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. 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. As stated by Frederick Brooks in an article from April 1987[2], 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.[1] 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.[4] 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 [3] 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 [5] 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 [1], these approaches involve counting the occurrences of certain attributes within programs, such as logical operators, operands, and keywords.

Much of the other related work 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 [1][3]. 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[5][3].

3 Design and Implementation

DRYness score We will define the DRYness score of a program to be in the range $[0, 1]$.

Although much of the current research on program similarity centers around plagiarism detection, similar techniques can nonetheless be used to evaluate similarity *within* a program. As stated above, we created various heuristics that center around structural similarity of subtrees, and then evaluated them on a number of example programs to evaluate their accuracy. Because there are very few other metrics for DRYness, these example programs do *not* technically count as labeled data, although we use them as if they are labeled. More specifically, for each example program, we have a rough idea of its DRYness score. We have created a few DRYness measures to analyze blocks of a program for lexical and structural similarities similar to [5]. ... For example, given two methods of a Java program, I will compare/contrast them by first counting the number of occurrences of certain attributes of these two methods, and then analyze the methods for structural similarity. I can then combine the results of these two metrics to produce a real number between 0 and 1, that will measure the amount of similarity between those two methods. If I then were to calculate this similarity between all methods in a Java program, and calculate measures of similarity between other attributes of the Java code, such as expressions and conditionals, I would be able to compute a measure of internal similarity for the program itself. This would also take the form of a real number between 0 and 1 (0 being completely "dry"), and would constitute the "dryness" metric.

It is important to note that this metric can be extended to other programming languages besides Java, although I propose to solely focus on Java due to its enormous popularity among amateur programmers. As an extension, I will also try to implement a system that gives an amateur programmer suggestions on how to improve their code to be more "dry". To do this, my program would first point out to the programmer which attributes of their existing program are not dry, and then suggest different ways of structuring their program that would decrease the dryness metric. Due to the undecidability of detecting whether two programs produce the same output, however, I suspect my program will not be able to give a programmer concrete recommendations of how to modify their existing program.

4 Results

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 heursitics, 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 []. 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 Izralevitz 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] 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. http://openurl.lib.rochester.edu/?ctx_ver=Z39.88-2004&ctx_enc=info%3Aofi%2Fenc%3AUTF-8&rfr_id=info:sid/summon.serialssolutions.com&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&rft.genre=article&rft.atitle=The+Use+of+Software+Science+in+Evaluating+Modularity+Concepts&rft.jtitle=IEEE+Transactions+on+Software+Engineering&rft.au=Baker%2C+A.L&rft.au=Zweben%2C+S.H&rft.date=1979&rft.pub=IEEE&rft.issn=0098-5589&rft.volume=SE-5&rft.issue=2&rft.spage=110&rft.epage=120&rft_id=info:doi/10.1109%2FTSE.1979.234167&rft.externalDocID=32&paramdict=en-US.

[2] Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. http://openurl.lib.rochester.edu/?ctx_ver=Z39.88-2004&ctx_enc=info%3Aofi%2Fenc%3AUTF-8&rfr_id=info:sid/summon.serialssolutions.com&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&rft.genre=article&rft.atitle=No+Silver+Bullet+Essence+and+Accidents+of+Software+Engineering&rft.jtitle=Computer&rft.au=Brooks&rft.date=1987&rft.pub=IEEE&rft.issn=0018-9162&rft.eissn=1558-0814&rft.volume=20&rft.issue=4&rft.spage=10&rft.epage=19&rft_id=info:doi/10.1109%2FMC.1987.1663532&rft.externalDocID=2&paramdict=en-US.

[3] M. Joy, M. Joy, and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=762946.

[4] D. Ratiu, R. Marinescu, and J. Jurjens. The logical modularity of programs. pages 123–127, 2009. http://openurl.lib.rochester.edu/?ctx_ver=Z39.88-2004&ctx_enc=info%3Aofi%2Fenc%3AUTF-8&rfr_id=info:sid/summon.serialssolutions.com&rft_val_fmt=info:ofi/fmt:kev:mtx:book&rft.genre=proceeding&rft.title=2009+16th+Working+Conference+on+Reverse+Engineering&rft.atitle=The+Logical+Modularity+of+Programs&rft.au=Ratiu%2C+D&rft.au=Marinescu%2C+R&rft.au=Jurjens%2C+J&rft.date=2009-01-01&rft.isbn=0769538673&rft.issn=1095-1350&rft.spage=123&rft.epage=127&rft_id=info:doi/10.1109%2FWCRE.2009.29&rft.externalDocID=5328618&paramdict=en-US.

[5] G. WHALE. Identification of program similarity in large populations. *COMPUTER JOURNAL*, 33(2):140–146, 1990. http://gateway.webofknowledge.com/gateway/Gateway.cgi?GWVersion=2\&SrcAuth=SerialsSolutions\&SrcApp=Summon\&KeyUT=A1990CZ98300007\&DestLinkType=FullRecord\&DestApp=WOS.