator feedback modifications were made for the version currently in use in 2004. Specifically, Problem 4 (Problems with basic structure) was split into two separate problems, the definition of Problem 6 (Basic mechanics) was refined, and all problem numbers were replaced with codes that made more explicit a grouping of Background problems (1 to 3 in the Appendix), General problems (4 to 6), and Specific problems (7 to 27).

The main tradeoff involved in designing such a list is between richness (implying a large number of problem descriptions that can support a detailed classification) and practicality (implying a small number of problem descriptions that different demonstrators / raters can become familiar with and use reliably).

3.3 Data Collection

The major factor constraining the collection of data is that any process adopted 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 adopted. 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 numbers. In order to try and provide some structure for standardising these decisions, two guidelines were stressed:

> At the end of a session with a student record the numbers 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 numbers, corresponding to the three most important topics that you helped with.

At the start of the course the 20 demonstrators (most of whom were already experienced with the study from previous years) were introduced to the problem list and trained on the data collection process. The most effective training was provided via a senior coordinating demonstrator who was present in most lab sessions to provide support and help adjudicate difficult cases.

All demonstrators completed checklists for all or almost all their scheduled lab sessions. We are confident that we achieved a very high degree of coverage for data collection. A major issue for this methodology, however, is inter-rater reliability. To attempt to assess this we had the senior demonstrator "shadow" each of the other demonstrators for at least one lab session, listening to the discussions with students, and making her own independent checklist entries which were then compared with the demonstrator's checklist. Most demonstrators had good agreement with the senior demonstrator. A minority had poor agreement, often recording a conspicuously narrow range of problem numbers, and seeming to overuse particular problem numbers (such as Problem 6). 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 largely reliable, although this is a factor that we intend to further explore and improve in future years.

4 Results

For our roughly 250 students over 25 labs (with a maximum of 3 problem numbers per help session) a total of 11240 problem numbers were recorded. Each one is associated with the lab session, the recording demonstrator, and a specific student (data were not collected from the 2.5% of students who declined consent). Clearly a range of possible questions can be explored with this data set, focusing on the frequencies of problem types, the problems experienced by different classes 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 the sections below we explore two of the basic characterisations of the whole data set, and a breakdown of the problem counts for selected example labs.

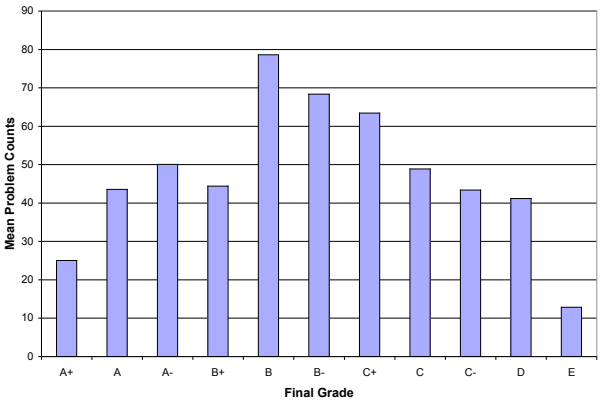

Figure 1: Mean problem counts by final grade

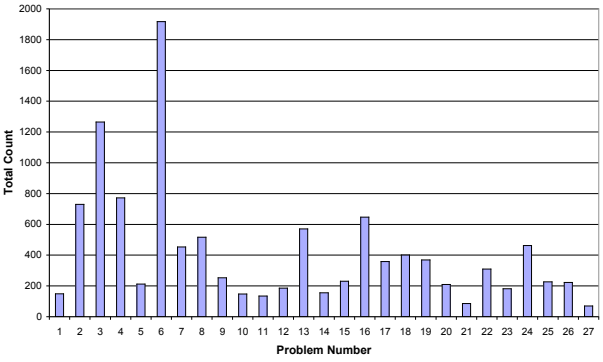

Figure 2: Total problem counts by problem number

4.1 Mean problem counts by grade

How many problems do students have, and are there systematic differences for students of differing ability? To explore these questions students were grouped according to their final grade in the course (A+ to E). The mean number of problems recorded for the students in each grade group is shown in Figure 1 (for example students who received an A- had on average 50 problems recorded over the 25 labs).

Given the anecdotal accounts of the multiple problems experienced by less able students, we were surprised to find that the mean problem counts have a roughly normal distribution. In retrospect, however, this distribution can be accounted for by two main effects. Firstly, for average to weak students (scoring "B" or lower), the more assistance they receive from demonstrators the better their achievement in the course. (This is a very reassuring validation of the excellent job done by the demonstrators!) The lower problem counts for weaker students may be a result of a reluctance to ask for help, and / or a result of a lower rate of lab attendance. At the other end of the spectrum the declining number of problems noted for the stronger (B+ or better) students is almost certainly a reflection of the fact that these students simply require less help.

4.2 Total problem counts by problem number

How common is each problem? Figure 2 shows the total number of times that each problem type / number (see Appendix) was recorded. We were very surprised at the clear dominance of Problem 6 (Basic mechanics). Although the problem definition is very broad, it is supposed to cover minor errors, which we expected to decrease in frequency as the labs progressed. Over the counts for each individual lab however, Problem 6 was remarkably robust, having the highest recorded count in the majority of labs. Similarly the breakdown of problem counts by students' final grades shows that Problem 6 dominated not just for weaker students, but at all levels. In short, students very persistently keep seeking assistance for problems with basic syntactic details[2].

The next most frequent problems are 3 (Stuck on program design), 4 (Problems with basic structure) and 2 (Understanding the task). For these very fundamental problems there was considerable variation when broken down over individual labs, which probably reflects the extent to which the different labs specified a clear structure for a program or left the design more open. There was also a clearly increasing trend in the incidence of these problems over the range of student abilities (collective they account for 18% of the problems experienced by A+ students, and 34% of the problems experienced by E students). The prominence of these problems overall supports the claims (noted in Section 2 above) that issues relating to basic design can be more significant than issues relating to specific language constructs. The more complex factors underlying the distribution of these problems (cf. Problem 6), however, suggest that such broad claims may mask more subtle interactions.

The next most frequent problems are 16 (Arrays), 13 (Data flow and method header mechanics), 8 (Loops), 24 (Constructors) and 7 (Control flow). It is notable that for these more specific problems the procedural / algorithmic aspects of Java are causing many more problems than the OO aspects. Arrays and loops are often noted in the literature as being particularly problematic, even in purely procedural languages. Control flow and data flow are of course procedural issues. The only clearly OO problem in this group is constructors, and the count for this problem may in part reflect underlying design problems that manifest themselves as not knowing when objects should be used. In short, the well documented problems that novices experience learning the procedural aspects of a programming language are clearly dominant in our study of COMP103. If it is indeed the case as Rist (1996)

[2] It is possible of course that high count for Problem 6 is an artefact of the data collection process. As noted above the less reliable demonstrators appeared to over-use this category, and it is possible that other demonstrators did so too when not being shadowed. We believe, however, that the total count is much too high to be entirely due to an artefact of this kind. We are attempting to further explore this issue with an even stronger definition of this problem in the version of the problem list in use in 2004.

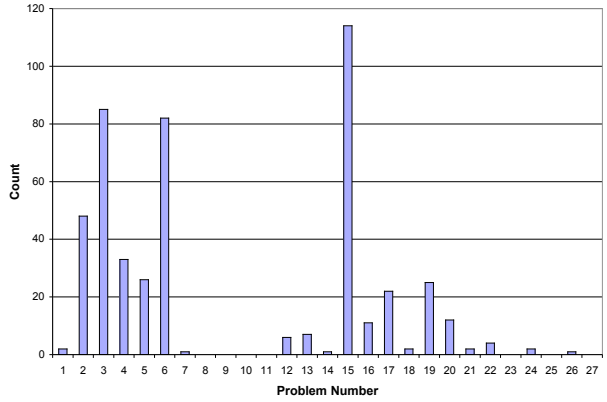

Figure 3: Problem counts for Lab 4

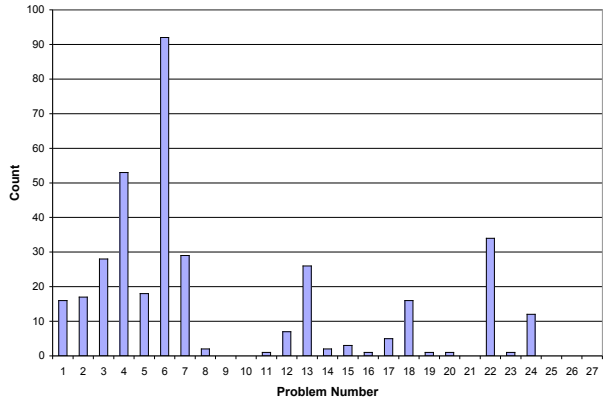

Figure 4: Problem counts for Lab 7

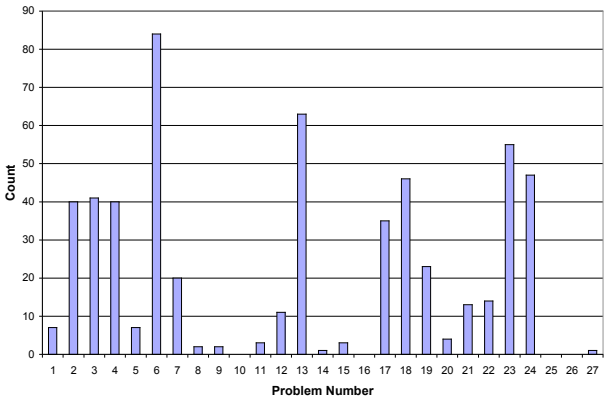

Figure 5: Problem counts for Lab 10

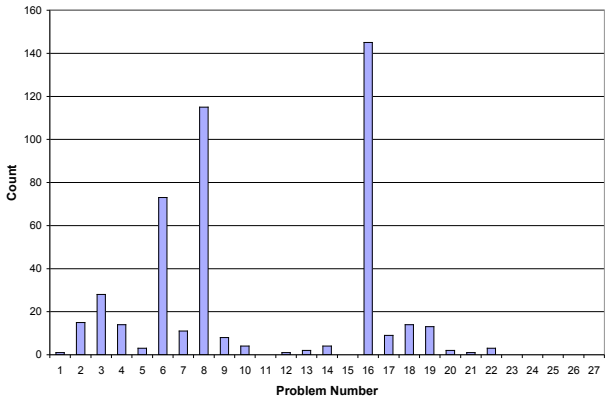

Figure 6: Problem counts for Lab 15

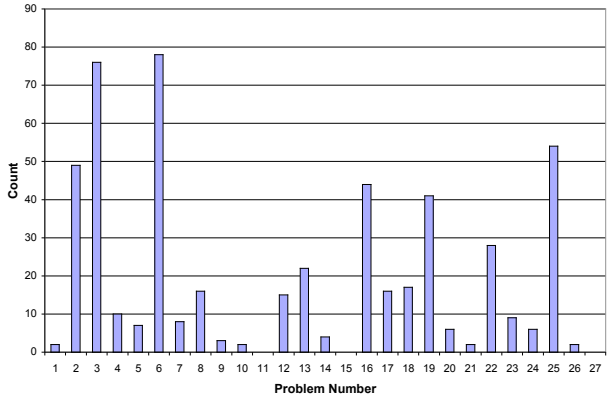

Figure 7: Problem counts for Lab 21

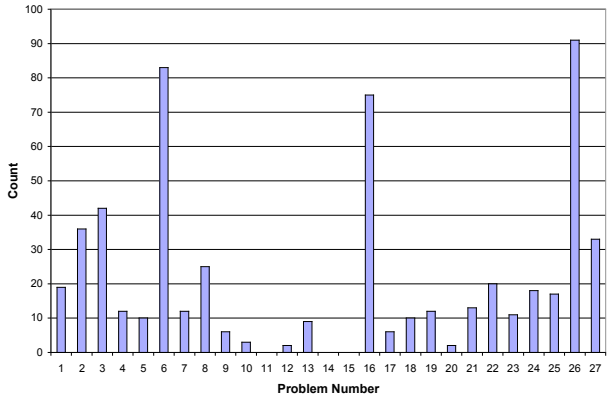

Figure 8: Problem counts for Lab 23

suggests, that learning an OO language adds an orthogonal component in addition to the task of mastering procedural concepts, then the problems caused by this additional component do not appear to be anywhere near as frequent as the problems caused by the procedural concepts themselves.

4.3 Problem counts for individual labs

When considering the problem counts for individual labs / tasks there is, of course, a steady progression of new problems being experienced as new topics are introduced. In addition to this basic trend, however, some further observations can be noted. Figures 3 to 8 show the problem counts for a range of example labs. Lab 4 is an example where a specific topic, strings, is introduced, and where the problem type associated with the new topic (Problem 15) clearly dominates. This is, however, the exception rather than the rule!

Lab 7 is the first lab to deal with explicitly creating and using an instance object. Problem counts relating to class vs. instance (Problem 22) and constructors (Problem 24) are not high however, and issues relating to basic mechanics (Problem 6) dominate. In cases like this where a new topic does not appear to cause problems we suspect that many students are successfully following instructions without fully engaging with the new concepts. Problems manifest themselves when these new concepts need to be used later in more creative ways (e.g. both Problems 22 and 24 have much higher counts in some later labs). In short, introducing a new topic without apparent problems does not necessarily indicate that the underlying concepts have been grasped.

The focus of Lab 10 is accessors and mutators (Problem 23), but issues relating specifically to the procedural matter of data flow via method headers (Problem 13) are just as significant, and both still fall behind the ubiquitous Problem 6. Lab 15 combines both loops and arrays. The relevant problem counts (8 and 16) clearly reflect the particularly problematic nature of these topics (note the scale of the *y* axis!). Lab 21 involves the use of an abstract class with instances of two subclasses. As expected Problem 25 (Hierarchies) features, but the unusually high count for Problem 3 (Stuck on program design) is probably also indicative of the complexity of this new concept. The very high count for Problem 2 (Understanding the task) may well suggest that the description of this lab needs to be rewritten and improved. Lab 23 is the second to employ Swing and explore GUI concepts. As expected Problem 26 (GUI mechanics) dominates, but note that arrays (16) and basic mechanics (6) are not far behind.

5 Discussion

In this paper we have described an ongoing study where a large amount of data, representing almost complete coverage for a sizable population of students, is being collected. To the extent that the problem list represents a valid taxonomy and is reliably applied by demonstrators, this data is a rich and potentially very useful characterisation of the problems encountered by students in our introductory programming course. We have described the tools and methodology of the study, and invite comments or feedback that could improve it further.

Given the goals that most computer science educators would espouse, an observation that recurs with depressing regularity, both anecdotally and in the literature, is that the average student does not make much progress in an introductory programming course (Linn & Dalbey, 1989; Kurland, Pea, Clement & Mawby, 1989). Winslow observes that "One wonders […] about teaching sophisticated material to CS1 students when study after study has shown that they do not understand basic loops..." (Winslow, 1996, p. 21). The results reported in this paper are certainly consistent with trends identified in the literature. Fundamental problems relating to basic program design are very persistent, as are problems relating to the procedural aspects of programming, and the particular topics of loops and arrays. We were surprised, however, at the persistence, frequency, and uniform distribution of problems relating to basic syntactic details.

As well as this "external" consistency, the observations reported in this paper are also internally consistent. In Figure 1 for example two very different measures – students' final grades and the number of problems recorded for them in labs – interact in very structured way (producing an approximately normal distribution). This converging evidence gives us some confidence that the tools and methods of the study are achieving at least reasonable levels of validity and reliability. Results collected in previous years have already been used to guide a reorganisation of the course (to spend more time on loops and arrays) and the creation of targeted resource materials (see http://www.cs.otago.ac.nz/comp103/help/). The analysis of individual labs, such as those described above, have also helped to identify areas where the course can be further fine tuned. In future years, with improved support for inter-rater reliability, we hope to explore a more detailed analysis of factors that predict success or failure, and the structure of individual student problem profiles.

Acknowledgments

This work has been supported by University of Otago Research into Teaching grants. Thanks to Janet Rountree, Nathan Rountree, Yerin Yoo, 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

Bruer J. T. (1993): *Schools for thought: A science of learning in the classroom.* Cambridge MA, MIT Press.

Détienne F. (1990): Expert programming knowledge: a schema based approach. In Hoc *et al.* (1990), 205-222.

Détienne F. (1997): Assessing the cognitive consequences of the object-oriented approach: a survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers* **9**:47-72.

Ertmer P. A. & Newby T. J. (1996): The expert learner: Strategic, self-regulated, and reflective. *Instructional Science* **24**: 1 - 24.

Gilmore D. J. (1990a): Expert programming knowledge: a strategic approach. In Hoc *et al.* (1990), 223-234.

Gilmore D. J. (1990b) Methodological issues in the study of programming. In Hoc *et al.* (1990), 83-98.

Hoc J. M., Green T. R. G., Samurçay R. and Gillmore D. J. (Eds.) (1990): *Psychology of Programming*. London, Academic Press.

Kurland D. M., Pea R. D., Clement C. and Mawby R. (1989): A study of the development of programming ability and thinking skills in high school students. In Soloway and Spohrer (1989), 83-112.

Linn M. C. and Dalbey J. (1989): Cognitive consequences of programming instruction. In Soloway & Spohrer (1989), 57-81.

Perkins D. N., Hancock C., Hobbs R., Martin F. and Simmons R. (1989): Conditions of learning in novice programmers. In Soloway and Spohrer (1989), 261 – 279.

Rist R. S. (1995): Program Structure and Design. *Cognitive Science* **19**:507-562.

Rist R. S. (1996): Teaching Eiffel as a first language. *Journal of Object–Oriented Programming* **9**:30-41.

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.

Soloway E. and Spohrer J. C. (Eds.) (1989): *Studying the Novice Programmer.* Hillsdale NJ, Lawrence Erlbaum.

Spohrer J. C. and Soloway E. (1989): Novice mistakes: Are the folk wisdoms correct? In Soloway and Spohrer (1989), 401-416.

Visser W. and Hoc J. M. (1990): Expert software design strategies. In Hoc *et al.* (1990), 235-250.

von Mayrhauser A. and Vans A. M. (1994): Program Understanding – A Survey. Technical Report CS–94–120, Department of Computer Science, Colorado State University.

Wiedenbeck S., Ramalingam V., Sarasamma S. and Corritore C. L. (1999): A comparison of the comprehension of object–oriented and procedural programs by novice programmers. *Interacting With Computers* **11**:255-282.

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

Appendix

The problem descriptions used in the study were:

1 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 6).

2 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).

3 Stuck on program design

They understand the task / solution (its not Problem 2) 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?".

4 Problems with basic structure

This problem number is meant to capture problems at the class / object / major structural level - they have a general design and classes (its not Problem 3), 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 (problems specifically with data fields or about or within methods, e.g. mixing up loops, will be some other problem number). Includes 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 22, or constructors Problem 24).

5 Problem naming things

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

6 Basic mechanics

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 18). Formatting output. Tidiness, indenting, comments.

7 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 13). Does not include event driven model, Problem 27.

8 Loops

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

9 Selection

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

10 Booleans and conditions

Problems with booleans, truth values, boolean expressions (except boolean operator precedence, see Problem 19). Problems with loop or selection headers / conditions will have to be judged carefully – is this a problem formulating the boolean expression (Problem 10) or understanding how the expression / result is relevant to the loop or selection (Problems 8, 9)?

11 Exceptions, throw catch

Problems with exceptions, throw catch.

12 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 24).

13 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 14.

14 Terminal or file IO

Problems with terminal or file IO / data flow (not including exception handling Problem 11, or output formatting Problem 6).

15 Strings

Strings and string functions. Does not include formatting output (Problem 6) or problems relating specifically to strings as reference types (Problem 21).

16 Arrays

Problems relating to arrays as a data structure, including array subscripts, array contents, array declaration and initialisation (cf Problem 17). Does not include failing to understand that an array as a whole is itself a reference type or may contain references (Problem 21).

17 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 4). Does not include cases more accurately described as problems with reference types (Problem 21) or arrays (Problem 16) rather than the concept of a variable.

18 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 6). 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 21).

19 Expressions & calculations

Problems with arithmetic expressions, calculations, notation such as "++", and all forms of precedence (including boolean operator precedence, c.f. Problem 10).

20 Data types & casting

Problems caused by failing to understand different data types and casting for primitive types (reference types are Problem 21).

21 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).

22 Class versus instance

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

23 Accessors / Modifiers

Specific problems (c.f. for example Problem 13) with the concepts of / purpose of an accessor or a modifier method.

24 Constructors

Specific problems (c.f. for example Problems 12, 13, 22) with the concept of / purpose of a constructor.

25 Hierarchies

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

26 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 25). Does not include the underlying concepts of event driven programming.

27 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.