UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Clone Detection for Student Programming Exercises

Bachelor-Arbeit

zur Erlangung des Grades eines
Bachelor of Science
im Studiengang Informatik

vorgelegt von

Michael Lellmann

Betreuer: Prof. Dr. R. Lämmel, Institut für Softwaretechnik
Erstgutachter: Prof. Dr. R. Lämmel, Institut für Softwaretechnik
Zweitgutachter: Andrei Varanovich, Institut für Softwaretechnik

Koblenz, im Juli 2011

Abstract

This bachelor thesis is about plagiarism in student programming courses. The target of this work is to define prerequisites for a program that should find code clones in an effective way. For that we will first define what a code clone is and afterwards define different types of clones. Following that we will have a look at those clone types, especially how we can detect them and how often we have to deal with them in our environment.

The second part of this work deals with the implementation of a program, that will fulfil the prerequisites defined before. We will have a look at the problems that arose while implementing it. Finally, we will do some empiric test on past and actual programming courses to show, that the program does work and yields useful results. That empiric work also should show, that the special values for different parameters in our program have been chosen in a well thought way.

This work also contains a documentation for the configuration file that is used to control the program.

Contents

Chapter 1

Introduction

For actual programming courses, students have to program little programs as their homework. This is normally done in small teams, and those exercises are then checked and given points by some course assistants. To be allowed to participate in an exam, the students have to get a specified number of points, something like 50

There have been problems with students copying code from other students. This is not welcome, because they should do the homework on their own (as a team) and not just copy it from another team. The other problem is, that they will not learn much from copying just the code, thus lowering the quality of education. Plagiarism is not welcome in general, because one should not say that he did the work, if he just copied the code from another student. Therefore, we are looking for an automatic tool, that has the ability to find clones in student solutions to support the course assistants.

We will start with a brief look at that clones are and define a few basic terms, so that we have the same terminology.

Afterwards, we try to work out important cornerstones for our program, what it has to be able to achieve and what the environment for using the program is. After that, we will look at the actual implementation of the program, what the problems were, how we solved them and discuss important program constants in deep. Afterwards we will have a look at how the program might be improved in future work.

Finally we will discuss, why we did not use existing tools, as there exist some.

There will be appendixes, one for documenting the configuration file of the program. Afterwards we will provide a few tables with empiric data on past courses. A least, we will have a short look at the performance of the program.

Chapter 2

Basics for clone detection

In this chapter we will have a brief look at the vocabulary, which is normally used in literature. We will use the same terminology as in [1].

The first term is the *Code Fragment.* As a code fragment we will consider any number of lines of a program. It can either be a function, a begin-end-block or any number of simple statements.

As a *Code Clone* we will consider two code fragments, what are similar in a before defined way.

At last, we will define different *Clone Types.* They are distinct in the steps that are needed to transform one code fragment into another to get a clone.

Type-1 Both code fragments will only differ in whitespace, layout and comments.

Type-2 The two code fragments may vary in whitespace, layout, comments, identifiers and literals.

Type-3 The two code fragments have further modifications then type-2 clones like added, removed or reordered statements.

Type-4 Two code fragments, that will do the same computations, but may use completely different syntactic constructs.

Chapter 3

Problem Analysis

In this chapter, we will have a detailed look at the needs of the program, what it should be able to achieve and which problems might arise while implementing the program. We will also have a short look at what the environment is, in that the program has to run. We will close this chapter with a discussion on what to concentrate in this work.

3.1 Requirements for the clone detection

The goal of the program is, to find as many copied code fragments as possible. This should be done in a way, that is fast (due to the size of the courses) and does not produce to many false positives. Too many false positives would mean a lot of extra work for the person in charge of the course, because he has to check all results.

We will start the problem analysis with a discussion, why a student team does copy code from another student team. Where are mainly two reasons. The first reason is, that the student team is unable to solve the problem, because they either did not visit the course or they just do not understand it. The other reason is, they are too lazy to do the homework.

As the exercises are in general not that big, if the team is too lazy to solve the problem on their own, they will most likely also not change much of the program if they copied it from another team. We can expect changes to the layout (automatic layout) and maybe the removing (or changing) of comments. Larger changes normally also means, that they have to invest time into them. In the case, that the student team is too lazy to solve the problem on their own, we expect Type-1 and Type-2 clones. Type-2 clones might arise from their goal, to hide the copying of the code with simple methods. The actual IDEs for Java e.g. do have an easy to use mechanism to rename variables. And, as variable renaming is an easy way to make the program look different to a normal programmer or course assistant, they might want to do it. In some rare circumstances we also might get some Type-3 clones, because they reordered statements (or rarely added some useless lines), but those changes will in most cases not be significant.

If, on the other hand, the students are unable to solve the problem on their own and therefore copy the solution from another team, we can also only expect Type-1 and Type-2 clones. Type-3 clones are not of interest to us, because if the student adds code, if he wants to obfuscate the cloning, the correctors will find that code. If he only does small changes, the program will still be a Type-2 clones for the most parts of the code fragment, and so our clone detection algorithm should find them. If the student is able to produce a Type-4 clone, he most likely understood the program. Due to the work needed, to change the structure of the program, it is relative unlikely that a student will produce a Type-4 clones to hide the copying of code.

But the main reason behind not looking for Type-4 clones, is that a Type-4 clone does normally not result from copying a code fragment. Type-4 clones will always happen, if a piece of code has to solve the same problem. And as all of the student teams have to solve the same problem (in some cases, there exist two or three exercises, from which they have to solve one, but still many teams will solve one exercise), we will have a lot of solutions which will do the exact same work, and so all correct solutions of the same exercise are Type-4 clones. Because of this, we will not be able to take the best (or state of the art) clone detection algorithm and use it on the solutions, because it may result in many false positives.

Another important matter is, that, to keep the exercises simple, the students are given a project, from which they should start. In such a case, the students only have to change code in a few files, and in them they only have to implement some functions. From this it results, that the program must have the ability to find unchanged code fragments and ignore them in the process of detecting clones. If the program will not have such an ability, we will get so many false positives, that we results from the program to not help a bit, because it most likely will tell us, that every solution is a clone to every other solution.

3.2 Program environment description

Let us now have a look at the environment in that the program should work. The students will be in teams, normally consisting of around 3 students (the number does not really matter). Each team has an unique name. This team name is used as a subdirectory of a svn-repository, in which the students have to check in their solutions. As there is normally no easy to access list of team names around, the program should be able to find the teams on his own, which could be done by scanning the root directory of the svn-repository. There might be a number of folders, that are used for auditing the course, so there should be a way, to ignore specific folders.

If we have found the teams, we have to find the solution for an exercise. In the team folders, there are normally 3 folders. One of which is normally called *solutions*, and, as this folder is created by the course administration, this is always there. In that solution folders, the students have to put their solutions. They may just upload a folder containing the solution or a zip-file, which contains the solution. The folder or the zip-file must have a reference to the exercise which they solve. It is typically emphasized, that the students call them after a schema like *Blatt01*, *Blatt 02*, ... and so on. Typically, not all students will do it this way, even if the teams will be advised to do so. Therefore the finding of the solution has to deal with the issue, that the name for the solution is varying.

3.3 What to focus on

After discussing the general requirements for finding the clones, let us abstract the requirements. As we have seen, the most important task is, to find Type-1 and Type-2 clones. There might be a few Type-3 clones, but they should not be our main target. On the other hand, we must make sure, that our program will not produce to many false positives due to the conditions the program has to work in. This especially means, that we must not try to find Type-4 clones. We also have to make sure, that we can constrain the program to parts of the code.

Chapter 4

Problem solving

In this chapter, we will have a closer look at how we started and what we implemented. We will also discuss program parameters, that we will use throughout the program, like the similarity threshold two code fragments must have to be considered a clone. But we first start with the issue of finding the files and making the program work with all needed files.

4.1 Finding the files

First, we implemented how the program find the files. There is an extra part of the program, that will scan a given directory for folders, which represents the student teams. As there are some folder, that are used for auditing the course, like *allgroups*, *staff* or that are utility folders for other programs that are used during the course, like *.svn*, there has been implemented a way, to filter out folders. Folders might be filtered out either by their full name, or a regular expression that matches the folder name. The number of filters is not limited to allow the program to be used as in many as possible environments.

After we have found the teams, we will scan each team folder for the solution of that team. First, the folder in that the solutions has to be uploaded to will be located. There is a parameter, that will name this folder. If the folder can not be found, we will inform the user of the program, that a team does not have the specified folder, so appropriate actions can be taken.

In the solution folder, we will look for a folder or zip file with a given name. The name of the solution may be given by its full name or a regular expression. For finding the solution of a specific exercise, it is preferred use a regular expression, that will check, if a name ends with the number of the exercise. For exercise 3, this can be achieved by using the following expression: ".*[ˆ1−9]3". The dot matches every char, so that expression means, that the word may start with as many random chars as needed, but it has to end with a 3 that is not preceded by any other number, except maybe 0. This actually makes it possible, to distinct between *Blatt3* and *Blatt13*, as both refer to different exercises, but both end with 3 (the above regular expression will only find the first one). If we did not find a folder with that name, we try to look for a zip file, whose filename (without the ending) does match the given parameter. Should there be more than one matching folder, only the first one is found and used in the program further on. If the student tries to obfuscate in such a way, the course assistants should note that while correcting the exercises and they can than take appropriate actions.

As the students may also check in zip-files, it was important to abstract from physical files, as we did not want to unzip all archives we find. In our program we will use a FileDescription object whenever we want to use a file. A FileDescription object may point to a physical file (in the file system) or to a file in a zip-archive.

If we do not find any files in the solution folder, we will assume, that the student team did not solve that exercise and remove it from the list of teams. This normally happens at the end of the course, if the student team is qualified for the exam.

4.2 The Diff algorithm

For our clone detection, we will calculate the minimal edit script. The minimal edit script is a cycle of actions, that exists of the three operations add, remove and change, that has the lowest amount of such actions too transform a sequence A into a sequence B. Out of that minimal edit script, it is possible to calculate a value for the similarity of two sequences.

The problem can be solved in almost linear complexity, as has been shown in [3]. The algorithm there has been developed for files specially, but it can easily be adapted for any ordered collection of objects, on those objects there must only be a natural ordering defined. For our program, we will use the implementation provided by DiffJ ([4]).

4.3 Diff on whole files - FileDiff

We first implemented a simple FileDiff algorithm. There are two main reasons why we did this. The first is, that it is really fast and finds those students, that just copied the code from another team without modifying it. While one might think, students will try to obfuscate, that is not done in all cases. The second and more important reason is, that a simple FileDiff will always work. If the solution does not compile, and methods working on an AST could not be applicable, the FileDiff algorithm still has a chance of finding copied code.

As simple as FileDiff is, it is as easy to trick out. In our implementation, we are comparing whole lines of a file. We talked about pretty printing the code before doing the FileDiff, but we decided against it, because FileDiff should not be the main clone detection algorithm used. Also, pretty printing will not work well, if the code is not correct, so we could not use it in such situations. On the other side, we do very simple pretty printing operations. But that only includes removing of whitespace at the beginning and end of a line. We will also remove empty lines from the collection containing the file content. Those operations only have minimal effect on the clone detection process, thought they might improve it a little bit, because indention style no longer matters, though whitespace using inside expressions still does.

We also thought about removing comments from the code, but this has not been implemented. First off, comments carry information, that, if it is not changed, will add to the clone detection process in a positive way as increasing the similarity of two code fragments. And as we had a token based clone detection algorithm in mind, removing comments does not yield any advantage in the overall clone detection process while the disadvantage would have stayed. First off, we can not guarantee, that the program has correct syntax. Therefore we must have had been very careful in the process of removing comments. Further on, removing stuff from the file also lowers the data the clone detection process has to compare, and therefore modifying the result.

After loading the files, we will first remove code, that is given in a defined starting solution. The first solution was, to remove files for which the FileDiff algorithm would result in a 100% clone. This however yields a few problems, as, if the students have to make only small changes in a big file, there would be a overall higher similarity between student solutions, resulting in possible more false positives.

The second attempt and actually implemented one is, that we will remove the lines of code that are in the starting solution. This is done in the way, that we first look for all files that have the same name. Afterwards we will calculate the difference between two files with the diff algorithm. The result of that calculation is then used to determine, to find the unchanged lines, which will be removed from the clone detection process. We constrained ourselves to files of the same name, because, if we would compare them to too many files, information will be lost. In most Java files, there are } on a single line, those would all be removed from the source file, which results in an information loss. As the editing of the students normally is to change a function or implement new functions to an already existing class, the filename is rarely if ever changed.

After implementing this, a problem occurred. In one exercise, the students had to add a function to an interface. Afterwards they had to implement that function in some derived classes. This resulted in the problem, that we had a lot of one line files, which contained the same function (as the function to be implemented was given). To remove such problems that occur from small files, a check for the file size has been implemented. If the file does have to few lines, in the standard configuration 10 lines, that file will not be considered in the clone detection process. This value can be changed with the configuration file.

The clone detection process works in the way, that we will compare one file of a solution A with all files of another solution B. The process is afterwards repeated for all the other files of solution A. After all files are checked, we will continue with two other solutions until all solutions have been checked against each other.

Let us now have a closer look at the clone detection process. If we have a File A and a File B, which are checked for clones, we will first calculate the Difference of the lines those files contain with the Diff algorithm. The difference of those lines is then used to calculate a similarity between both files. This is done, that we will count the number of lines that are existent in each of the files. Afterwards we will calculate the percental value of unchanged to total lines. This will be used as the similarity of the two files. There has been proposed another way of calculation the similarity in the way, that we would add up the characters of the lines instead of just the lines.

```
int foo1()
{
  int x;
  return /*something very long*/;
}

int foo2()
{
  int y;
  return /*something very long*/;
}
```

As seen in the above code, the two functions *foo1* and *foo2* differ only in their second line. It does not matter, how long the line with the return statement gets, the similarity would always be 0.8. That means, that short lines have the same weighting as long lines. So it seems logically, to use the characters of as line for calculating the similarity. That however results in other problems:

```
int foo1()
{
  return /*some long calculation*/ +1;
}

int foo2()
{
  return /*some long calculation*/ + 1;
}
```

In this example, the lines with the return statement differ. (There is an extra space after the plus in *foo2*.) As we are only comparing whole lines, the Diff algorithm would result in telling us, those lines will be different. So, in a character based approach those lines would be weighted much more than the rest of the code, although they only differ in one char.

As seen in the above two examples, both ways of counting have disadvantages. We decided, to use the line counting approach, because it is faster and easier to implement. The problems described above can only be solved with pretty printing the code prior to calculating the difference. Those problems also do not occur in a token based clone detection approach, so we do not try to avoid them in this FileDiff, because it is not really worth the effort needed.

The last problem we had was, what should be the threshold for the similarity for when two files are considered code clones. This question is not easy to answer, because it depends on the circumstances. On a general note, there are no Java code files, that will have a similarity of 0, because the lines that contain } will be counted in the process of the clone detection. Also many other lines tend to be the same, like import statements. If both of the files are coded from scratch, a value of 0.75 has shown to provide good results[1].

With all the problems of the FileDiff algorithm, it can not be more than a rough detection mechanism. However, if we find a clone with the FileDiff with a very high similarity, the changes that a team did copy a whole file and not tried to obfuscate it.

4.4 A token based approach - FunctionDiff

As a purely line based approach does not really help us, we tried another approach with a token based variant. As we still have to make sure, that, if the students start from an existing project, it does not result in to many false positives, we tried to find a way to sort out already existing code. The basic idea behind the sorting out is, that the students always have to implement whole functions. Therefore, the easiest way to remove existing code is, to remove unchanged functions, because the code of functions can easily be extracted.

For the token based approach, we will constrain ourself further, as we will only compare code in functions. This does exclude some information, but that is mainly import statements and class fields. Import statements are normally generated from an IDE and as the exercises have to solve the same problems or have to work on the same data structure, those import statements do not differ that much in general. Class fields also do not carry much information. With actual IDEs it is very easy to rename them. Therefore, it has been decided, to just focus on the functions and ignore the other parts.

First, we will generate an abstract syntax tree (AST) for each file. If this has been done, we will extract all functions from the AST. Those functions are then compared to the functions from a given starting solution. If we find an unchanged function, we will ignore it further on in the clone detection process.

[1]For more details on how to determine a good threshold, refer to appendix B.

As in Java encapsulation is very important, that will result in many small functions as seen below:

```
class foo
{
        private int x;

        public int getX()
        {
                return x;
        }

        public void setX(int x)
        {
                this.x = x;
        }
}
```

The class *foo* has one member field for which two methods exist, one to set it and on to get the value of it. Both of them are very small. Those small functions would result in many false positives, so it has been decided to remove small functions. The functions above have 5 and 8 tokens respectively. But, as we want to remove all easy functions, we looked for a function, that will set a value after checking it for null, and if it is null, that function will throw an exception. This is still an easy function that will often occur in programs.

```
public void setTesting(testing test)
{
  if (test == null)
  {
    throw new IllegalArgumentException("foo");
  }

  this.test = test;
}
```

The *setTesting* functions consists of a total of 23 tokens. A good value for minimum function length therefore is a value of 25, giving us some extra tokens for little modifications. That value can be changed in the configuration file. If no removing of short functions is wished, setting this value to -1 will remove that ability.

The other problem we had to decide was, if we would use a token or node based way for calculating the similarity. As we are using PMD, an open source java parser, that will generate an AST, we used a token based approach. The problem with a node based approach would have been, that we must have defined a natural ordering for the different nodes, because without such an ordering, we would not have been able to use the Diff algorithm. Defining this ordering might be easy on the paper, but with the extensive use of inheritance, as done in PMD, that would result in a not that easy to understand part of the program.

The main clone detection process will again be done by the Diff algorithm only now calculating the smallest difference between two lists of tokens. For calculating the similarity of two code functions, we will calculate the number of tokens that occur in both functions, and afterwards will calculate the percentage of tokens occurring in both functions to the overall number of tokens used in both functions. For the threshold, a value of 0.9 has shown to provide good results. The value has to be set higher than for FileDiff, because there are many tokens that will occur in every function and therefore there will always be a certain similarity between two functions[2].

This token based approach can also be used to find variable renaming. We did not implement some kind of variable name mapping, because that could easily be passed by reordering the variables. To provide good results, it would have to check all possible mappings for what will result in the greatest similarity. We used a simpler way, which included an option that would allow the program to ignore the name of identifiers. This results in the program not being able to be obfuscated by variable renaming. But on the other side, a lot of information is lost too. As now all identifiers match, only the structure of the program does count. As the overall similarity will go up, it is advised to set the threshold higher to get not too many false positives.

[2]More information can be found in appendix C.

Chapter 5

Implementation

In this chapter, we will have a short look at the main components of the program, what their functionality is and how they provide it. We will also have a brief look at the third party tools we used.

5.1 Main program components

There are basically three main components of the program. The first is the input component. It consist mainly of the abstract class *FileDescription*. The main use for this class is the abstraction from the Java file object. The FileDescription class provides a function, that will provide a Java InputStream as the return value. This InputStream is used in all the functionality for reading the files and it is the only part that needs adjustment, if a new subclass of FileDescription would be implemented that e.g. will point to a file on the Internet.

The next main component is the clone detection mechanism. The two used algorithm have been implemented as functions of the *CloneDetection* class, although the main work is done in third party tools described below. If a clone is found, it is posted to an instance of an output class. The output class will store these clones as an object and after the clone detection process is finished, they can be printed with the *print* method. The output class will sort the clones so that the most obvious ones are printed before the other ones.

The CloneDetection class also functions as the main part of the program. It will initiate the loading of the options and files, the clone detection process and finally the output of the program.

5.2 Third party tools

DiffJ

The DiffJ[1] tool provides a way to compare two Java files for similarities. It was first planned, to use it for doing the main work, but the output format was hard to use in determining the similarity between two files, as the program output is plain text, but some of the code has been reused and changed, so that the output will be more helpful to us. DiffJ uses PMD to parse Java files and provides a lot of useful helper functions, to make the work with PMD easier.

pmd

PMD[2] is a tool for Java, that will look for potential problems in the source code. We are not interested in that functionality but for the Abstract Syntax Tree (AST) it can generate. This AST is used in the token based clone detection process we use in our program. There are other java parsers out there, but we used PMD mainly because it was used by DiffJ which we first wanted to use as our main token based clone detection program.

[1]http://www.incava.org/projects/1042574828
[2]http://pmd.sourceforge.net/

Chapter 6

Literature

Of course, there do exist a lot of tools, that are designed to find clones. But after taking a closer look at them, they did not really fit the special needs for detecting clones in student programming exercises. In [1] several clone detection algorithms are compared to each other, also in the way which clones they are able to detect. From that research, we can say that graph and metrics based algorithms have the ability to detect Type-4 clones. That ability might be low, but it does exist and might result in many false positives, especially if those tools will get better with the time. Only Token-based, Tree-based and some Text-based methods do not have the ability to detect Type-4 clones. After having a closer look at some of the programs of that type, it got clear, that clone detection algorithms will not be that helpful for us.

Clone Detection algorithm in general do not have the ability to ignore parts of the code, therefore lowering their usefulness to us[1]. As this is one the most important things, we looked for tools that are specialized in finding plagiarism in student programming solutions. In [2] they did a comparison of different tools for that task. They first sorted out the tools, removing those from the list, that had no documentation and other weaknesses.

We will also constrain ourselfs to those 5 tools. The first tool is JPlag[2] ([5]). JPlag is a web based tool, there the actual plagiarism detection is done

[1]This however could be implemented, if the algorithm is open source. Though, if the algorithm is open source, we could also implement it into our program. That would have the advantage, that we can have a self defined output format, which might be used in future extension to the course ware system.

[2]https://www.ipd.uni-karlsruhe.de/jplag/

on a server. The files are uploaded to a server and after the job has been done, you will get informed about it and get a result. Allthough, it seems that the results come in reliable, as a advisor over a course you do not have any influence as to when you will get the results. Also, the uploading of data to a foreign server is generally a task, that should be thought of before. As of these problems, we only wanted to have a tool that has the ability to run on a local machine, therefore JPlag does not suit us.

MOSS[3] (described more closely in [6]) is also an online tool for plagiarism detection in software programming courses, therefore it is out too.

Let us now have a look at the last three tools. They can all be run localy, so the online restriction is not a problem for them. The first of those tools is Marble. As it is also described in [2] it is really hard to find any information about Marble in english. The tool is closed source too, which is a cause against it, because we want to be able to make improvements or changes to the program if they are needed. Therefore, Marble is also not the tool of choice.

The last two tools, Plaggie[4] and SIM are both open source and can be run on a local machine. So they meet all the requirements we had so far. As discussed in the paper ([2], page 19), Plaggie did a relativ bad job, that resulted in some false positives to be higher than the real clones. Plaggie would have been needed to be improved a lot to make it useful, but in the actual state its result are not that useful to us.

SIM on the other side did not provide a way to filter out code, from that the students had to start. Because of this, we would also have needed a lot of work to make SIM work to our satisfaction.

As described in the conclusion of the before mentioned paper ([2]), these tools are vulnerable to many small changes. Therefore, using them might have given us some advantages, but would also have constrained us to that tool. The approach we did with our program was to implement two different clone detection mechanism. It might be a consideration to integrate Plaggie and SIM into the program to get more results and afterwards combine the results of all the used clone detection algorithms. This would certainly lower the amount of false positives and would make the program harder to circumvent.

[3]http://theory.stanford.edu/ aiken/moss/

[4]http://www.cs.hut.fi/Software/Plaggie/

Chapter 7

Future work

This program has shown to function and found some plagiarism in this years P̈rogramming Techniques and Technologiesc̈ourse. Nonetheless, this program still is in a very early state. It can be improved in many way, either its clone detection mechanism, the usability of the tool and the integration in existing tools for courses.

The biggest step could be an integration in an automatic course ware system, that will check the programs with unit tests, and if they produce the right results (and are not overly complicated by some metrics) would just give the points to the students without interaction of a course assistant. With such an automatic tool, it would be possible, to allow the removing of teams, so that every student has to do the homework on his own. This is not doable int the moment because of the work involved. If every student has to do the homework, the overall quality of the course would go up, because it is actually common, that some team members do not contribute much to the team work.

For such a tool, it would also be a good idea, to allow for more ways to access files. If the program could work directly on a svn-directory, the need for a separate copy of the student home work would no longer be needed, therefore removing issues with maintaining an up to date copy on a local hard drive. The program could be programmed in a way, that it would fetch a specific version of a file (actually, the version of the file as it existed on the dead line for that exercise).

While we found some plagiarism in courses, our clone detection has some weaknesses. It could be a goal, to implement more and better ways of detecting clones. The results of those clone detection algorithms could afterwards be used, to determine in a more save way, if it is a clone or not. This could be achieved in a way, that we will say two code fragments are clones if e.g. three quarter of the detection algorithms at least say, it is a clone. This would lower the chance of getting a false positive.

Bibliography

[1] Chancal K. Roy, James R. Cordy, Rainer Koschke. *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach* in: Science of Computer Programming 74 (2000)

[2] Jurriaan Hage, Peter Rademaker, Nike van Vugt, *A comparison of plagiarism detection tools* http://www.cs.uu.nl/research/techreps/repo/CS-2010/2010-015.pdf (2010)

[3] Eugene W. Myers, *An O(ND) Difference Algorithm and Its Variations* (1986), cached download from http://xmailserver.org/diff2.pdf, July 2011

[4] incava.org, *Open-source software, primarily for Linux, Ruby and Java development.* DiffJ: http://www.incava.org/projects/1042574828, version used: 1.1.4, released 26.04.2009

[5] Lutz Prechelt, Guido Malpohl, Michael Phlippsen, *JPlag: Finding plagiarisms among a set of programs*, a technical report about JPlag from the developers. (2000) - research report

[6] Saul Schleimer, Daniel S. WilkersonDivision, Alex Aiken, *Winnowing: Local Algorithms for Document Fingerprinting*, found on-line at http://theory.stanford.edu/ aiken/publications/papers/sigmod03.pdf (2003) - research report

[7] Aleksi Ahtiainen, Sami Surakka, Mikko Rahikainen, *Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises*, In: Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006 (2006) - research report

[8] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, Amit Seker, *Shared Information and Program Plagiarism Detection*, found on-line at

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.76&rep=rep1&type=pdf
- research report

Appendix A

Documentation for the configuration file

To use this program, you have to name a configuration file as the parameter of the program. If there is no given parameter, the program will look for a file called *config.xml* in the actual runtime environment directory. If it cannot find the file, the program will close with an error.

The program uses a xml-file for the configuration of the clone detection process. The *clone_detection_options*-element does function as the root element of the file.

The *base_dir*-element points to the root directory there the team folders have to be located. The path can be absolute or relative to the actual runtime directory, from which the program is started.

The *groups*-element groups parameters for finding the student teams, actually it only accepts *ignore*-elements. There can be as many as needed ignore options. The ignore option takes a value, which will be the name of the directory that will be ignored. It can also be extended by the regex="true" option, which means that the value will be treated as a Java regular expression[1].

The *solutions*-element group does describe the solution folder. The first element is the *search_dir*, that is the sub folder of the team folder, in that the students have to put their solutions. The *find*-element describes the folder or zip file (without the .zip file ending) in that the solution is located. It can be either a set value or a regex expression. It is also possible to ignore files

[1]It uses the Java regex class. For a documentation about the allowed syntax check out the documentation on-line at http://download.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html

```
<clone_detection_options>
        <base_dir>path to the root directory</base_dir>

        <groups>
          <ignore value=".svn" />
        </groups>

        <solutions>
          <search_dir>solutions</search_dir>
          <find value=".*[^1-9]6" regex="true" />
          <ignore value=".svn" />
        </solutions>

        <start_solution>
          <dir>path to the start solution</dir>
          <ignore value="stuff.*" regex="true"/>
        </start_solution>

        <option name="thresholdFunctionDiff" value=".9" />
</clone_detection_options>
```

Figure A.1: A simple example configuration file for the program

and directories using the before mentioned *ignore*-element.

The *start_solution*-element is used to describe the solution the students have to start with. If that element misses, it will be assumed that the students had to start from scratch. The *dir*-element contains the path to the directory that holds the code. It can either be an absolute or a relative path to the current runtime directory. It is not possible, to have more than one project that the students had to start with. This however can be circumvented in the way, that the different starting projects are located in a common directory which then is designed to be the starting project. As the program will search recursively for files, it will find all files of all the projects. Again it is possible to ignore unwanted files and directories.

The last element is the *option*-element. With that element it is possible to set a named parameter to a user defined value. There are some values that the program will use to modify its behaviour. Those elements are set to standard values internally at the start of the program, so they can also miss in the configuration file. There are currently 4 supported options[2] recognized by the program. The first one is the *thresholdFildDiff* parameter. It is set to 0.75 at the initialization of the program and can be set to any value, but it should be between 0.0 and 1.0. It defines the threshold for when a code fragment is considered a clone[3] for the FileDiff algorithm. A value of 1.0 means, that both files are 100% similar, while 0.0 means there is not a single similar line.

The minimum number of lines a file must have to be considered during the clone detection process can be set with the *minLinesFileDiff* option. Any integer value is approbate. If this functionality is not desired, a value of -1 will result in no files being sorted out due to being to short.

The same options exist for the FunctionDiff part. The threshold for the function diff part can be set with the *thresholdFunctionDiff* option. The parameter *minFunctionLenth* sets the minimum number of tokens a function must contain to be considered in the clone detection process, while a value of -1 does deactivate this functionality if desired.

[2]it is important to note, that the option names are case sensitive

[3]actually, the program uses $<$ for the comparison, so 1.0 can not be used for finding 100% clones.

Appendix B

Empiric data for FileDiff

In this appendix, we will have a look at the data this program calculated using only its FileDiff part. If no other information is given, we will use the standard values for the program parameters, meaning a threshold of 0.75 and a minimum number of lines of 10.

To get a good threshold value for the FileDiff algorithm, the data for a course was analyzed[1]. As seen in table B.1, the lower the threshold is the more clones the program finds, getting us more false positives and therefore increasing the work needed to check them all. What should be the threshold for the program? There is no definite answer to that question. We can only say, that a specific value does yield a good trade off between finding clones and not finding to much false positives. From the table, we can say, that a value of 0.75, meaning three quarter of the code will be copied, will result in good results, that are useful for the course assistants.

Table B.2 shows us, what the average similarity of two programs is. If we compare files with the same name, it is noticeable higher than comparing all files. This is the result of files with the same name normally targeting the same task and wherefore having more in common than general files.

[1]The data here is given for the ”Programmierung” course of 2010. Other courses show the same characteristics.

exercise	1	2	3	4	5	6	7	8	9	10
solutions	19	21	20	19	19	19	18	13	15	15
files	102	102	128	151	144	259	183	173	240	191
0.0	4141	1792	4307	6122	5039	19149	9591	8564	19430	10335
0.1	905	280	445	661	400	1675	579	713	2892	661
0.2	37	150	89	65	39	138	27	36	610	27
0.3	3	94	72	40	14	70	9	21	292	8
0.4	1	58	67	34	7	57	8	19	140	7
0.5	0	24	49	20	6	44	8	19	68	7
0.6	0	10	38	12	6	38	7	17	59	6
0.7	0	1	27	4	2	30	5	14	52	3
0.8	0	0	11	2	1	23	2	10	49	0
0.9	0	0	3	1	1	18	1	8	49	0
1.0	0	0	0	0	0	0	0	0	0	0

Table B.1: Number of clones detected with different threshold values in the ”Programmierung” course of 2010

exercise	1	2	3	4	5	6	7	8	9	10
solutions	19	21	20	19	19	19	18	13	15	15
files	102	102	128	151	144	259	183	173	240	191
all files compared										
file checks	4847	4874	7539	10403	9282	30932	14436	12861	24843	15469
cloned files	0	1	25	4	2	26	5	13	52	3
avg sim(%)	6.1	2.5	3.4	3.5	2.7	3.4	3.2	3.6	5.3	3.4
only files with the same name compared										
file checks	207	535	199	200	154	279	101	90	132	104
cloned files	0	0	13	1	1	21	1	9	50	1
avg sim (%)	11.7	10.0	18.1	11.7	7.1	18.9	7.6	15.3	41.7	8.9

Table B.2: Average similarity of compared files in the ”Programmierung” course of 2010

Appendix C

Empiric data for FunctionDiff

In this appendix, we will have a look at the data this program produced only with its FunctionDiff part. If no other information is given, we will use the standard values for the program parameters, meaning a threshold of 0.9 and a minimum number of tokens for functions of 25.

As it can be seen in table C.1, the average similarity is a lot higher with this token based approach than it was for the FileDiff algorithm. This is due to the fact, that all ‘simple‘ tokens like (,), {, } . . . will match, which is for FileDiff in most times not the case, because of different layout styles. To get not to many false positives, it has shown, that a value of 0.9 will result in good results with not to many false positives.

exercise	1	2	3	4	5	6	7	8	9	10
solutions	19	21	20	19	19	19	18	13	15	15
functions	111	366	309	270	234	432	255	311	310	336
all functions compared										
checks	5605	63270	43333	32762	24320	85561	2778	41806	40805	48033
avg sim(%)	18.4	27.8	21.2	19.2	17.9	17.9	16.5	18.1	18.0	17.1
only functions with the same name compared										
checks	491	2745	1275	1208	436	764	632	451	886	672
avg sim (%)	24.1	66.1	48.7	53.4	32.7	42.8	30.0	40.0	37.7	30.2

Table C.1: Average similarity of compared functions in the ”Programmierung” course of 2010

Appendix D

Performance

In this chapter, we will have a short look at the performance of the program. The computer used for testing had an Intel Core 2 Duo processor with 2.4GHz and it had 2 Gigabyte of main memory (RAM).

For a sample solution existing of around 20 teams, which resulted in around 100 files and 500 functions, the program needed less than a minute to finish looking for clones. This should be sufficient enough to be used in a working environment.

Due to the fact, that we removed multiple reads of files form the hard disk, the program does need quite a lot of memory. The amount of memory used for the above solution was 250Megabyte, which should be alright with the computers having more today.

The runtime of the program is $O(n^2)$ (n is the number of files found) due to the fact, that we have to compare each file to each other, resulting in two loops iterating over each file.