UNIVERSITY OF OSLO
Department of Informatics

Detection of plagiarism in Java programming assignments in introductory computer science courses at Ifi

Short master thesis

Jose Luis Rojas

May 2012

Acknowledgments

Most of all I thank my supervisor Arne Maus, for his great help and advice during the whole process of preparing this thesis, his insightful reviews of my work, and also his support of my related earlier work way back many years ago when I first became interested in the topics discussed here. His numerous stories and anecdotes also helped make our discussions very instructive and interesting.

A very special thank-you must also go to Ragnhild Kobro Runde and Dag F. Langmyhr for their encouragement and help in getting me started with this master's thesis. The lecturers that have given me the opportunity to apply my plagiarism detection system to their courses through the last years are also thanked.

I am also grateful to my family and friends for their support and understanding during the time when I was working with this thesis.

Jose Luis Rojas K.,
mail@joselr.com
Oslo, May 2012

Contents

Acronyms

ACM	Association for Computing Machinery
Antics	Anti-Cheating System for source code assignments
API	Application programming interface
ATM	Attribute counting metric (algorithms)
CGI	Common Gateway Interface (web technology)
DBMS	Database management system
EAR	Entity-Attribute Relationship (database diagram)
FS	*Felles Studentsystem* ("Joint student system")
GLS	*Godkjentlistesystemet* ("Approved-solutions list system")
GUI	Graphical user interface
HTML	HyperText Markup Language
Ifi	*Institutt for informatikk* (Department of Informatics, UiO)
INF1000	Introduction to object oriented programming (course at Ifi)
INF1010	Object oriented programming (course at Ifi)
INF2220	Algorithms and data structures (course at Ifi)
Java SE	Java Platform, Standard Edition
JDBC	Java Database Connectivity (API in Java SE)
MVC	Model-view-controller (design pattern)
NYU	New York University
ORM	Object-relational mapping
PRD	Preprocess-Reorder-Diff (algorithm)
SM	Structure metric (algorithms)
TA	Teaching assistant
UI	User interface
UiO	*Universitetet i Oslo* (University of Oslo)
URL	Uniform resource locator
VPN	Virtual private network

Chapter 1

Introduction

Plagiarism is unfortunately a common problem in undergraduate computer science courses that involve assessment of programming assignments (Vamplew and Dermoudy, 2005), and the Department of Informatics (Ifi) at the University of Oslo is no exception. One way of addressing the problem is to use tools that can automate the detection of suspected plagiarized solutions.

In this thesis I describe an algorithm for identifying suspicious student solutions of programming assignments, and a new stand-alone system built around the algorithm. A similar version of the underlying algorithm has been used for several years in three courses in Java programming at the department. The algorithm has been incorporated into the Joly application, which I introduce in section 2.7.1 (p. 16), and it has proven effective in identifying cases of plagiarism. It has been applied as part of an ongoing effort to reduce code plagiarism, and there are already indications of improved learning outcomes in one of the courses where it is used (see table 7.1 (p. 56) or Maus and Lingjærde (2009)). I will call the algorithm Preprocess-Reorder-Diff (PRD), which indicates the main steps it uses to compare the structure of program codes. The new stand-alone application is called Antics—Anti-Cheating System for source code assignments.

The application of my algorithm has until now required making a number of adjustments each time it was used on a new assignment, not only selecting appropriate thresholds in the algorithm itself, but also running several supporting programs and scripts to convert the file format of submissions to one that the algorithm accepted, and to collect desired information about the submissions. In one of the courses where it is regularly used, INF1010, I adapted it to four different assignment submission systems during the last few years, and to several different assignment characteristics in each system. As Freire and Cebrian (2008, p. 6) explain:

> "The existing variability in submission delivery formats, generally imposed by the graders, frequently becomes a major usability bottleneck. Plagiarism detection tools expect submissions to be structured according to rigid specifications, while each university or subject will probably adhere to a different standard. [...] Without [tools to automate the conversion of submission formats], the manual intervention required to reach a common format would probably discourage many graders from performing automated plagiarism detection at all."

Regarding the use of suitable thresholds and other internal algorithm parameters, they also note that "the effectiveness of these algorithms is strongly dependent on "details" such as thresholds and parsing methods" (Freire and Cebrian, 2008). A goal of my new stand-alone application is to automate these tasks as much as possible so that teachers and teaching assistants be able to use it easily.

Another goal is that the application be relatively easy to incorporate into other systems like Devilry, which is being used for submitting solutions in many courses at the department.

1.1 Research question

The following is the research question I got at the beginning of my work with this thesis, on January 16th, 2012. The thesis is my attempt to answer these questions. I include the original Norwegian version first, followed by my English translation, in which I added "(a)–(n)" labels as explained below, for use later in this thesis to refer back to specific parts of the research question.

Fuskesjekk for obligatoriske oppgaver skrevet i Java i begynnerkursene på Ifi

I Joly er det laget en stor database over innleverte obligatoriske oppgaver i Java, og særlig da i kurset INF1000. Kandidaten har tidligere laget et Java-program som sjekker om en nyinnlevert oblig er en kopi, eller nær kopi, av en av de 'gamle' besvarelsene i Joly-databasen.

Oppgaven nå er todelt:

1. Det lages et frittstående anti-fuskesystem (AntiF) med et meget brukervennlig grafisk brukergrensesnitt for gruppelærere/forelesere som kan ta enkeltfil eller hele filområder med innleverte obliger i et kurs og sjekke hver av innleveringene for likhet mot Jolydatabasen. For hver av filene lages en % score hvor mange essensielt sett like linjer denne innleveringa har med den/de mest like obligene i databasen (Dersom den filen som testes også er levert i Joly, skal den selvsagt ikke sammenlignes med seg selv). Det vurderes å legge de sjekkede filer inn i basen dersom man har nok opplysninger om kandidat-navn og helst også : Kurs, oblignr og år. Om mulig forbedres og feilsikres også fuskesammenligningsalgoritmen.
2. Det foretas deretter en omfattende vurdering om hvordan AntiF kan inkluderes i Devilry slik at det blir meget lett (eller automatisk) å foreta slik fuskesjekk av alle innleveringer av Java-programmer til Devilry , hvor det bl.a må tas hensyn at Devilry baserer seg på en fil per innlevering mens AntiF og Joly bruker en database; og at Devilry ikke bare har innlevering av obliger i Java, men også i nyttes i flere andre fag og for andre programmeringsspråk. Slik integrering implementeres dersom det synes relativt enkelt. Det er da viktig at innlevinger til Devilry også selv legges i databasen slik at kopi av nyere og 'årets' andre innlevinger kan oppdages.

Detection of cheating for programming assignments written in Java in the introductory courses at Ifi

In Joly, there is a large database of submitted mandatory assignments in Java, and especially in the course INF1000. The candidate has previously made a Java program that checks whether a newly delivered solution is a copy, or close copy, of one of the 'old' solutions in the Joly database.

The task now is twofold:

1. (a) A stand-alone anti-cheating system (Antics) shall be made (b) with a very user-friendly graphical user interface for teaching assistants/lecturers (c) so that they can take single files or entire directories with mandatory assignments submitted in a course and (d) check each of the submissions for similarities in the Joly database. (e) For each file a % score shall be computed representing the number of essentially equal lines that this submission has with the most similar submissions in the database ((f) If the file to be tested is already in Joly, it should obviously not be compared with itself). (g) It should be considered whether

to add the checked files to the database if enough information is available about the candidate's name and preferably also: Course, assignment number and year. (h) If possible, the cheating comparison algorithm shall be improved and secured against errors.

2. (i) Afterwards a comprehensive assessment shall be conducted regarding how Antics can be included in Devilry (j) so that it be very easy (or automatic) to run such a cheating check of all submissions of Java programs to Devilry, (k) where among other things it must be taken into account that Devilry is based on one file per submission while Antics and Joly use a database, (l) and that Devilry does not only contain submissions of mandatory assignments in Java, but also in several other courses and in other programming languages. (m) Such integration is to be implemented if it seems relatively easy. (n) It is important that submissions to Devilry themselves also be added to the database so that copies of more recent and "this year's" other submissions can be detected.

1.2 Overview of tasks

In order to make it easy to refer back to the different parts of the research questions in section 1.1 (p. 8), I introduce the following shorthand notation of "tasks". The (a)–(n) labels above show what the tasks refer to, and the following overview shows some of the sections in the thesis where I discussion or illustrate these tasks:

Task 1.(a) (stand-alone system): Chapter 5 (p. 33); Chapter 6 (p. 49)

Task 1.(b) (user-friendly GUI): Section 6.1 (p. 49), section 6.1.1 (p. 49), figure 6.1 (p. 50), figure 6.3 (p. 52)

Task 1.(c) (directory import): Section 6.1.1 (p. 49)

Task 1.(d) (run against Joly database): Section 5.4.1 (p. 47), section 6.2 (p. 52), section 6.1.1 (p. 49), figure 6.3 (p. 52)

Task 1.(e) (percent score): Section 4.4.3 (p. 29), section 4.2 (p. 25)

Task 1.(f) (skip same-submission): Section 6.1.1 (p. 49)

Task 1.(g) (consider export to Joly database): Section 6.1.1 (p. 49)

Task 1.(h) (algorithm improvement): Section 4.6 (p. 32)

Task 2.(i) (assess Devilry inclusion): Section 6.3 (p. 53), table 5.3 (p. 39), table 5.4 (p. 41),

Task 2.(j) (automation in Devilry): Section 6.3 (p. 53)

Task 2.(k) (file vs. database storage): Section 5.4.1 (p. 47)

Task 2.(l) (language identification): Section 5.2.3 (p. 37)

Task 2.(m) (integrate if easy): Section 6.3 (p. 53)

Task 2.(n) (Devilry to Joly database export): Section 6.3 (p. 53)

1.3 Overview of chapters

The chapters of this thesis are organized as follows.

Chapter 1 (Introduction): This chapter presents the topics I have worked on in the thesis, and the main reasons it was needed. This is followed by the research question that the thesis is going to address, and finally an overview of the sub-tasks of the research question, and of the chapters of the thesis is given.

Chapter 2 (Background): In this chapter I describe the processes and systems in use at Ifi that can be complemented or improved by the work done in this thesis. This includes an account of the procedures around electronic delivery of student solutions at Ifi, detection of suspected plagiarism in these, and the management of the cases that arise. I also discuss how my new application will relate to the various systems in use.

Chapter 3 (Source code comparison algorithms): This chapter summarizes the various techniques that can be used for automatic detection of plagiarism in source codes. Mine and other related algorithms developed at Ifi are positioned in this larger context.

Chapter 4 (Design of the PRD algorithm): Here I describe the various cheating stragegies that should be uncovered by an algorithm for detection of plagiarism, and how I designed and programmed the PRD (Preprocess-Reorder-Diff) algorithm to identify these automatically.

Chapter 5 (Design of the Antics application): This chapter describes the goals of the application I built around the algorithm in chapter 4, how I designed it to fulfull these goals, its overall architecture, and why I chose the technologies selected for it.

Chapter 6 (Implementation): This chapter goes into more detail about the implementation of the Antics stand-alone application, especially its web-based graphical user interface, and how it can be used to detect plagiarism in computer courses.

Chapter 7 (Results): Here I present some results of using my PRD algorithm for plagiarism detection at Ifi, both by itself and when used through the new stand-alone application.

Chapter 8 (Conclusion): The last chapter summarizes the experiences I acquired during the work with the thesis, pointing out what was accomplished regarding the research question, and what remains to be done. The thesis concludes with some suggestions for further work.

Chapter 2

Background

In this chapter I describe in more detail the background and motivation for this thesis, including: the current status of plagiarism detection in some beginning Java programming courses at the Department of Informatics (Ifi), University of Oslo; the student submission systems that have been used; and the goals for a new tool that can help in finding better practical solutions to the problems raised. My proposals for solutions will be discussed starting from chapter 4 (p. 24).

2.1 Main themes of the thesis

In order to give the reader a good idea of what to expect in this thesis, here is a bird's eye view of the work undertaken and the main topics that will be covered:

- **Short master thesis:** This thesis belongs in a relatively new category of master's degree programs offered at Ifi where the written thesis is a smaller portion of the work compared to traditional master's—which are expected to be written during $1\frac{1}{2}$ years of work. The short theses, like this one, are intended to be written during only 17 weeks. This has implications for the breadth and depth of the research undertaken.
- **Specific tasks:** The aim of this work is to answer the specific research questions posed in section 1.1 (p. 8). I will refer back to these questions throughout the thesis, using the shorthand notation of "tasks" explained in section 1.2 (p. 9)—where you can also find an overview of the main sections of the thesis that address each of the research questions.
- **Practical approach:** The tasks in section 1.1 (p. 8) have a high degree of practical focus instead of theoretical, and this is reflected in the type of work and discussions presented in this thesis.
- **Programming codes:** The practical algorithms and systems that will be discussed are concerned with comparing programming language source code instead of natural language running text. The analyses of these two types of data are normally very different. The second type is generally better understood and tools are more widely used (Lim et al., 2011), one such system for plagiarism detection in natural language texts used at the University of Oslo and elsewhere in Norway is Ephorus. In this thesis I focus solely on the problem of detecting plagiarism in programming language codes.
- **Home assignments:** Sometimes plagiarism detection is incorrectly stated as detecting cheating on examinations. This thesis has probably little applicability to exams, where

the students are normally monitored constantly and the types of cheating that may occur are likely very different from the ones seen in home assignments. Instead the focus is on detecting cheating on mandatory assignments that the students normally solve during weeks or months and can work on at home. This type of assignment naturally provides more opportunities for cheating.

- **Old algorithm in new clothes:** As pointed out in the preamble of the research question in section 1.1 (p. 8), the PRD algorithm that I describe in this thesis is not new, I made the first version already in 2007, and that version could identify practically the same cases of plagiarism as the latest versions. What is new is the description and discussion of the algorithm, some new improvements in the latest version (an example of a new improvement is one that doubled the speed by reducing the number of comparisons done by half), and especially the new Antics stand-alone application for ease of use.

2.2 Courses as test cases

The courses in which earlier versions of the alogrithm have been used regularly are the first three courses in Java programming that students normally take at Ifi:

INF1000 - Introduction to object oriented programming.

INF1010 - Object oriented programming.

INF2220 - Algorithms and data structures.

In several of the study programs that students can follow at Ifi these courses are scheduled to be taken the first, second, and third semester of their studies, in the order listed above. The first course can be substituted with INF1100, which uses a different programming language (Python), and the third course is not mandatory in all study programs based at the department.

These courses have between 150–500 students in one semester, and all involve 2–4 mandatory programming assignments in Java as an important part of the course. The students must get their solution to each assignment approved before they can take the final examination in the course. In order to correct the assignments the courses employ 5–20 TAs (teaching assistants).

It is difficult to prepare completely new mandatory assignments each semester, so in many cases the courses reuse assignments from previous years with only minor modifications. This has led to a proliferation of earlier student solutions which new students can often acquire, for instance from friends or from the Internet. Unfortunately such plagiarism happens frequently in universities, and this makes it difficult for the teachers to appraise the students' work fairly (Lim et al., 2011).

2.3 The need for automation

Despite various efforts at our department and in other universities, plagiarism in introductory programming courses remains very difficult to eliminate, and "will not likely disappear any time soon" (Vogts, 2009). A more reasonable goal to begin with is to try to reduce the incidence of "unacceptable" plagiarism—for instance copying of more than 50% of the code in an assignment (Vogts, 2009). Before we started applying plagiarism detection tools systematically to all deliveries in these courses, some lecturers at Ifi estimated that at least 20% of the students in the introductory courses cheated in mandatory assignments (Kielland, 2006). This is in line while several reports of rates in the order of 10% to 30% plagiarized solutions within individual assignments (e.g. Zobel and Hamilton (2002) and Culwin et al. (2001), as quoted in Vamplew and

Dermoudy (2005)), although the estimates in other universities are lower (Kielland, 2006). After having worked with this for a few years I have collected some evidence, which I present in section 7.1 (p. 56), that suggests the rates of plagiarism in the first course mentioned above actually were around 20% before we started using automated plagiarism detection tools, and that our efforts have had a positive effect since then, apparently reducing the rate to around 2–5%.

To get an indication of the size of the problem of detection, let's take as an example Assignment 3 in the course INF1000 the previous semester, Fall 2011. 310 students delivered their solutions. A number of them delivered several versions of their solution (it is a big assignment, and the students can deliver several versions before the deadline, and some of them need to correct errors in their first version after the initial deadline and deliver an updated one). In total 441 `.java`-files were delivered for this assignment in the Joly system, counting all versions the students delivered. Now, for plagiarism detection I compared these 441 files amongst themselves and also against the 2709 older deliveries I have collected that were student solutions to essentially the same assignment, from 2004 to 2010, for a total of 1.3 million comparisons needed, calculated as follows:

1. First, each of the 444 new solutions in the current semester needs to be compared against all the 2706 older deliveries of the same assignment, which gives a total of 1.2 million comparisons: $(441 \times 2709) = 1,194,669$.
2. Next, all the 441 new solutions in the current semester needs to be compared against each other, which means 97,240 additional pairs of programs to be compared: $(441 \times 441)/2 = 97,240$.

The total for both types of comparisons was in this case 1.3 million.

Even though it's possible to compare a few pairs of programs manually, it is a very error-prone and time-consuming approach (Kar, 2000), and it is impossible to do a million program comparisons by hand. The situation is further complicated by assignments with several source code files which sometimes can be developed independently of each other, and therefore may need to be compared individually. An example from the last plagiarism check I did in INF1010 this semester consisted of 39 million comparisons.

> If there are many source codes in a program set, it is nearly impossible to compare all pairs of programs manually. For example, if there are more than 1,000 source codes in a program set, the number of required comparisons is about 500,000. Humans cannot manually search for all plagiarized cases in a program set with so many source codes. Therefore, we need an automatic plagiarism detection system (Lim et al., 2011).

As we can see, a tool for performing these comparisons needs to be automatic. In general, the number of comparisons C needed when the program set has N_{new} source code files from the current semester and N_{old} from earlier semesters is:

$$C = \frac{N_{new}^2}{2} + N_{new} \cdot N_{old} \tag{2.1}$$

A version of the PRD algorithm that I'll describe in this thesis has been the main tool used to identify possible cases of plagiarism in the courses mentioned above for several years (in INF1000 since 2007, in INF1010 since 2008, and in INF2220 since 2009). Generally, we always find very likely cases of plagiarism in each assignment, though the number of these has been diminishing.

2.4 Dealing with suspected plagiarism

It's important to emphasize that the results identified by automatic tools are only suspected cases. Computers are not the judges nor executioners, a human always has to review the results before serious action is taken,

> "Plagiarism detection tools can only point to probable cases of plagiarism, but the final decision is always in the hands of a human grader. The value of automated plagiarism detection is in narrowing down the search by highlighting the clearest cases." (Freire and Cebrian, 2008)

The procedure used in the three Ifi courses I cover has been that the lecturer compares the suspected submissions manually, possibly after an initial review and recommendation by the TA that would normally assesses the particular student solution. Then the lecturer usually summons the suspected student for a meeting to let them present their side. There can be great variety in the types of cases, from very clear-cut obvious copies of solutions from earlier years, to collaborations between students that both delivered their solution the same semester. If it is found out that one party clearly helped the other, only the case with student that received too much "help" is pursued.

2.5 Dealing with confirmed plagiarism

The University statutes require that action be taken when plagiarism is confirmed, so then a report is made to the university disciplinary committee. Other universities have similar rules (Vamplew and Dermoudy, 2005), but this is not necessarily the only way to proceed. Some studies have investigated possible benefits of other courses of action (Vogts, 2009), (Sheard et al., 2003).

I'm not sure how the mayority of cases are ultimately resolved once they arrive at the higher levels of the system, but I have been told the penalties at the university if found guilty are usually one or two semester's suspension. A problem that appeared when the first cases were sent to the disciplinary committee was, as Kar (2000) also found: "Even [when certain types of program code plagiarism are] detected, it may be difficult to prove such cases to the judicial panel [...] because most of [the] panel members usually come from no programming background at all".

2.6 Practical considerations

In this thesis I'll focus on the technical aspects of plagiarism detection systems, and not the philosophical or practical implications. I'll only mention some opinions about these in this section.

On the **positive** side, one of the main goals of plagiarism detection is to reduce cheating, and if this is achieved the students will likely learn more. For instance, the tool could help prevent cases like those Zobel (2004) notes (as cited in (Vamplew and Dermoudy, 2005)), where students who cheat in the hope of catching up later could get caught in a "cycle of plagiarism" which can prevent them from ever developing the skills necessary to solve the assignments independently and doom their educational career. If plagiarized submissions are not detected and penalised, the efforts of the more honest students are devalued (Vamplew and Dermoudy, 2005). Also, some interesting papers that advocate alternative courses of action instead of punishment, still describe the usefulness of having a working automatic detection system to accomplish other desirable goals (Vogts, 2009), like detecting when novice programmers require additional educational support. According to Vamplew and Dermoudy (2005), "The primary tool in combating

plagiarism should be education of students about the issue. The need still exists however, for means to detect plagiarism when it does occur, and automated tools can provide valuable assistance in this task".

Another positive effect, mentioned in Vamplew and Dermoudy (2005), can follow especially from some types of systems that can reduce plagiarism even if their detection capabilities did't work, or indeed if the system didn't even exist at all. For instance a system like APE/Gorilla, that could guarantee detection of the easiest ways to plagiarize (file copying and copy-pasting) would in practice work more by deterring plagiarism rather than detecting it, as it would be anticipated that very few students will be foolish or desperate enough to carry out such actions when they have been told about the system (Vamplew and Dermoudy, 2005). In general, one may expect that plagiarism can be reduced by merely telling the students that any kind of system to detect plagiarism is being used, but one should also consider some dangers, like creating an arms race, where students look at the detection system as a challenge and attempt to find ways to beat the system, leading to the need for incresingly more complex detection systems, and so on.

Negative opinions include those that question the whole enterprise of plagiarism detection and prosecution. One popular example is the blog post "Why I will never pursue cheating again", published on the web by a professor at NYU (Ipeirotis, 2011), in which he explained the problems he experienced after pursuing plagiarism in a computer science course. An atmosphere of suspicion affected the course dynamics, and negative reviews by the students even affected the professor financially. Interestingly, he later removed the contents of the blog post, with this notice: "The post is temporarily removed. I will restore it after ensuring that there are no legal liabilities for myself or my employer." In my opinion it was a revealing account of the negative aspects of plagiarism detection, and he made a compelling case for creating assessment procedures that minimize the problem by design. I like the idea of trying such alternatives when viable, instead of always using detection tools. Vamplew and Dermoudy (2005) and Hwang and Gibson (1982) also discuss such alternatives, including grading methods that can discourage the practice of plagiarism.

Negative side effects of the conventional ways of dealing with cheating include the large amount of extra work they can impose on teachers to control manually the suspected cases, prepare documents for the case (marking similarities manually in pairs of source code listing, and such, especially if the detection tool doesn't help with this). This can divert resources from other, possibly more productive activities, and can feel like a waste of time and resources. It's certainly not fun for the students involved either, which can be required to attend interviews and hearings at various levels, write accounts of their actions, and can be suspended, etc. A comprehensive approach should weigh the expected benefits versus drawbacks. Hopefully, as stated by Kar (2000) such systems "can save time and resources for teaching large programming courses successfully and satisfactorily."

Finally, I will mention a way of reducing cheating that has been shown to work, student run **honor codes**. This model is not incompatible with the use of automatic tools for detecting plagiarism, but one difference is that one can expect less cheating in general, and therefore less cases to deal with. According to the Wikipedia article on academic dishonesty, "Several studies have found [that] students who attend schools with honor codes are less likely to cheat than students at schools with traditional integrity codes (McCabe and Trevino, 1993). Another study found that only 28% of schools with honor codes have high levels of cheating, whereas 81% of schools with mixed judicial boards have high rates of cheating (Bowers, 1964)." These results come from universities in the USA with very strong student run honor codes, the kind where even examinations can be completely unsupervised, and penalties for cheating are severe. While this type of full honor code may be difficult to implement in other universities, research by McCabe et al. (2002), as cited in Wikipedia, suggests that an intermediate solution can also be effective. This type of "modified honor code" retains the traditional proctored exams, but puts students in charge of the judicial hearing process, making it clear that it is the students' responsibility to stop cheating amongst themselves (Wikipedia).

2.7 Assignment submission systems at Ifi

In the three courses mentioned earlier (INF1000, INF1010, INF2220) we have used a number of different submission and assessment registration systems over the last few years. Since the assignments were often only minor modifications of earlier versions of the same assignments, it was often necessary to collect together in a coherent format the earlier submissions from various systems before I could apply the plagiarism detection algorithm.

This is one of the problems I'm addressing with the new system presented in this thesis, and it's one of the reasons I personally ran the algorithm during the last few years, especially in the courses INF1010 and INF2220.

As an example, the course INF1010 has used all the mentioned systems the last few years: E-mail submissions in 2007, Joly in 2008, Upload.cgi in 2009 and 2010, an early version of Devilry in 2011, and an updated Devilry in 2012. My algorithm has been the main one used for plagiarism detection in that course every year since 2008.

In INF1000 the problem was smaller because fewer different systems were used for submission of solutions—Joly has been used in most semesters since 2006, but I still often did the plagiarism detection in this course as I'll discuss in the next section (2.7.1 (p. 16)).

One major goal of the new system is to make this process more user friendly, and easy to adapt to new submission systems. In the following sections I give an overview of each of the assignment submission systems that have been used in beginning Java courses at the department; then some advantages and disadvantages of the system regarding its use at the department generally; and then some information about the way they have been used with my algorithm earlier and prospects for how they the new Antics system will relate to them. Since Antics will be tightly integrated with the Joly database and Devilry, I'll come back to these in much more detail in later chapters.

2.7.1 Joly and the Joly database

This is the online assignment submission system that really made possible the automation of plagiarism detection for the introductory Java courses at Ifi. It was made in collaboration between several master students under the guidance of associate professor Arne Maus: Steensen and Vibekk (2006) implemented most of the initial system, programmed in Java, with a web interface, and using the Jetty web server and MySQL database; Kielland (2006) also participated in developing the web interface and provided one of the first algorithms used to compare student solutions. The system was named after Norwegian-French corruption hunter and politician Eva Joly.

In this thesis I distinguish between "Joly", meaning the Java online application, and the "Joly database" that it uses for storage. Joly has been used in the course INF1000 for most semesters since 2006, when the first complete version was ready, except in 2010. It has also been used in INF1010 as mentioned above. My involvement has mainly been implementing a version of my algorithm, now called PRD, in Joly in 2007, when we found it could provide more suitable results than the alternatives we had available, as will be discussed in section 3.1 (p. 20). However, there have since been some problems apparently related to the integration of my algorithm in Joly. The algorithm was therefore turned off in Joly and I ran it manually for the last few years in this course. More recently Darisiro (2012) appears to have identified the problem in Joly, he said after adding a test for null value he was able to run thousands more student solutions through the algorithm in Joly. In my local runs of the algorithm outside of Joly I have not had these problems.

Pros: Large database with well organized student submissions; Target of active research; The files are stored with useful metadata in the first line, meaning they are easy to organize and

export to other systems even if when extracted from the database without the accompanying context.

Cons: Runs in a server that is not directly accessible from outside the university, though it's possible to access it using VPN; Very limited authentication of students submitting assignments; Imperfect stability, the system has gone down sometimes; No built-in function for downloading more than one delivery at a time; The current version of the Joly application can be used in INF1000 where assignments consist of one `.java`-file, but at least a small improvement is needed in the application for use in INF1010 or INF2220 to allow the students to deliver several `.java`-files in an assignment. The database doesn't need to be changed if the Joly application puts together these files into one in a suitable way. At the moment the application also allows students to deliver a second file of any type, but the second file is only sent to the examiner by e-mail and is currently not stored in the database.

To work with Joly earlier I made scripts that automated the download of all submissions of an assignment, and that entered them into my algorithm. In this thesis I instead implement communication with the Joly database directly, which is necessary in order to add new solutions to the database.

2.7.2 Upload.cgi

This refers to a Perl CGI script maintained by system administrator Peder Stray which was mostly used in INF1060, but also in the three courses I mentioned earlier in a few semesters.

Pros: Good authentication of students; Stable and reliable online system; The most important data about the delivery is in the file names (username of student, assignment number, and date and time of delivery); Each submission can only consist of one file, but it can be any type of file, including a compressed archive, so students can submit solutions in any programming language, or solutions that consist of several `.java`-files by packing them into a Zip file or similar.

Cons: No history, deliveries are only available for one semester; Little structure, there is only one folder for each course, and all the deliveries during the semester are uploaded to this folder, including all versions of all the student's submissions and all assignments. The last time I used it as TA (INF1010, spring 2010) the directory ended up with 1213 files; The built-in download script (`download.cgi`) requires clicking on each single file that a TA wants to download and the files are retrieved with the wrong file suffix (`.tar.gz` regardless of actual file type).

To cope with this I made a script that automated the download of all submissions from a given set of students or assignment number, correcting the file suffix, and organizing them in a better folder structure. Like my script for Joly, I shared it with the other TAs to ease downloading of the submissions they were to examine (otherwise they would have to scroll through the 1000+ list of files to find the ones they were to assess, and click on each one individually to download it). This delivery system is no longer in use in the three courses I cover, nor in INF1060, so in the current thesis work I'll only implement importation of files or collections of files that were earlier downloaded from this system, but no direct communication with its system. That should not be too difficult to add later though, if ever needed, because the simple structure of the storage folder provides the information needed about each file.

2.7.3 Devilry

This is a new delivery system initiated a few years ago by several students that worked as TAs (teaching assistants) in several courses, led by Espen A. Kristiansen. It has lately been widely adopted by many courses at the department. It is an open source project built using the Django framework for Python.

Pros: Good security features; Stable; Open source; Relatively user-friendly for students; Easily accessible online from outside the university network; Has a function for downloading a Zip file with all the deliveries a TA is to correct; Plugin architecture with a stable and reliable small core easily extendible with plugins; The Django base provides attractive characteristics (Holovaty and Kaplan-Moss, 2009) regarding ease of development, maintainance, user extensions, adaptability to several database backends, etc.

Cons: The web interface for lecturers and TAs that Devilry has had in its first and current versions is not very user friendly, and has caused various problems and extra work for the teachers managing Devilry; Performance penalty because of the Python base (as opposed to what could likely be achieved if programmed in type-safe languages like Java or C++); The default option for downloading student submissions together as a Zip archive is currently missing information about the date and time of delivery.

Regarding earlier use of my algorithm with Devilry, as usual I first made some scripts to import the student solutions into my algorithm's data structure given the format provided by Devilry. The first time Devilry was used in one of the three courses I cover was in INF1010 on the spring of 2011. An example of a new complication I had to adapt my plagiarism tools for was that single deliveries in Devilry can consist of one or any number of files. This is in contrast to the previous systems where the submission systems only stored one file (possibly a Zip archive) for each delivery. Now a single delivery could contain several Zip archives and files of various types.

A problem I found the next semester I used Devilry was that the deliveries from the previous year were no longer accessible in the new version of Devilry (but I had saved them in my local machine, so I was able to use them for plagiarism detection in 2012). This has later been solved in Devilry. Another problem was that the format of the default Zip file provided by Devilry with all submissions of a given assignment had changed, requiring new adaptation of my scripts, since this was the file I used for running the algorithm. The folder structure was changed and the date and time when the students had made their deliveries was no longer included in the Zip file (the time stamps were Jan 1st 1970). This is part of the information I've always saved for all solutions I've collected earlier, because it is useful when prosecuting plagiarism cases, however, this information is still available in the web interface of Devilry, so this is not a big problem. There are also other ways of integrating my algorithm with Devilry to avoid this problem.

2.7.4 E-mail delivery

E-mail has also been used as a means of delivering assigmnents, where the student sends his solution directly to the TA that will assess it, mostly as an attachment in an e-mail.

Pros: No custom-built system required.

Cons: It's very cumbersome to collect e-mail deliveries in the courses covered here because they often have more than 10 TAs and each has around 30 students, which deliver 4 assignments each. The e-mails would be sent directly to these TAs, ending up in many different mail boxes. The result is a large number of different e-mails that would need to be collected in one place.

This form of delivery has not been used in these courses since 2007 in INF1010 and 2008 in INF2220, which is why I have only collected the deliveries after these times, and used them in the comparisons. Going forward the only allowance for this type of submissions in my system will initially be the capability of uploading individual files, folders, or compressed archives containing deliveries; after the user has transferred the e-mails to one of these formats.

2.8 Assessment registration systems at Ifi

Another type of related systems are those for registering the grades or assessment results of the assignments. Since the examiners who assess the student solutions need both to access

the submissions (using the assignment submission system), and to register their assessment of these, it can be an advantage that both systems integrate well or are part of the same overall system. The students also need access to both systems in order to submit their answers and check the assessment these receive.

It is also possible to combine both systems with a plagiarism detection tool. One scenario where it would be useful to integrate my detection algorithm with the assessment registration system is to provide a notice for the examiner on student solutions where a high probability of plagiarism was detected.

At ifi there are several assessment registration systems:

- "GLS" (*Godkjentlistesystemet*, wwws.ifi.uio.no/system/oblig), which was traditionally used by examiners to register results until about a year ago.
- Then "Devilry" (devilry.ifi.uio.no) took over and is now used both for assignment submission and assessment registration in many courses at Ifi. GLS still has to be accessed to find older results, though the examiners no longer have direct access. There are plans to transfer the older data to Devilry.
- Another related system is "FS" (fs.usit.uio.no), which has some of the assessment registrations, including information about students that completed all mandatory assignments in a given course or that have taken the final exam.
- There is also an ongoing project to extend "Joly" (Local URL: plagiat.ifi.uio.no) to include this functionality.

Finally, there are also three command-line tools (netgroups(1), ng(1), ustat(1)) in Ifi's Unix network for accessing a local NIS (Network Information Service) directory service with related information, including information about which examiner is assigned to assess which students' solutions in many courses. This information is useful in my system, so I have used it extensively when running the algorithm manually earlier, and I'll discuss its integration with the Antics system in section 6.1.1 (p. 49).

Chapter 3

Source code comparison algorithms

In this chapter I give an overview of the different types of program code comparison algorithms and systems found in the literature for use in detecting plagiarism, and show where my algorithm and GUI application fits in. I also summarize other such algorithms that have been developed and studied at Ifi.

3.1 Types of code similarity algorithms

A number of diverse algorithms are described in the literature that can be used to detect code similarity with respect to plagiarism. One way to categorize them is into the following four main types according to the code analysis technique they use (combining the categories from Al-Khanjari et al. (2010) and Kar (2000), and using the acronyms in Al-Khanjari, Fiaidhi, Al-Hinai, and Kutti (2010)). I include examples of each type of algorithm from Ifi, where algorithms in all these categories have been implemented and studied.

1. **Attribute counting statistics (ATMs):** These algorithms are based on statistics that count keywords, identifiers, operators, words, etc. in program codes (Kar, 2000). Compared to the other techniques these are the easiest to implement and run very fast (Kar, 2000). At Ifi Kielland (2006) developed an ATM algorithm and incorporated it into Joly in 2006. (the plagiarism detection software I introduced in section 2.7.1 (p. 16)). Other examples include PlagDetect (Al-Khanjari et al., 2010), and Girer's "Accuse" system (Verco and Wise, 1996).

2. **Program structure metrics (SMs):** These schemes convert each student program into a stream of tokens, and then compare the streams corresponding to different student programs in order to find common segments (Al-Khanjari et al., 2010). Implementation of these techniques is relatively difficult since it requieres sophisticated parsing techniques in order to identify the modules in a program (for example methods and classes) and their interrelationships (Kar, 2000). At Ifi, an example of an SM algorithm is my PRD algorithm, which has been used extensively at Ifi since 2007, and was also incorporated into Joly in 2007 (though there have been some problems with this integration). I explain the algorithm in detail in chapter 4 (p. 24). Its running time is around 1 millisecond for each comparison of two program files, using a normal laptop computer. This is fast enough for our current uses at Ifi. Several other systems have been released that use the SM technique, including YAP3, JPlag, and SIM, see table 3.1 (p. 21) (Lim et al., 2011).

3. **Grammar based:** Another possibility is a scheme that analyzes program grammar using a compiler-like system (Kar, 2000). These require much more computing resources and are

normally much more difficult to implement than ATMs and SMs schemes (Kar, 2000). At Ifi Ligaarden (2007) implemented and studied two grammar-analyzing algorithms in 2007. The running time was around 1 second for each comparison of two programs, which is a bit slow considering that around a million program comparisons may be needed in certain assignments at Ifi.

4. **Hybrid systems:** Some systems combine ideas or algorithms from several techniques. An example is the system presented in Freire and Cebrian (2008). Joly can also be considered as hybrid system since it has built in algorithms implementing several of the techniques mentioned above, and it has been used to compare the results of these algorithms during testing, but further work is needed to streamline the application of multiple algorithms during the normal use of Joly with live incoming student submissions. Another type of hybrid system, summarized in section 3.3 (p. 22), can combine the techniques mentioned above with information gathered during the program creation phase.

Given all these possible techniques for automatic plagiarism detection, it would be interesting to find out which may be best in general. One of the best known and most extensive studies that compared several ATMs and SMs is Verco and Wise (1996), which concluded that the SM technique "is consistently equal to or better in performance" than ATM. Whale (1990) also states that "popular approaches [...] based on counting program attributes are shown to be inadequate. A two-stage method of identifying similar pairs based on structural features is proposed, and the superior performance of this technique is established". On the other hand, some studies have found better results with ATMs, like Al-Khanjari et al. (2010).

At Ifi a number of tests were performed, comparing the ATM algorithm in Joly against a version of my SM algorithm after I had incorporated it into Joly. The results were encouraging for both types of techniques, but mine was chosen as the preferred algorithm. One of the main reasons was that the results it provided,—a percentage of essentially equal lines in two programs,—was much easier to understand for lay people than the vector angles of the ATM algorithm. This is important because the disciplinary committees that handle the cases usually consist of non-programmers, as I discussed in section 2.5 (p. 14).

Lim et al. (2011) include a table shown released systems for both plagiarism detection in plain texts and in program codes. I reproduce the entries related to program codes in table 3.1 (p. 21).

System Name	Type	Base Method	Usage	Cost
YAP3	SM	Greedy-String-Tiling	Stand-alone	Free
Clonechecker	(?)	(Unknown)	Stand-alone	Commercial
MOSS	ATM	Winnowing	Web service	Free
JPlag	SM	Greedy-String-Tiling	Web service	Free
SID	SM	Data Compression	Web service	Free
SIM	SM	Local Alignment	Stand-alone	Free
CodeMatch	Hybrid	Fingerprint	Stand-alone	Commercial

Table 3.1: Plagiarism detection tools for program codes. All entries except the "Type" column are taken from Lim et al. (2011).

3.2 Where my algorithm fits in

My algorithm is, as mentioned above, of the second type (a structure metric or SM). Some of its advantages and disadvantages as compared to the other types can be summarized as follows:

- **Compared to ATMs** it is slower and more difficult to implement, but appears to give better results, and is fast enough for our current uses.

- **Compared to other SMs** described in the literature and table 3.1 (p. 21), my algorithm is, as I will suggest in the next chapter, easier to implement, faster (has lower overall complexity), and still appears to assign high similarity scores given most of the plagiarism strategies that seem to occur in the courses in which it has been used. However, it's likely the results can be improved by borrowing some ideas from other systems.
- **Compared to grammar-based** systems it is several orders of magnitude faster, which makes it much more convenient to use with the large program sets I have applied it to.

I will come back to these points and provide some evidence for them in the next chapter, and at the end of this thesis I will suggest possible improvements to the algorithm.

3.3 Other approaches

Another type of approach that has been reported in the literature and found promising is the application of special tools during the time when the students initially write their programs, like having them use customized editors that make the copying process more labour-intensive, or that store extra information, normally hidden from the students, that record document manipulations they have done (Vamplew and Dermoudy, 2005), (Vogts, 2009). One such system is APE/Gorilla, which I mentioned in section 2.6 (p. 14).

The system I present in this thesis doesn't currently employ these techniques, and is concerned with detecting suspected plagiarism based solely on the finally delivered files.

3.4 Open source vs. closed source algorithms

Even though a number of plagiarism detection systems exist that can be used free of charge, the systems are mostly closed-source. As Freire and Cebrian (2008) explains:

> Many systems lack comprehensive documentation on their internal logic and on the algorithms they use. Some authors oppose public access to this information, since this could lead to circumvention of these algorithms by cheaters. This can be considered akin to "security by obscurity". [...] [F]ew systems have [...] opened up their source code to external scrutiny.

The article goes on to suggest that the typical closed, centralized development model is nevertheless "unlikely to provide the necessary flexibility", for example in cases when students discover that some types of plagiarism are more likely to be caught than others (Freire and Cebrian, 2008). One problem is that most of the released systems seem to have some weaknesses. "Even though all of them have been shown effective in locating plagiarism [...] little correlation between them has been observed." (Freire and Cebrian, 2008) The similarity rankings produced by MOSS and JPLAG being clear examples (Culwin et al., 2001). Freire and Cebrian (2008) is therefore developing his "AC" system as an open-source tool.

At Ifi, Joly is currently a closed-source system. **Devilry** on the other hand is open source, and released under the **BSD license**. Antics is intended to be easy to integrate with Devilry, at least in-house at Ifi. A full integration that allows packaging Antics together with the Devilry distribution for use at other institutions would however require changing some code in Antics.

As I mentioned in section 4.4.3 (p. 29), there is one part of the PRD algorithm that is currently **GPL**-licensed, the Java-based Diff library it uses. This license doesn't fit well with Devilry's BSD license,—combining code from the two would normally require making the whole combined

package GPL-licensed. The Diff code should therefore be replaced with another version or another algorithm before Antics code can be released together with Devilry under the BSD license. This is not a problem at this time since the PRD alogrithm and Antics are only expected to be used in-house at Ifi initially, and such use is allowed under the GPL license without requiring release of the source code. It is also possible to release Antics with its current Diff library if the combination is given a GPL license. Also, there are LGPL versions of Diff for Java that can be used in Antics if one wants to release it without the source code.

Another problem with releasing Antics at this point is that it has some **weaknesses** that could be exploited by students if they or their friends found out about them. An open source process could help plug some of these holes, though not necessarily all:

> Even though all of [the algorithms proposed in the literature] have been shown effective in locating plagiarism (although in a rather unstructured experiments) little correlation between them has been observed. The similarity rankings produced by MOSS and JPlag are clear examples of this (Culwin et al., 2001, p. 14), (Freire and Cebrian, 2008).

> It is understood that no detection system is fool proof. Despite that[,] an automatic plagiarism detection system can act as an effective deterrent to reduce plagiarism considerably in large introductory programming courses (Kar, 2000).

Chapter 4

Design of the PRD algorithm

In this chapter I review the design decisions that went into making the PRD algorithm. I named it for the main conceptual stages it performs: Preprocess-Reorder-Diff.

4.1 Cheating strategies

One of the first things to consider when designing an algorithm for detecting similarities is what types of techniques the students use to disguise their plagiarized code.

The following is a well-known list of such stragegies, taken from Whale (1990), "in increasing order of sophistication":

S1 Changing comments or formatting.

S2 Changing identifier [names].

S3 Changing the order of operands in expressions.

S4 Changing data types. [e.g. int to double]

S5 Replacing expressions by equivalents.

S6 Adding redundant statements or variables.

S7 Changing the order of independent [modules].

S8 Changing the structure of iteration statements.

S9 Changing the structure of selection statements.

S10 Replacing procedure calls by the procedure body.

S11 Introducing non-structured statements.

S12 Combined original and copied program fragments.

I would add: **[S1b]** Changing printing statements (both those used when requesting input from the user, and output).

The first cheating strategy (S1) is the easiest to do for students, they introduce, change, or delete comments, or change the formatting of the code (indentation and other spacing). These techniques can make the programs look very different, even though they don't have any effect on

how the program works. The next strategies, S1b—changing the contents of String literals,—and S2,—changing the names of identifiers,—are also very easy to do and very common in plagiarized code. In fact, many of the plagiarism cases I have seen involve only these strategies, as noted by Kar (2000). These cases are easy to uncover using both simple ATM algorithms and simple SM algorithms because both the operators, operands, and order of statements and modules is unchanged.

The other techniques also generally preserve most of the original code and only add to it, reorder modules, or make relatively small changes to certain lines of the code. These changes can be more difficult to detect because the numbers of operators and order of statements and modules can be very different.

4.2 Summary of the PRD algorithm

The idea I had for my similarity detection algorithm was that if it removed insignificant program text (comments, S1); standardized the names of identifiers (S2), String literals (S1b), and layout (S1) to a common form; and reordered independent modules (S7) to a common order, then almost any two programs would now either look very similar if they had a common source, and if not they would probably look very different, regardless of the other plagiarism strategies (S3–S6, S8–S12). These other strategies would mostly only produce changes in certain lines of code, without reducing the overall structural similarity too much, assuming the program is not too short. They also require more effort from the student, are encountered much less frequently (Kar, 2000), and after a certain threshold the differences would make it very difficult to prove plagiarism anyway (see section 2.5 (p. 14)) and the student would have had to learn a good deal just to apply them effectively.

And that is what my algorithm does. It can be thought of as a three-step process.

1. **Preprocess:** It reduces each program to its essential structure in a standardized way, then
2. **Reorder:** reorders independent modules to a common order, and finally
3. **Diff:** compares the resulting, "preprocessed program skeletons" to each other using a version of the Unix "diff" tool programmed in Java, refined with some extra logic that compares surrounding characters instead of only whole lines.

The reason this technique works is that there are almost an **infinite number of ways** to solve even simple programming tasks. The students seldom "appreciate the [wide] range of implementation possibilities" (Whale, 1990), but a teacher that grades programs soon learns this. An example given by my supervisor is a problem like "sort three values in ascending order". The variety of solutions that different programmers come up with to solve this is amazing. You can use if-else sentences arranged in many different ways, or switch statements, or arrays, maybe with for or while loops, or methods, or do some of the logic with boolean operands, or use the ternary operator, or various combinations of these. Even students chosing the same general strategy still end up with different-looking code. And so in general, the longer the program and more complicated the assignment the more infinitesimal the probability of two students structuring their whole solution the same way.

The result of the comparison is a percentage **similarity score**, from 0 to 100%, indicating the amount of essentially the same code that the two programs share. The algorithm also uses a few other tricks to minimize the effect of some of the other plagiarism techniques, and to speed up the processing, which I will explain in more detail in section 4.4 (p. 28). The percentage scores are computed from the number of common characters found by the Diff stage in the preprocessed program skeletons, and are normalized so that the score of program A compared to program B is the same as that of program B compared to program A.

As it turns out, this scheme is very similar to **other structure-metric** (SM) algorithms, so there is limited novelty in the approch. I will explain some of the differences as compared to the algorithms used in YAP and YAP3 in section 4.5 (p. 30), which are very similar.

The results of **applying this algorithm** in the three courses where I have used it has been very good, the programs pointed out by the algorithm as having the highest suspicion scores usually turn out to be clear cases of plagiarism (practically everything above 80–85% similarity score). Then follow results with varying degrees of plagiarism (70–80% score), usually with the clearest cases getting the higher scores. Below a certain threshold, the cases begin to look less and less obvious. Depending on the complexity of the assignment and the size of the resulting programs, a good threshold for cases that should be checked manually appears to be around 67% score (indicating about 2/3 very similar code even after attempted obfuscation) in INF1000. Most of the programs getting these scores will be suspect. In more advanced courses where the students probably cheat less, and those who do maybe employ more advanced cheating strategies the threshold may be around 60%. This will be illustrated graphically in figure 7.1 (p. 60) in the "Results" chapter.

The majority of student submissions nomally get even lower scores, below about 50% similarity, and above 30%. After all, the programs are solutions to the same assignment, so some similarity is expected. Fortunately there are relatively small number of student submissions that get scores above 50%, and there appears to be a good correlation between the score produced by the algorithm and how clear the cases are after double-checking manually.

Sometimes a few suspects get a little lower scores than they should (in the 60–70% range). These are still normally pointed out by the algorithm as having higher than normal scores, but it can require some work to sift through these if there are many more clear cases. A good strategy is to start at the top of the result list and continue down as far as resources allow.

The more code the students are expected to write and the more room for differing solutions, the better the results from the algorithm appear to be. When there are many possible ways to solve the assignments, usually two solutions will only be very similar through plagiarism.

4.3 Selecting a programming language

The comparison stage is most critical for running time, since hundreds of thousands or even millions of comparisons may need to be run in a single assignment, as pointed out in section 2.3 (p. 12), and each comparison may need to call the "diff" algorithm, and do some other manipulations of the files or programs being compared and the results.

Anything non-trivial that will run that many times could have an impact on running time, so a reasonably fast programming language is a good choice, like C, C++, or Java. Other parts of the algorithm, that only need to be run once for each submitted solution, or that handle the slower interaction with users are on the other hand suitable for a larger variety of programming languages, including the ones mentioned above and scripting languages.

For the time-critical parts of the algorithm, I decided to use Java, because it is fast, portable, and well known by the people who likely will be using it in the next few years at the department. C and C++ could have improved the speed at the cost of portability, and likely making it more difficult for some of the users in these courses. Luckily, the speed of the Java version turned out to be more than good enough for the purposes that the algorithm has been used so far.

The choices of programming languages for the Antics application are a different story, as I will discuss later in section 5.2.1 (p. 35).

Figure 4.1 Two very similar programs showing low level plagiarism techniques.

Program 1 (original)

```
import easyIO.*;
//  Oblig2 [...]

class Olje{
    In tast = new In ();
    Out skjerm= new Out();
    String  [][] eier = new String [10] [15];
    int [][] utvunnetOlje = new int [10][15];

    [...]

    // Deloppgave 4:
    void oppdaterOljeutvinning () {

        for (int i = 0; i< eier.length; i++ ) {
            for(int j = 0; j<eier[i].length; j++){

                if (eier[i][j] != null) {

                    System.out.print(" Antall fat utvunnet i felt "+ i
                      +  "-" + j + " de siste 6 mnd er ( tidligere "
                      +  utvunnetOlje[i][j] + " total fat) : ");
                    int fat = tast.inInt();
                    utvunnetOlje[i][j] += fat;

                }

            }
        }
    }

    // Deloppgave 5:

    void finnRadenMedHoystOljeutvinning ()  {
        int maks = 0;
        int rad = 0;
        int sum;

        for (int i = 0; i< eier.length; i++ ) {
            sum =0;

            for(int j = 0; j<eier[i].length; j++){
                sum+= utvunnetOlje[i][j];
                    }
            if(sum > maks) {
                maks = sum;
                rad = i;
            }
        }
        System.out.print(" Rad med høyest oljeutvinning er rad: "
          +  rad );
 }
[...]
```

Program 2 (original)

```
import easyIO.*;
// Oblig 2 [...]
//Jeg lasta inn programmskissen som ble gitt i oppgaveteksten og
//bygde videre fra den.
class Olje {
    In tast = new In();
    Out skjerm = new Out();

    String[][]eier= new String[10][15];
    int[][]oljeutvinnet= new int[10][15];

    [...]

    void finnRadenMedHoystOljeutvinning() {
        //Går gjennom radene for å finne hvilken rad som har
        //høyest utvinning
        int rad= 0;
        int sum= 0;
        int temp = 0;
        for(int i=0; i<oljeutvinnet.length;i++){
            for(int k=0; k<oljeutvinnet[i].length;k++){
               temp +=oljeutvinnet[i][k];
            }
            if(temp>sum){
               sum=temp;
               rad=i;
            }
        }
        System.out.println("Rad med høyest oljeutvinning: rad " +rad
          + "(" +sum +" fat)");
    }

    void oppdaterOljeutvinning() {
        //Bruker samme for løkke som i forrige oppgave, og printer
        //ut teksten om utvunnet olje. Så spør jeg bruker om hvor
        //mange fat som ble utvunnet.
        for(int i=0; i<eier.length;i++){
            for(int k=0; k<eier[i].length;k++){
                if(eier[i][k]!=null){
                    skjerm.out("Antall fat utvunnet i felt " + i +
                      "-" + k + " siste 6 mnd. (tidligere total "
                      + olkeutvinnet[i][k] + " fat): ");
                    int fat= tast.inInt();
                    oljeutvinnet[i][k] += fat;

                }
            }
        }
    }

[...]
```

Figure 4.2 Results of preprocessing and reordering the two programs in figure 4.1 (p. 27).

Program 1 (preprocessed & reordered)

```
119 Lines  1563 Chars  2 Classes  7 Methods
 Oblig2 2,8 81 (5-15) 1m:  main 3,6 72 (6-12)
 Olje 10,110 1666 (16-179) 6m:  listeSolgteFelt 13,9 191 (101-111)
 [...]
class C {
        String[][] i1 = new String[10][10];
        int [][] i0 = new int [10][10];
        void i0() {
                for (int i0 = 0; i0 < i1.length; i0 ++) {
                for (int i0 = 0; i0 < i1[i2].length; i0 ++) {
                if (i0[i1][i2] != null) {
                P("t" + i3 + "t" + i4 + "t" + i5[i3][i4] + "t");
                int i0 = i1.inInt();
                i0[i1][i2] += i3;
                }
                }
                }
        }
        void i0() {
                int i0 = 0;
                int i0 = 0;
                int i0;
                for (int i0 = 0; i0 < i1.length; i0 ++) {
                i0 = 0;
                for (int i0 = 0; i0 < i1[i2].length; i0 ++) {
                i0 += i1[i2][i3];
                }
                if (i0 > i1) {
                i0 = i1;
                i0 = i1;
                }
                }
                P("t" + i3);
        }
[...]
```

Program 2 (preprocessed & reordered)

```
126 Lines  1774 Chars  2 Classes  7 Methods
 Oblig2 2,8 81 (4-14) 1m:  main 3,6 72 (5-12)
 Olje 10,117 1891 (15-171) 6m:  listeSolgteFelt 13,9 191 (88-103)
 [...]
class C {
        String[][] i1 = new String[10][10];
        int [][] i0 = new int [10][10];
        void i0() {
                for (int i0 = 0; i0 < i1.length; i0 ++) {
                for (int i0 = 0; i0 < i1[i2].length; i0 ++) {
                if (i0[i1][i2] != null) {
                P("t" + i2 + "t" + i3 + "t" + i4[i2][i3] + "t");
                int i0 = i1.inInt();
                i0[i1][i2] += i3;
                }
                }
                }
        }
        void i0() {
                int i0 = 0;
                int i0 = 0;
                int i0 = 0;
                for (int i0 = 0; i0 < i1.length; i0 ++) {
                for (int i0 = 0; i0 < i1[i2].length; i0 ++) {
                i0 += i1[i2][i3];
                }
                if (i0 > i1) {
                i0 = i1;
                i0 = i1;
                }
                }
                P("t" + i3 + "t" + i4 + "t");
        }
[...]
```

4.4 How the algorithm works

Figure 4.1 (p. 27) shows parts of two very similar programs that were delivered by two students in the course INF1000. I will use these as examples as I explain in the following sections how the PRD algorithm works, in each of the conceptual stages: Preprocess-Reorder-Diff.

4.4.1 Preprocess stage

At first sight, the two program snippets in Figure 4.1 (p. 27) look different, but on closer scrutiny we can see that the two methods are only reversed in order and that otherwise the program logic and structure is almost identical.

In order to make a tool that can identify these similarities, my algorithm first converts the two programs to the "preprocessed" versions shown in figure 4.2 (p. 27). This stage only needs to be done once for each program, and the results can be saved in the database for reuse when comparing to all other programs.

During the preprocessing, these transformations are performed, many of them designed so that the Diff phase can recognize essentially equal parts of two programs as such:

1. All comments are removed.
2. All indentation and in-line spacing is changed to a format similar to the Java Code Conventions, except with fewer levels. Mainly three levels are used, for class, method, and method contents. This hels the Diff stage to not lose track when the contents of a method was copied into a slightly deeper block level.
3. All String literals are converted to the single value ''t''.
4. All identifiers are converted to a short, standardized form: All declared classes are renamed C, so that they can be matched later on, and all student-defined method names, variable names, class names inside methods, and other identifiers are renamed i0, i1, i2, and so on depending on their position within the line in which they appear. Repeated instances of the same identifier in the same line get the same name. For example. the line: "`sum = sum + value;`" is converted to "`i1 = i2 + i1;`".
5. Reserved words in the programming language and commonly used classes and methods from the standard library of the language are retained unchanged, except that some near synonyms are replaced with one common name.
6. Printing commands of various kinds are renamed P. Since the algorithm uses numbers of equal characters to estimate similarities, we want printing statements to be short and thus have less impact on similarity scores.
7. Any run of more than three equal consecutive lines is reduced to a maximum of three lines, except equal consecutive printing lines, which are reduced to one line.
8. Some unimportant lines are removed, like "`In tast = new In()`" in figure 4.1 (p. 27), which refers to a package used in INF1000, which can be declared inside or outside methods. Various other tricks have also been used in the different courses, depending on the specifics of the assignments, though these are more difficult to automate.

Some general information about the program is also gathered, including the number of lines and characters (counted after preprocessing, so that these not be affected by long comments, string literals, or variable names). The number of classes and methods are also recorded. These counts will be saved along with the preprocessed version of the program for use, among other

things as cut-offs to avoid comparing very different programs. Also, the starting and ending line of each method and class is recorded, for use later during the reordering phase.

It is also possible to produce other types of summarized metrics computed for each method, class, or for the the entire program for use in later phases, but at the moment the ones mentioned above are the most important.

More technically, in order to accomplish this phase, what is done is a simplified parsing of the program text. The most complicated part is identifying the start and end point of the classes and methods in the program. For most of the other parsing Java's StreamTokenizer class is used. It helps to identify and remove all comments easily, and helps with the parsing of the other tokens in the program, by separating them into a few token types. The algorithm uses this for the reformatting, which needs to find a standard location as end of statements and blocks. This is done by looking for "{", ";" and " }".

One could also introduce a full parsing of Java syntax, but this would make the algorithm slower and much more complicated. The simplified parsing used seems do the job well for the current uses.

4.4.2 Reorder stage

When the whole file has been preprocessed, the reordering stage is done. Here the algorithm tries to determine a standardized order for the classes and methods of the program so that the Diff stage can find most of the code in about the same order if two programs are very similar. The reordering used now is based on the number of characters in each method and class after preprocessing. During reordering enough data is collected about the original starting and ending line of each method and class, and their reordered locations, so that a tool can analyze the results and reconstruct the likely reordering the student did to the program if found to be plagiarized. Some examples of this are shown in figure 4.2 (p. 27). Other metadata about the submission is also saved by the Antics application so that it be clear which student, course, semester, etc., it came from.

4.4.3 Diff stage

The Diff stage is the most time-consuming part of the process, as all program files may need to be compared to all others in a program set, leading to possibly millions of comparisons of pairs of programs (section 2.3 (p. 12)).

When two preprocessed programs are to be compared, the algorithm first checks whether the sizes of the programs are within a certain range in relation to each other. If they are too different (more than double the size or less than half) we skip any further comparison since the result would inevitably be less than 50%. This cutoff can typically reduce the number of comparisons needed by around 20%, depending on the size of the assignment. Larger assignments with more room for varying solutions can give a wider range of answer sizes, permitting more cutoffs. The algorithm also checks that the program to be compared is above a minimum size (very short programs that are not large enough to have solved much of the assignment are probably unnecessary to compare and could give high percentages against other equally as unfinished responses).

Next, a comparison is run with a normal line-wise "diff". The Diff algorithm provides a quick solution to the longest common subsequence problem. I am currently using the GPL-licensed Diff algorithm for Java from `bmsi.com`. It is the only part of the PRD and Antics systems that I am aware of that is GPL-licensed. It has to be changed with another implementation of Diff or another algorithm if Antics is to be released outside Ifi without a GPL license. More on this

in section 3.4 (p. 22). The Diff algorithm currently used is a translation to Java of the GNU Diff algorithm, which uses the basic algorithm is described in Myers (1986).

If this Diff result provides a certain similarity between the programs (depending on the thresholds chosen for this run of the algorithm), then a more detailed comparison is made that checks for character matches in the lines above and below the lines that diff matched. This refinement is only minor, and is only carried on a small number of the comparisons, often less than 1% depending on thresholds chosen and the complexity of the assignment. The result is a small improvement in the final score. Actual plagiarism cases tend to give more matching characters here, while independently developed programs don't give as many extra matches.

The final result of the comparison is the percentage difference in the number of characters that were matched. This is computed from the results of line-wise Diff if that didn't result in a high enough score, and from the line-wise Diff refined with extra character matches for those programs that went on to this stage.

I normalize the result so that the similarity of program A vs. B is equal to B vs. A. When both comparisons are made the normalized results are practically always the same both ways, or within 1% of each other. This indicates that the underlying Diff algorithm has good symmetry (it finds the same matches when taking the Diff of A vs. B as that of B vs. A).

It is also possible to get some information from the unnormalized results, for example, if program A is slightly smaller, but all its contents are in B, then one guess could be that A was 100% taken from B, while B may be considered 80% equal to A, with 20% additional code. However, this is not necessarily the case, the copy could be B, which only added some unnecessary extra code. In my experience, the simplicity and beauty of the single percentage score for both programs adds to the appeal of the PRD algorithm without distracting complications, which could have limited value. The differing results could still be shown to the user in an "advanced view" or "more detailed report".

Another important point: Many students take these courses several times, and they are allowed to reuse their own earlier solutions when a similar assignment is used a different semester. My algorithm takes this into account and avoids giving distracting results about almost identical solutions from the same student delivered in different semesters. This is done by always keeping track of the user name of the student that submitted a solution, since user names are unique at UiO over time.

A cutoff idea that I have not implemented but that could speed things up a bit is to skip comparisons of slightly different versions of the same assignment delivered by the same student (they can deliver several times and often do), if a student delivered two versions of the same program that are 99% alike it is likely enough to use only one of them for the comparisons with other deliveries.

Drawbacks: A problem with this algorithm can be the **block mismatch problem** discussed in Wise (1996). It can occur when there are a number of big and similarly-sized methods in a class, and some of these methods have been altered with a little more code in some and a little less in others, or similar alterations. This can sometimes confuse the reordering phase, and if that reordered the methods the wrong way (not the same way as another student's similar program), some whole blocks of code may be missed by the Diff algorithm. This problem has been investigated and can be reduced by using Greedy-String-Tiling, or Local Alignment as discussed in Lim et al. (2011) and Wise (1996).

4.5 Compared to YAP3

As it turns out, my PRD algorithm is very similar to other structure metric algorithms, particularly the algorithms in the YAP, YAP2, and YAP3 systems. These systems in turn are very similar

to each other. The main difference between them is the underlying Diff-type algorithm they use (Wise, 1996). They use similar techniques to preprocess the programs, they reorder the program codes, however in a different way, and then apply a Diff-like algorithm.

YAP uses sdiff (which gives the same results as Diff, only formatted in a slightly different way, it's only a postprocessor for Diff). YAP2 uses Heckel's algorithm, which can match reordered code better in some cases but less well in others, particularly if extra print statements are introduced in one of the programs (Wise, 1996). YAP3 uses a more advanced algorithm for matching the possibly transposed code blocks, Running-Karp-Rabin Greedy Tiling (RKR-GST). This algorithm is slower but gives better results than Diff (Wise, 1996).

The differences between the YAPs and the PRD algorithm are not very large, but include:

- My algorithm is applied to Java programs, and parses and reorganises the modules (classes and methods) in this language. YAP3 appears to have been designed for C initially, though its ideas could equally well be applied to Java.
- **Simpler and faster** to run than YAP3, since the PRD algorithm uses a version of Diff which runs in linear time, instead of YAP3s Running-Karp-Rabin Greedy Tiling (RKR-GST) This is not necessarily best in all cases. In The reordering is simpler also. For entry level courses it is often enough to detect the even simpler plagiarism techniques (Kar, 2000), and my algorithm appears to work well, even though some blocks of reordered code can be missed by the Diff tool my algorithm currently uses. The programs in question still often get relatively high similarity scores, but these can be missed by busy TAs that can't check the whole list.
- My algorithm currently doesn't determine the **static execution sequence** of methods in the source codes, as YAP3 does; it uses a different scheme for reordering the token streams into a hopefully common form to use during the comparison phase. The result is not necessarily better performance in identifying plagiarism, though it is simpler and likely faster at least. For comparing program sets involving more advanced plagiarism strategies the YAP3 approach is probably better.
- **Identifiers** are not removed entirely, as the YAP3 paper seems to suggest they possibly do. This would reduce the structure information in the preprocessed versions. Instead a simple renaming scheme is applied to them, and the number and positions of identifiers are retained and used for comparisons.
- Preprocessed program lines remain as **strings**, and I use the character length of strings matched to compute the final similarity percentage. YAP3 appears to discard some of this information since it transforms the tokens to numbers instead of keeping the strings. I think that taking into account the varying lengths of preprocessed lines depending on the lexicon of the programming language, can help produce a more accurate program structure similarity score, but I haven't tried the numeric approach yet.
- **Indentation levels** are treated especially to allow effective use of diff that help match methods and function starting points, yet avoids Diff's getting lost too easily if the student has made minor changes that add an indentention level of similar code within methods.
- The Diff algorithm doesn't get totally lost in the presence of extra print statements, or **decomposition of compound assignment** statements into several simpler ones (for example `int a, b, c;` changed to `int a; int b; int c;`). These can often just become an unmatched line beween otherwise matched content, and the character-by-character after-matching step I use can help find a little similarity in the beginning of these short lines. Depending on the size of the total program these differences will likely have a small effect.
- I have developed solutions to the surrounding **practical problems** encountered when applying these tools, regarding unpacking and reassembling numerous souce code files in

single deliveries to a form easier to use in detection tools, usable in course assignments with techer-supplied precode, removing test files and small files from the programs to be compared, formatting everything together for whole-solution matching

4.6 The latest improvements

Compared to the version of the PRD algorithm I have used "manually" the last few years, the most important improvement (see task 1.(h) (algorithm improvement) (p. 8)) done during this thesis is probably the doubling of speed on current-semester comparisons because of the better **management of metadata** about the deliveries. Earlier I just let it run all comparisons both ways, program A compared to B, and B to A. By just keeping track in the Java algorithm of what student solutions have been compared the algorithm can now avoid comparing B to A.

Another improvement is changing **number literals** to a reduced set of possibilities in the Preprocess stage. All numerical constants in the programs are now changed down to the nearest multiple of 10 (1–9 become 1, 10–99 get 10, and so on), but everything above 1000 is left as 1000 to avoid giving too much weight to these literals when characters counts are used to compute the similarity scores (section 4.4.3 (p. 29)). Negative numbers are treated similarly. This will help give a little better results in the assignments where the only change from the previous year is in some number constants, as we have done many times. I noticed the algorithm gave slightly smaller similarity scores in these cases just because the numerical constants didn't match, though the difference is not very large and plagiarism cases where still clearly identified.

Chapter 5

Design of the Antics application

In this chapter I describe the design of a new stand-alone application for plagiarism detection that I created during my work with this thesis, called Antics (Anti-Cheating System for source code assignments). This application is a kind of wrapper around the PRD algorithm presented in the previous chapter.

5.1 Software requirements

The requirements for the Antics application are essentially to fulfill the main points presented in the **research question**, section 1.1 (p. 8). Several of the tasks allow room for evaluating what is feasable within the time frame of this short master's thesis.

The most important goal for the application is to make it **easy to use** the PRD algorithm in actual university courses, in the form of a stand-alone system that can help organize many student solutions, and apply the plagiarism detection algorithm to these. The **intended users** of Antics are just the teachers and TAs in these courses. A typical use case would be a lecturer or TA that wants to check all student submissions in a current assignment for plagiarism. The student submissions must be collected using other systems, though another goal is to be able to integrate Antics into such larger systems, like Devilry.

The application should help selecting which student solutions will be compared with others; automate the running of the comparisons for selected courses, semesters, and assignments; keep track of the relevant information about each student solution so that the system can show this information together with the results of the comparisons.

The system will need to organize information about which students delivered each of all the pairs of solutions that are found to be very similar, when these were delivered, what the similarity score was, and so on. From my experience with using the algorithm without such a supporting application, I think it would be useful if the application allows importing solutions in many different file formats that are used in the relevant courses, including single or multiple source code files, or delivered as compressed archives containing single or multiple source code files or directories with these, arranged in various folder structures, which can be intermixed with various file types, not just source code files. Antics should also keep track of the various versions of the solutions that students delivered in the same assignment.

To summarize, these are some of the **main goals** for the application:

- Easy to use.
- Easy to install.

- Easy to integrate with Devilry.
- Allows importing student solutions in various file formats.
- Allows importing and exporting solutions to and from the Joly database.
- Allows comparing selected student solutions, giving similarity percentages.
- If possible, sort the similarity results both by top scores and by correcter.
- Collects the actual student-submitted files that gave high scores in an easy to use package.

In the following sections I discuss the various choices of technologies I considered for Antics.

5.2 Selecting technologies

The main system that Antics needs to be able to interact with is the Joly database (section 2.7.1 (p. 16)), which is a MySQL database that runs in a server at Ifi. It could also be fine if Antics was fully usable as a stand-alone system without connecting to the Joly database, though this was not a requirement. Another design goal was that it be possible to integrate Antics with Devilry (section 2.7.3 (p. 17)), an open-source web application written in Python using the Django framework.

This combination of MySQL-access, ease of integration with a Django web application, while also remaining a stand-alone application easily installed and used by itself, and being possible to develop in a short period of time, meant that selecting suitable technologies was an important task. Table 5.1 (p. 34) shows an overview of the technologies used in the Joly application, Devilry, and Antics.

Technologies	Joly	Devilry	Antics
Programming language	Java	Python	Python & Java
Programming framework	Spring	Django	Django
Database	MySQL	(Supports several)	SQLite, MySQL
File system storage	(No)	(Yes)	(Partial)
ORM	Hibernate	Django ORM	Django ORM
Web server	Jetty	Apache, and others	Django runserver
Web interface	JavaServer Pages	Django templates	Django templates
Authentication	Non-encr. passw.	Auth. plugin	Non-encr. passw.
License	Closed-source	BSD license	(Not release-ready)

Table 5.1: Main technologies used in the Joly application, Devilry, and Antics.

Both Joly and Devilry were programmed in **object-oriented** languages, Joly in Java, and Devilry in Python. Both use **relational databases** for storing information about the student deliveries. Joly uses a MySQL database, while Devilry is designed to be compatible with several database backends, including MySQL, PostgreSQL, Oracle, and SQLite. The current deployment of Devilry at Ifi stores the actual student deliveries in the file system, but uses a database for managing these and other metadata. This database is by default SQLite in the developers' distribution of Devilry.

The combination of object-oriented language and relational database lead both Joly and Devilry to use **object-relational mapping** (ORM) for converting the inherently incompatible type systems of object-oriented languages and relational storage. The ORM used in Joly is the Hibernate library, while Devilry was developed using Django's object-relational mapper. Django is an open source web application framework written in Python, that adheres to various well-known

design patterns including the model-view-controller (MVC) pattern. It comes with an ORM that translates between Python classes and its relational database model. This is programmed using Python classes containing special data types designed for this interaction. It has a simplified query language for ease of access to various databases, while still allowing normal SQL statements when necessary.

In order to limit the number of technologies that Antics would be based on, a reasonable idea was to employ some of the technologies already present in either Joly or Devilry, and avoid introducing too many new ones. A good starting point for Antics could therefore be an object-oriented approach with a suitable ORM, so that the data model be close to that of either the Joly database or Devilry. Antics will need to manage the same type of data that both Joly and Devilry already manage with databases, including courses, semesters, assignments, students, deliveries, and so on, and could therefore benefit from a database.

5.2.1 Programming language

There are several differences between the Joly data model and the Devilry data model. The one in Devilry is much more general and detailed (see figure 5.2 (p. 36), which illustrates one of the more that 20 Django apps that Devilry consists of), and includes capabilites beyond the scope of Antics, for managing many types of student submissions and users, and funcitonality for student submission of deliveries, and grading and feedback system for these, anonymous correcting, etc. The Joly on the other hand only consists of 14 tables (which will be illustrated later in figure 5.1 (p. 43) and figure 5.3 (p. 45)).

For Antics a small subset of Devilry's data model could be enough. One problem is that this subset has various **inconsistencies** with how similar data is organized in the Joly database. If I just used the data model from the Joly database, the inconsistencies could also make it more difficult to integrate Antics with Devilry. But if I just copied the more complex data model from Devilry I would have had both lots of possibly unnecessary complexity, and some extra work translating all the relevant data from Joly to this model. The application might not end up being as easy to install and small as desired.

I considered whether to use a **Java ORM** or the **Django ORM**. The easiest of these to use and program against appeared to be the second one, and it had the advantage of making it easier to adapt Antics to Devilry later, since both would be using the same technologies. A Java ORM could also be a solution, and it could have an advantage with respect to running speed, since Java is generally faster than Python. If I chose a pure Java solution I would probably need to include both a database backend and a JDBC driver with Antics if I wanted it to work out-of-the-box as a stand-alone application. On the other hand, since version 2.5, Python now comes with a good embedded database backend, SQLite, which could be suitable for Antics—it is "the most widely deployed SQL database engine in the world" according to its home page.

The Django ORM even extends this capability by providing a query language that makes it easy to select among several database backends without changing the application code, by only changing a few settings in one file. Antics could therefore include a ready-to-use SQLite database while allowing the user to select another database if preferred, if I used Python and Django. Django also offered an attractive framework for developing the user interface of Antics, and included a database introspection tool that could help connecting Antics to the Joly database.

The **main problem with Python** and Django, was that they would be slower than languages like Java, C, or C++ for the large amount of computation that Antics needs to do for plagiarism detection, as suggested in section 2.3 (p. 12). On the other hand, a big advantage if I chose Django was that it could potentially make it easier and faster for me to complete Antics. According to the Django home page it is "The Web framework for perfectionists with deadlines. Django makes it easier to build better Web apps more quickly and with less code" (Django, 2012).

Table 5.2: All 28 classes in the Devilry core data model, and their superclasses in parentheses.

All classes in the Devilry core app
AbstractApplicationKeyValue (models.Model)
AbstractIsAdmin (object)
AbstractIsCandidate (object)
AbstractIsExaminer (object)
Assignment (models.Model, BaseNode, AbstractIsExaminer, AbstractIsCandidate, Etag)
AssignmentGroup (models.Model, AbstractIsAdmin, AbstractIsExaminer, Etag)
AssignmentGroupTag (models.Model)
BaseNode (AbstractIsAdmin, SaveInterface)
Candidate (models.Model, Etag, AbstractIsAdmin)
ShortNameField (models.SlugField)
LongNameField (models.CharField)
Deadline (models.Model, AbstractIsAdmin, AbstractIsExaminer, AbstractIsCandidate)
Delivery (models.Model, AbstractIsAdmin, AbstractIsCandidate, AbstractIsExaminer)
DevilryUserProfile (models.Model)
Examiner (models.Model, AbstractIsAdmin)
FileMeta (models.Model, AbstractIsAdmin, AbstractIsExaminer, AbstractIsCandidate)
EtagMismatchException (Exception)
Etag (object)
Node (models.Model, BaseNode, Etag)
Period (models.Model, BaseNode, AbstractIsExaminer, AbstractIsCandidate, Etag)
PeriodApplicationKeyValue (AbstractApplicationKeyValue, AbstractIsAdmin)
RelatedUserBase (models.Model, AbstractIsAdmin)
RelatedExaminer (RelatedUserBase)
RelatedStudent (RelatedUserBase)
RelatedStudentKeyValue (AbstractApplicationKeyValue, AbstractIsAdmin)
SaveInterface (object)
StaticFeedback (models.Model, AbstractIsAdmin, AbstractIsExaminer, AbstractIsCandidate)
Subject (models.Model, BaseNode, AbstractIsExaminer, AbstractIsCandidate, Etag)

Because of these reasons I decided to take the **best of both worlds**. I implemented most of Antics in **Python** using the Django framework, including an internal database and the code to communicate with the Joly database, while I programmed the PRD algorithm in **Java**, since it was the most critical part of the system regarding running time. The Python code calls the Java algorithm directly when necessary, as I will cover in section 5.4 (p. 46).

5.2.2 Object-relational mapping

Regarding the **data model** for Antics, I also decided to use a combined solution for the internal database in Antics, consisting of many of the tables from the Joly database, but translated into a model nearer what Devilry uses, and adopting the terminology of Devilry.

I selected **Django's ORM**, like Devilry, and created two data models, one that emulates Devilry and one that communicates with the Joly database:

Antics core data model: This is the main model containing all the classes used normally by Antics, except only when importing or exporting to the Joly database. It is intended as a subset of the Devilry core model, reflecting the data I needed for Antics that Devilry also uses, but simplified, and with some additional classes and attributes related to plagiarism detection.

Antics' model of Joly: I also created an additional model representing a subset of the Joly database. This model is only used to import and export data to the Joly database. The Django ORM requires making such a model with classes and fields that represent the external database one wants to connect to (the Joly database in this case).

It was necessary to create at least two data models if I wanted to use Django's database synchronization tools because they don't allow synchronizing part of a model to a database, each Django ORM data model is normally synched completely to a database, and Django "protests loudly" if there is any class or attribute in the model that is not also present in the corresponding database. The model can have fewer classes than the database, but not more.

So one of my models could only have classes already present in the Joly database. And since I wanted to persist some additional information related to plagiarism detection, and use classes and attributes like those in Devilry that are not in the Joly database, this had to be programmed in a separate model.

5.2.3 System architecture

Having decided to use the Django framework, a natural choice was to follow its architecture guidelines, including the **MVC** (model-view-controller) design pattern. The model, view, and controller parts of the application are separated and located in different directories:

- The **model** is described in two directories: `core`, which contains the main data model for Antics, and `jolydb`, which contains the emulated model of the Joly database. I programmed these models using the Django ORM, with Python classes (which I illustrate later in table 5.3 (p. 39)) that represent the tables in the two databases. The attributes in these classes represent the columns in the databases, and the objects that are created of the classes are saved as the data in rows of the databases.
- The **view** section is located in the `templates` directory, and contains web pages and forms written in the Django template language (which is a combination of HTML with simplified Python-like directives).
- The **controller** code is located in the root `antics` directory and is written in Python. It is further separated into several files, including: `urls.py`, which controls which Python functions are called when certain URLs are accessed by the user; and `views.py`, which contains the code of these functions.

Antics can also be described as a **distributed application**, based on the client-server model. A web browser (the client) is needed to access the user interface, while another process runs the application code in the small Django development web server (the server). It is also possible to adapt Antics to run on any other web server that implements the WSGI interface (Web Server Gateway Interface), which runs Python code, but since Antics should be considered as a work in progress, it is still using the development web server in Django.

This server is not intended for use by many clients at the same time, and has some overhead because of its debugging features, but a few users should be handled ok, which is enough for Antics at the moment. It can also help if the teachers using it want to make changes.

Most of the code currently runs on the **server side**, and the client just shows the dynamically generated web pages or forms sent by the server, and sends requests when the user fills out a form or clicks on a Web page link.

Another way of characterizing the Antics architecture is as as an **event-driven** architecture. I will cover the main events in section 6.1 (p. 49) in the next chapter, but I mention the menu

choices here: Import files, Import from Joly, Export to Joly, Detect cheating, Antics database, and Configure.

In addition to the directories and files related to the MVC pattern mentioned above, there are some other related files and directories in the Antics package, including:

- The file **settings.py** contains the most important user-editable settings for the application.
- The default internal **SQLite database**, which is implemented in a single file in the root directory of Antics, and has the same filename as the corresponding default Devilry database for development, `db.sqlite3`. This database has no password or user authentication, and can be inspected and edited easily using the `sqlite3` client application.
- The **filedata** directory is where the file system versions of preprocessed files and `.java` student solutions are located during plagiarism checking.
- The **prd** directory currently contains the PRD algorithm programmed in Java.
- The simple password infrastructure is implemented using the Django app **password_required**, located in the directory of the same name. This is a small app with BSD license that I bundled with Antics.
- Another useful utility function I added was **magic.py**, which helps to identify file types (for example whether it is a Zip archive, Tar file, text file, etc.). Python had a built-in tool for file type identification but it appears to be merely based on the file extension (the last part of the file name), and is thus probably not reliable. It is easy to rename a file with any file extension even if the file is a completely different type. This tool is necessary for instance if one is to import student submissions that were downloaded with Download.cgi (section 2.7.2 (p. 17)), since it often names the delivered files with the wrong file extension. That is where `magic.py` comes in, and helps determine more reliably the file type. Program codes are actually identified as "text files", but this is a good start, and Antics can easily add a lookup into the file contents when it knows it is a text file, and if it finds typical Java syntax (comments, package, import, a class or a Java interface) it can assume it is a Java file and run it through the palgiarism detector. It can help identify the actual language of student submissions in Devilry if Antics is built into or connected to Devilry (see task 2.(l) (language identification) (p. 9)).

5.3 Data model

Since Antics is programmed in **two object-oriented** languages (Python and Java), but communicates with two relational databases, it needs object-relational mapping (ORM). I selected **Django's ORM** as mentioned above. This maps each Python class in the Antics data model to a table in the relational database; class attributes are saved as columns in the respective tables; and objects of a class are represented by rows of the table. Table 5.3 (p. 39) shows an overview of the tables in the Joly database, the **classes** in the Devilry core data model, and how they were translated into the Antics model of Joly, and Antics' own core data model.

All the internal data about courses, semesters, assignments, students, deliveries, and so on is persisted in **Antics' internal database**. The communication with the Joly database is completely optional, and is done only when the user asks to import or export data from or to the Joly database. At that point the user can select (as I illustrate later in figure 6.1 (p. 50)) which specific courses and semesters he wants to import or export, and these are then transferred to or from the Antics' internal database.

Whenever the **plagiarism detection** function in Antics is run, it uses only the data that has been specifically imported earlier, either from Joly, or from local files or directories, instead of accessing the Joly database behind the scenes. The user can get an overview of the contents of the

Joly database model	**Antics' Joly model**	**Antics core app**	**Devilry core app**
–	–	–	Node, BaseNode
Course	Course	Subject	Subject
(Part of Course)	(Part of Course)	Period	Period
Coursegroup	Coursegroup	(examiners)	(examiners)
Assignment	Assignment	Assignment	Assignment
(Part of Assignment)	(Part of Assignment)	(Part of Assignment)	Deadline
Studentsolution	Studentsolution	Delivery	Delivery
Group_Student (link table)	(*)	(examiners)	(examiners)
Languagetype	Languagetype	FileType	–
Student	Student	(Student)	AssignmentGroup, Candidate, DevilryUserProfile
Employee	–	–	(DevilryUserProfile), Examiner
–	–	FileObject, DeliveredFile, ExtractedFile	FileMeta
Employee_Course (link table)	(*)	–	(DevilryUserProfile)
Comparison (SM)	–	Comparison (SM),	–
Preprocessed (SM)	–	Preprocessed (SM)	–
–	–	Assignments-Checked (SM)	–
Elementoccurrence (empty), (ATM)	–	–	–
Elementtemplate (ATM)	–	–	–
Reportcase (empty) (ATM)	–	–	–
–	–	–	StaticFeedback

Table 5.3: Main classes (and database tables) in the data models of Joly, Antics' model of Joly, Antics core, and Devilry core. Abstract and utility classes not shown. Classes and attributes shown in parentheses are not exact matches, the differences are explained in the text. (*): Can't be used through the Django query language due to multi-column primary key.

Antics database at any time (as I illustrate later in figure 6.3 (p. 52)). Deliveries imported from the Joly database are marked as such interanlly,—allowing source identification when plagiarism is detected,—and are not imported again if the user asked for it when it was not necessary.

In order for Antics to connect to the Joly database the user needs to provide a suitable Joly password, as I'll explain in section 6.4.1 (p. 54) about security.

Table 5.3 (p. 39) shows an overview of the data models in Joly, Antics, and Devilry:

- **Joly database model:** The left-most column shows all the 14 tables that are currently in the Joly database (the names without parentheses).
- The two middle columns show the two data models built into Antics:
  - **Antics' model of Joly:** The left one is the model that emulates Joly and is used only for importing and exporting data from Joly.

  - **Antics core app:** The middle right column shows the classes in the main internal data model in Antics, which is implemented as a Django app called "core".

- **Devilry core app:** The right-most column shows the main classes in the Devilry "core" app data model. The class names are aligned according to what class they resemble the most in the other models, though there are important differences in all classes. An overview of all the classes in this Devilry core app can be found in figure 5.2 (p. 36).

In addition to this, each class has a number of **attributes** (fields) which also follow the same philosophy. In the Antics' model of Joly these emulate the Joly model, while the Antics core app emulates a subset of the Devilry core app, with some tacked-on extra plagiarism-related fields.

Table 5.4 (p. 41) shows the attributes of three of the main classes in the data model (Course or Subject, Assignment, and Studentsolution or Delivery), illustrating the complexity in harmonizing the fields in the Joly and Devilry models for the purposes of Antics. The extra attributes related to plagiarism detection added by Antics are also shown.

5.3.1 Antics' model of Joly

As can be seen in figure 5.3 (p. 39) and figure 5.4 (p. 41), most of the classes and attributes present in the Joly database are replicated in the Antics' model of Joly. In this section I explain some of the differences.

Since Antics is a stand-alone application intended to be used by one teacher, or maybe just a few teachers, for checking the deliveries in their own course for plagiarism, it was not critical that I include the management of employees that Joly or Devilry have. That is why the class **Employee** is in the left-most column in the first table but not in the two middle columns. The feature in Antics that most resembles the Employee management in Joly is Antics' requirement for a password to access the GUI, as I will describe in more detail in section 6.4.1 (p. 54), but it doesn't keep track of different users at the moment. The current version assumes that each user (or the users in each course) just uses his own copy of Antics.

Regarding the bottom left part of the table, that shows the tables in the Joly database related to the **SM and ATM** algorithms in Joly, I have not implemented importing the data from those classes to Antics. This is why there are no corresponding classes in the column for the Antics model of Joly. Instead of copying this data from Joly, Antics creates its own plagiarism-detection data. The data is similar to that in Joly, but a little updated.

All the other tables from Joly are replicated in the Antics Joly model, except for the two link tables marked "(*)" in the table. Unfortunately, the Django ORM doesn't currently support database tables with **multi-column primary keys**. It requires that all tables in the database,—including link tables,—have a single-column primary key. This primary key column is usually an integer serial counter, but another data type could also be used, as long as it is a single column, meaning a single attribute in the corresponding class, that is unique for the table. The Django ORM adds such an integer field to every class automatically when designing a new data model if one doesn't specify another field as being the primary key, but in this case the Joly database is already defined and in use by anothe application (Joly), so its schema should not be altered.

Fortunately, the result of this limitation is only that I can't include the link tables as tables in the normal model of Joly that Antics uses, but instead I can use **raw SQL** statements to access these tables from Django. This is also possible and is the solution I adopted. In addition, I only need to access one of the tables in question, called Group_Student, which contains the information about which group (TAs section) a student belongs to. This is implemented very differently in Devilry however, as there every single delivery can be associated with any "examiner" (usually a TA), instead of having whole groups (sections) fixed to a TA. I haven't implemented this flexibility in Antics completely yet.

Joly database model	Antics' Joly model	Antics core app	Devilry core app
Course:	**Course:**	**Subject:**	**Subject:**
courseID	courseid	id	id
semester	semester	(periods)	(periods)
courseCode	coursecode	short_name	short_name
courseName	coursename	long_name	long_name
–	–	–	parentnode
(in Employee)	–	–	admins
–	–	–	etag
Assignment:	**Assignment:**	**Assignment:**	**Assignment:**
assignmentID	assignmentid	id	id
assignmentNo	assignmentno	nr	–
–	–	short_name	short_name
assignmentText	–	–	(long_name)
–	–	–	anonymous
–	–	–	scale_points_percent
courseID	courseid	(in period)	(parentnode)
deadline	deadline	deadline	(in assignmentgroups)
–	–	–	publishing_time
–	–	(deliveries)	–
languageID	languageid	(languageid)	(delivery_types)
algorithm	–	–	–
–	–	checkedAgainst	–
limit1	–	(results1)	–
limit2	–	(results2)	–
(in Employee)	–	–	admins
–	–	–	etag
Studentsolution:	**Studentsolution:**	**Delivery:**	**Delivery:**
studentsolutionID	studentsolutionid	id	id
–	–	(jolynr)	(number)
assignmentID	assignmentid	assignment	(assignmentgroup)
text	text	(version), (text)	(filemetas)
bytesize	–	(deliveredfiles), (extractedfiles)	(in filemetas)
–	–	(path)	–
deliverytime	deliverytime	deliverytime	time_of_delivery
–	–	–	deadline
–	–	–	after_deadline
–	–	–	successful
username	username	(username)	(delivered_by)
–	–	sectionnr	–
–	–	examiner	–
–	–	–	feedbacks
–	–	–	etag

Table 5.4: The attributes (or database columns) of three of the main classes (tables) in the data models of Joly, Antics' Joly model, Antics core, and Devilry core. This illustrates some of the complexity of harmonizing the differents data models between Joly and Devilry, and some of the work done in this direction in Antics. Parentheses show inexact matches and foreign keys pointing back to these classes from other classes. Primary keys are underlined.

The other link table, **Employee_Course**, is used in Joly to represent the other many-to-many relationship in the Joly database, linking lecturers and TAs to the courses where they have higher privileges to access Joly than normal students. For the current purposes of Antics I don't need to replicate this data. It consists mainly of lists of former TAs that had their own password in Joly for accessing the solutions submitted by their students. The Antics application should be perfectly usable without copying this authorization data from Joly, by implementing its own security measures, see section 6.4.1 (p. 54).

The rest of the classes and fields in Antics' model of Joly follow the tables in the current Joly database very closely, as required by Django for communication with the Joly database. One minor difference is that Django forces the attribute names in such legacy database tables to be lowercase-only.

To avoid confusion remember that in this thesis I distinguish between the "**Joly**" application and the "Joly database". The requirements for Antics, described in section 1.2 (p. 9), only call for Antics to communicate with the database (both read and write to it), but not necessarily with the Java application "Joly" directly. Since both applications will use the same database, it is important that any changes Antics makes to the database don't cause problems for Joly. In order to ensure this I decided that Antics will not make any change to the schema of the Joly database, and should only fill in data in the same columns that the database has, and in a manner consistent with how Joly would add such information.

5.3.2 Connecting to the Joly database

In order to generate the code for these classes I used Django's database introspection tool ("**inspectdb**"), which is often used when integrating Django code with legacy databases. This gave me a half-ready Django ORM data model, as shown in part in figure 5.2 (p. 44), with all the classes and attributes from the Joly database, and their respective data types, but without the relationships between the classes (foreign keys and many-to-many fields).

I then proceeded to add the **foreign keys** and many-to-many fields. Figure 5.1 (p. 43) shows these relationships in the original Joly database. After that version the database has been expanded with two more classes related to the SM algorithm (Comparison and Preprocessed) shown in the figure 5.3 (p. 45). The latter figure is missing the class Student, but it is still present in the database. The first figure has a class Checkedelement, that was apparently intended for use with an ATM algorithm, but has since been dropped from the database. Two of the tables are empty (Elementoccurrence and Reportcase), but those are related to the ATM algorithm and thus not needed by Antics.

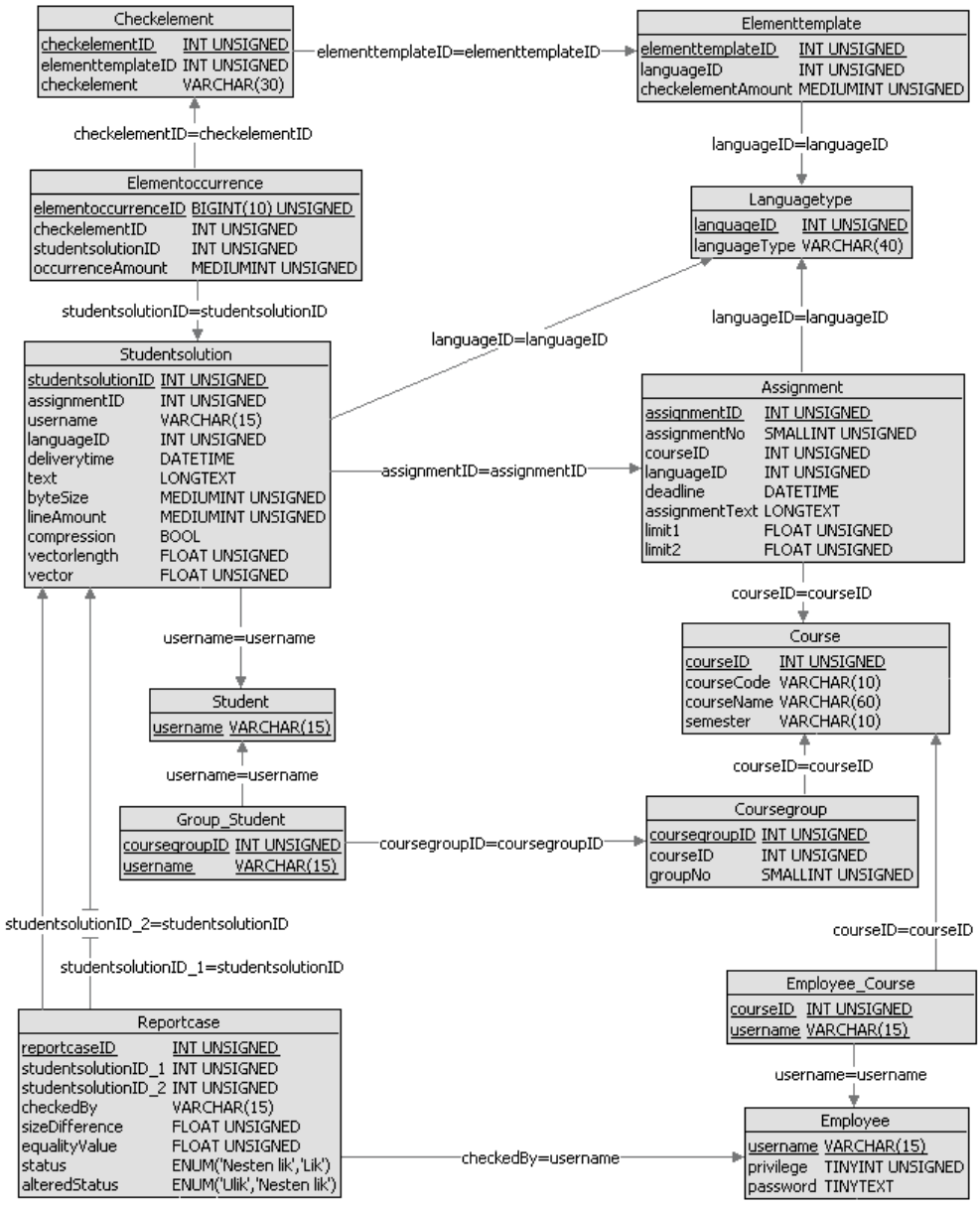
The resulting model was enough to connect the Joly database to my application for **downloading** student solutions from it (and import them into Antics). Table 5.3 (p. 39) and table 5.4 (p. 41) show the Django ORM model I used for this under the heading "Antics' Joly model". The only problem when downloading through this Django ORM model is that it's impossible to get the information about the sections ("groups") the students are part of, because this information is only present in the Django-incompatible link table "Group_Student" mentioned above. The solution is to use raw SQL to get this information out of the Joly database. I haven't implemented this yet (I'm putting all students in "group 1") but I will try to implement it.

For **uploading** to the Joly database there was also a similar problem with the multi-column primary key of the same table ("Group_Student"), requiring the same type of solution, as I also described above in section 5.3.1 (p. 40).

In order to get an idea of the data that Antics would be importing from the Joly database I made an exact **count of the rows** in each of the tables, as shown in table 5.5 (p. 44).

There is data about **14 courses**, but a Course in Joly is defined as a combination of subject and semester, for example INF1000 fall 2009 is one course, and INF1000 fall 2011 is another. For

Figure 5.1 Original Joly database schema. The current version of the database has one table less (Checkelement) and two more related to the SM algorithm: Preprocessed and Comparison (see figure 5.3 (p. 45). Taken from Steensen and Vibekk (2006, p. 185)



Antics I translated this into a hierarchy of two classes, Subject and Period, like in Devilry. This makes it a little easier to treat the data about one course together, but it is still possible with the Joly model.

I also noted that two of the courses had no deliveries of student solutions, even though they had Coursegroups and other information. I therefore built into Antics a check that the courses actually contain student solutions in the function for importing student solutions from Joly, so that the user not be confused when he selects a course but nothing is imported.

I found that only two actual courses had any student deliveries in Joly: INF1000 (with data from 10 different semesters, and a total of 9685 student solutions), and INF1010 (with student submissions from two semesters, 1041 deliveries), for a total of **10726 deliveries**. By the way, I com-

Figure 5.2 Results of the Django "inspectdb" tool run on the Joly database, to help make a Django ORM model for the Joly database.

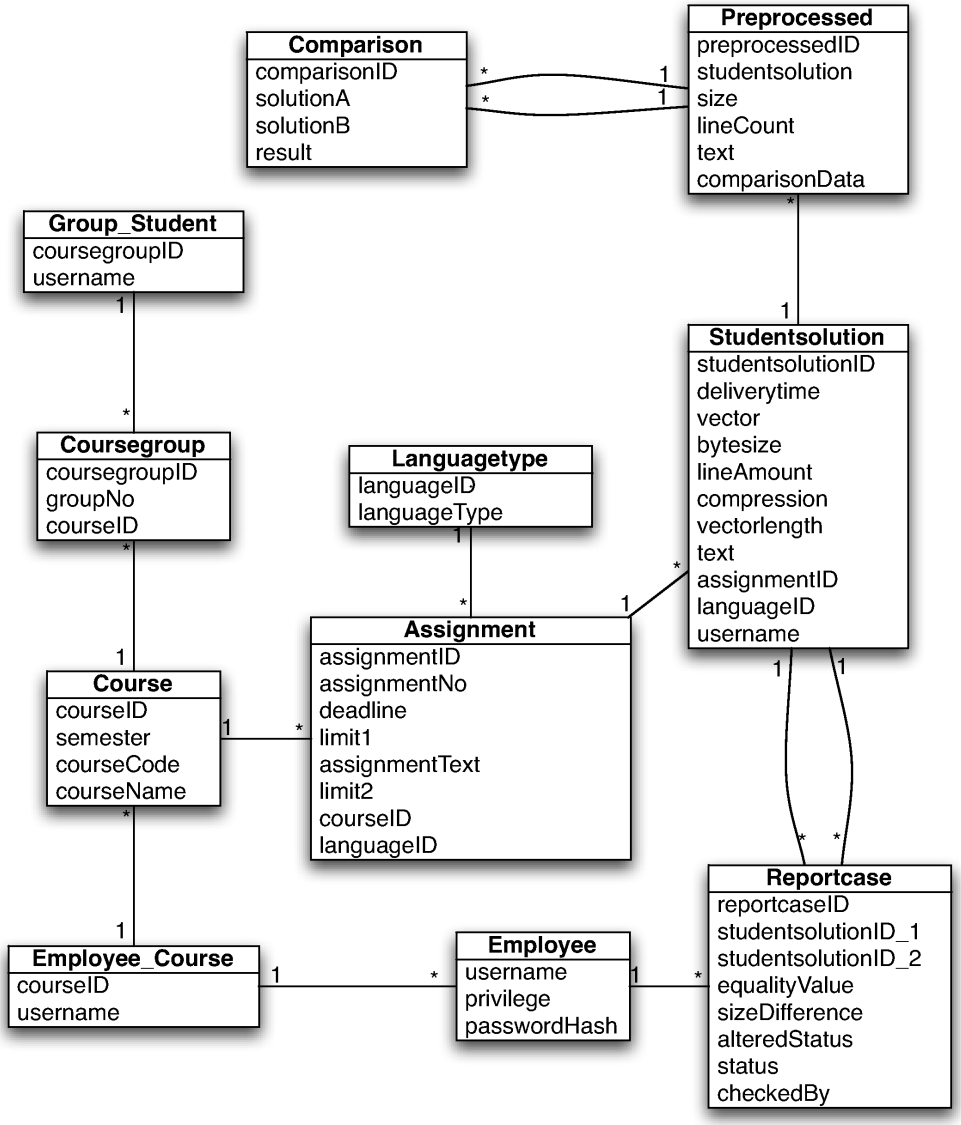
```
# This is an auto-generated Django model module.
# You'll have to do the following manually to clean this up:
#     * Rearrange models' order
#     * Make sure each model has one field with primary_key=True
# Feel free to rename the models, but don't rename db_table values or field names.
#
# Also note: You'll have to insert the output of 'django-admin.py sqlcustom [appname]'
# into your database.

from django.db import models

class Course(models.Model):
    courseid = models.IntegerField(primary_key=True, db_column='courseID')
    semester = models.CharField(unique=True, max_length=30)
    coursecode = models.CharField(unique=True, max_length=30, db_column='courseCode')
    coursename = models.CharField(max_length=180, db_column='courseName')

    def __unicode__(self):
        return self.coursecode + '-' + self.semester

    class Meta:
        db_table = u'Course'

class Assignment(models.Model):
    assignmentid = models.IntegerField(primary_key=True, db_column='assignmentID')
    algorithm = models.CharField(max_length=765)
    assignmentno = models.IntegerField(db_column='assignmentNo')
    deadline = models.DateTimeField()
    limit1 = models.FloatField(null=True, blank=True)
    assignmenttext = models.TextField(db_column='assignmentText', blank=True)
    limit2 = models.FloatField(null=True, blank=True)
    #courseid = models.IntegerField(db_column='courseID')
    courseid = models.ForeignKey(Course, db_column='courseID')
    languageid = models.IntegerField(db_column='languageID')

    def __unicode__(self):
        return u'Assignment ' + unicode(self.assignmentno)

    class Meta:
        db_table = u'Assignment'
[...]
```

Table name	Exact row count
Course	14
Coursegroup	130
Assignment	45
Studentsolution	10726
Languagetype	1
Group_Student	3875
Student	2758
Employee	79
Employee_Course	121
Preprocessed (SM)	10726
Comparison (SM)	3192
Elementtemplate (ATM)	1
Elementoccurrence (ATM)	0
Reportcase (ATM)	0

Table 5.5: Summary of data in the Joly database. For each table the number of rows with data is shown.

puted these numbers in two different ways, using a MySQL client connected directly to the Joly database (see table 5.5 (p. 44)), and computed by Antics as the number of deliveries it could import from Joly, and both counts were luckily the same. The Antics application has a menu choice called "Antics database" that shows a summary of the student deliveries currently imported into the internal database (as I illustrate later in figure 6.3 (p. 52)). It is a little rudimentary summary at the moment, but still useful.

Figure 5.3 Updated Joly database schema. The current version has three tables not shown here: Student, and two tables related to an ATM algorithm: Elementoccurrence, and Elementtemplate. Taken from the Joly documentation.



5.3.3 Antics core app

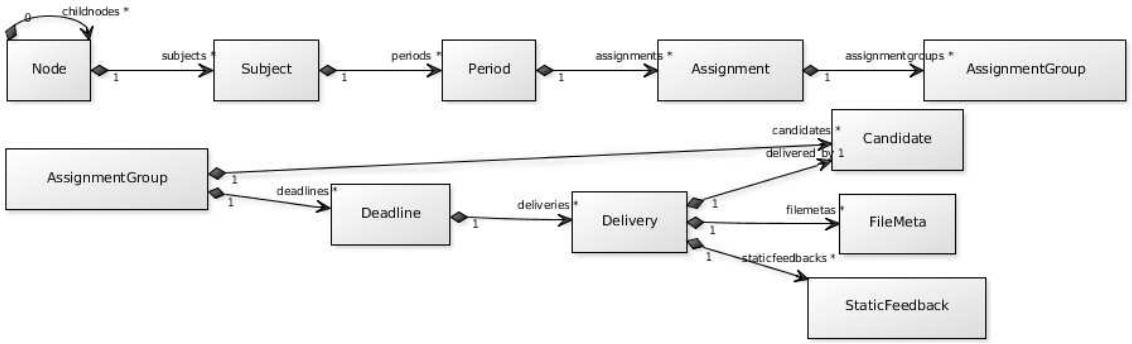
Figure 5.4 (p. 46) shows the main classes in the Devilry core data model, and figure 5.2 (p. 36) shows all the classes.

The Devilry code is organized as more than 20 "Django apps", and several other utilities, APIs, clients, etc. The most important app is called "core", which is intended as the basic and stable core with the most important classes, around which other functionality can be added, plugged in, or connected in various ways. I only replicated a few of the core classes and put everything into a "core" Django app for Antics.

A different setup that could have been used would be to make a separate Django app for the 2–3 plagiarism-related classes (marked "SM" in the "Antics core app" column in table 5.3 (p. 39)). That could result in a better organized model, and maybe easier adaptation to Devilry, but I also

wanted to simplify things, and decided to reduce the number of Django data model apps to only the two apps shown in the table, one for connecting with Joly, and one for everything else.

Figure 5.4 Devilry core data structure. From http://devilry.org/devilry-django/dev/core.models.html



Some of the differences in semantics between Joly and Devilry include the lack of fixed **sections** in Devilry, called "groups" in Joly, as mentioned above. In Joly these represent smaller classes of 20–40 students that usually get their deliveries corrected by the same TA (called "examiner" in Devilry) throughout the course. Devilry implements this differently, without fixed sections, but allowing each student delivery to be related to an examiner independently of other deliveries by the same student. The three courses I have applied my algorithm to use such sections, but a problem has been noted with some submission and assessment registration systems, including Joly, when the students change the section (or TAs' classes) they attend. When building Antics I had to decide which solution to use, the fixed one from Joly, or the flexible but more difficult to implement flexible examiners. Unfortunately, the Django ORM made it complicated to extract this infomation from the Joly database, as mentioned above and in section 5.3.1 (p. 40), so I haven't finished implementing a good solution for this yet when importing student solutions from Joly, but when importing solutions from other sources the sections should be as they are in the data. I'm working on finishing this in Antics.

5.4 Antics' Python-Java connection

Another interesting theme I had to consider was how to connect my Python code to my Java PRD algorithm. There are several possibilities, depending on how much data needs to be transferred between the languages, whether access to the database is needed from the foreign language, etc. Devilry itself has no Java bindings currently, but there are a couple of APIs built into Devilry that help communicate with code written in other languages, including a "RESTful" web API and a "simplified" API. In order for the foreign language code to communicate with Python, the Django apps in Devilry require some extra code implementing APIs for accessing their data models through the simplified or RESTful Devilry APIs.

So one possibility for Antics could be to copy or reimplement some of these **APIs from Devilry**, then update my core app to expose its data through the API for access from Java. This should be easier to do when Antics is actually integrated into Devilry.

Another possibility could be to access the Antics database directly from Java using **JDBC drivers**, and use Python to start the Java code and wait for it to finish. This could be very complicated if I wanted to maintain the Devilry philosophy of allowing the user to chose between at least four database backends, now from Java. It's not very important anyway, and SQLite could be selected as the only internal database for Antics.

Due to these complications, and limited time, I decided to use an easier solution, I just created a simple scheme for **communicating through files** in the file system. There is only one type of communication that is needed, which is the Python code calling the Java algorithm when the user asks for a given plagiarism detection run that has not been done earlier and stored in the Antics database. When the user asks for this, the Python side makes sure the necessary data is in the file system, then starts a Java application, which provides some output of its progress in its own window, and when it has finished, control is regained by the Python application, which gets the results from a file and enters these in the database. To allow multiple users at the same time in Antics, the file used for the results is given a unique name for each plagiarism detection run, so that several such runs can be executing at the same time.

A related question was what type of **files to use for communication**, and whether these would be too wasteful of disk space. The files used include the preprocessed program skeletons, and a few small text files that control the Python-Java communication (one describing the tasks for the Java algorithm and another for the results).

In order for the Java side to **discover the usernames** of the students' solutions it checks for plagiarism, the scheme uses the usernames as the folder names in which the files are located when "communicating through files". These usernames practically the most important information from the database that the Java algorithm needs (in order to skip comparisons between earlier solutions of the same student, and similar cases), and so it is that the simple file communication provides all the information the Java algorithm needs. When done, the Python app recollects the results produced from Java, containing these file names, and updates the Antics database accordingly.

5.4.1 Database vs. file system storage

An interesting feature of Devilry is its combination of database and file system storage. This is a controversial issue, so I start with some quotations. I then summarize my view of the matter at the end of this section.

Sears et al. (2006) conducted an extensive study that compared the preformance of one database system (SQL Server) against one file system (NTFS), and found that for objects "larger than **one megabyte** on average, the **file system** has a clear advantage. If the objects are under **256 kilobytes**, the **database** has a clear advantage. Inside this range, it depends on how write intensive the workload is, and the storage age of a typical replica in the system."

The **Django** ORM framework encourages storing uploaded files and images in the file system and not the database, "for performance" reasons (Django, 2012). Its data model specifically lacks a field designed for database BLOBs, and the only built-in file- and image-related fields are designed to store the contents of these in the file system (Django, 2012). However, there are ways around this, one can use the built-in TextField type to store files, and the model can implement automatic encoding and decoding between binary data and text.

Devilry is currently using the file system for storing the deliveries submitted by students at Ifi, though all other data, including all metadata about the same submissions, is stored in a database. The lead developer of Devilry, Espen A. Kristiansen, gave me some of the reasons for this choice: [My translation:] "When they ported Devilry to Django [(Devilry had started as a Java project)], they decided to adopt an API for files with these characteristics:

- "It had to support data streaming, to make it possible to deliver and retrieve large files, and streaming of audio and video.
- "It had to support large files, if possible limited only by the capacity of the server.
- "It had to support changing the storage back-end.

"Even though all relational databases support BLOBs, not all support streaming of data, and those that do, don't do it in the same way. This makes it difficult to use relational databases for files in Devilry without having to lock Devilry in to a single database.

"For the time being Devilry uses a filesystem-based backend for storage. The main reasons for this are that it is a simple and stable solution, with good performance. It's also easy to scale to distributed file systems without the need to change Devilry."

In the case of **Antics**, I am currently using a **dual solution**, chosen mainly for simplicity when interfacing Java and Python, and that fulfills the main requirements for my application. All the data is kept in the application's own database, and only a subset of the data, that can easily be re-generated from the data in the database, is stored in the file system. This is different from both the Joly and Devilry approach, but somewhere in between. I keep the convenience of Joly, of having all the real data in one database, so that a simple database dump is enough to back up all the data, while still taking advantage of the file system when its simplicity helps to achieve the application's goals, somewhat like Devilry.

The database engine for Antics can still easily be selected among SQLite, MySQL, PostgreSQL, and Oracle, but only the SQLite option comes bundled with the application and ready to use. The other three can be selected if the user already has the desired database installed elsewhere, in which case she can use that by editing six fields in the `settings.py` file: database engine, name, user, password, host, and port (if not using the default for the given database).

The **file system** is used for storing both the `.java` files of student submissions (possibly extracted from compressed archives if that is what was imported into Antics) and the preprocessed program skeletons produced by the Java algorithm from these extracted files. Both these types of files are accessed from Java when the user asks for a new cheating detection run, because this makes it easier for the Java algorithm to access the data. The copying of data from the Antics core database to the file system is not a big problem for performance, since it only needs to be done once for each file, and the files copied are `.java`-files, which are usually small in size (I present a test run with timing information later in section 7.2 (p. 57), where copying from database to files a thousand deliveries took about 10 seconds with basic hardware). The complete collection of 10726 student submissions that are currently in the Joly database only take 109 megabytes of space when transferred to the Antics SQLite database, (and a little more when copied to disk files), and this is very little disk space these days.

Copying is skipped for files that were already copied to the file system in an earlier detection run. The copying also does not have a large effect on the running time of the algorithm, which is dominated by the many program comparisons done by the Java algorithm (this will also be illustrated later, with timing details in section 7.2 (p. 57)). What could have an effect is the fact that the Java code is accessing files instead of database data directly. As Sears et al. (2006) found, files of these small sizes are generally handled faster by databases. This is however not a big concern for Antics at the moment. The most important thing for such an early version of the application is that it just works and allows teachers to detect plagiarism, and is fast enough for this. In later versions one can add refinements that improve running time.

Regarding the **performance** issue, I found that the student deliveries in the three courses I concentrate on in this thesis are on average smaller than 256 kilobytes, so in these particular courses a database-only solution would probably have better performance than a file system solution. However, Devilry is intended for use in many more courses and situations, potentially the whole university, and including scanned examinations, or courses with various types of large deliveries, so I think there are **other factors** to consider besides performance, and the file system alternative (task 2.(k) (file vs. database storage) (p. 9)) seems like a reasonable choice, like Espen suggests. It also follows the philosophy of the Django framework which is an integral part of Devilry.

Chapter 6

Implementation

This chapter describes the implementation of the Antics stand-alone application in more detail, especially its web-based graphical user interface, and how it can be used to detect plagiarism in computer courses.

6.1 Graphical user interface

The Antics GUI is a typical **event-driven** interface. It provides a web page with a menu, and does nothing until the user selects from this menu some event—defined in Wikipedia as "a significant change in state"—that she wants to occur.

Some of the reasons for selecting the Django **Web interface** for Antics (see task 1.(b) (user-friendly GUI) (p. 8)), instead of using GUI technologies I knew a little better before I started this project (like the Java Swing toolkit, Java applets, Tcl/Tk, Tkinter, or even CGI scripts), are that Devilry uses a Web interface, and one of my tasks was to consider Devilry integration (see task 2.(i) (assess Devilry inclusion) (p. 9)); that I wanted to learn more about this Web interface; that Django looked suitable for quicker application development; and as it turned out had very good support for developing and debugging the Web UI also—particularly compared to CGI scripts which are a nightmare to debug in comparison (Langtangen, 2009).

Both the server process and the client browser are currently intended to be run on the same host computer, and the user should have access to the command window where the server was started to see some additional program output. The web server I'm using is just the development server that comes with Django ("**runserver**"). It is not an advanced solution intended for handling many users at a time and is slower and less secure than full-blown web servers like Apache, but it gets the job done, and is the easiest way to get up and running. Some of the main goals for this version of Antics were that it worked and could easily be used for plagiarism detection.

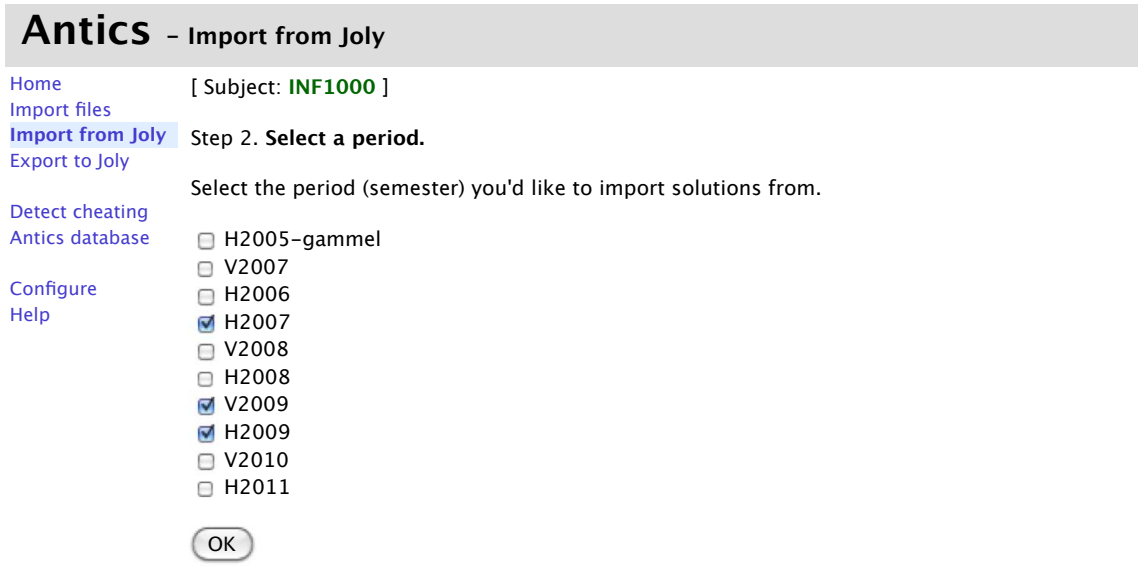
6.1.1 Antics' menu options

The Antics application presents the user with these main choices from its left menu:

Home: The home page describes the main functions of the application.

Import files: (See figure 6.1 (p. 50)). This menu option lets the user select a file or directory (see task 1.(c) (directory import) (p. 8)) in the file system of her local computer (the one that

Figure 6.1 Example from the user interface of "**Import from Joly**" in Antics. At this point the user has selected the subject (course) INF1000 and is selecting which semesters of that course to import data from the Joly database to the Antics database.



is running the browser client), and then imports all the student solutions from that file or directory to the Antics database. This requires that the files are organized in one of the supported import formats. One example format is *COURSE/Semester/assignmentnumber/student/version/solution* (for example `INF1000/H2011/ob3/josek/v1/Oblig3.java`). The user can also enter a specific course, semester, and possibly assignment number if desired, and then Antics would only import solutions for the specified semester, without requiring these fields in the directory structure to be imported. Another file format I'm working on integrating is the Zip files downloaded from Devilry with many student solutions in a given assignment. In this case Antics would unpack the Zip file, recognize the file format (using magic, see p. 38), and import the student solutions in it, organizing them according to the meta data that Devilry includes in the Zip file. When importing from files or directories while Antics is running on a Linux machine in the Ifi network, it may be possible for Antics to determine the section ("examiner group") a student belongs to using the Ifi command-line tools discussed in section 2.8 (p. 18). This is not implemented yet.

Import from Joly: (see figure 6.1 (p. 50)) Lets the user select a course and a set of semesters, and imports all the student solutions in that semester from the Joly database to the Antics database. Antics always keeps track of the username of the student, the semester, assignment, and course that each student solution belongs to, and will put submissions by the **same student** on the same assignment associated together in the internal database (as two versions), whether these were imported from a file or Joly (task 1.(d) (run against Joly database) (p. 8)). When the plagiarism detector is called such submissions will be labeled with the same username, and thus will not be counted as possible plagiarism if two solution from the same student match, as asked in task 1.(f) (skip same-submission) (p. 8). Note: In order to use this function the user needs to enter a valid Joly password in the "Configure" menu item (see section 6.4.1 (p. 54)).

Export to Joly: This will allow the user to transfer student solutions from Antics to the Joly database (as asked for in task 1.(g) (consider export to Joly database) (p. 8)). The solutions should first be imported to Antics using the "Import files" function mentioned above. That function ensures the solutions are properly categorized with course, semester, assignment, students, etc., so that they can be uploaded to the Joly database with the proper

Figure 6.2 Example of output from the "**Detect cheating**" function. The usernames of the students have been scrambled and the times of delivery changed, but otherwise these are the actual results from the last use of the algorithm in the course INF1000, Fall 2011, Assignment 3, and correspond to the last line in table 7.1 (p. 56). The matches with "= h10-" are likely plagiarism of deliveries from the previous year. "g#" stands for section (examiner group).

Antics – **Detect cheating**

Home
Import files
Import from Joly
Export to Joly

Detect cheating
Antics database

Configure
Help

[Subject: **INF1000**]
[Period: **H2011**]
[Assignment: **oblig3**]

Results of the plagiarism detection on oblig3:

```
Sorted by most similar:

100%:  g4/remab/Oct21_1757 = g3/wknols/Oct21_0308
  100  g4/remab/Oct21_1318 = g3/wknols/Oct21_0308
   75  g4/remab/Oct21_1757 = g8/xknh/Oct21_0329
   75  g4/remab/Oct21_1757 = g8/xknh/Oct21_1723
   75  g4/remab/Oct21_1318 = g8/xknh/Oct21_0329
   75  g4/remab/Oct21_1318 = g8/xknh/Oct21_1723
100%:  g3/wknols/Oct21_0308 = g4/remab/Oct21_1757
  100  g3/wknols/Oct21_0308 = g4/remab/Oct21_1318
   75  g3/wknols/Oct21_0308 = g8/xknh/Oct21_0329
   75  g3/wknols/Oct21_0308 = g8/xknh/Oct21_1723
 99%:  g7/oztlia/Oct20_1934 = g7/uankf/Oct20_2051
 99%:  g7/uankf/Oct20_2051 = g7/oztlia/Oct20_1934
 84%:  g11/hwenx/Oct31_1656 = h10-g12/oanux/20101022.0227
   84  g11/hwenx/Oct27_1822 = h10-g12/oanux/20101022.0227
   72  g11/hwenx/Oct31_1656 = g1/emilmen/Oct28_1908
   72  g11/hwenx/Oct27_1822 = g1/emilmen/Oct28_1908
   59  g11/hwenx/Oct24_0122 = h10-g6/hjxiov/20101022.0258
 83%:  g118/gwxg/Oct20_2054 = h10-g2/yxrzmsi/20101022.1234
   64  g118/gwxg/Oct20_2054 = g113/xaizc/Oct21_1302
 76%:  g1/hwocit/Oct21_1355 = h10-g11/kowet/20101022.0256
   73  g1/hwocit/Oct31_0333 = h10-g11/kowet/20101022.0256
   62  g1/hwocit/Oct31_0333 = g11/eailcol/Oct21_1710
 75%:  g8/xknh/Oct21_0329 = g4/remab/Oct21_1757
   75  g8/xknh/Oct21_0329 = g4/remab/Oct21_1318
```

Download high-scoring files

(Click this button to downloaded all the files in the results list as a Zip file, which will also include the results list as a text file.)

meta data. Since the Joly database only allows one file per delivery, which must be a `.java`-file, Antics will first unpack the files if they are compressed, and if there were several files in a single delivery the `.java`-files in it will be collected together into a suitable larger `.java`-file.

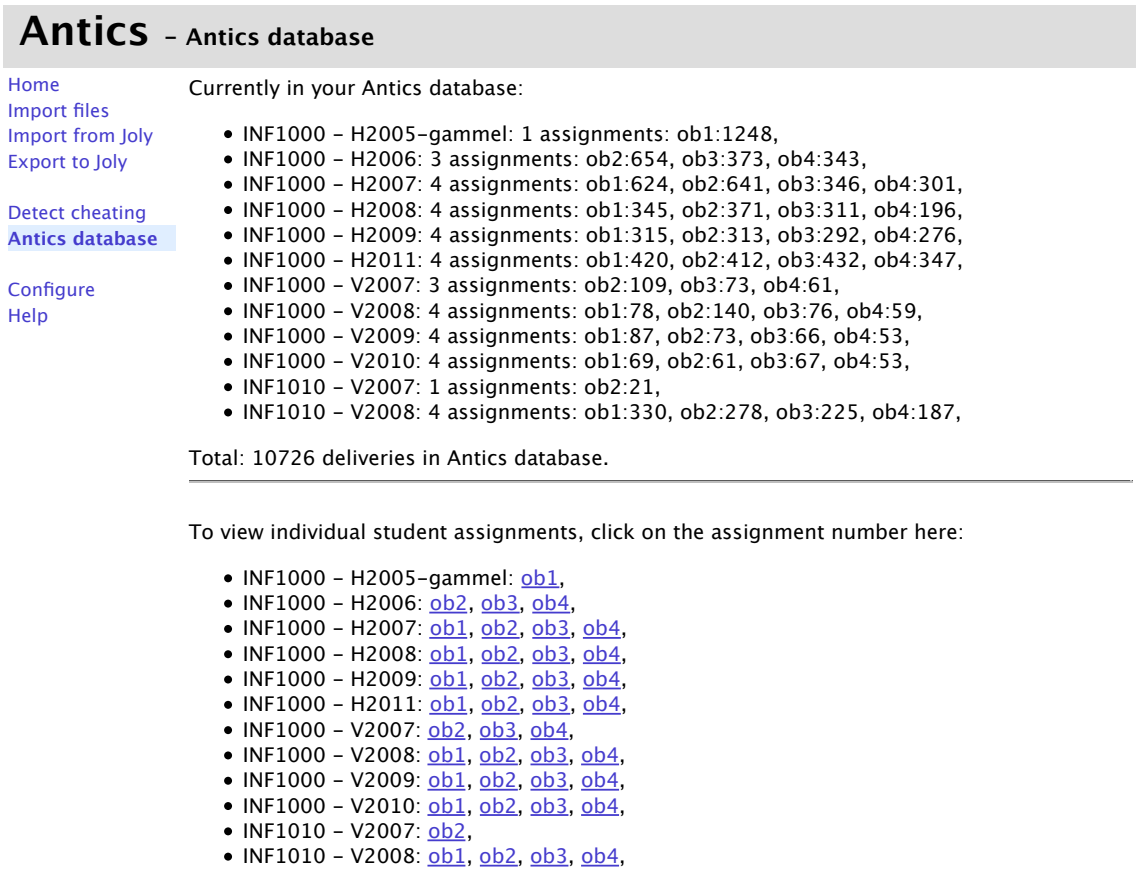
Detect cheating: (See figure 6.2 (p. 51)). This allows the user to select a course, semester, and assignment from his local Antics database, and then runs the Java plagiarism detection algorithm on them, showing the results when this is done. More details about this process can be found in section 5.4 (p. 46).

Antics database: (See figure 6.3 (p. 52)). This menu option shows the contents of the user's Antics database, with a summary of the numbers of deliveries in each semester and each course from which the user has imported student solutions. It will also allow the user to select any delivery in the database to download it or view it in the browser window.

Configure: Allows configuring a couple of global settings for the user's instance of Antics, including entering a Joly password to enable the Joly import and export features, and changing the Antics password.

Help: A short user guide.

Figure 6.3 GUI for the menu item "**Antics database**", which shows an overview of the current contents of the Antics database, and allows the user to open an individual student assignment. In this example all the 10726 student solutions in Joly were imported to Antics. "ob" is a shorthand notation for "obligatory assignment. An improvement that would be useful here is a text field to enter the username of a student, allowing the download or viewing of her submissions.



6.2 Cheating detection user interface

Here is a possible rundown of the whole process that the system can go through when the user selects the menu option "**Detect cheating**" in the Antics application.

The Python "views.py" function associated with the URL "`http://127.0.0.1:8000/detect/`" starts by showing him a list of courses, periods (semesters), and assignments that are currently stored in the Antics database. The user **selects the assignment** and choses which other semesters to include in the plagiarism detection run. The main choices will be to only compare the student solutions in the selected assignment with each other, or to also include earlier submissions of the same assignment.

Having selected this the Python function checks that the necessary data is in the file system, as described in section 5.4.1 (p. 47), and if not copies the necessary data. Then the **Java PRD** algorithm is started, with the necessary input telling it what the user asked for, and this Java application runs the plagiarism detection. Section 7.2 (p. 57) shows an example of a test run like this with timing information.

This process starts by checking whether the preprocessed versions of the files to be compared are already created, as explained in section 4.4.1 (p. 28), if not these are made by the same Java

program. Both this step and the possible previous copying step are relatively quick, a test run on an older laptop took about 6 seconds for around 1100 deliveries (section 7.2 (p. 57)). Both processes write each file only once.

Then the real task starts, of **comparing all** the student submissions of the user-selected assignment against themselves, and against previous submissions of the same assignments if the user selected that option. The number of needed comparisons can be quite large as explained in section 2.3 (p. 12). For large program sets this process can take a few minutes.

While this is happening, the web interface is currently not updated, but some data is given to the user about the algorithm's progress, printed to the command window. The plan is to make either a small window showing the progress or maybe some information about the results so far.

When all program comparisons finish, control is returned to the Python application, which loads the **results into the Antics database**, from the location where Java left it, which is currently in a specific file in the file system. This file is located in a specially created directory for this communication, having a folder name with both a date and time component and a random number, so that multiple plagiarism detection runs can in principle be run at the same time by different users. These directories can be deleted when Antics has received and organized the results they contain, or they can be left in case a few more deliveries are downloaded later and an updated detection run with these is desired.

Finally, after the data is ready in the Antics database, the user will be shown a summary of the results (see figure 6.2 (p. 51)), including a list of **the most suspect** deliveries in the set, and depending on the information available about the section (class) where the students in question are registered, a sublist is given with the results for each section. In beginner courses there is usually a good number of suspected cases identified by the algorithm, though this number has decreased a lot over the years in INF1000. Table 7.1 (p. 56) shows some examples of results found using the algorithm over many years.

The course lecturer is usually most interested in "the most suspect" results for the whole course, while specific TAs, which correct the assignments of a few students, would mostly be interested in the results for their own students.

Depending on the policy of the course lecturer or department, the lecturer or TAs may now control the suspected cases from their sections manually, maybe send summaries to the lecturer, which calls in the students in the most apparent cases for interviews, etc., as described in section 2.4 (p. 14) and section 2.5 (p. 14).

6.3 Devilry integration

I designed the Antics application with a goal that it would be easier to integrate into Devilry (see task 2.(i) (assess Devilry inclusion) (p. 9), and task 2.(m) (integrate if easy) (p. 9)) than if I had developed it without regard to Devilry. This had an effect on many design decisions, as explained throughout chapter 5. The most obvious consession to Devilry is using **Python and Django** to control the functions of Antics. This should help integrate the two systems, by keeping Antics as a Django app, which it already is, but maybe located under the apps directory of Devilry.

The **core data model** of Antics emulates some of the Devilry core data model classes and attributes, and uses similar names. It is a closer to the Devilry model than if I had just replicated the Joly data model. This should also help a little during the integration.

Since both applications currently have their versions of classes like Subject, Assignment, and Delivery, two alternatives for integration (task 2.(i) (assess Devilry inclusion) (p. 9)) are either to change the code in Antics so that it uses Devilry's versions of these directly; or keep the classes in Antics and just copy the data between the respective class objects when needed. The first

alternative provides a tighter and more elegant integration. The second may be easier and presumably could work ok since only a minority of the courses in Devilry would likely use the Antics functionality. With both alternatives the data elements that are unique to Antics (those relating to **plagiarism detection**) should of course be left in their own app separate from the Devilry core.

When it comes to the controller part of Antics, these functions (menu choices) can be a good starting point for implementing the same functions in a suitable Devilry controller (see task 2.(j) (automation in Devilry) (p. 9)), maybe still coupled with the new Antics app in Devilry. The translation of the code should be helped by the facts that Antics is already programmed using the same Django framework as Devilry and uses a similar though reduced data model. For instance, references to the class `antics.core.models.Subject` would become something like `devilry.apps.core.models.subject.Subject`, and so on.

The user interface templates are somewhat rudimentary in Antics at the moment, but some of the code should help design a suitable view for the Antics functionality when integrated into Devilry.

As a bonus, when integrating Antics with Devilry, one part that could probably be used directly with the least changes is the integration of Antics with the Joly database. Antics already has the necessary infrastructure to **translate the Joly data** to a form suitable for Devilry, so this particular data model in Antics should help if data from Joly is to be imported into Devilry (see task 2.(n) (Devilry to Joly database export) (p. 9)), or if data from Devilry is to be exported to Joly at discrete points in time like this functionality is provided in Antics. The same problems I encountered with this integration would be found when connecting Devilry with the Joly database, and the same solutions should help solve these problems.

6.4 Installing and running Antics

One of my goals for Antics was ease of installation. I tried to make it so that it would be enough to download and unzip the **antics-1.0.zip** file and start using it immediately without any installation, and so that it would work equally well on any Linux, Windows, or Mac computer. However, I haven't tested all the possible scenarios yet, so there could be some bugs. The application can be downloaded for Ifi-related purposes from: `http://josek.at.ifi.uio.no/antics` (password: `secrt`) in mid-2012. After that, you can try www.CodeAntics.com.

The requirements are mainly that Python 2.5, 2.6, or 2.7 is installed in the computer and Java 1.5 or later (but check the accompanying README file for possible updates to this). These versions can be checked with the commands "`python -V`" and "`java -version`". The reason for those particular versions of Python is that Django doesn't support Python 3 yet, and that Antics uses the SQLite database built into Python since version 2.5. The reason for Java 1.5 is that the PRD algorithm uses several of the capabilities of Java that were introduced with version 1.5.

There will be a README file in the above-mentioned Zip file explaining how to start the Antics application. The procedure will likely be something similar to typing the following two commands in a command window: "`cd antics`" (to change to the directory where the Zip file was unpacked), and "`python run.py`" (to start the application). You then open a URL like "`http://127.0.0.1:8000/antics`" in any browser that is running in the same computer.

To stop the application one can press "Ctrl-c" in the same command window where the application was started.

6.4.1 Security in Antics

Antics is a work in progress. It should be reasonably usable and user friendly by now but not very polished yet. It currently has no user management, it only asks for one bare password

(without accompanying username), when the user wants to access the various functions of the GUI (graphical user interface), and this password is transferred unencrypted to the server. The password is stored unencrypted in a settings file. You keep the file only user-readable so that other osers can't see its contents directly, though some determined hacker may possibly be able to find it if he has remote access to the computer where Antics is being used (because the development web server used to run Antics easily is not designed to be very secure). Normally they won't know the simple password and have a little trouble logging into Antics.

The reason I included such a password is that the GUI is web-based and can be accessed by other people if they are logged in to the same host computer or network. It is currently using Django's development web server ("runserver"). This runs the application with the user privileges of the user that started it, instead of a user with limited privileges as would be the case with a larger web server. As I understand it this server accepts connections from anybody that can access the host computer, so if there was no password the students that learned when their teacher was using the Antics application could possibly access it if they had remote access to the host computer that was running it and found the URL being used, and had the password.

To prevent this possibility the teacher can take measures like these:

1. Put their copy of Antics in a **directory** that is not world-readable. This way the application source code and its database file will not be accessible directly by remote shell, and hackers would need to wait till the teacher actually runs the application for it to be accessible, and then only the user interface could be accessible to them and not the underlying files. It is not necessary to locate Antics in a directory managed by another web server.
2. Run Antics in a **host computer** that is not accessible by students remotely. That way the students can't access the web interface even when the application is being run. At Ifi, the computers in the lecturers' offices are normally not accessible by students, so a good solution would be to run the application in such computers, or in their own laptop or home computer.
3. Change the **password**. Antics comes with a simple password for accessing the web interface, and a "Configure" menu choice for changing it. It is probably a good idea to change the password if one is going to run Antics in a host that is accessible by others.
4. Change the **port**. Antics allows the user to select a port number for the web interface, but uses a default value when this is not supplied. The port number is part of the URL address needed to open the web interface to use Antics.

On the other hand, the **password timeout** is a little short at the moment, so I included instructions for turning off the password check if the user wants. This is at the bottom of the `settings.py` file. It should be ok after doing step 2 in the list above.

Step 1 (changing the mode of the settings file to only **user-readable**) should always be done if the Joly database password has been configured in Antics. The only thing this configuration does is store the password in the `settings.py` file in the Antics root folder. Therefore it's important that this file not be world-readable in the file system if it is stored in a computer network where students or other people have access to files. The development web server should not expose the contents of this file to others if it's only readable by the user.

Another password that Antics manages is a **password** to connect to the **Joly database**, as mentioned above. It is not meant to be included with Antics normally, but maybe a password to the Joly development database or similar may be made available. The Joly database appears to only be accessible from the UiO network, so this is a requirement for connection. Antics has a menu choice called "Configure" where the user can enter his Joly database password and other information (host, user, port, etc.), if the user wants to connect to the Joly database with Antics.

Chapter 7

Results

In this chapter I present some results of using my algorithm for plagiarism detection with actual student deliveries at Ifi, including a detailed test run using the new Antics stand-alone application.

7.1 Plagiarism detection over time

Semester & Assignment		Number of students with max. score above the given %				Total no. of students with similarity score above 75%	
		100%	90%+	80%+	75%+		
h2004	ob2	8	+41	+42	+38	= 129/499	25.9%
	ob3	13	+16	+19	+12	= 60/411	14.6%
	ob4	12	+15	+22	+17	= 66/339	19.5%
h2006	ob2	3	+13	+29	+17	= 62/487	12.7%
	ob3	5	+13	+25	+7	= 50/327	15.3%
	ob4	8	+5	+5	+4	= 22/316	7.0%
h2007	ob2	0	+3	+7	+6	= 16/296	5.4%
	ob3	1	+0	+5	+3	= 9/251	3.6%
	ob4	2	+5	+4	+4	= 15/230	6.5%
h2008	ob2	2	+4	+16	+12	= 34/283	12.0%
	ob3	0	+1	+1	+0	= 2/244	0.8%
	ob4	0	+1	+0	+0	= 1/204	0.5%
h2009	ob2	0	+1	+7	+7	= 15/269	5.6%
	ob3	0	+3	+1	+2	= 6/236	2.5%
	ob4	0	+2	+2	+1	= 5/218	2.3%
h2010	ob2	0	+3	+4	+23	= 30/327	9.1%
	ob3	0	+6	+4	+5	= 15/286	5.2%
h2011	ob2	0	+2	+2	+12	= 16/331	4.8%
	ob3	2	+2	+2	+3	= 9/310	2.9%

Table 7.1: **History of plagiarism detection in the course INF1000**, showing a decrease in the number of obvious plagiarism cases over time. **Example:** The first line means that on the 2nd obligatory assignment, 8 students' solutions were 100% equal to another solution (see "**Notes**" below, p. 57), 41 students' solutions scored between 90% and 99% similarity, and so on, and 129 students of the 499 that delivered the assignment got a score of 75% or higher similarity.

The table above shows a summary of results of using my PRD algorithm for detection of plagiarism in the course INF1000 at Ifi. All the results were computed with essentially the same version of my algorithm, each time comparing the deliveries in the given semester with all the earlier ones and the ones from the same semester.

Notes:

- The obligatory assignments are labeled "**ob**". There are 4 of these in INF1000 each semester, but the first one is too small for plagiarism detection (it can be solved with 10–15 lines of Java code), so only assignments 2–4 are checked.
- The table shows the number of students that delivered high-similarity scoring solutions **when compared to all other** submissions the same semester or earlier, without separating between the two. All the matches against earlier semesters are very likely plagiarism, but in pairs of matches the same semester one may be a legitimate original answer and the other a copy, or there may be multiple copies and one original. The actual number of plagiarizers is therefore likely a little lower than the numbers shown. However, the enablers aren't helping us solve the problem, and are difficult to separate automatically, so they are counted here.
- The **percentages** are computed based on the number of students still remaining in the course at each assignment. It could also be computed based on the initial number of students in the semester, giving lower percentages in the later assignments.
- The **4th assignment** was not checked for plagiarism in 2010 and 2011. It was a completely new assignment that has only been used in those two years so far, eliminating the possibility of plagiarizing older solutions. Also, in 2010 collaboration between pairs of students was allowed in this assignment.
- There was a minor **jump up** in the number of suspicious results from 2009 to 2010. One possible contributing factor was that the course got new lecturers at this point which had not worked much with plagiarism detection earlier. There are many variables that can influence these results, and the lecturer's experience, and the warnings they give the students at the beginning of these course are two such variables.
- In the fall of **2004** no actual plagiarism check was carried out, but the deliveries were collected, allowing a plagiarism check after the fact. In this case it revealed widespread copying, especially in assignment 2.
- **Joly** was introduced in 2006 (with an ATM algorithm), and my SM algorithm has been used every semester since 2007.
- Only data from the **autumn** semesters is shown. The course has also been given in the spring semester, but the number of students is much lower in these (60–80 instead of 300–500).

7.2 A test run of the system

To give a better idea of the results produced by the PRD algorithm and Antics' plagiarism detection function, I will present a detailed test run with deliveries of Assignment 3 in INF1000.

In this test I used a subset of the same data that produced the results seen in the last line of table 7.1 (p. 56) (for autumn 2011). The only difference is that I am only using the student deliveries from the last 3 semesters (1144 `.java`-files in total), instead of the data from 12 semesters which I used in the real run last year (3150 deliveries). In this case all the top plagiarism matches were from these semesters anyway.

Table 7.2: Overview of student solutions of Assignment 3 in INF1000 from 3 semesters, for use in a test run of the algorithm. The semesters are labeled "h" for *høst*, meaning autumn.

Semester	No. of students that delivered solutions	No. of versions they delivered
h2009	236	296
h2010	286	407
h2011	310	**441**
Total	832	1144

Table 7.2 (p. 58) shows an overview of the deliveries that I used in the test. As an example, the last line shows that 310 students delivered 441 submissions of assignment 3 in the autumn of 2011. Each delivery consisted of only one `.java`-file, but there are more deliveries than students because many students deliver several versions of their solution, either several unfinished ones while they are working with it before the deadline, or additional versions after the deadline if the TA asks for improved versions. This is the biggest assignment in the course and many students end up needing one or two extra attempts.

The test run will try to find plagiarism suspects in the last semester. This means that the 441 submissions from that semester will be compared against themselves, and against the 703 older deliveries $(296 + 407)$.

Going back to formula (2.1) in page 13, this means that the system will need to perform about 400,000 comparisons:

$$C = \frac{N_{new}^2}{2} + N_{new} \cdot N_{old}$$

$$C = \frac{441^2}{2} + 441 \cdot 703$$

$$C = 407,263$$

The procedure used to perform the comparisons was described in section 6.2 (p. 52), but I recap it shorly, and include information on running time:

@T0s: File creation: When the user has selected the course, semester, and assignment to use as N_{new}, and which semesters to compare them to (N_{old}), the first thing Antics does is check whether these students' submissions are in the file system, if not it copies them. In my test run it took 12 seconds for Antics to transfer the files from the Antics database to files.

@T12s: Preprocess & Reorder: Then the Java algorithm is called. It first "preprocesses and reorders" the 1144 `.java`-files, creating the preprocessed program skeletons. This took 6 seconds in my test run.

@T18s: Diff: Then the same Java application continues on to perform the needed comparisons between the programs. This took 3 minutes and 18 seconds in my test run, but I was using a laptop with relatively slow CPU and hard drive (a MacBook Pro with Intel Core 2 Duo processor), so this should be faster on newer hardware.

@T216s: Results: The results are shown on-screen and should be entered into the database. The database entry part is not completely finished, but the results will be as shown in figure 6.2 (p. 51) in the previous chapter.

Table 7.3: Statistics showing how many deliveries (students' solutions) got their maximal similarity score in each 10%-range. *Note: There were* **3 deliveries** *scoring 100%, delivered by* **2 students**, because one of them delivered two versions scoring 100%. This table counts the deliveries, while table 7.1 (p. 56) only counts students (scored with the maximum reached by their deliveries). See "(*)" in p. 59 for info about the Cut off.

Maximum score	**No. of deliveries**
100% alike	3
90-99%	2
80-89%	3
70-79%	10
60-69%	17
50-59%	34
40-49%	208
30-39%	120
20-29%	34
10-19%	0
00-09%	0
Cut off (*)	10
Total	441

By the way, that figure has the actual results from INF1000 autumn 2011. The usernames and times are scrambled but the percentages and other data are not, and it can be seen how the last line of table 7.1 (p. 56) was computed: 2 students had 100% score, 2 had between 90–99% and so on. The table counts only the maximum result for each student.

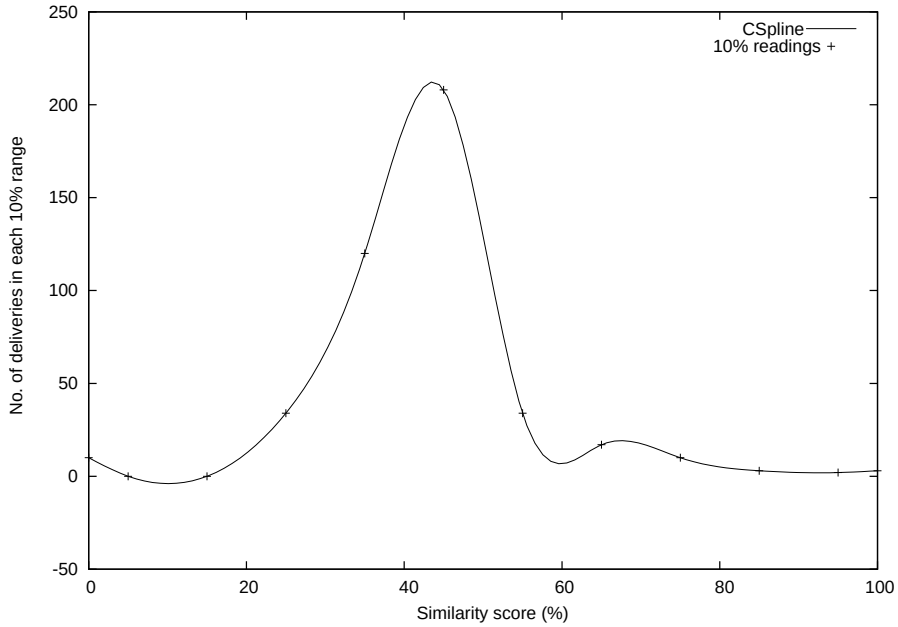
The Java algorithm also shows some **statistics** with the distribution of maximum scores for each submitted delivery, see table 7.3 (p. 59). Each delivery is counted once and located in the 10%-range where it got its maximum similarity score when compared to the other deliveries. This data is currently based on deliveries and not students, so students submitting two versions of their solution will be counted twice. For example, there are **three 100% deliveries** (see figure 6.2 (p. 51)), but these came from only **two students**. The reason they were counted as three in these statistics is that one of them delivered 2 versions (this can be seen from the timestamp of the two deliveries at the top of the figure). This is computed further into 2 students in table 7.1 (p. 56), since this table is based on numbers of students and not versions of deliveries.

The rest of table 7.3 (p. 59) is computed in a similar manner, it shows that there were two deliveries with maximum similarity score between 90–99%, three in the range 80–89% and so on. These values are the highest score each delivery got when compared against all others. The value comes from the normalized Diff results as explained in section 4.4.3 (p. 29). For the top scoring deliveries an extra step is performed, with character-wise refinement of the line-based initial Diff (see section 4.4.3 (p. 29)). In these cases it is the adjusted result that is used.

For a few deliveries (about 10 in this case, marked "(*)" in table 7.3 (p. 59)), no actual Diff was run, either because no single comparison met the size range criteria. This usually means the size of the delivered program was too small for a meaningful comparison, but it could also be because it was too large. The algorithm only runs comparisons between programs that are within 50–200% of each other in size as computed by their number of characters after preprocessing (that is, comparison is cut off if one is less than half the size of the other or more than double), as explained in section 4.4.3 (p. 29).

Figure 7.1 (p. 60) shows a graphical representation of table 7.3 (p. 59). The data points are set in the middle of each percentage range, so the deliveries reaching 90–99% are plotted at $x = 95$, those between 80–89% at 85 in the x-axis and so on. The deliveries that were cut off from any comparison are plotted at $x = 0$. To aid visualization of this data I connected the points

Figure 7.1 Histogram for the results of applying my algorithm to **Assignment 3 in INF1000**, autumn 2011. For each delivery (students' solution) the maximum similarity score was recorded, after comparing it to the 1144 other solutions in the program set. The number of these falling within each 10% range is represented by the points in the chart. To aid visualization the data points were interpolated using a CSpline (with Gnuplot), but see the text (p. 60) about the right "shoulder formation" which does not represent the underlying data, it's just an interesting coincidence.



using a cspline (with Gnuplot). These plots always have the general shape of a **bell curve**. The majority of the student solutions fall in the middle ranges, between 30–39% and 40–49% as their maximum similarity score from all comparisons, and only a few get high similarity scores.

The Gnuplot cspline created a little shoulder formation in the right hand side of the curve. It almost seems to magically identify the area where the plagiarizers are expected to fall in the curve (above 60% similarity), so I kept it, but it is more likely just an artifact of these particular data points and the cspline.

Chapter 8

Conclusion and further work

In this final chapter I summarize the tasks I managed to fulfill out of those I started with and those that still need further work. I also look at the experiences I gained along the way, both positive and negative, the contribution this work represents, and present some recommendations for further work.

Note: In mid-2012, the Antics application can be downloaded for Ifi-related purposes at `http://josek.at.ifi.uio.no/antics` *(password:* `secrt`*). See section 6.4 (p. 54) for more information.*

Table 7.1 (p. 56) provides very convincing evidence that this work has had a positive effect at the department. From the beginning of these efforts until now the number of clear plagiarism cases appears to have diminished to less than 1/4 of what it was (20% to 5%). It is clear this work has helped address an important issue at the department.

The present thesis is my attempt to create a system to enable this work to continue. The result is an application that works and can be used for this purpose, although it still needs a lot of polishing before it is ready for a wider audience.

I tried to make this application a meeting point between two other systems, each with its own view of the world, and learned that this is not always an easy task to accomplish in a short period of time. Integration with the Joly database was achieved, but integration with the Devilry system is only in the beginning stages.

8.1 Experiences and missteps

One **positive** outcome of this work is certainly that I have learnt a lot about various technologies and tools, including an attractive application framework (Django) which however has some unexpected idiosyncracies, database accessing and ORM data modeling, Web UI programming, I re-learnt Python and LaTeX. I also learnt about the plagiarism detection literature and writing theses, and I learnt something about what my earlier fellow master candidates had investigated and recommended.

Some of the **missteps** I also tried along the way (hopefully) taught me valuable lessons:

- I found out CGI scripting was not a very developer-friendly technology on which to build complex web applications;
- I didn't quite sort out the Devilry inclusion task and was only able to provide a step along the way;

- I may have gotten somewhat caught in the hype of Django, and at least one of the choices this lead me to, regarding the inelegant file system data sharing between Python and Java, could probably have been solved better by enabling the Java side to communicate with the Antics database directly;
- I concentrated maybe too much on the server side and neglected client-side technologies like Javascript, jQuery, or Ajax which could have made the UIs more user friendly and responsive (the current forms often require several steps which could have been reduced in number with the aforementioned technologies);
- I had forgotten that the Diff-algorithm I had been used was GPL-licensed code and didn't have time to change it to a more flexible solution;
- I realised my algorithm idea was not entirely new, and instead resembled very much a trio from the 1990s (the YAPs);
- I also didn't calculate enough time to polish my application, and some of the functions remain buggy or unfinished, though I'll try to improve them in the days to come;
- I also rushed some of the text in this thesis probably leaving some uncorrected technical and grammatical errors.

8.2 Recommended future work

Going forward, Ifi now has three related systems, two with support for plagiarism detection (Joly and Antics), and two with functionality for student delivery of assignments (Devilry and Joly). Each has its strengths and weaknesses. The larger Devilry system lacks anti-cheating tools, but could be integrated with one. The other (Joly) has newly been restored to hopefully its former better self (Darisiro, 2012), but it has been noted that it could benefit from an assessment registration system and some other desirable functionality; and the last (Antics), is specialized in one of the areas, but lacks support for other functions.

It remains to be seen how well Joly will perform now. If it works well, some of its features for plagiarism detection, could compete with Antics in some courses. Technically, Joly is a cleaner design, and runs faster, but the two systems can also complement each other.

However, at the moment I see the most potential in approaches that consider Devilry. It also has its share of problems, but if these are sorted out,—and work on this is going on,—it could be developed into an even better system. The Joly data model is well designed for its purpose, but this purpose appears to be a little limited and doesn't really compete with Devilry when it comes to some other courses besides INF1000.

At the moment Antics hasn't even started its first real trial, so it remains to be seen how much staying power it has. I think it is important to continue the work on plagiarism reduction, and one tool or the other should be continued. Compared to Joly, Antics has a few advantages for certain courses, but both could be improved. The idea behind Antics of allowing a teacher to import any set of solutions easily and check them for similarities is very useful function, and is probably a little better in Antics at the moment.

My recommendation at the moment is to let the three systems battle it out a little bit and see which is most used or preferred by the intended users. Those who want to continue with similar work will probably have a preference for one or the other and should be allowed to push them forward, resources permitting, until a clear winner or two begin to stand out.

The next person interested in improving Antics, should consider completing the tasks in section 1.2 (p. 9), along with trying out changing the reordering phase of PRD to use static execution order, as YAP–YAP3. That could improve detection results that today can get lower similarity scores than they deserve because of the block mismatch problem I discussed in section 4.4.3

(p. 29). This doesn't appear to be a big problem in the introductory courses but I've noticed it in a few cases. These were often still detected, but with lower scores than they should have gotten. Another way to improve this problem that could be investigated is changing the Diff phase of PRD into Greedy-String-Tiling, or Local Alignment. This will make the algorithm more complex and slower but probably probably not too much. It should be interesting to try, especially for plagiarism detection in more advanced courses.

Bibliography

Z. A. Al-Khanjari, J. A. Fiaidhi, R. A. Al-Hinai, and N. S. Kutti. "PlagDetect: a Java programming plagiarism detection tool". *ACM Inroads*, 2010. Volume 1, Issue 4.

W. J. Bowers. "Student Dishonesty and its Control in Colleges". *Bureau of Applied Social Research, Columbia University*, 1964.

F. Culwin, A. McLeod, and T. Lancaster. "Source Code Plagiarism in UK HE Computing Schools: Issues, Attitudes, and Tools". *Technical Report SBU-CISM-01-02*, 2001. South Bank University.

S. Darisiro. "The Recovery of an Old Lost System". *Master thesis, Ifi, UiO*, May 2012.

Django. Django Software Foundation: "Django documentation". 2012. djangoproject.com.

M. Freire and M. Cebrian. "Design of the AC academic plagiarism detection system". *Technical report, Escuela Politecnica Superior, Universidad Autonoma de Madrid*, Nov 2008. http://tangow.ii.uam.es/mfreire/pub/ac-whitepaper-08.pdf.

A. Holovaty and J. Kaplan-Moss. *"The Definitive Guide to Django – Web Development Done Right"*. Apress, second edition, 2009.

C. J. Hwang and D. E. Gibson. "Using an Effective Grading Method for Preventing Plagiarism of Programming Assignments". *ACM SIGCSE Bulletin 14:1*, Feb 1982. pp. 50-59.

P. Ipeirotis. "Why I will never pursue cheating again". *http://www.behind-the-enemy-lines.com/-2011/07/why-i-will-never-pursue-cheating-again.html*, Jul 2011.

D. C. Kar. "Detection of plagiarism in computer programming assignments". *Journal of Computing Sciences in Colleges*, Mar 2000. Volume 15, Issue 3.

C. K. Kielland. "Metoder for likhetsvurdering av innleverte obligatoriske oppgaver i Java". *Master thesis, Ifi, UiO*, Jul 2006.

H. P. Langtangen. *"Python Scripting for Computational Science"*. Springer, 3rd ed., 2nd printing edition, 2009.

O. S. Ligaarden. "Detection of plagiarism in computer programming using abstract syntax trees.". *Master thesis, Ifi, UiO*, Nov 2007.

J.-S. Lim, J.-H. Ji, H.-G. Cho, and G. Woo. "Plagiarism detection among source codes using adaptive local alignment of keywords". *ACM ICUIMC '11: Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication*, Feb 2011.

A. Maus and O. C. Lingjærde. "Lasting effects of automatic plagiarism detection in an introductory course in programming". *INTED2009 Proceedings*, pages 233–240, Mar 2009.

D. L. McCabe and L. K. Trevino. "Academic Dishonesty: Honor Codes and Other Contextual Influences". *The Journal of Higher Education 64, no. 5*, Sep-Oct 1993.

D. L. McCabe, L. K. Trevino, and D. L. Butterfield. "Honor Code and Other Contextual Influences on Academic Integrity: A Replication and Extension to Modified Honor Code Settings". *Research in Higher Education 43, no. 3*, Jun 2002.

E. Myers. "An O(ND) Difference Algorithm and its variations". *Algorithmica Vol. 1 No. 2*, 1986. p 251.

R. Sears, C. V. Ingen, and J. Gray. "To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem". *Tech report, Microsoft Research*, Apr 2006.

J. Sheard, A. Carbone, and M. Dick. "Determination of factors which impact on IT students' propensity to cheat". *ACE '03: Proceedings of the fifth Australasian conference on Computing education*, Jan 2003. Volume 20.

T. Steensen and H. Vibekk. "DHIS and Joly: two distributed systems under development: design and technology". *Master thesis, Ifi, UiO*, May 2006.

P. Vamplew and J. Dermoudy. "An anti-plagiarism editor for software development courses". *ACE '05: Proceedings of the 7th Australasian conference on Computing education*, 2005. Volume 42.

K. Verco and M. Wise. "Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Couting Systems". *Proceedings of 1st Australian Conference on Computer Science Education, Sydney*, Jul 1996.

D. Vogts. "Plagiarising of source code by novice programmers a "cry for help"? *SAICSIT '09: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, Oct 2009.

G. Whale. "Identification of Program Similarity in Large Populations". *The Computer Journal 33:2*, 1990. pp. 140-146.

M. J. Wise. "YAP3: Improved detection of similarities in computer program and other texts". *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, Mar 1996. Volume 28, Issue 1.

J. Zobel. "Uni Cheats Racket: A Case Study in Plagirism Investigation". *Proc. Sixth Australasian Computing Education Conference*, 2004. Dunedin, New Zealand.

J. Zobel and M. Hamilton. "Managing Student Plagiarism in Large Academc Departments". *Australian Universities Review*, 2002. 45(2): 119–126.