Keeping DRY: A Comprehensive Measure of Modularity Within a Computer Program

J. Hassler Thurston

Abstract

DRY, which stands for "Don't Repeat Yourself", is a principle in software engineering that discourages programmers from writing duplicate code while programming. Since following the DRY principle has many benefits, it is important that it be measured. We present a few heuristics for measuring DRY, mostly followed from results of detecting similarity between two programs. We show how these heuristics lay the foundation for future work, and how they may turn out useful for encouraging amateur programmers to improve their coding style.

1 Introduction

DRY, or "Don't Repeat Yourself", is a programming ideal and standard that urges programmers to avoid duplicating code or structures of code in order to make programming more efficient and readable. It was first formulated by Andrew Hunt and David Thomas in 1999 in their book "The Pragmatic Programmer", in which they state that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [5]. Following DRY means writing more modular code, which by definition means separating each part of a program into different pieces, each piece solving a specific sub-problem. A program that does not follow the DRY principle is termed WET, which some have satirically expanded as "write everything twice" and "we enjoy typing" [2].

As stated by Frederick Brooks in an article from April 1987[4], software engineering will always face four essential difficulties: complexity, conformity, changeability, and invisibility. Following DRY can help alleviate the troubles of complexity and changeability since breaking up code into small pieces makes it easier to understand, and changing code that is already modular is almost by definition easy.

We thus set out to create a measure of "DRYness", so that we can evaluate how modular people's programs are. By creating such a measure, we can not only analyze code in a new, objective way, but we can also try to encourage amateur programmers to reflect on their code design and improve their coding style if necessary.

However, coming up with an objective measure is subject to many difficulties. First, we have searched the literature and have only found a few similar measures described, so we are one of the first to try to objectively define a DRYness measure. Second, once we have a measure of DRYness, it is hard to argue objectivity of the measure, since in some sense the way we measure modularity of code is subject to personal interpretation. Because of these difficulties, we have chosen to take a slightly different approach. We have created several heuristics for measuring the modularity of code, and compare the heuristics by testing and comparing them on a number of different hand-written programs which we feel captivate many cases of modular or non-modular design.

2 Related Work

As stated above, we think we are one of the first ones to come up with a measure of DRYness. There have been some attempts in the past to evaluate program modularity, which for purposes of this paper will be considered the same as program DRYness. Baker et. al.[3] describe an approach to measure effort expended while writing a program in order to determine when program modularity is beneficial. To Baker et. al., effort expended is defined as the ratio of program volume to the "level of implementation", which are both based on counting the number of total and distinct operands and operators in a program. They then use their measure of effort, which they claim measures program modularity accurately, to evaluate over 500 programs to determine when it is useful to modularize code. Ratiu et. al.[10] develop a heuristic that determines the difference between how code is modularized and how it *should* be modularized based on the underlying logic. However, their heuristic evaluates program structure as a whole, meaning that they analyze packages, classes, and functions/methods (and their names) but not the content of these functions or methods.

Besides the two papers described above, we have found that much related work has been done on a similar topic, namely program similarity. That is, given two programs, we would like to know how similar they are in style to each other. This is useful for detecting plagiarism among amateur student programmers, and detecting whether other concepts or programs have been implemented before. Joy, Joy, and Luck [6] note that since much of student plagiarism involves changing a program structurally (e.g. replacing a for loop with a while loop) and lexically (e.g. rewording comments), they can detect similarity with an incremental comparison method, whereby they study different possible modifications of two programs. Whale [11] takes a slightly different approach, whereby he counts the number of occurrences of certain attributes of programs (such as the number of unique operators and operands), and combines this metric with a similarity measure over optimized versions of code at runtime. Similar to [3], these approaches involve counting the occurrences of certain attributes within programs, such as logical operators, operands, and keywords.

Much of the other related work[7, 8] we have found centers on two different ideas for detecting similarity or modularity (one of which was just described): counting occurrences of certain attributes and detecting similarity of parse trees. Counting occurrences of certain attributes is relatively easy, as we can determine these counts by a single pass through a program. Moreover, many people have found this approach to detect similarity somewhat accurately [3][6]. However, there are many instances when counts of logical operators and operands could be the same, but programs could be very different. Thus the need for detecting structural similarity.

Detecting structural similarity is relatively hard, since we first must perform a lexical analysis and a parse of a program, which takes multiple passes over a program. Once we have parse trees for programs, it is also then hard to determine which subtrees to compare for similarity, and furthermore how to compare them. However, such an analysis can have the potential to be very useful and beneficial for determining similarity[11][6].

3 Design and Implementation

3.1 DRYness score and setup

We will define the DRYness score of a program to be in the range $[0, 1]$. A DRYness score close to 0 means the program is "very dry", and a score near 1 means the program is "very wet", i.e. repetitive. A DRYness score of -1 means the Java program does not compile.

Our DRYness score is computed using Java, meaning that our heuristics only work on Java files. However, similar heuristics could be used to detect DRYness for most other programming languages. We chose to implement our heuristics for Java files since Java is a popular programming language among amateur programmers, and we found a convenient external library to lexically analyze and parse Java files efficiently [1].

3.2 Test dataset

We compute the DRYness heuristics (described below) on seven different test Java files, which we feel captures the basic properties of DRYness somewhat well. Of these seven test files, one consists of an empty class (for a baseline measurement), and the other six are paired, three of which are simple, modular programs; and three of which are WET versions of the modular programs. We thus expect the wet versions of the pair to have a DRYness score close to 1, and the dry versions of the pair to output a score close to 0. We also test our heuristics on our own Java programs, to give us some insight on how well our DRYness program is in itself modular. By testing whether our heuristics return relevant values on these pairs of files, we can somewhat objectively evaluate the success of our heuristics.

3.3 Heuristics

We have so far implemented four heuristics (three non-trivial ones) and tested them on our test dataset. These heuristics have varying levels of success and employ various techniques to evaluate DRYness structurally. Our trivial heuristic always returns a DRYness score of 0 (for a baseline), and is defined in `ZeroHeuristic.java`. We also have yet to design the last of three heuristics, for which we give brief descriptions of how they will operate, but no results.

Iteration 1 Heuristic The Iteration 1 Heuristic defins DRYness recursively, starting from the root node of a program's parse tree. For the heuristic to be well-defined, we must define the DRYness score of each type of parse tree node, and output a number between 0 and 1. For example, our definition of the DRYness score of an if statement will be defined differently than the DRYness score of a variable assignment statement. Thus, for each type of parse tree node, the Iteration 1 Heuristic mostly computes the average of the DRYness scores for each sub-tree of a given node. For trivial parse tree nodes, the Iteration 1 Heuristic returns either 0 or 1. Furthermore, for a given Block Statement (comprising a sequence of statements), the Iteration 1 Heuristic compares each statement pairwise for equality. If there are pairs of statements that are equal, the heuristic increases the DRYness score by a small amount.

All Pairs Naive Heuristic The second non-trivial heuristic, the All Pairs Naive Heuristic, computes DRYness iteratively. For each parse tree node that the heuristic visits, it first adds the node and its contents to an `ArrayList` of other similar parse tree nodes. This means that after visiting all nodes in a parse tree, the heuristic has a sequence of `ArrayList`s in memory, with each `ArrayList` comprising all subtrees of a given type. The heuristic then compares the elements of each ArrayList pairwise, checking for equality. If there are pairs of subtrees that are equal, the heuristic will thus increase the DRYness score by a small amount. The final DRYness score is taken to be the average of the results of the pairwise equality checks, where each pairwise equality check is computed as follows:

$$\frac{\#equalPairs}{\#totalPairs}$$

, where $\#equalPairs$ is the number of pairs of subtrees of a given type that are the same, and $\#totalPairs$ is the total number of pairwise comparisons made for a given type of node.

All Pairs Weighted Heuristic The third non-trivial heuristic, the All Pairs Weighted Heuristic, is a slightly modified version of the All Pairs Naive Heuristic. Similar to the All Pairs Naive Heuristic, this heuristic compares parse tree nodes of a given type in pairs, checking for equality. However, the All Pairs Weighted Heuristic weights each type of equality check differently, so that, for example, equal statements within a Block Statement are penalized more heavily than equal comment strings. It thus returns a weighted average of the result of each pairwise equality check.

Tree Edit Distance Heuristic The first yet-to-be-implemented heuristic, the Tree Edit Distance Heuristic, will operate very similarly to the All Pairs Naive Heuristic. However, instead of checking whether two parse tree nodes are equal, this heuristic will employ tree edit distance to check the structural similarity of two nodes. Given two subtrees, the tree edit distance between these two trees is defined to be the number of insertions, deletions, or renaming of nodes in one tree such that the trees are identical[9]. Thus, a tree edit distance of 0 implies that the two trees are equal.

The Tree Edit Distance heuristic will then compute tree edit distance for each pair of subtrees, and increase the DRYness score in a way that is inversely proportional to its tree edit distance. However, computing tree edit distance is relatively expensive, requiring $O(n^2)$ time for each call, where n is the number of nodes in the largest tree [9].

Tree Edit Distance Weighted Heuristic The Tree Edit Distance Weighted Heursitic, also yet-to-be-implemnted, will be exactly the same as the Tree Edit Distance Heuristic, except that we will assign weights (similar to the All Pairs Weighted Heuristic) to each type of parse tree node.

Baker Heuristic The Baker Heuristic will be based off of a proposed heuristic for measuring program "volume" proposed by Baker et. al.[3]. As described above, this heuristic indirectly computes DRYness by way of counting the number of logical operands and operators within a program. Unlike the previous heuristics, this is a heuristic based on lexical similarity instead of structural similarity. We plan to compare the results of this heuristic to our other six heuristics to determine how well a simple lexical heuristic performs relative to the others.

4 Results

Our results are contained within `results.csv`. Here, we can see that the Iteration 1 Heuristic outputs DRYness scores that are significantly higher for sample programs that are wet, which is to be expected. We see that the All Pairs heuristics also have this property, although the DRYness scores for all sample files are very close to 0 (mostly less than 0.1). As expected, the All Pairs Weighted Heuristic gives values closer to 1 than the All Pairs Naive Heuristic for wet Java files.

Further results and analysis will be discussed once the other three heuristics are implemented and tested. We also plan to provide an in-depth discussion of the consequences of our results and ideas for future lines of research. For now however, we can conclude that the given heuristics compute DRYness to a somewhat accurate level.

5 Conclusion

Evaluating the DRYness of a program or its degree of modularity is currently considered to be an unsolved problem, even though there has been a lot of research on a similar aspect of program similarity – that is, examining two different programs for plagiarism detection. Similar to a couple of earlier heuristics, our heuristics involve computing the structural similarity of a Java program by examining the similarity between subtrees of a given Java parse tree. We have compared these heuristics against each other, and found that the heuristics compute a measure of DRYness to a somewhat accurate extent. More research must still be done however in order to improve and optimize the given heuristics, and argue subjectively why they are decent.

6 Acknowledgements

The author is deeply indebted to Chen Ding and Joe Izraelevitz for their support, suggestions, and guidance over the past two months. The author would also like to thank Victor Liu, Jacob Bisnett, Ben O'Halloran, Evan McLaughlin, Brandon Allard, and the rest of the CSC200 student body for their recommendations, and Tyler Hannan for his help with Java generics, Unicode support issues, and visitor patterns.

References

[1] javaparser - java 1.5 parser and ast - google project hosting. `https://code.google.com/p/javaparser/`. Accessed: 2014-12-03.

[2] The wet cart - the daily wtf. `http://thedailywtf.com/articles/The-WET-Cart`. Accessed: 2014-12-02.

[3] A. L. Baker and S. H. Zweben. The use of software science in evaluating modularity concepts. *IEEE Transactions on Software Engineering*, SE-5(2):110–120, 1979.

[4] Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[5] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[6] M. Joy, M. Joy, and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.

[7] Donald F. McGahn. Copyright infringement of protected computer software: An analytical method to determine substantial similarity. *Rutgers Computer and Technology Law Journal*, 21(1):88, 1995.

[8] Kevin A. Naud, Jean H. Greyling, and Dieter Vogts. Marking student programs using graph similarity. *Computers & Education*, 54(2):545–561, 2010.

[9] Mateusz Pawlik and Nikolaus Augsten. Rted: A robust algorithm for the tree edit distance. 2011.

[10] D. Ratiu, R. Marinescu, and J. Jurjens. The logical modularity of programs. pages 123–127, 2009.

[11] G. WHALE. Identification of program similarity in large populations. *COMPUTER JOURNAL*, 33(2):140–146, 1990.