Software Reuse In Final Year Projects: A Code of Practice

J. Paul Gibson[1], Department of Computer Science,
National University of Ireland, Maynooth.
Date: November 2003 (updated September 2004)

Technical Report: *NUIM-CS-2003-TR-12*

Key words: plagiarism, code reuse, software reuse, code of practice

Abstract

In general, university guidelines or policies on plagiarism are not sufficiently formalised to cope with the complexity of software oriented assessment. Final year projects, in computer science, are the area in which undetected (and unpunished) plagiarism can have the most impact on a student's degree result. We argue that a policy for software reuse is the most explicit (and fair) way of overcoming this problem; and we specify such a policy that can be equally applied to all assessed practical work which involves the development of any software artefacts, not just *code* as written in some programming language. In the report, we use 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, tests, maintenance and versioning. Concrete examples are used to show acceptable and unacceptable forms of reuse (mostly at the design and implementation stages). These examples are represented in Java, although they should be easily understood by anyone with software engineering experience. The lessons learned from the examples can be applied to other languages and artefacts, at all stages of the software development cycle. We conclude with a simple code of practice for reuse of software based on a *file-level* policy.

Contents

[1]email: pgibson@cs.may.ie

1 Software Reuse and Plagiarism

1.1 Plagiarism

At all stages of education and learning, including University, reuse of other peoples' work and ideas is fundamental. It is actively encouraged, it is not forbidden and it does not constitute plagiarism. What is not encouraged is when the work of another person is presented as your own (willfully or not). This is plagiarism and will not be tolerated.

It is your responsibility - whether in your role as a student, or as a scientist or as an engineer - to ensure when you 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[5]. Professional bodies (with publishing houses) also provide guidelines on plagiarism (the ACM student magazine *Crossroads* is a good example[2]. Universities generally have their own policies (or guidelines) on plagiarism[3] and students in all departments must be made aware of them. Individual departments may also provide more specific guidelines[4], 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 reused is software. It is the role of this document to try and clarify what is meant by plagiarism in this context.

1.2 Software Engineering and Reuse

In software engineering, software is usually not built from scratch[7]. Normally, already existing software artefacts (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 reused, in a wide range of ways, in the construction of a 'new' software system.

Software reuse is one of the least-well understood elements of the software engineering process[3]. A discipline of reuse has yet to be rigorously accepted or applied and, as a consequence, software reuse is often done in an ad-hoc fashion. It is much more challenging to reuse software artefacts in a controlled, systematic way – the quality of software is compromised if a rigorous engineering approach to reuse 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 reuse of other peoples' software.

1.3 Requirements for a code of practice

We propose that a code of practice for software reuse 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

[2]See www.acm.org/crossroads/doc/information/wg/plagiarism.html

[3]NUI Maynooth make an explicit statement in the University Calendar — in 2004 this can be found on pages 77 and 78 — providing a *Guide to Students* complemented by a statement of the *Disciplinary Consequences*. We make reference to this document, where appropriate, later in this report.

[4]The Computer Science Department at NUIM provides a final year project handbook[6] that covers plagiarism.

- 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[5] 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 project supervisor (see section 5 for details).

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. We will return to these requirements in the final section of this report.

In order to guide the formulation of the code of practice, we propose that concrete examples of acceptable and unacceptable forms of reuse be examined. These examples are not intended to be complete. The examples were chosen because they represent the most common forms of reuse that we have witnessed in final year projects (both acceptable and unacceptable). As such reuse has mostly been at the design and implementation stages of development, we focus our examples on these levels of abstraction. However, the lessons to be learned are applicable to the reuse of all software artefacts.

We chose Java as the modelling language used to build the software artefacts (models) in our examples. This choice reflects our students' software modelling experience, where the majority are most comfortable working with object oriented programming languages, in general, and Java in particular.

2 Unacceptable Software Reuse — why we need a policy

In this section we explicitly identify — through a single, simple piece of Java — a number of unacceptable ways of reusing other peoples' software. The types of reuse in the example also illustrate bad engineering practice when reusing your own[6] software. Such reuse is common in software engineering because the construction of most models is incremental, where some, or all, of the model/code in a previous increment is reused in a following increment. Although the reuse of your own code in the way illustrated by the example in this section is not explicitly forbidden, we do recommend that you follow our guidelines for acceptable forms of reuse, even when reusing your own code!

2.1 The Original Software — a realistic example

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

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

[5]The impossibility may also be due to the nature of the project - if it explicitly requires ad-hoc re-use then it is clear that the code of practice cannot be followed. In situations like these, it should be the responsibility of the project supervisor to guide the student through such software engineering bad practice.

[6]We do not address the question of *ownership* or Intellectual Property in this report. We use the notion of your own code to refer to code you have written yourself; of course, you may or may not actually have *ownership* over it.

at the end of the file listing, we see that the code *appears*[7] 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.

```
// Example1.java - sorting an integer array
// Author Dr J Paul Gibson
// Version 1 -  29/10/2003
// For use in TR-2003-12
// Code of practice for reuse of software

class IntArray {

int size;
int max;
int numswaps;
int 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);
```

[7]We write *appears* because we cannot be sure that this is its behaviour without performing some adequate tests; unless, of course, the author provides a formal specification of what it does and a guarantee that it meets the specification, which is unlikely!

```
}// 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 {

public static void main(String[] args){
System.out.println("Example1 - NUIM-TR-2003-12");
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

/* TYPICAL OUTPUT
Example1 - NUIM-TR-2003-12
Randomly generated integer array
 size: 12 max: 8 values:  5 5 7 1 7 6 5 6 1 5 6 1
 number of swaps = 0 number of comparisons = 0
Array after it is sorted
 size: 12 max: 8 values:  1 1 1 5 5 5 5 6 6 6 7 7
 number of swaps = 15 number of comparisons = 66
*/
```

Using this example, we illustrate unacceptable types of software reuse 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 as that which would be typically produced and submitted by such a student:

```
// MyArray.java
// Author 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]);
}
}
```

The amount of time taken to produce the `MyArray` class could typically vary between 10 minutes to 10 days, depending on the ability of the student. Note that, to anyone familiar with programming, this submitted code is plagiarised[8] from the original `Example1` class. The student has made a number of minor changes:

- the layout and indentation has changed,
- comments have been changed,
- names/identifiers have been changed; although not all of them,
- code has been removed (in this case, the student saw no need for counting `swaps` and `comparisons`, or for a `toString` method),

[8] In particular, note that the author's details in the header comments have been removed, suggesting *deliberate deception* rather than simple *engineering incompetency*.

- In the compare method, a '>' is changed to a '<' (to sort in descending rather than ascending order), and
- 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 reused. However, the degree of acknowledgement could vary from a vague and imprecise: *This code was inspired by the work of Paul Gibson*; 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 reuse is the most common form of reuse that is found in submitted work! In section 4, we illustrate more rigorous types of reuse of `Example1` that are considered to be acceptable, and do not require *cut-and-paste* programming. First, we must mention other forms of plagiarism.

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 reused without proper acknowledgement can be hard to identify due to its 'softness'! To clarify this, we illustrate other forms of software plagiarism, where the reuse is less obvious[9] than that shown in the previous Java example, but which is nevertheless considered to be unacceptable.

3.1 Re-use through *collusion*

In all practical projects, it is considered normal practice to be given help. This help must[10] 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 (perhaps as a co-author of the paper, or in the form of financial payment for their time). 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. Software collusion involves a third party writing the code for at least one of these components, and a student submitting this code as their own. (The worst case is where a third party has written the code for all of the components.) In such an instance, both parties are guilty of breaking the law with respect to plagiarism. Such an instance will

[9] Note that this form of plagiarism will be *obvious* to an experienced software engineer, but could be 'missed' by a non-expert.

[10] The main professional bodies — the ACM and IEEE - practice blacklisting when help goes unattributed. Consequently, we chose to use the word 'must' rather than 'should'.

be treated as deliberate deception and will result in the student (or students) being sent to the disciplinary committee[11]:

> "*. . . Penalties imposed may involve suspension or expulsion from the course and from the University . . .*"

3.2 Re-use through unacknowledged reverse engineering

Often software engineers will look at some code and be able to reverse engineer[4] some abstract property of that code in order to reuse 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 reuses 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 innovative design should be recognised in the evaluation of any project. Reusing other engineers' designs without proper acknowledgement is as bad as reusing their code in the same way. In industry, this would usually be considered a *worse case scenario* because of the higher risk of patent infringement.

3.3 Re-use through 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 reuses it to generate[12] 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 Re-use through 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). This is not always bad practice[13] when it is properly acknowledged. It is very bad practice if not properly acknowleged.

[11] NUIM University Calendar 2004, page 78.

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

[13] It is usually a result of bad practice when done by students during their final year projects!

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 A Simple Guideline: No Reuse Without Test

From the examples above, it seems that you need to be very careful about acknowledging any reuse of code. There is a simple guideline to ensure that you never forget 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 examples that follow, in the next section, we do not explicitly state how the reused artefacts should be tested in order to show that they *do what they are supposed to do*. In the case that a student does not properly test the software that they are reusing, this student should be advised that the reuse is unacceptable. The following guideline[14] is suggested:

- If you don't know how to test it then don't reuse it.
- If you don't know what to test it against then don't reuse it.
- If you know what to test and how to test it, then reuse 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 artefacts that they reuse: if their system fails due to a defect in another person's software then this is the student's responsibility! Of course, reuse of good quality software (like that found in library classes) does not require as much testing as reuse of some piece of code found on the web, for example.

4 Acceptable Forms of Software Reuse

We should note that it is usually a policy of a University that[15]:

> *"Examiners, tutors and markers are required to report instances of suspected plagiarism ..."*

Thus, it is the responsibility of the student to ensure that their code is above suspicion. In this section we identify useful strategies for reuse that would leave an examiner in no doubt about what has been re-used and what has been the original contribution of the student.

4.1 Composition/Aggregation

Consider a problem where the chosen solution (in Java) requires two `IntArray` components. Typically, in high level languages, such `IntArray` behaviour can be reused without having to edit the original file in which the code was written. In Java, the `import` keyword is used to reuse a *package* of classes (specified as a directory in

[14] We use the word 'test' to represent some sort of validation or verification, and we use the word 'it' to represent any software artefact.

[15] NUIM Calendar 2004, page 78.

which the `.class` files are found). An example of this is given below, where we note that the reuse of code is properly acknowledged as a comment in the code[16]

```
// Composition.java - reuse of IntArray
// Author Dr J Paul Gibson
// Version 1 - 29/10/2003
// For use in TR-2003-12
// Code of practice for reuse of software
// Re-use of IntArray class written by J P Gibson
// Original source file:  Example1.java

import IntArray.*;

class Composition1{

public static void main(String[] args){
IntArray first,second;
first  = new IntArray(12,8);
second = new IntArray(12,8);
System.out.println("Composition1 - NUIM-TR-2003-12");
System.out.println("first: "+ first);
System.out.println("second: "+ second);
}

} //ENDCLASS Composition1

/* TYPICAL OUTPUT

Composition1 - NUIM-TR-2003-12
first:  size: 12 max: 8 values:  3 4 1 3 6 3 3 2 2 1 4 2
 number of swaps = 0 number of comparisons = 0
second:  size: 12 max: 8 values:  6 5 8 8 3 4 7 4 8 8 2 1
 number of swaps = 0 number of comparisons = 0
*/
```

4.2 Inheritance

Consider a problem whose solution requires an integer array that would be sorted into descending order. The `IntArray` class provides very similar behaviour — it can sort the integers into ascending order. Typically, software engineers could reuse the `IntArray` in many different ways to provide the required behaviour:

- (1) Create a new class whose internal state is an `IntArray` component (like in the previous example). Then, provide a new method to reverse the values in the array. Then, redefine a new sort to call the original sort routine, followed immediately by a call to reverse. (This outline design appears quite straightforward, but if you try to code it up you will see that it is open to different implementations, all of them fairly 'messy'.)
- (2) Create a new class which extends the `IntArray` with a new sort method, taking a boolean parameter to specify whether the sort is to be done in ascending or descending order.

[16]This reuse must also be more explicitly mentioned in the design documentation. Furthermore, the reuse must be verified — by writing test code that shows that the `IntArray` class provides the required behaviour.

- (3) Create a new class which overrides the `IntArray` sort code by making it sort in descending order
- ...

Option (2) is the nicest design, but option (3) is the easiest to code[17]. For simplicity we chose a variation on option (3) in the code below:

```
// Extension1.java - reuse of IntArray
// Author Dr J Paul Gibson
// Version 1 -  29/10/2003
// For use in TR-2003-12
// Code of practice for reuse of software
// Re-use of IntArray class written by J P Gibson
// Original source file:  Example1.java
import IntArray.*;

class Extension1 extends IntArray{

// Overrides sort method to sort 'downwards'
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

Extension1 (int x, int y){super(x,y);};

public static void main(String[] args){
Extension1 descending = new Extension1(12,8);
descending.sort();
System.out.println("Extension1 - NUIM-TR-2003-12");
System.out.println("descending: "+ descending);
}

} //ENDCLASS Extension1


/* TYPICAL OUTPUT
Extension1 - NUIM-TR-2003-12
descending:  size: 12 max: 8 values:  8 8 7 6 6 5 4 4 3 3 2 1
 number of swaps = 34 number of comparisons = 66
*/
```

4.3 Template Reuse - genericity

In software engineering, high level languages usually provide a means of specifying parameterised behaviour. Typically, we see this in the form of generic data structures. A classic example is that of a stack (with 'LIFO' behaviour offered through methods push and pop). This provides great opportunity for reuse: if you need a stack of integers and someone provides a generic stack then all you need to do is reuse their generic stack by instantiating the type parameter to integer.

[17]Students should be encouraged to discuss such design (implementation) decisions in their code. Too often, an examiner is presented with a final piece of code with no idea of the choices that were made in its synthesis.

Java currently does not provide such templates as part of the standard language. Typically, if you require a stack of 'somethings' you need to directly edit an already existing stack of 'somethingelses' by renaming all occurences of 'somethingelse' by 'something'. This is cut-and-paste programming, albeit in a more structured and constrained style. There have been proposals to introduce genericity to the Java programming language (for example, in the form of **GJ**ava[1]). If you are using Java in any project we advise you to look at the generic extensions that are available. In particular, Java 1.5 [2] has generics functionally equivalent to templates.

When working on any software engineering project, the choice of programming language is central. If you require reuse in the form of generic components then you are advised to make sure that your choice of language supports it in some way. If it does not then you have to be very careful about how you acknowledge the contribution of other people's work when you reuse their generic structures. In particular, if no proper acknowledgement is given then this would be considered as plagiarism through a specific type of reverse engineering.

5 File-Level Reuse: A simple yet effective code of practice

As the examples in the previous sections show, unacceptable forms of software reuse 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 reuse 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 reuse 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 reused will not 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 reused software will be properly acknowledged in the documentation, and the student must clearly[18] distinguish between the software that they have reused and the software that they have written themselves; and they must note in their own software where the reuse occurs.
> **(iii)**: All reused software must be *adequately*[19] tested.
> **(iv)**: All students who are found to have plagiarised software - intentionally, or not - will be punished[20].

To conclude, we argue that this code of practice meets all our original requirements (from section 1): it is simple to understand, it is simple to implement and enforce, it is consistent with University policy, and it is fair to the students.

Acknowledgements

Many thanks to all staff and students who commented on numerous versions of this report. Particular thanks to Des Traynor for his advice on almost every aspect of the document.

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

[19] The notion of *adequate* depends on the nature of the code being reused. At one extreme, classes from a standard library can almost be considered as part of the language and should not require much testing to show that they do what they are supposed to do; provided, of course, that the person reusing the library classes understands them! At the other extreme, code that is cut-and-pasted from the web should be exhaustively tested; it is very likely that it does not do what is required to do, or claims to do!

[20] Following the guidelines set down in the department's final year project and the unversity calendar.

References

[1] E. Allen and R. Cartwright. The case for run-time types in generic java. In *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines*, pages 19 – 24. ACM International Conference Proceeding Series archive, 2002.

[2] G. Bracha. Generics in the java programming language. Sun online tutorial, Sun, July 2004. "http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf".

[3] C. Gacek, editor. *Software Reuse: Methods, Techniques, and Tools, 7th International Conference, ICSR-7, Austin, TX, USA, April 15-19, 2002, Proceedings*, volume 2319 of *Lecture Notes in Computer Science*. Springer, 2002.

[4] 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.

[5] G. Harvey. *Writing with sources: A guide for students*. Hackett Publishing Company, 1998. ISBN: 0872204340, URL: "http://www.fas.harvard.edu/ expos/sources/".

[6] D. O'Donoghue. Final year project handbook. Computer science department report, NUI Maynooth, Oct. 2004. "http://www.cs.may.ie/internal/fourthyearhandbook.pdf".

[7] R. Rada. *Software Reuse*. Intellect Books, 1994. ISBN: 1871516536.