Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

Plagiarism in programming: a review and discussion of the factors

J. Traxler
School of Computing and Information Technology, University of Wolverhampton, Wulfruna Street, Wolverhampton, WV1 1SB, UK

Abstract

In the light of recent press coverage of Wolverhampton University's attempts to sensitise students and staff to the dangers of copying and plagiarism in assessed coursework, it is timely to look at one facet of this problem peculiar to software engineering, namely the copying of source code by novice programmers, especially those novice programmers on high-volume modular programmes of study.

This paper attempts to review the issue of plagiarism and copying from a wide variety of angles and in the light of this review draws on informal surveys and ongoing project work at Wolverhampton to highlight possible improvements.

The paper starts by reviewing the most general factors that influence academic misdemeanors. These factors are common to all subjects and cover not only the institutional climate and its regulatory framework, but also the professional (as opposed to academic) context of courses and aspects of course management, particularly the assessment regimes, staff:student numbers, strategies for deploying staff and the extent of permeability and modularity.

Moving on specifically to plagiarism in novice programmers' source code, surveys of our teaching staff and feedback from students inform a discussion of why students copy, how they go about it, how staff currently detect such copying and how they feel computer support would help to fight the problem. Having got this far, it is clear that we need to review how the choice of initial programming language and environment is a factor in the occurrence of copying and plagiarism. Moving onto a more practical treatment, we look at the technical issues involved in implementing computer support for copying detection, specifically detection-in-the-small, that is dealing with mere handfuls of programs, and homes in on some strategies investigated by student projects.

Finally the paper tackles the problem of detection-in-the-large, that is dealing with high volumes of programs from large introductory programming modules, and of validating such detection programs. The purpose of this paper is to stimulate discussion of all these issues and the ways in which they are inter-related.

1 Introduction

Recently the University of Wolverhampton has run a large-scale poster campaign to sensitise academic staff and students across the institution to the problem of plagiarism. The campaign has been based on the assumption of widespread ignorance amongst students of the exact definitions of cheating, collusion and plagiarism; and the assumption that these problems are being experienced across much, if not all, of the British HE sector [1], [2]. The poster used, designed by a art student, was captioned "Have YOU got AWAY with WORDS?" and won the editorial approval of the Times Higher Education Supplement [3]. Amongst other things, the THES said that the poster campaign "represents a bold attempt to face up to a problem which would appear to be on the increase and which is possibly much deeper than many universities would dare to admit". It cited a British Psychological Society survey showing that one in eight undergraduates admitted cheating and plagiarism in their examinations. The editorial went on to infer that this state of affairs was due to the combination of the continued depression of the graduate employment market combined with novel assessment methods.

2 The Background to This Paper

This paper looks at one specific aspect of this real or imagined problem , namely cheating in programming assessments by novice Pascal programmers.

The motivation was an interest in aspects of the computer-assisted delivery of initial programming teaching to large classes, the opportunity to assist in a couple of postgraduate and undergraduate projects and a consciousness of weakness of current practices in the face of increasing student numbers and the fragmentation produced by modularity.

The presentation looks at the possible occurrence of cheating and the strategies to prevent or detect it and draws on several sources: students canvassed on cheating strategies, staff canvassed on detection strategies, a survey of the literature and feedback on the issues involved in attempting to construct a tool for the detection of cheating.

The findings are informal and provisional and the intention is merely to provoke discussion and further investigation. The following remarks merely give an outline of the approaches taken and provisional findings and observations.

3 Institutional Factors

Whether or not cheating takes place, successfully or otherwise, is probably influenced by a number of institutional factors.

As we have seen above, one of these factors could be institutional acceptance that cheating is a possibility that it should face up to. This in turn leads to a consideration of how best deal with it in terms of institutional regulations and disciplinary procedures. On the one hand , the institution could deal with it lightly as a merely academic irregularity and the students in question required to replace or resubmit the offending piece of work or, on the other hand, it could treat it far more harshly, leading to the students in question being withdrawn from their courses. The first option seems almost to condone cheating but has no large administrative overhead whilst the second option demands a considerably greater level of formality and proof, and is something that teaching staff may, for several reasons, be unwilling to invoke. Unfortunately if they chose instead take milder informal measures, they then put themselves in

a position that is unsupported by their institution. Thus the institution must find some middle ground that is practical as well as just.

The straightforward institutional picture is however often confused by the fact that many courses are validated by professional bodies and consequently the courses are required to develop their students' understanding of social, legal and ethical issues. Such courses might have amongst their objectives an expectation that students will understand and observe , for example, BCS or IEE Codes of Conduct. Proven cases of cheating raise problems in relation to this expectation of professional behaviour.

Turning to the specifics of computing, folklore relating to the probability and consequences of being caught influence the extent of cheating. Some lecturers, apocryphally, merely tell classes that detection tools are in use - even though they are not - and this alone suffices to inhibit cheating. Other lecturers might be known for letting culprits off lightly.

Teaching and assessment regimes will also have an impact on student behaviour : under a manual marking regime, the detection of cheating also depends on the way in which a module leader allocates markers to the various assessments and possibly to the various teaching, tutorial and workshop roles. For example, detection and uniformity are optimised if only one lecturer marks the entire assessment from the whole cohort but conversely better teaching and feedback are achieved if each tutor works with their own tutorial group. Workshop staff might have sound intuitions about students' performance but not feed this through to teaching staff.

Technology is now widely being used to deal with the problem of teaching rising student numbers. Currently the nature of this technology is limited but some institutions already keep notes, program code and examples on-line for students rather than print them as hardcopy. In a dispersed modular system of teaching, it becomes quite easy for students to submit to one lecturer as their own work what another lecturer has provided on a network for other students as teaching material.

Personal experience suggests that substantiating an allegation of cheating in programming to the satisfaction of a non-technical panel can be extremely difficult in a technically complex case whilst conversely the panel may have difficulty finding "expert" and impartial advice that is not already involved in the allegations, especially in a more obscure aspect of programming.

4 Feedback from Teaching Staff

In order to get some indication of the current state of affairs, staff teaching introductory programming using Pascal within the School of Computing at the University of Wolverhampton were canvassed by email on the tactics they currently use to detect cheating and those they would like to use were there software support available. The characteristics or criteria put to them as indicating cheating were :

i. control structures especially the pattern of loops and conditions; repeat & while; case & if
ii. data structures especially the pattern of records, arrays and user-defined types; integer & real; boolean & integer flags; menu options as integers or characters;
iii. static analysis covering cyclic complexity measures and also coding anomalies such as global variables, declared but unused variables, incorrect "var" parameters, side-effects, unused assignments, uninitialised variables, looping conditions duplicated as selection conditions inside the loops

iv. program structure as indicated by a module hierarchy or structure chart
v. textual indicators such as program layout, text and strings; dialogue and menus; the style and extent of data validation
vi. dynamic analysis, execution using test harnesses
vii. extrinsic factors such as previous convictions, anomalous performance, tip-off

Feedback from staff was that during each semester-module they each detected a handful of episodes of cheating each involving three or four students. These were usually copying (ie one does, many hand-in) but sometimes were collaboration (ie equal effort) and could take place anywhere in the semester. The factors that seemed to encourage cheating were the assignments being too difficult or too long, unorganised or unmonitored groupwork, a high number of assignments all due at same time, students not coping with the material, two or more struggling students, struggling students being friendly with a less struggling student and allowing late work to be handed in. On the other hand, the factors that inhibited cheating were getting caught and the use of small time-constrained exercises.

As for detection, the survey showed that most lecturers working manually said they relied on control structures, data structures, coding anomalies, design similarities and textual indicators roughly equally to reveal similarities. Dynamic analysis and demonstrations were time-consuming and anyway never revealed good cheats. Several lecturers mentioned that they watched students who had been caught cheating before and finally many said that what revealed cheating (rather than what they looked for) was anything peculiar, unnecessary, obscure, over-elaborate or eye-catching.

The same lecturers were also asked about the most attractive strategy to be used for computer-supported detection of cheating. There was a fairly even spread across the various proposed strategies with several observations that performance would a significant issue if a detection tool were to be used on realistic numbers of students.

A smaller survey was later conducted of lecturers teaching FoxPro and Visual Basic .This corroborated the previous study in relation to the extent of cheating but one lecturer pointed out that informal collaboration and assessed groupwork made the issue very unclear in many cases.

Demonstrations, especially where they reveal that students do not understand their own programs, were mentioned to a greater extent than with the Pascal group as a means of detection but other methods were seen as more problematical because the simple problems set had only a limited set of solutions and thus similarities arose fairly naturally.

Finally, we should bear in mind that there is always little point in looking at a specific indicator of cheating if students have yet to be introduced to it. For example, a module hierarchy chart would be of little use on students ignorant of procedures. Conversely, some of the criteria only show up in defective programs rather correct ones (eg static analysis) and so might show up failures as cheats and successes as honest.

5 Feedback from Students

In view of the fact that asking lecturers how they catch cheats only reveals how lecturers catch failed cheats (namely the ones who got caught), we also attempted to get some picture from students themselves as to how they cheat successfully. Several students were approached by the project students as to how they would or had cheated. The conversations

were necessarily guarded and usually second-hand and most of the techniques were fairly obvious.

Any techniques involving a lot of work, even only clerical work, were dismissed. For example, one project student suggested substituting each procedure call with the corresponding procedure
body but this was dismissed as hard work even though it would probably be undetectable.

Other possible tricks include combinations of simple search-and-replace operations on variable, type, procedure and functions; resetting tab-stops and so on.

In order to catch cheats, it is worthwhile to consider not just how they cheat but why. At least two alternatives exist : either two or more students of comparable ability collaborate (because of laziness, pressure of work or a misunderstanding about the nature of groupwork) or one student asks another for assistance (because of the inability of the first student to do the assessment unaided, due to innate inability or pressure of work). Sometimes, one student will copy from another without asking - this has happened in supervised practical exams by swapping and copying disks and with students searching hard-drives for undeleted coursework.

The methods used to disguise the fact that copying has taken place would quite possibly differ in these two scenarios.

The sophistication of the cheating tactics would also be influenced by the sophistication (and volume and weighting) of the programming task being assessed. Certainly, there is little to be gained by investing large amounts of time and expertise to cheat in an small, unweighted and undemanding assessment.

The discussion so far has assumed we are dealing with large initial programming classes. It seems likely that as students' programming matures and they learn more languages, the analysis of cheating behaviour becomes considerably more difficult than the initial stages discussed here. Even in these initial stages there may be problems. If part-time students, using say C at work, start using Pascal on their course, they may produce stylistically similar programs because of their similar background. Conversely, looking for coding anomalies in C as a second language in students whose first language was Pascal may only reveal transliterations of Pascal showing through the C.

6 Choice of Programming Language

Although this paper deals principally with Pascal, it is obvious that the ease and attraction of cheating and the strategies for detecting it are to some extent a function of the choice of introductory programming language, the programming environment and the hardware platform. An initial investigation by Hathway[8] looked briefly at the techniques previously used in other languages and showed considerable scope for identifying coding anomalies in C using tools such as lint. Many of these anomalies, such as referencing an uninitialised variable, cannot actually be used in newer or arguably better languages such as Pascal, Modula-2 and Ada.

Clearly the nature and traceability of copying has changed as HE institutions moved away from the use of a central mini running VMS or Unix with student accounts to networked PCs (supplemented by students' own PCs) with floppy-disks as the storage medium.

7 Detection-in-the-Small

There is a considerable literature that covers the issues of plagiarism and "similarity" in large classes of novice Pascal programmers. Some of this literature has been summarised by Leech [4] and draws on a wide range of current sources. They are all based on examining pairs of programs for specific characteristics and using these characteristics as measures of similarity. Many of the measures are complex and sophisticated and may be inappropriate for novice programmers and/or small programs. Some of the work looks at surface and typographical features and may or may not focus on the specifics of Pascal whilst other authors look for what one could call language-independent deep structure, related to control flow, and may have a universality but may waste effort both in preparation and in analysis.

Leech [4] shows how the tactics suggested have evolved from systems based on a crude use of Halstead metrics, for example the system suggested by Parker and Hamblen [5] (and subsequently criticised by Salt [6]). The metrics in question were the measured length of a program and its so-called vocabulary. The problem with comparing short programs is that of coincidental correctness. This method was neverheless elaborated to include control structures and then control flow by later workers described by Parker and Hamblen [5]. Berghel and Sallach [7] increased the number of factors to include different types of variables and different types of program line (ie blanks, comments and continuations) in an attempt to catch padding and restructuring of program text. Oman and Cook [8] saw previous attempts as to some extent dealing with surface features and attempted to improve them by defining sixteen measures of their own that show considerable typographical sophistication, presumably own the assumption that cheats only notice and change crude typographical features in their copying. Their measures (eg blocked or bordered comments, number of multiple statements per line, BEGIN or THEN followed by statement on the same line) were specific to Pascal-like languages but only to their typography and did not include any measure of variable, types of keyword or modularisation.

The most complex system was proposed at roughly the same time by Faidhi and Robinson [9] who used Pascal as their working language and were thus able to define language-specific characteristics. Their twenty-four measures include typographical ones (eg number of blank lines), control flow ones, structural ones (eg number of modules) and Pascal-specific ones (the proportions of simple expressions, terms and factors as defined by empirical formulae). They were intended to identify similarity in both novice and expert programmers' code.

The work of Whale [10], [11] was an attempt to synthesise several methods into a viable system culminating in a system called PLAGUE. This first generates a structure profile containing details of selection and iteration and then subjects nearest matches to futher comparison using operators. Structure is used as a criterion to catch cases where one keyword has been substituted for an equivalent one (eg IF and CASE)

Wise [12] reacted to perceived problems with PLAGUE to produce YAP but the problems in question were mainly the effort required to implement versions for new languages and dialects. YAP implements a preprocessor. The other problem with PLAGUE is reported to be the difficulty of interpreting its raw numerical output.

In their work Faidhi and Robinson [9] also provide a tentative hierarchy of cheating, running from level 1 to a maximum level 6. These are cumulative and start with changes in comments and indentation, followed by changes in identifiers, extra redundant code, alterations to function and procedural code, alterations control loop and finally changes in control and decision logic. Whilst it is clear that a scale indicating the severity of cheating would be helpful in calibrating the range or effectiveness of detection tools, it is not at all clear that this is a

convincing one. At the cost of increased complexity, it might be necessary to use different scales; one for typographical changes, one for variable and types, one for logic and control structures and so on.

To look at a contrasting and specific option , Hathway [13] looked at source-code for stylistic anomalies. This meant any features of program syntax that although they compiled, were in some sense or other self-evidently silly, unsafe, pointless or redundant. A much longer list is given later but unreachable code, uncalled procedures or unreferenced variables would be typical examples.

The exact extent of this list depends of course on the language and implementation involved. In software engineering terms, a better and newer language such as Ada or Modula-2 would offer less scope for such anomalies whereas Fortran and C being older are less safe in this respect.

A preliminary attempt at a list includes :

unreachable code
uncalled procedures and functions
functions being used as procedures
extra semicolons, especially after last statement
BEGIN/END around single statement
BEGIN/END inside REPEAT/UNTIL
loop counter changed inside FOR loop
assignments made inside loops independently of loop-related variables
globals
declaration of anonymous variables, especially arrays
references to uninitialised variables
declarations of unused variables, types, constants
successive assignments to a variable without an intervening reference
variable parameters in functions
unused parameters in procedures/functions
variables parameters not assigned or changed in procedures

Hathway chose only a limited subset of this list and at least demonstrated that it was a technically viable approach but it also has several flaws, the most obvious being that the pairs of suspect programs may have no anomalies and thus be given a clean bill of health. Nevertheless the approach represents one attempt to identify a programmer's characteristic "signature".

Incidentally, this topic has attracted continued interest as a student project. It is quite a worthwhile topic because it is amenable to a variety of approaches (in a variety of languages) with spin-off in programming, statistics and experimental design. A crucial issue is the testing and validation of any cheating detector. Ethics and the availability of experiential subjects constrain this testing. There are several possible strategies but basically one wants neither a system that never detects cheats nor a system that detects cheats where none exist.

Starting from one end, the system could be presented with pairs of programs where one was a literal copy of the other. Once this was detected, the copy could be gradually modified in a variety of controlled ways to explore the sensitive of the system to different styles of plagiarism, a sort of mutation testing. Starting from the other end, the system could be presented with pairs of different programs to ensure it never gave "false positives". Either

of these strategies could then be used with a variety of programming constructs that might be taught and various cheating techniques.

8 Detection-in-the-Large

With larger populations, it is necessary to explore performance issues since many of the methods proposed are complex and typical introductory programming classes may number a hundred students. This means any method involving one-to-one comparison will be extremely time-consuming both administratively and in terms of processing.

Any large-scale system would probably need a mix of methods, some early ones acting as filters and later ones working more intensively on candidate pairs of programs. The system would need to be tunable in the sense of adjustable for different programming assignments and would probably aim to produce a handful of indicative measures for each pair of candidate programs. A sensible strategy would be to combine this approach with subsequent human inspection to identify the quirky, eye-catching and tell-tale random features that would confirm or refute suspicions.

9 Conclusion

This is clearly an interesting if minor field with many technical challenges, possibilities and problems. It is an absorbing topic for student projects and may also have some real relevance if the trends identified in the THES continue. There has been considerable research looking at different technical strategies but none of these seem to be completely convincing in terms of accuracy and efficiency and any technical strategies should only be considered in a wider institutional and pedagogic context.

References

1. Targett, S. (1995) Poster Campaign Warns Against Plagiarism. Times Higher Education Supplement, No 1159 p1
2. (anon). University Combats Plagiarism, Network, Issue 19, p6, University of Wolverhampton, 1995
3. (anon). Plagiarism or Pastiche. Times Higher Education Supplement, No 1159 p11
4. Leech, M. (1995) A Critical Evaluation of Existing Literature Concerning Plagiarism Detection Systems with Reference to FoxPro, unpublished MSc dissertation, University of Wolverhampton
5. Parker, P. and Hamblen, J. O. (1989) Computer Algorithms for Plagiarism Detection. IEEE Transactions on Education, Vol 32 No 2, pp94 - 99
6. Salt, N. F. (1982) Defining Software Science Counting Strategies. SIGPLAN Notices, 17, p58 - 67
7. Berghel, H. L., Sallachi, D. L., (1984) Measurements of Program Similarity in Identical Task Environments, SIGPLAN 19 (8) p65 - 76
8. Oman, P. W., Cook, C. R., (1989) Programming Style Authorship Analysis. ACM pp320 - 326
9. Faidhi, J. W., Robinson, S. K., (1987), An Empirical Approach for Detecting Program Similarity and Plagiarism Within a University Environment, Computers in Education ,11 p11 - 19
10. Whale, G. (1990) Software Metrics and Plagiarism Detection. Journal of Systems Software, Vol 13, pp131 - 138
11. Whale, G. (1988) Identification of Program Similarity in Large Populations. The Computer Journal, Vol 33 No 2, pp 140 -146
12. Wise, M. J. (1992) Detecting Similarity in Student Programs : YAP'ing may be preferable to Plague'ing. ACM pp 268 - 271
13. Hathway, D. F. J. (1992), An Investigation into Pascal Programming Style and the Implementation of a Pascal Static Analyser, unpublished BSc project report, University of Wolverhampton