Software Reuse and Plagiarism: A Code of Practice

J Paul Gibson
Telecom & Management SudParis
9 rue Charles Fourier, Evry, France
paul.gibson@it-sudparis.eu

ABSTRACT

In general, university guidelines or policies on plagiarism are not sufficiently detailed to cope with the technical complexity of software. Software plagiarism can have a significant impact on a student's degree result, particularly in courses were there is a significant emphasis on large-scale projects. We argue that a policy for software reuse is the most explicit, and fair, way of overcoming this problem. In our policy, we specify the notion of software to cover all the documents that are generally built during the engineering of a software system — analysis, requirements, validation, design, verification, implementation and tests. Examples are used to show acceptable and unacceptable forms of reuse, mostly at the design, testing and implementation stages. These examples are represented in Java, although they should be easily understood by anyone with software engineering experience. We conclude with a simple code of practice for reuse of software based on a *file-level* policy, combined with emphasis on re-using only what is rigorously verified.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; K.7.4 [**The Computing Profession**]: Professional Ethics—*Codes of good practice*

General Terms

Documentation, Legal Aspects

Keywords

Plagiarism, ethics, software re-use, testing, student projects

1. SOFTWARE REUSE AND PLAGIARISM

This paper is based on a code of practice that was first presented in an internal technical report [7] that was produced in response to a perceived problem of plagiarism in final year projects of computer science and software engineering students. These projects are the assessed work in which undetected (and unpunished) plagiarism can have the most impact on a student's degree result. Consequently, we proposed a policy for software reuse as the most explicit (and fair) way of overcoming the problems that arise in preventing, detecting, assessing and punishing project work that contains plagiarised material. In the 6 years that have followed the creation of this code of practice, the author of the report has moved to a new higher-level institute where he has lectured students on the issue of plagiarism (as part of teaching about ethics in software development) and this article reports on the impact that this has had on student's re-use of secondary material in their own assessed project work.

1.1 Plagiarism

At all stages of education and learning, students' reuse of other peoples' work and ideas is fundamental and it is to be encouraged. What is *not* encouraged is when the work of another person is presented as a student's own. This is plagiarism and must *not* be tolerated.

It is each student's responsibility to ensure that when they include (directly or indirectly) the work of others that this contribution is fully and properly acknowledged. Guidelines on the acknowledgment of the work of others can be found in a text by Gordon Harvey [9]. Professional bodies (with publishing houses) also provide generic guidelines on plagiarism — the ACM student magazine *Crossroads* is a good example for students [10]. Universities generally have their own policies (or guidelines) on plagiarism. Individual departments may also provide more specific guidelines and one must be careful that these documents are consistent.

In our experience, however welcome these documents are, they do not cover many of the more difficult, technical issues which arise when the work that is being re-used is software. It is the role of a *code of practice* to try and clarify what is meant by plagiarism in this context. This article is concerned with such a code of practice; it is not about defining different types of source code plagiarism [4] — although research into formalising plagiarism types in software engineering is complementary to our work. Furthermore, we do not present any new results on plagiarism detection [14, 2, 11] — although such research is necessary for students to be convinced that the risk of detection is high. We also do not provide any new ideas concerning policy for punishing plagiarism, but we do agree that a holistic approach is required [12]. Finally, this article is not a rigorous analysis of whether the problem of plagiarism is increasing or decreasing in students' software projects — although our personal experience does reflect the more formal analysis that has been carried out in demonstrating that the problem is growing [1].

1.2 Software Engineering and Re-use

In software engineering, software is usually not built from scratch. Normally, already existing software artifacts (from the set of documents and models that are built during the engineering of a software system - analysis, requirements, validation, design, verification, implementation, tests, maintenance, versioning and tools) are re-used, in a wide range of ways, in the construction of a "new" software system.

Software re-use is one of the least-well understood elements of the software engineering process. It is much more challenging to re-use software artifacts in a controlled, systematic way – the quality of software is compromised if a rigorous engineering approach to re-use is not followed. Clearly, any piece of assessed work incorporating the development of software (including final year projects) can be considered to be of "poor" quality if it has relied upon non-rigorous, ad-hoc re-use of other peoples' software.

1.3 Requirements for a Code of Practice

We propose that a code of practice for software re-use offers many advantages to students submitting work for evaluation: makes explicit what constitutes plagiarism with respect to software; provides guidelines which, if followed, should ensure that a student is not wrongly accused of plagiarism; defines structures to help examiners to objectively check for plagiarism in a consistent and fair manner; and improves the quality of the software that students produce.

We acknowledge that any code of practice will obviously restrict the way in which software can be developed. Furthermore, such a code will almost certainly be too restrictive in the sense that there are sure to be specific requirements for some system that would be impossible to meet in the time-frame of a project if the code of practice is enforced, but otherwise could possibly be met. For this reason we propose that exceptional cases be dealt with by the lecturer or project supervisor. The fundamental requirements for the code of practice are that it is: simple to understand, apply and enforce; consistent with wider plagairism policy; and as fair as possible to all students.

In order to guide the formulation of the code of practice, we propose that concrete examples of acceptable and unacceptable forms of re-use be examined. These examples are not intended to be complete. The examples were chosen because they represent the most common forms of re-use that we have witnessed in final year projects (both acceptable and unacceptable).

2. UNACCEPTABLE SOFTWARE RE-USE — WHY WE NEED A POLICY

In this section we explicitly identify — through a single, simple piece of Java — a number of unacceptable ways of re-using other peoples' software. Further, we recommend that you follow our guidelines for acceptable forms of re-use, even when re-using your own code. This is known as self-plagiarism [3] and may also be considered bad practice by lecturers setting software development assignements.

2.1 Original Software — a Realistic Example

In this subsection we introduce a software artifact/model, in the form of Java source code, which forms the basis of discussion about unacceptable forms of re-use. Note that, like much of the code that is re-used by students, this artifact was not designed for re-use!

Let us now examine what this code does and how it may be *reusable*. Firstly, we note that the code consists of 2 classes: `Example1` and `IntArray`. The `Example1` class appears to be a simple test driver for the `IntArray` class. By running the code, and through examination of the sample execution which is provided as a comment at the end of the file listing, we see that the code *appears*[1] to generate an array of 12 integers whose values are initialised randomly to be in the range 1 to 8. It then prints out this array, sorts it and prints out the array again. It also counts the number of comparisons and swaps that were performed in the process of sorting the array.

```
class IntArray {

int size, max;
int numswaps, numcomparisons;
int [] values;

IntArray (int size, int maxvalue){
reset(size, maxvalue);}
// end IntArray constructor

public void reset(int sizeIn, int maxIn){
size = sizeIn; max = maxIn;
values = new int [size]; randomize();
}// end reset

public void randomize(){
for (int i =0; i<size; i++)
 values[i] =(int)(Math.random()*max)+1;
numswaps=0; numcomparisons=0;
}// end randomize

public boolean compare(int i, int j){
if (i<size && i>=0 && j<size && j>=0){
   numcomparisons++;
   return values[i]> values[j];
}else return false;
}// end compare

public void swap(int i, int j){
if (i<size && i>=0 && j<size && j>=0){
   numswaps++;
   int temp = values[i];
   values[i] = values[j];
   values[j] = temp;}
}// end swap

public void sort(){
        for (int i =0; i<size; i++)
           for (int j = i+1; j<size; j++)
               if (compare(i,j)) swap(i,j);
}// end sort

public String toString(){
String str =" size: "+size+
             " max: "+max+" values: ";
for (int i=0; i<size;i++) str = str+" "+values[i];
str = str+"\n number of swaps = "+numswaps+
            " number of comparisons = "+numcomparisons;
return str;
}// end toString
}// endclass IntArray

class Example1 {
```

[1]We write *appears* because we cannot be sure that this is its behaviour without performing some adequate tests

```
public static void main(String[] args){
IntArray test = new IntArray(12,8);
System.out.println("Randomly generated integer array");
System.out.println(test);
test.sort();
System.out.println("Array after it is sorted");
System.out.println(test);
}// end main
}// endclass Example1
```

Using this example, we illustrate unacceptable types of software re-use that are typically seen in the projects submitted by students. All of these involve use of *cut-and-paste* functionality provided by most text editors and operating systems.

2.2 Cut-and-Paste Plagiarism

Imagine that a student's project requires them to provide code to sort integers in descending order; and imagine also that this is a non-trivial task for a computer science or software engineering student. Such a student may "find" the `IntArray` code, above, through searching the web, for example. It appears to sort integers, but it sorts them into ascending order. The student could clearly make a minor change to achieve the required behaviour. We propose the following code (fragment) as that which would be typically produced and submitted by such a student:

```
class MyArray {
int size;int max;int [] values;
MyArray (int size, int maxvalue){
reset(size, maxvalue);}

public void reset(int sizeIn, int maxIn){
size = sizeIn;
max = maxIn; values = new int [size];randomize();}

public void randomize(){
for (int i =0; i<size; i++)
 values[i] =(int)(Math.random()*max)+1;}

public boolean compare(int i, int j){
if (i<size && i>=0 && j<size && j>=0)
{return values[i]< values[j];}else return false;}

public void swap(int i, int j){
if (i<size && i>=0 && j<size && j>=0)
{int temp = values[i];
values[i] = values[j];values[j] = temp;}
}

public void sort(){
for (int i =0; i<size; i++)
for (int j = i+1; j<size; j++)
 if (compare(i,j)) swap(i,j);}

public static void main(String[] args){
MyArray test = new MyArray(10,20);
System.out.println("Randomly generated array");
System.out.println("Values: ");
for (int i=0; i<test.size;i++)
 System.out.print( " "+test.values[i]);
test.sort();
System.out.println("\nArray sorted (descending order)");
System.out.println("Values: ");
for (int i=0; i<test.size;i++)
 System.out.print( " "+test.values[i]);
}}
```

Note that, to anyone familiar with programming, this submitted code is plagiarised The student has made a number of minor changes: (1) The layout and indentation has changed. (2) Comments have been changed. (3) Identifiers have been changed; although not all of them. (4) Code has been removed (in this case, the student saw no need for counting `swaps` and `comparisons`, or for a `toString` method). (5) In the compare method, a '$>$' is changed to a '$<$' (to sort in descending rather than ascending order). (6) Design structure has been altered — rather than having a separate testing class, the test is included as a main method of the 'new' `MyArray` class.

Typically, plagiarised code will be altered in at least one of the ways illustrated in the example above. Even though the "new code" is different from the original, plagiarised code, the work done in making the changes could be considered to be insignificant. A poor student (or a student who deliberately wishes to mislead an examiner into believing that they wrote `MyArray` from scratch) would "forget" to acknowledge the original author of `Example1`; and would probably make as many insignificant changes as possible. A good student may indeed acknowledge the original `Example1` software artifact that was re-used. However, the degree of acknowledgement could vary from a vague and imprecise: *This code was inspired by the work of A. Programmer*; to a complete acknowledgement, including a listing and reference to the original code, together with a *difference file* showing exactly the changes that were made. Such a potentially wide variety of acknowledgements makes it impossible to fairly credit such software submitted in this way.

The problem we face is that this *cut-and-paste* type of software re-use is the most common form of re-use that is found in submitted work!

3. DIFFERENT FORMS OF PLAGIARISM

All plagiarism involves claiming other peoples' work as your own (or assisting someone to make such false claims). In software engineering, work that is re-used without proper acknowledgement can be hard to identify. To clarify this, we illustrate other forms of software plagiarism, where the re-use is less obvious than that shown in the previous Java example, but which is nevertheless considered to be unacceptable.

3.1 Collusion

In all practical projects, it is considered normal practice to be given help. This help must be publically acknowledged when the work is presented for evaluation or publication. When the help is significant then it is normal for the person who has given the help to be credited in a more formal way. Where help has been given, and there is *collusion* between the parties involved, it is a simple matter for no public acknowledgement of this to be made. In such a case, there is no direct re-use of software in the classical engineering sense. However, this *collusion* is plagiarism.

Software — of any reasonable complexity — is structured and has different components. Collusion in software development involves a third party writing the code for at least one of these components, and a student submitting this code as their own.

3.2 Unacknowledged Reverse Engineering

Often software engineers will look at some code and be able to reverse engineer [5] some abstract property of that code in order to re-use that abstraction to help them write their own code, usually as a solution to a different, yet similar, problem. When the original piece of code is not acknowledged then this is also commonly known as "stealing someone else's idea(s)". In final year projects, this type of plagiarism often results when a student re-uses the design of a software system (or part of a software system) as a structure, template or pattern for their own code.

Students should not be discouraged from engineering software in this way (it is a reasonably advanced technique) but they should be strongly encouraged to correctly acknowledge where the original design (idea) originated. Software design is a challenging part of the software engineering life cycle; and good design [8] should be recognised in the assessment of any project. Re-using other engineers' designs without proper acknowledgement is as bad as re-using their code in the same way.

3.3 Unacknowledged Translation

Imagine that a student is required to write code that provides exactly the same behaviour as seen in `Example1`, but is required to code it in C++. The student "finds" the Java code and re-uses it to generate[2] C++ code. This can be thought of as a specific form of re-use through abstraction.

Again, this may be considered a good approach in some circumstances, provided the original code is properly acknowledged. A student who choses not to acknowledge the original code will be considered to have attempted to deliberately deceive the examiners of their work, and this will result in them being brought to the disciplinary committee.

3.4 Unacknowledged Code Generation

Software engineering tools, often found as part of complex development environments, can be used to automatically generate code. Any such generated code must be explicitly identified and correctly acknowledged. Note that these tools usually credit themselves, so removing these credits would be considered as deliberate deception on the part of the student, and disciplinary action would follow.

A common form of plagiarism is to use a tool to reverse engineer design documentation from implementation code (from Java to UML, for example). The code generation can also go in the other direction (from abstract to concrete). For example, there are tools to generate C++ code from data flow diagrams. This type of automated software engineering is good, provided the role of the tool is properly acknowledged.

3.5 No Re-use Without Test

From the examples above, it seems that care needs to be taken about acknowledging any re-use of code. There is a simple guideline to ensure that a student never forgets the acknowledgement, avoiding the risk of being accused of deliberate deception when the plagiarism is a result of incompetency: *Explicitly acknowledge the use of someone else's code — no matter how small — by testing it against your requirements.* In the case that a student does not properly test the software that they are re-using, this student should be advised that the re-use is unacceptable. The following guideline[3] is suggested: (1) If you don't know how to test it then don't re-use it. (2) If you don't know what to test it against then don't re-use it. (3) If you know what to test and how to test it, then re-use it only after the tests are successfully completed.

Supervisors should advise students that it is the students who are responsible for the behaviour of the artifacts that they re-use: if their system fails due to a defect in another person's software then this is the student's responsibility.

[2]This generation may, or may not, be assisted by tools — see the next subsection.

[3]We use 'test' to mean some sort of validation or verification, and 'it' to represent any software artifact.

4. ACCEPTABLE SOFTWARE RE-USE

In this section we identify useful strategies for re-use that would leave an examiner in no doubt about what has been re-used and what has been the original contribution of the student. It is important that concrete examples of each acceptable form of re-use are presented to the students [7]: *Composition and Aggregation, Inheritance, Templates and Genericity, Design Patterns and Architecture, Interface Design and Specification of requirements (including tests).* We note that the examples of acceptable code re-use must also illustrate the best techniques and tools for testing the code (models) that are being re-used.

5. FILE-LEVEL RE-USE

Unacceptable forms of software re-use are most easily identified by files that contain *cut-and-paste* code: i.e, code that has been produced by more than a single author. We cannot preclude the re-use of multi-authored software; however, we can preclude the submission of the student's own work in a multi-authored file. All the examples of acceptable forms of re-use are directly supported — at the file level — by the vast majority of modelling languages that are used in software engineering. In other words, they do not require one to write *cut-and-paste* code. We suggest the following code of practice:

(i): All software that is re-used will <u>not</u> be found in the same file as software that is submitted by the student for evaluation, unless authorised by the supervisor and justified in the documentation.

(ii): All re-used software will be properly acknowledged in the documentation, and the student must clearly[4] distinguish between the software that they have re-used and the software that they have written themselves; and they must note in their own software where the re-use occurs.

(iii): All re-used software must be *adequately* tested.

(iv): All students who are found to have plagiarised software — intentionally, or not — will be punished following a standard set of public guidelines

This code of practice meets all our original requirements.

6. EVALUATING THE CODE OF PRACTICE

The author has much experience of using puzzles and games for teaching about software engineering [13, 6]. One problem which has proven successsful is that of the matches game: *The game begins with a random number of matches. There are 2 players who play alternately: each time removing 1,2 or 3 matches. The winner is the player who leaves their opponent with the final match.*

6.1 Plagiarism continues

When these students have not been explicitly warned about plagiarism through presentation of the code of practice, we

[4]Through intelligent use of comments, fonts, colors, etc...

find that, on average, half the students plagiarise material from the web. For example, consider the code which is publically available on the web and which can be easily found using a search engine:

```
private boolean losing(){
// is the number of matches left a losing position?
return ( (numMatches-1)%(currMaxR+1) == 0 );}

private int pickwell(){
// how many matches to remove to leave a losing position
return ( (numMatches-1)%(currMaxR+1) );}

public int pick(){
// the number of matches to remove (depending on level)
if ( (numMatches-2)<currMaxR ){
   appendHistory("You left me with an easy win...");
   return numMatches-1;};
if ( losing() ){// in losing position so pick randomly
   appendHistory("I have a difficult choice...");
   return 1+(int)(Math.random()*currMaxR);};
int temp = (int)(Math.random()*4);
boolean pickgood = (currLvl > temp );
if (pickgood){// leave other player a losing position
   appendHistory("I feel I can make a good choice...");
   return pickwell();
   };
appendHistory("I am not sure what to do...");
// make a random pick
return 1+(int)(Math.random()*currMaxR);
}// end method pick in Matches
```

Students that are asked to program a perfect player for the 3 matches game have often plagiarised the original code to arrive at code such as:

```
private boolean losing(){
// is the number of matches left a losing position?
return ( (numMatches-1)%(3+1) == 0 );}

private int pickwell(){
// how many matches to remove to leave losing position
return ( (numMatches-1)%(3+1) );}

public int pick(){// the number of matches to remove
if ( (numMatches-2)<3 ){return numMatches-1;};
if ( losing() ){// in losing position so pick randomly
              return 1+(int)(Math.random()*currMaxR);
return pickwell();
}}// end method pick in Matches
```

Typically, students that find the original code then replace all instances of `currMaxR` with the constant 3, remove the code for different levels and edit out the code for writing messages to the screen.

6.2 Re-use when code of practice is presented

After students are presented with the code of practice, a significant number continue to submit plagiarised material. Although, the percentage falls by half compared to the case when the code of practice is not presented. An interesting observation is that the strong students tend to re-use code that it is found on the web but they successfully manage to do so by following our policy. Furthermore, many of the students that re-used code in this way identified, through testing, errors in source code that they found in the web. This was a valuable lesson for them to have learned.

7. CONCLUSIONS AND FUTURE WORK

Evidence for the success of adopting a code of practice is currently anecdotal. Other lecturers have reported that students, who have been presented with the code of practice, are much more rigorous about re-using code in a range or modules and projects. In future research, we hope to validate our intuition through a more rigorous, scientific experiment.

8. REFERENCES

[1] R. F. Boisvert and M. J. Irwin. Plagiarism on the rise. *Commun. ACM*, 49(6):23–24, 2006.

[2] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *Information Theory, IEEE Transactions on*, 50(7):1545–1551, July 2004.

[3] C. Collberg and S. Kobourov. Self-plagiarism in computer science. *Commun. ACM*, 48(4):88–94, 2005.

[4] G. Cosma and M. Joy. Towards a definition of source-code plagiarism. *Education, IEEE Transactions on*, 51(2):195–200, May 2008.

[5] G. C. Gannod and B. H. Cheng. A framework for classifying and comparing software reverse engineering and design recovery techniques. In *Proceedings of the 6th Working Conference on Reverse Engineering.* ACM International Conference Proceeding Series archive, Oct. 1999.

[6] J. P. Gibson. A noughts and crosses Java applet to teach programming to primary school children. In J. F. Power and J. Waldron, editors, *Proceedings of the 2nd International Symposium on Principles and Practice of Programming in Java*, volume 42 of *ACM International Conference Proceeding Series*, pages 85–88. ACM, 2003.

[7] J. P. Gibson. Software reuse in final year projects: A code of practice. Report NUIM-CS-TR-2003-12, Department of Computer Science, NUI Maynooth, 2003.

[8] J. P. Gibson, E. Lallet, and J.-L. Raffy. How do I know if my design is correct? In *Formal Methods in Computer Science Education (FORMED 2008)*, pages 61–70, 2008. To appear in ENTCS.

[9] G. Harvey. *Writing with sources: A guide for students.* Hackett Publishing Company, 1998.

[10] C. Jordan. At a crossroads: plagiarism. *Crossroads*, 13(1):2–2, 2006.

[11] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, New York, NY, USA, 2006. ACM.

[12] R. Macdonald and J. Carroll. Plagiarism — a complex issue requiring a holistic institutional approach. *Assessment & Evaluation in Higher Education*, 31(2):233–245, 2006.

[13] J. O'Kelly and J. P. Gibson. Software engineering as a model of understanding for learning and problem solving. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 87–97, New York, NY, USA, 2005. ACM.

[14] A. Parker and J. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, May 1989.