How can software metrics help novice programmers?

Rachel Cardell-Oliver

School of Computer Science and Software Engineering
The University of Western Australia,
M002, 35 Stirling Highway, Crawley, WA, 6009
Email: rachel.cardell-oliver@uwa.edu.au

Abstract

Many computing education studies have reported poor learning outcomes in programming courses for novices. Yet methods for measuring students' ability to generate computer programs remains an open research problem. In this paper we review some limitations of existing approaches for assessing student programs. We then propose a set of valid and reliable metrics for the direct measurement of novices' program code. Distributions of each metric are given for student populations. The metrics can be utilised for both diagnostic and formative assessment. Examples of formative assessments are given and a new diagnostic metric is presented for Perkins' "stoppers" and "movers" learning styles. This metric captures the multi-dimensional distance of a student's program from a target solution. The metric can be used by instructors to triage a large set of submissions and to tailor formative feedback to individuals.

1 Introduction

Programming courses for novices have two main learning outcomes: the ability to comprehend (read) program code and the ability to generate (write) program code. Although many studies have reported poor learning outcomes, the research problem of how to *measure* those outcomes has received surprisingly little attention. In particular, while criterion-based analysis of learning outcomes related to program comprehension has been the subject of several major studies (Lister & Leaney 2003, Lister et al. 2004, Decker 2007), the problem of criterion-based measurement of students' code generation skills remains an open problem.

How can learning outcomes related to novices' ability to generate program code be measured ? The first part of this paper (sections 2 and 3) reviews some limitations of existing approaches for measuring student programs and in particular highlights problems with the use of students' final grade as a metric to validate claims in empirical studies. In the second part of the paper (sections 4 to 6) we propose a suite of software metrics to measure the quality of students' programs, and so to inform strategies for improving programming skills. The focus of this paper is on software metrics that provide useful feedback to students (formative assessment) and to their lecturers (diagnostic assessment) (Crisp 2008). Our metrics can also be used for grading assignments (summative assessment) but that is not our primary goal here.

This paper addresses two research questions:

1. which software metrics are appropriate (or not) for measuring novices' ability to generate program code? and
2. how can measurement with software metrics contribute to student learning?

In Sections 2 and 3 we answer the first question by reviewing some representative studies that evaluate novice students' ability to write computer programs. Several common pitfalls in the use of metrics in such experiments are identified. For example, we show that final grade is neither a valid nor reliable metric for drawing conclusions about students' code generation ability. In section 4 we propose a list of significant attributes for novices' programs together with metrics for each attribute. These commonly used metrics are both valid and reliable for the attributes they measure (Fenton & Pfleeger 1998). Sample distributions of the metrics are shown for large student populations for a selection of different Java programming exercises.

Sections 5 and 6 address the second research question by demonstrating two ways in which metrics can be used in programming courses for formative and diagnostic assessment. Section 4 presents examples of the formative feedback generated by the measurement tools used by students during their laboratory sessions. Section 6 considers the role of metrics for diagnostic assessment, that is feedback to lecturers about problems encountered by a cohort of students. We show how students can be classified according to Perkins' "stoppers", "movers" and "tinkerers" learning styles (Perkins et al. 1989) using a distance measure between the metrics vectors of a canonical solution program and the code submitted by each student.

2 Assessment of Code Generation Skills

This section focuses on related work that proposes specific methods for measuring the programs generated by novices. These approaches can be evaluated in terms of the validity, reliability and cost of the metrics they propose using the validation framework of Kitchenham *et al* (Kitchenham et al. 1995). *Validity* is the extent to which a measurement reflects the property of interest, in our case novices' ability to generate correct, efficient and readable computer programs. Establishing the validity of a metric requires examination of detailed and explicit criteria and their measurement instruments (Kitchenham et al. 1995). *Reliability* is whether the measurement is repeatable and is in agreement with other measures for the same property. Poorly documented grading schemes may be interpreted differently by different markers, and so they have low reliability. *Cost* of a metric is the

effort required to make the measurement. Metrics that can be evaluated automatically by a computer have lower measurement cost than those that require detailed inspection by a human marker.

In 2001 an international study was undertaken to measure code generation skills (McCracken et al. 2001). 217 students from different institutions took a laboratory examination in which they wrote programs for a set of mathematical problems about reverse-polish and infix calculators. The students' programs were evaluated against a set of well documented general evaluation criteria that included executable tests for correctness and expert inspection of program style. For programs that did not work a "degree of closeness" metric was also used as a subjective evaluation of how close a student's source code was to a correct solution. The performance of students in all the test groups against all these criteria was much lower than their instructors expected with an average general evaluation score of 22.9 out of 110 and an average degree of closeness score of 2.3 out of 5.

A number of flaws with the study were identified by the authors and others (McCracken et al. 2001, Decker 2007). The test had limited validity for its purpose because results were affected by a number of external factors. The mathematical focus of the problem set, the difficulty of the required data structures and problems with presentation and instructions for the test all created cognitive overload for the test that many students were unable to overcome. Detailed rubrics for assessing the exercise meant that grades were reliable for the given exercise, but the reliability of the measures have not been tested on any other programming tasks. The results presented using averages and standard deviations are inappropriate summary measures for these skewed populations. Instead, the median and quartile ranges should be presented. Overall, the metrics used proved neither valid nor reliable for general use and the cost of collection was high.

Autograder is a system for automatically grading student programs (Morris 2003). Laboratory exercises and automatic marking scripts are developed using a 10 step process starting with a canonical instructor-written solution and test suite, to detailed specifications for the students and finally post-facto feedback on the quality of the grading process. This assessment has criterion-referenced validity in that tests are provided for each of the requirements given to students in the specification. However, the system appears to mark only functional correctness, and does not consider planning, style, efficiency or other non-functional quality attributes of programs. Furthermore, the problem of degree of closeness of non-working solutions is addressed by providing work-arounds for "minor" errors such as naming and output format errors using regular expression matching, Java reflection, and overriding of directly or indirectly dependent methods. Although these work-arounds meet the goal of satisfying students that the marking is "fair" in recognising their effort, they lower the validity of the metric for the overall goal of measuring program generation ability. In our view a better approach is to set a target standard for assessment, but then allow students to refactor their code and resubmit it rather than for the instructor to create artificial work arounds. For these reasons the system has limited validity for the purpose of measuring students' overall program generation skills. Autograder is fully automatic once the programming exercise and its marking script have been developed, so it has low measurement cost even for large cohorts of students.

Lister and Leany present a criterion-based framework for assessing both code generation and code comprehension skills (Lister & Leaney 2003). The focus of that paper is on how to *specify* criteria for outcomes rather than how to *measure* outcomes as in this paper. Explicit and clear criteria are communicated to the students for the grades of fail, pass, credit, distinction and high distinction. For example, the criteria to be satisfied by a credit or distinction student include "to have demonstrated, within a small well-defined program context, that you are able to: (a) Apply the basic OO programming concepts of classes, instances, events, and methods. (b) Apply the basic program control constructs of sequence, selection and iterations. (c) Take an informal problem description and translate it into a readable, working program. (d) Apply the basics of testing and debugging." Programming skills were assessed in an assignment with several parts: 5 exercises adding small amounts of code (10 to 15 lines) to an existing system, and the final exercise to add an entire new class. Details of the measurement protocol and marking criteria are not given in the paper but the approach is certainly amenable to the specification of detailed metrics and measurement protocols for each criteria as introduced in this paper, in which case the validity, reliability and cost of the tests could be assessed.

There are many other published studies that use software metrics for assessment. Their goals range from detecting plagiarism to teaching software quality. In this paper we can only discuss a representative sample of such studies. More broadly, this paper draws on prior research on how students learn to program (Robins et al. 2003), on measuring learning outcomes in CS1 programming courses (Lister et al. 2004, Decker 2007, Robins 2010), on the automatic assessment of programming (Ala-Mutka 2005), and the theory of software metrics (Fenton & Pfleeger 1998, Kitchenham et al. 1995).

3 Evaluation of Final Score as a Metric

Many studies of student learning in CS1 draw conclusions based on students' final letter grades or percentage scores. In this section we identify some difficulties with this approach and caution against using either final grades or final scores as the measurement baseline for empirical studies.

Final grades in University courses are assigned as a weighted sum of assessed components. They may be adjusted to meet institutional requirements for pass rates and grade distribution. This is called norm-referenced grading (Lister & Leaney 2003). In CS1 courses assessed components include programming exercises, programming projects, tests and examinations. Final score is most often not a valid metric and studies based on final grades can be difficult to interpret because "when using overall course grade as the success marker, one should know if there was a curve placed on the grades, or even the basic breakdown of what is considered A work". (Decker 2007).

Furthermore, final score or grade is not a reliable metric because it is highly sensitive to minor changes. Figure 1 shows typical distributions of student scores for different types of assessment taken from a particular cohort of 180 students, being those students who sat the final exam out of 200 who enrolled initially. The scores shown are for 4 of the 8 components used for the final mark. Each component has been scaled to a mark out of 20 for easier comparison. The top figures (a) and (b) are practical work scores and the bottom figures (c) and (d) are examination scores. Three of the components (a, b and c) assessed code generation skills while exercise (d) assessed code comprehension skills.

It is interesting to note that practical work components (a) and (b) exhibit the typical bimodal distribu-

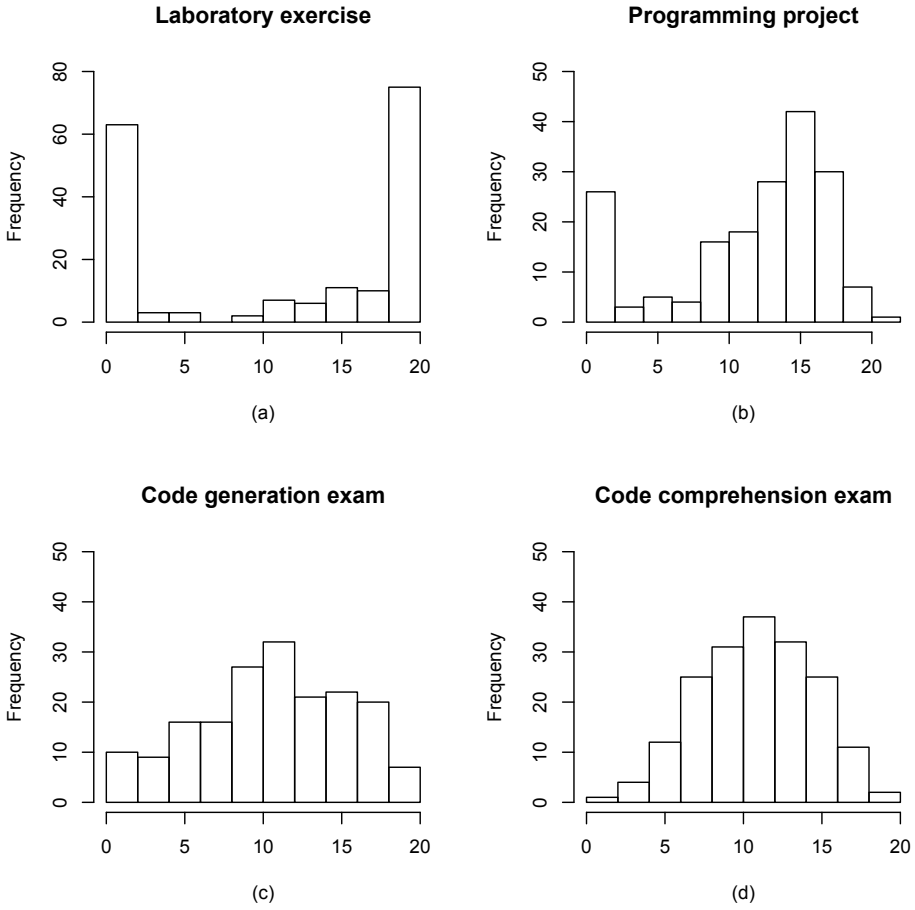

Figure 1: Distribution of Scores for Practical (a,b) and Written (c,d) Components assessing Code Generation (a,b,c) and Code Comprehension (d)

tion cited in many studies (Robins 2010). However, written components (c) and (d) are much closer to a normal distribution. Low scores in practical work occur when students either fail to submit, or submit almost nothing. At the other end of the marking scale, students who submit a "mostly working" program typically score well in the practical work, since in most cases lab exercises are designed for formative (rather than summative) learning. To this end, students are provided with test cases and style checking software so that most should be able to complete the exercise correctly. It is dangerous to make assumptions about the students who do *not* to submit, that is the lower group of the bimodal distribution. For example, students' behaviour in first year courses is shaped by the way they cope with the new environment of university study and the need for many to adopt more independent learning strategies than they used at school. On the other hand, we have observed that bimodal distributions occur in coursework components for courses at all levels of our degree and in both programming and non-programming subjects. The common factor seems to be that in courses that have a high number of assessed components, and where the work required to obtain a good mark for a component is not reflected in the assessment weight, that students make strategic decisions about whether to submit that work or not. A typical student will complete some but not all of the low-weighted lab exercises.

Finally, we mention some unwanted side effects of the standard practice of calculating a final score in computer programming courses as a weighted sum of assessed practical and written components. Two different types of distribution have been observed for assessed tasks in CS1: normal and bimodal. The choice of weights for the summed components leads to either the bimodal distribution observed in some institutions, or the normal distribution observed in others. That is, final score is not a valid nor a reliable metric. For these reasons extreme caution must be used about any conclusions drawn from correlations with the distribution of final marks. This includes claims about failure rates (high or low) since the failure rate is highly sensitive to the weighting of assessment components.

It is outside the scope of this paper to present a detailed alternative to the norm-referenced schemes used for summative assessment in most existing courses. A better approach is to use criterion-referenced (rather than norm-referenced) assessment for final grades (Lister & Leaney 2003) and certainly for empirical studies that aim to measure the effectiveness of new pedagogies. The main problem with the final grade metric is that it combines different aspects of a multi-dimensional attribute (students' ability to read and to generate code) into a single measurement. This approach almost always leads to problems with validity and reliability (Kitchenham et al. 1995). From the measurement theory point of view the correct approach is to use a *vector* of measurements rather than a single value to capture multi-dimensional properties such as the quality of program code. Full details can be found in textbooks on software metrics (Fenton & Pfleeger 1998). In the next sections of this paper we demonstrate ways in which

such vectors of measurements can be used to help novice programmers.

4 Software Metrics for Novice Programmers

Software metrics are used to measure the quality of the software produced by professional software engineers. The rationale for software measurement in industry is to improve the quality of the software production process and the quality of the software end product. Can software metrics offer the same benefits for novices?

In software metrics practice, first the specific goals of measurement are identified and then attributes that contribute to each goal are chosen together with measures for each attribute (Fenton & Pfleeger 1998). This practice is essentially the same as defining a criterion-referenced grading scheme (the goals) together with measures and measurement instrument is provided for each criteria (the attributes and their measures).

In education there are three main goals for measurement: diagnostic, formative and summative (Crisp 2008). *Diagnostic assessment* is measurement of the performance of a cohort of students in order to identify any individual learning difficulties, common problems, and areas of strength. *Formative assessment* provides feedback directly to students with the goal of helping them to improve their understanding of the subject. *Summative assessment* measures how well students have achieved learning objectives. This paper focuses on measurement for diagnostic assessment and for formative feedback to students.

Five areas for measurement and a collection of established metrics for each have been identified for our overall goal of measuring students' ability to generate program code. Each metric is valid and reliable for the attribute it measures and most of the metrics can be collected automatically using open-source software engineering tools. A measurement vector can thus be produced automatically for each Java class submitted by a student.

Program Size Non-comment lines of code, lines of code, number of methods, number of fields, and subsets of these based on Java modifiers. Measured using the Java Reflection library and a lines of code counter application.

Functional correctness Vector of individual test case results (pass, fail or error), number of test cases run, number of tests passed, code coverage of a test suite, input coverage of a test suite, coverage of other properties such as algorithmic complexity. Measured using JUnit with instructor-written or student-written test cases and the Emma code coverage tool.

Efficiency Execution time for a given test set. Measured using JUnit.

Program Style Number of code violations for naming conventions, hiding, complexity, coding conventions, magic numbers. Measured using Checkstyle and PMD.

Client Validation A client acceptance test of a sequence of user inputs and defined responses, assessed against a checklist of items. This metric is not fully automatic but requires some expert inspection. It can be used for assessing graphical programs.

Figure 2 shows the population distributions of metrics for the attributes of program size, functional

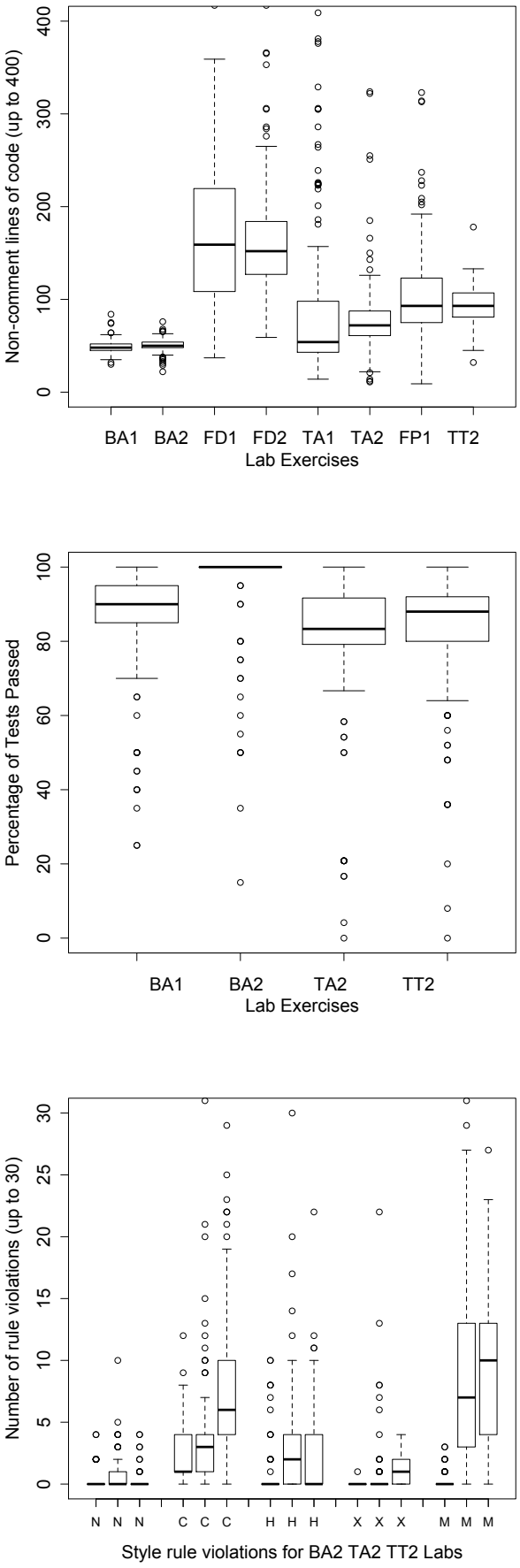

Figure 2: Metric Population Distributions for Program Size (top), Functional Correctness (middle) and Program Style (bottom) for categories N=naming, C=coding, H=hiding, X=exceptions, M=magic numbers

correctness and program style. Programming exercises are identified by three letter identifiers, each a single Java class. The exercises are presented in the order they appear during the course: that is, from easiest to hardest. The number of submissions for each of these exercises ranges from 132 to 200. The central line of each box is the median, and the box shows upper and lower quartiles of the population. Limits of the main population (99% range) are shown by the whisker dotted lines and outlier values are indicated by dots.

In a typical laboratory exercise the signatures of all public methods are given and the students provide implementation code for each method. Given these constraints on student classes we might expect the distributions for each metric to have small variation. However, Figure 2 shows that the range and variation for each metric is large and that most populations have a long tails. These examples of student populations for selected software metrics show how much student programs differ from one another. Section 6 explores some possible reasons for this variation.

5 Formative Feedback

Our software metrics are simply a number or vector for some attribute of a Java class. The metrics are therefore not much use on their own to improve student learning. Fortunately, the measurement tools we use can be configured to generate informative messages that alert students to the defects in their code and provide some advice about how to fix them.

During the laboratory session students use JUnit, Checkstyle and PMD to check their code. JUnit tests are written by the instructor. If the student's code fails a test case then an error message is provided to explain the problem and help them to identify what needs to be done. For example, the following messages are generated when a student mistakenly initiates a character count variable to 1 instead of 0.

```
Initial character count should be 0
     expected:<0>  but was:<1>
There are no alpha chars in the string " ***  42  !!"
     expected:<0> but was:<1>
Error code ? expected for mostFrequent in empty string
     expected:<?> but was: <a>
```

PMD identifies style violations and returns feedback messages that refer to a line of code and the problem identified. For example, the following error messages are a sample of those generated for code that passed all the functional tests, but was highly complex and inefficient.

```
32 Avoid unnecessary comparisons in boolean expressions
180 Avoid really long methods.
180 Method names should not start with capital letters
293 The method 'isChar' has Cyclomatic Complexity of 54.
```

After the student has completed a lab exercise they submit their source code for assessment. The program is assessed using the same tools as used in the lab and a summary of the results is emailed to the student. The following example shows the type of formative feedback provided in these emails:

```
Highly inefficient code:
Your class executed in 10.207 seconds and the cohort
median execution time was 0.049 seconds.
If your execution time is much higher than the
median then you can improve efficiency.
Ask a lab demonstrator for help with this.

Your class has  00255 non-comment lines of code
(NCLOC).  The expected NCLOC was around 70.
Warning (only, no marks): If your code is very long
(say, over 100 lines) then it may be using some
Java types such as arrays incorrectly.
Ask a lab demonstrator for help with this.
```

6 Metrics for Diagnostic Assessment

This section demonstrates how metrics vectors of attributes of student code can provide diagnostic information about the difficulties students are having. This in turn can be used to modify course delivery, providing ways for students to catch up and focussing help where it is needed to address problems in student learning.

Many CS1 courses offer a sequence of laboratory programming exercises for formative assessment. In an ideal world, students should correct problems in their code as soon as possible, rather than waiting for a submit, mark and return assessment cycle. For this reason, in our lab classes students are provided with JUnit and instructor-written test cases, as well as the style checkers PMD or Checkstyle with a selection of coding rules specifically configured for novice programmers. That is, all the metrics discussed in the last section are available to students from when they first start working on their assignments. However, as shown in Figure 2 this does not mean that when the exercises are submitted that there are no defects.

Our approach to diagnostic assessment is illustrated using Perkins' classification of novice programmers' learning styles as "stoppers" and "movers".

> "When novice programmers see fairly quickly how to proceed, naturally they do so. When, however, a clear course of action does not present itself, the young programmer faces a crucial branch point: what to do next? ... Some students quite consistently adopt the simplest expedient and just stop. They appear to abandon all hope of solving the problem on their own." (Perkins et al. 1989), page 265

Stoppers are students who tend to stop and give up when they can not immediately see how to proceed with a problem. Non-starters are stoppers who may make some progress in lab classes but choose not to submit their programs for assessment. Movers, on the other hand, will try different approaches when faced with a problem. Movers can be further divided into two types. Extreme movers are called "tinkerers". When faced with a problem they make changes but more or less at random:

> "Students often program by means of an approach we call tinkering - they try to solve a programming problem by writing some code and then making small changes in the hopes of getting it to work." (Perkins et al. 1989), page 272

The ability to diagnose a students' learning style is useful because different types of learners require different types of feedback. Movers are the most successful learners. They are adept at solving problems on their own and are likely to thrive whatever teaching and learning environment they are in. Tinkerers and stoppers both have learning strategies that interfere with their progress in solving programming problems. We have observed in our own classes that stoppers can be discouraged by receiving a detailed list of defects in their work, and this can easily turn stoppers into non-starters. On the other hand, movers and tinkerers can usually make good use of detailed feedback and will re-evaluate and improve their programs. To gain maximum benefit from explicit feedback all

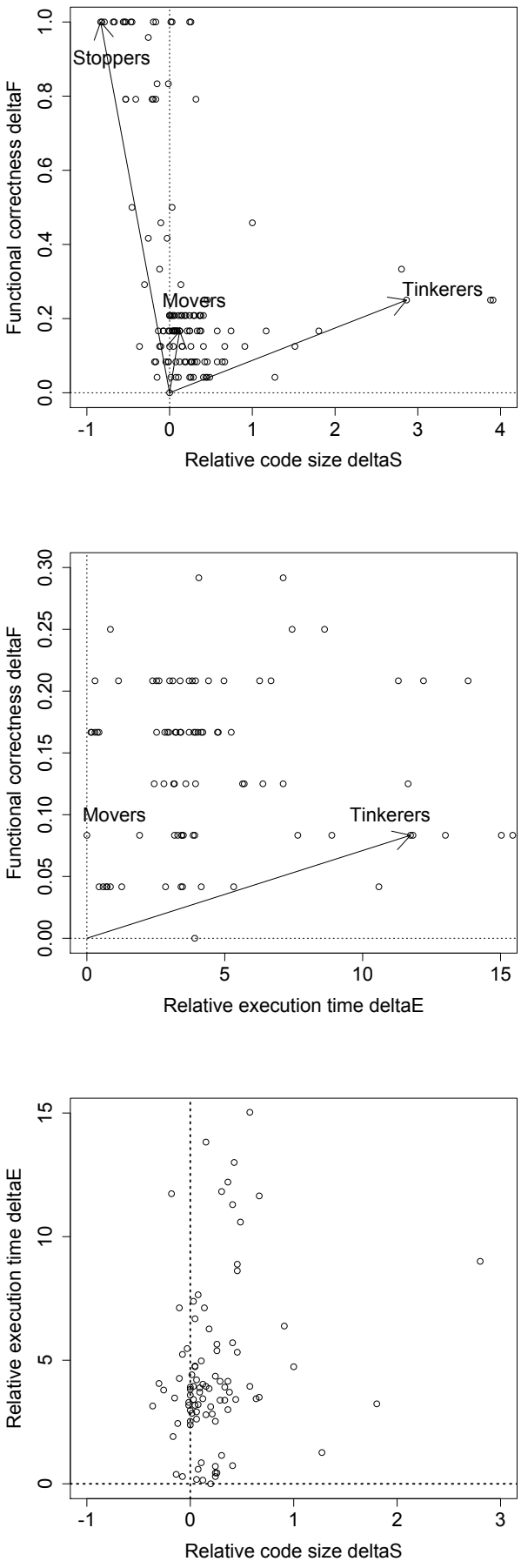

Figure 3: Relative code size vs Functional correctness vs Efficiency distances. Canonical solution shown by dotted lines. Distance from 0,0 is stopper, mover or tinkerer strength. Extreme outliers are not shown.

students should be given the opportunity to refactor their programs and re-submit. In many cases we conjecture that incomplete submissions are the result of a lack of organisation or lack of time to refactor rather than serious difficulties with the material. However, if these problems are not addressed, then this can lead to learning edge momentum problems, that is, inability to deal with new concepts when related ones have not been mastered (Robins 2010).

We now show how to identify a students' learning approach using measurements of the programs they generate and submit. In order to classify students we considered different attributes of their programs: size, functional correctness, efficiency and style. We then set out to characterise learning style in terms of *distance* from a target solution for each attribute. These classifications were validated by hand inspection of students' code noting whether the submission was not completed, completed but by tinkering, or completed to a good standard. Three of the attributes (size, correctness and efficiency) provided meaningful information about students progress. Each was normalised by taking its ratio to the same metric for a canonical solution. The attribute of number of style violations was not used since code inspections suggested that style violations were typically the result of careless mistakes rather than a lack of understanding of programming tasks.

The size distance Δ_S of of a student submission S from a canonical solution C is defined as

$$\Delta_S = ((size(S) - size(C))/size(C))$$

where the unit of measurement for size is non-comment lines of code which has a ratio scale. By construction Δ_S has a median of 0 (best) and ranges from -1 (no code submitted) to a positive maximum $m > 1$ where the size of m depends on the particular exercise. For example, the maximum is 6.167 for the TA2 exercise and 0.798 for TT2.

The target for the number of test cases passed by a student submission S is T the number of test cases in the instructor-provided test class. The proportion of tests passed is not necessarily a ratio scale for any test suite, but it can be made so where certain conditions are satisfied construction of the test suite. The details of such a construction will be discussed in another paper. The functional correctness metric

$$\Delta_F = (T - passes(S))/T$$

defines the distance of a submission from its functional correctness target. Δ_F ranges from 0 (best where every test is passed) to 1 (no tests are passed). Classes for which the tester could not be run are recorded as 0 tests passed giving $\Delta_F = 1$.

The target for efficiency is the number of milliseconds taken to execute a canonical solution class C using an instructor provided JUnit test suite designed for performance evaluation. The class with the lowest execution time amongst the functionally correct programs (hopefully the instructors' class but not always) is chosen as canonical solution class for the execution time target. Efficiency distance is defined by

$$\Delta_E = (time(S) - time(C))/time(C)$$

where the unit for measuring execution time is milliseconds and it has a ratio scale. Since some student submissions do not terminate, we use an upper bound timeout U for the efficiency test. Δ_E is only meaningful for programs that implement the full functionality, and so non-submissions or submissions that do not pass most tests are not considered. Δ_E ranges from the target of 0 for the most efficient class to a maximum threshold of $(U - time(C))/time(C)$.

We can now describe Perkins' learner types in terms of a triple of Δ_F, Δ_E and Δ_S measures. The symbol $\approx$ means close to the target value while $\ll$ and $\gg$ mean far from the target value.

Non-starter No submission or one that can not be compiled or run: $\Delta_S = -1 \vee \Delta_F = 1$

Stopper Incomplete submission that fails most tests and is smaller than required: $\Delta_S \ll 0 \wedge \Delta_F \gg 0$

Tinkerer Writes verbose and inefficient code: $\Delta_S \gg 0 \wedge \Delta_E \gg 0$. Tinkerers, however, often have a good functional correctness score, because they will continue to add to their classes until the most obvious measurable objectives (tests passed) are met.

Mover Submits code that meets all the criteria of functional correctness, efficiency and readability and good design: $\Delta_S \approx 0 \wedge \Delta_F \approx 0 \wedge \Delta_E \approx 0$

Figures 3 show the pair-wise relations between each of the three metrics. Each dot on the graphs represents an individual submission. The target for each metric is indicated by a dotted line. The x,y distance from the target gives an indication of the students' distance from the expected solution. Students to the right of the vertical line in the top and middle plots of Figure 3 are movers or tinkerers and to the left are stoppers. The bottom plot of Figure 3 shows that the defects of large code size and large execution times are largely independent of one another. It can thus be seen that all three measures are necessary for diagnostic assessment since each provides new information for distinguishing between student submissions.

The classification of submissions provided by this metric triple can be used to create a triage ranking of the submissions of a large cohort students. That is, a list of possible tinkerers is generated automatically. The instructor then examines the submissions of each student by hand, and can offer one-to-one help for students with specific difficulties. Stoppers may need more tutorial time or simply the opportunity to resubmit. Tinkerers may need further training on particular programming techniques such as debugging or planning. Movers are already making good use of the tools that are available in the lab. The range of performance across these three metrics suggests that marking schemes used in many institutions (including our own) that are based only on functional correctness can provide misleading feedback to students about their progress.

7 Conclusion and Discussion

We have posed the question: how can software metrics help novice programers? This paper contributes several answers to that question. Limitations in the validity or reliability of metrics proposed in previous studies of students' code generation skills are identified. Problems with norm-referenced metrics such as final score or final grade that have been widely used in previous studies are detailed. We argue that existing grading schemes can be improved by introducing measurable targets based on software metrics. Meaningful targets can be set by studying distributions of these metrics for populations of novice programmers and metrics for sample solutions to lab exercises. Finally, we have shown that using software metrics for assessment should not be seen simply as a boon to busy academics by automating their marking duties, but that software metrics have a valuable role to play in formative and diagnostic assessment.

This study has suggested several ways in which teaching code generation skills to novice programmers could be improved.

1. Instructors should provide clear and measurable criteria on what students are expected to be able to do.
2. Assessment of programming tasks should be based on several different attributes, not just functional correctness.
3. Care should be taken to reduce the cognitive overload in program generation tasks as much as possible.
4. Students should be (strongly) encouraged to refactor and improve their code, rather than submit and forget.
5. Refactoring and quality improvement should be taught with reference to the professional context of software quality and software metrics.
6. Programming courses should allow for students who learn at different speeds, and allow for students with different levels of programming skill.

Our recommendations echo those of other studies including (Lister & Leaney 2003) and (Robins 2010). The contribution of this paper is to provide further evidence that these approaches are necessary and to offer a framework in which the effectiveness of such strategies can be scientifically evaluated based on the theory and practice of software metrics.

Acknowledgements

The author would like to thank the students of CITS1200 at The University of Western Australia and research students and colleagues Lesley Zhang, Adam Khalid, Terry Woodings, Nick Spaddacini and the anonymous referees for their valuable feedback and suggestions.

References

Ala-Mutka, K. (2005), 'A survey of automated assessment approaches for programming assignments', *Journal of Computer Science Education* **15**(2), 83–102.

Crisp, G. (2008), 'Raising the profile of diagnostic, formative and summative e-assessments. providing e-assessment design principles and disciplinary examples for higher education academic staff.', online. Retrieved February 2010.
URL: *http://www.altc.edu.au/resource-raising-profile-eassessments-crisp-adelaide-2008*

Decker, A. (2007), 'How students measure up: An assessment instrument for introductory computer science'. PhD thesis, The State University of New York at Buffalo, USA.

Fenton, N. E. & Pfleeger, S. L. (1998), *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing Co., Boston, MA, USA.

Kitchenham, B., Pfleeger, S. & Fenton, N. (1995), 'Towards a framework for software measurement validation', *IEEE Transactions on Software Engineering* **21**(12), 929 –944.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004), 'A multi-national study of reading and tracing skills in novice programmers', *SIGCSE Bull.* **36**(4), 119–150.

Lister, R. & Leaney, J. (2003), First year programming: Let all the flowers bloom, *in* T. Greening & R. Lister, eds, 'Fifth Australasian Computing Education Conference (ACE2003)', Vol. 20 of *CRPIT*, ACS, Adelaide, Australia, pp. 221–230.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001), 'A multi-national, multi-institutional study of assessment of programming skills of first-year cs students', *SIGCSE Bull.* **33**(4), 125–180.

Morris, D. (2003), Automatic grading of student's programming assignments: an interactive process and suite of programs, *in* '33rd Annual Frontiers in Education', Vol. 3, pp. S3F – 1–6 vol.3.

Perkins, D., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989), Conditions of learning in novice programming, *in* 'Studying the Novice Programmer', Lawrence Erlbaum Associates, New Jersey, pp. 261–279.

Robins, A. (2010), 'Learning edge momentum: a new account of outcomes in CS1', *Computer Science Education* **20**(1), 37–71.

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