How Open is Open Source? Metrics for measuring software mutation

Charles Hathaway

May 14, 2015

Abstract

Open source software is toted as being ”openly accessible” to many people, thus allowing greater innovation and complex new methodologies. However, given the complexity of software and how difficult it is to design and implement, the question of whether or not secondary communities adapt and modify the software needs to be addressed. This paper poses the question of determining the involvement of multiple communities to a project by measuring their contributions in terms of complexity, and discusses how we would go about measuring contributions to a project in a meaningful way.

1 Introduction

Open Source software, which includes such well known projects as Linux, Apache, and OpenSSL, has a large portion of todays market share in their respective segments. According to reports available from w3techs [1], *nix systems account for 67.7% of the internet web servers; 52.7% of these are running a Linux distribution. Important projects, such as OpenSSL, attract world-wide media attention when severe bugs appear; take for example the numerous articles published because of the April 2014 bug called ”Heartbleed” [2] [3].

After this particular crisis, it was realized that there are numerous open source projects that are too big to fail. Large corporations agreed to contribute money to these projects, in hopes that it will prevent another such bug [4]. It is here where we can begin to understand the need to figure out who is working on projects, how much are they doing, and how complicated the projects are.

Understanding who is working on a project is no simple matter; we must consider multiple roles (programmer, tester, project manager, to name a few), along with the amount of work they are putting into that project (are they spending 1/8 of their day on it? 16/24 of their day?). This paper, will propose a method for further research that will allow one to quantify the contributions of at least one role in a software developers world; the role of the programmer. Namely, what we want to discover is how we can measure the change that a developer causes in the codebase; are they simply renaming variables, adding UI elements, or changing core algorithms? This question of the distribution of coding contributions might have several applications: for example knowing the degree to which modified code has changes in

complexity arises in assessment of students' work in education; in the assessment of originality in legal domains; and even the assessment of biological material if we think of genetics or amino acid sequences as code.

This brings us into the field of software complexity. It is important not to confuse this field with other forms of complexity we often look at in the computer science discipline, such as time and space complexity. The goal of this field is to determine how difficult a program is to understand, test, and write.

There have been many attempts over the years to measure software complexity; some of the most well known ones by McCabe [5], Oviedo [6], and Halstead [7]. Our goal in this paper is to develop a method which can eventually be used to evaluate the most common complexity measures, and look for correlations between how real people percieve a program to be, and how the metrics understand the complexity of a program. To conclude the paper, we will run a small-scale experiment on 5 people, asking them to rank 5 projects (which produce similar output) on how complex they appear to be. Later research will repeat this experiment on a larger, and more diverse, group of subjects.

For simplicity, this paper will utilize the visual programming environment created for the 3Helix program at Rensselaer Polytechnic Institute, CSnap. These scripts are best stored (for long term access) as images, and will be included at the end of the paper.

2 Literature Review

As a form of literature review, will use a subset of well-known papers that represent some of the most important works in this field. Most of these articles have been reprinted **Software Engineering Metrics Volume 1** [8], including:

- **A Complexity Measure** [5]
- **Control Flow, Data Flow, and Program Complexity** [6]

It is worth mentioning that most of these metrics have had their accuracy questioned, and for the most part been proven to fail in one or more ways. A very well known paper that described the failure of these metrics is **Evaluating Software Complexity Measures**, written in 1988 by Elaine J. Weyuker [9]. Weyuker's has been used as the core for many following software complexity papers, including the recent publication by Hongwei Tao, **Complexity measure based on program slicing and its validation** [10].

2.1 Evaluating Software Complexity Measures

In this section, we will summarize and describe previous metrics, with particular attention to Weyuker's analysis. For convenience, I have copied the 9 properties of complexity measures she proposed here. Note, however, that they are

explained in greater detail in the paper.

In the following statements, P and Q represent entire programs. In order to satisfy each property, the metric result of each of the proposed methods must satisfy all of these properties to be considered correct; the use of carnality operators in the context returns the result of a particular metric algorithm instead of the length of a program.

- Property 1: $(\exists P)(\exists Q)(|P| \neq |Q|)$
  - There exists a program P and a program Q such that the complexity of program P does not equal the complexity of program Q. ie, not all programs have the same complexity
- Property 2: Let c be a nonnegative number. Then there are only finitely many programs of complexity c.
  - There is a limit to the number of programs at complexity c
- Property 3: There are distinct programs P and Q such that $|P| = |Q|$
  - There are two programs that do have the same complexity measure
- Property 4: $(\exists P)(\exists Q)(P \equiv Q \ \& \ |P| \neq |Q|)$
  - Programs P and Q produce the same output, but don't have the same complexity
- Property 5: $(\forall P)(\forall Q)(|P| \leq |P;Q| \ and \ |Q| \leq |P;Q|)$
  - In all cases, adding together programs doesn't make them less complex
- Property 6a: $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \& |P;R| \neq |Q;R|)$
- Property 6b: $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \& |R;P| \neq |R;Q|)$
  - Taken together, 6a and 6b are meant to say that adding the same program to existing programs, with the same complexity, doesn't necessarily mean that the two programs will still have the same complexity; for example, variables may interact in such a way as to make it much more complicated.
- Property 7: There are program bodies P and Q such that Q is formed by permuting the order of the statements of P, and $|P| \neq |Q|$
  - The order of the statements has an impact on complexity; ie, the same statements in a different order can mean something more complicated
- Property 8: If P is a renaming of Q, then $|P| = |Q|$
  - Simply changing the name of our variables shouldn't impact the complexity of the program
- Property 9: $(\exists P)(\exists Q)(|P| + |Q| < |P;Q|)$

– Adding the complexity of two programs could result a complexity number lower than adding the two programs together and taking the complexity again

2.2 Summarization of Control Graphs

Control graphs are used by many metrics, and rather than repeating their design in every section, I will describe them here. There has been some change in the nature of computer programming which makes many of the graphs demonstrated in older literature obsolete, namely the case where a GOTO drops execution into a loop or conditional from elsewhere. We will set aside this case, to make the generation much more simple.

Given a program start point, P, add every statement to a node N until a conditional (or loop) node is reached. At this point, create a new node to represent the conditional, and follow all new paths. When they end, join them into a new node. If we are dealing with a loop, when we reach the end of the loops inner block, draw a line back to our loop node. Then connect the loop node to a new node. To help clarify, some sample programs, (along with their graphs) are given below.

2.3 Cyclomatic complexity

Thomas McCabe proposed a complexity measure in his 1976 paper [5]. In this section, I will briefly describe how it works, how to calculate it, and discuss some responses to it (specifically the responses made by Elaine Weyuker in her 1988 paper [9]).

2.3.1 How it works

The McCabe metric (Cyclomatic number) analyses the control flow of a program to determine how complex it is. There are 3 primary items that is used the calculation of the cyclomatic number:

- Number of nodes, denoted as n
- Number of edges, denoted as e
- Number of compontents, denoted as p

Given these three variables, we can calculate a complexity (μ) of a program using the following equation:

$\mu = e - n + 2p$

It should be noted that we use 2p instead of p; some books use p, but draw an exra edge back to the start node from the end node in each component. These two methods result in identical complexities, but I prefer the second because it makes it easier to identify the start and end nodes in my program.

The real challenge is to create the graph so you can find values for these variables. To do this you must break a program into parts such that there is a starting node, and ending node, and a sequence of nodes between that two have the following key attributes:

- A compound statement is only one node; ie, int i=0; i = b*x; ... counts as only one node until the next branch occurs
- A branch is a conditional, either in the form of an if statement or a loop (with a loop, connect the node to both the following statement(s) and the code that gets executed after it exits)

As a general rule of a thumb, we have a component anywhere a loop occurs. We can have multiple components inside a component (embedded loops).

Once we have created a graph, it is a trivial matter of counting the number of edges, nodes, and components and applying the equation.

2.3.2 Example

Looking at the control graph for the first program, see figure 6, we can calculate the cyclomatic number pretty easily. There are 2 edges, 3 nodes, and 1 component. Therefore, our cyclomatic number is:

$\mu = e - n + 2p = 2 - 3 + 2(1) = 1$

In this figure, we can see that all statements get put into one node. More complicated graph, such as the one for program delta in figure 10, allow to see how the control flow graph is generated more intuitively.

2.3.3 Analysis

This metric was included in the analysis performed by Weyuker in her 1988 paper [9]. Ultimately, she concluded that McCabes metric failed to address properties 2, 6, 7, and 9, for the following reasons:

- Property 2: We can easily imagine an infinite number of programs that don't branch, but are just repeats of the same statement over and over (x+=2;x+=2;x+=2..). These programs would have a complexity of 1, and there are an infinite number of them

- Property 6: Since McCabes complexity measure doesn't account for the interaction of variables, appending a program (R) to the end of another program (P) would always result in $|P| + |R|$, whichs means that if $|Q| = |P|$, then $|Q| + |R| = |P| + |R|$.
- Property 7: The order of statements in McCabe's metric have no influence on complexity; we will always have the same number of branches, same number of nodes, and components
- Property 9: McCabe's metric fails this for the same reason as property 6

Additional critique was made by Martin J. Shepperd in his 1988 paper [11], however we found some discrepancies between this critique and the model developed by McCabe.

2.4 Normalized Compression Distance

Proposed by Rudi Cilibrasi and Paul Vitanyi in their 2005 paper, **Clustering by compression** [12] puts forth a very straightforward way of calculating complexity changes between two pieces of code, without even knowledge of the structure of the content. They proposed using a standard compression method (such as gzip or PPMZ) to approximate the Kolmogorov complexity of a program, and using this solve the equation in figure 1.

Figure 1: Normalized Compression Distance

$$NCD(x,y) = \frac{C(xy) - min(C(x),C(y))}{max(C(x),C(y))}$$

There is some additional difficulty involved in computing this from CSnap projects; a large portion of the XML file is simply constant data, such as the program version, stage size, etc. Therefore, to make this more accurate, we only compress and compare the script sections of the XML file. This still isn't perfect, since XML has so much additional data, but it a better approximation than using an entire file.

2.4.1 Example

If we attempt to compare projects alpha and beta, we can get the NCD by extracting the scripts, compressing them, and performing the outlined calculation listed above. The raw XML code is too long to insert here, however, it can be obtained from the data repository in the files "sample_csnap_applications/alpha_script.xml" and "sample_csnap_applications/beta_script.xml".

If we compress these files using gzip, the record the sizes of the new files, we can do the calculation.

```
[charles@aura sample_csnap_applications]$ cp alpha_script.xml alpha_beta_script.xml
```

```
[charles@aura sample_csnap_applications]$ cat beta_script.xml >> alpha_beta_script.xml
[charles@aura sample_csnap_applications]$ ls *_script.xml
alpha_beta_script.xml  alpha_script.xml  beta_script.xml
[charles@aura sample_csnap_applications]$ gzip *_script.xml
[charles@aura sample_csnap_applications]$ ls -l *_script.xml.gz
-rw-rw-r--. 1 charles charles 380 May  7 14:41 alpha_beta_script.xml.gz
-rw-rw-r--. 1 charles charles 278 May  7 14:37 alpha_script.xml.gz
-rw-rw-r--. 1 charles charles 315 May  7 14:37 beta_script.xml.gz
```

With our variables not set to:

- $C(x) = 278$
- $C(y) = 315$
- $C(xy) = 380$

We can use the equation to calculate NCD:

$NCD(x,y) = \frac{C(xy)-min(C(x),C(y))}{max(C(x),C(y))}$

$NCD(x,y) = \frac{380-min(278,315)}{max(278,315)}$

$NCD(x,y) = \frac{380-278}{315}$

$NCD(x,y) = 0.3238095238095238$

2.4.2 Analysis

Weyuker did not analyze this metric in her paper, and as of the time of writing this paper, no one has attempted that comparison. This is may be due to the fact that we don't normally analyze the distance between two sets of code as a metric, even though many of Weyukers metrics are entirely about finding a change between two pieces of code.

We provide a brief pass over all of Weyuker's properties with NCD in mind, and give an explanation as to why each is or isn't satisfied. We will not do rigid proofs, but rather leave those for an area of later research.

- Property 1: $(\exists P)(\exists Q)(|P| \neq |Q|)$
  - This can easily be demonstrated; just look at the example we gave.
- Property 2: Let c be a nonnegative number. Then there are only finitely many programs of complexity c.

  - Given that for NCD a program is just a compression of source code, we could expect a program of size a to smaller than a program of size abc

- Property 3: There are distinct programs P and Q such that $|P| = |Q|$
  - This is harder to envision, but if we had two programs that were the same code moved around, then compressed, we would get the same complexity size. Take a program (in some hypothetical language) that consisted of the code "ab", and it's brother program that consisted of "ba". Compressing these programs results in a NCD metric of 29 for both

- Property 4: $(\exists P)(\exists Q)(P \equiv Q \ \& \ |P| \neq |Q|)$
  - As long as the two programs aren't identical, and aren't permutations of each other, we could imagine two programs with different NCD's and the same output

- Property 5: $(\forall P)(\forall Q)(|P| \leq |P;Q| \ and \ |Q| \leq |P;Q|)$
  - Although in our examples it *may* be possible to prove this one wrong, this is most likely due to our not-quite-perfect compression; one could imagine that with a perfect compression, you could get a metric that is the same size, but not smaller

- Property 6a: $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \& |P;R| \neq |Q;R|)$

- Property 6b: $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \& |R;P| \neq |R;Q|)$
  - NCD will account for this; if program R has subsets of program Q in it, it would result in a different size than program R;Q

- Property 7: There are program bodies P and Q such that Q is formed by permuting the order of the statements of P, and $|P| \neq |Q|$
  - This can be proven by taking the beta_script.xml and moving the clear block; the resulting file has a larger size than the original

- Property 8: If P is a renaming of Q, then $|P| = |Q|$
  - NCD would fail in this respect; longer variable names would result in longer compressed scripts

- Property 9: $(\exists P)(\exists Q)(|P| + |Q| < |P;Q|)$
  - It is unlikely this would be true for NCD; adding two programs together gives the compressor more to work with, and therefore it would be able to do a better job. Just adding the size of the smaller compressed files would fail to work (just think about the overhead we have to add in! Program "ab" went from size 3 to size 29 when we compressed it).

2.5 Data flow complexity

Enrique Oviedo proposed a metric based on data flow in his 1980 paper [6]. This metric is based on 2 key components; data flow (DF) and control flow (CF). He concludes with the equation $C = \alpha CF + \beta DF$ (where $\alpha = \beta = 1$).

2.5.1 How it works

Control flow is simply the cardinality of the program flow graph. The program flow graph, and how to construct it, is clearly defined in the paper. It more or less follows the same structure as the graphs we construct for the Cyclomatic number, with a more formal definition being given to how a "program" (function in modern terminology) is denoted. Formally,

$CF = \| E \|$

Where $\| \ \|$ stands for set cardinality.

The more complicated part of this metric is the data flow measure. The key terminology for this measure is the distinction between a locally available variable and a locally exposed variable. A variable is locally available when the variable is defined in the block (a block is a "node" in our control flow). A variable is locally exposed when it is referenced without being defined in the block. Another important note is that a variable can be "overridden" (killed) in a block if it is locally exposed, and then made locally available.

$$DF_i = \sum_{j=1}^{\|V_i\|} DEF(v_j)$$

Where V_i is the set of exposed variables in this block and $DEF(v_j)$ counts the number of available definitions for v_j; that is, for j from 1 to length(the set of variables that reach block i) add the number of definitions of variable j to the running total. The example given by Oviedo is:

> For example, if block n_i consists of the statements *X = Y; A = B * Y * (C - B); Z = 2 + A;* and the set of variables definitions that can reach block n_i is $R_i = C_j, B_j, Y_k, B_k$ (ie, if the definitions of C and B in block n_j and the definitions of Y and B in block n_k can reach block n_j), then, according to Definition 8, we have $V_i = C, B, Y$ and
>
> $$DF_j = \sum_{j=1}^{3} DEF(v_j) = 4 \text{ [6]}$$

Although some authors (for example, the author of the very well known Principles of Compiler Design [13]) utilize DEF alongside other functions, such as KILL and USE, these are not described in Oviedo's paper.

$$DF = \sum_{i=1}^{\|S\|} DF_i + DF_f$$

Where S is all nodes except the start and the end nodes.

2.5.2 Analysis

This metric was included in the analysis performed by Weyuker in her 1988 paper [9]. Ultimately, she concluded that the Dataflow metric failed to address properties 2 and 5.

2.6 Other metrics

In addition to the above stated metrics, we will also consider lines of code (also referred to as statement count). This metric is almost universally accepted as not indicative of the complexity of a program, and will provide a sane reference point to verify that we are not just finding correct data.

Lines of code is very dependent on the language in which we are evaluating programs. Since we are using CSnap as a small-scale staging run, we will consider a ”line of code” to be a command block (as defined by Snap! and CSnap). These blocks include things such as ”if”, ”go to x coordinate”, etc.

3 Measuring change

There are several issues that need to be addressed by our algorithm of we are to effectively measure change in software over a large period of time. These include:

- Accounting for ”non-changes”; ie, making a change to a project, testing it, then changing it back
- Discerning between changes that required little to no effort, and changes that may only have been to a line or two, but required a great deal of effort
- Considering things that change the output as part of the metric; not all changes will change the end-result of a program, but will rather optimize the process; other changes will cause the behavior of the program to dramatically alter. How do we differentiate these things?

4 Experiment proposals

In order to test our complexity measure, we will first apply it to the domain of education and student progress. The first step of entering this domain is to interface with the primary stakeholders; teachers and educations that work in the schools, with the primary audience; students. Rather than trying to ”ask” opinions and talk in an abstract way, we have

developed a system to measure the change in student work on our CSnap ”community site”, which teachers can use to help us find a relationship between the various metrics discussed and complexity as viewed by teachers.

Before getting input from teachers, a survey needs to be developed and tested. It will consists of a small number of scripts, which have varying degrees complexity. The teachers will ultimately look at these scripts, and provide an absolute and relative complexity ranking that we can use to help calibrate our metric to present them with a very solid example to base their more qualitative feedback on. Before presenting to the teachers, we will test the survey on fellows in the GK12 grant to verify the correctness and understand-ability of the survey. In the next section, our initial results will be documented.

5 Pre-experiment Results

For this first draft, we created a survey consisting of the following questions

1. Please rank the above projects from most complex (5) to least complex (1)
    - Alpha
    - Beta
    - Bravo
    - Charlie
    - Delta
2. How complex would you rate each project (1 = not very complex, 100 = very complex)
3. What is your gender?
    - Female
    - Male
4. Which race/ethnicity best describes you? (Please choose only one.)
    - American Indian or Alaskan Native
    - Asian / Pacific Islander
    - Black or African American
    - Hispanic American
    - White / Caucasian

- Multiple ethnicity / Other (please specify)

5. What was your major?

The scripts presented to the subjects can be seen in figures 6, 7, 8, 9, and 10

Figure 2: Calculate Metrics for Scripts

	Cyclomatic	Dataflow	NCD	Lines of Code
Alpha	1	2	0.054	23
Beta	2	6	0.031	14
Bravo	1	2	0.039	11
Charlie	6	27	0.056	25
Delta	7	23	0.06	19

5.1 Program Description

The Python program used to generated the metric results for this paper is available, at request, from the author. It is not open source yet, as it is in a repository that also contains copyrighted material. However, in this section we will explain how the program works.

5.1.1 Control Graph Generation

Generating the control graph turned out to be more difficult than thought, due to the complexity related to storing things in XML. The current program parses XML scripts saved from CSnap, extracts the sprites and all their scripts, then constructs directed graph objects from these scripts. These graph objects have several notable methods, namely:

- Graph.components – Returns a list of lists, each of which contains a component seperated from the others
- Graph.lines_of_code – Returns the number of lines of code, summed from the number of lines in each node
- Graph.to_png – Give an input filename, outputs a PNG containing the control flow graph for all components

The graph consists of nodes, which are represented by Node objects. These objects are constructed by the parse_xml script, and contain the following attributes:

- parents – The parent nodes of this object, if any. If is important to note that this needs to be a list because it is possible for a node can have multiple parents; namely if it's the statement that follows an if-else block
- variables – A list of variables that get set in this node
- references – A list of variables that get referenced in this node

- children – A list of children of this node
- name – A string representing a ”name” for a node; in our implementation, this is a list of the statements in the node
- node_type – This merely indicates if this node a loop node, if node, or ”data” block
- lines_of_code – A number representing the number of lines of code in the node

The most interesting methods on these objects, not counting getters and setters are:

- start_node – Returns true if this node has no parents, indicating that it is a start node, or False otherwise
- end_node – Returns true if this node has no children, indicating that is a end node, or False otherwise
  - These two things are not mutually exclusive; consider a graph which has only statements and no loops or branches. Although we later add a program entry and exit point, during the generation this node is simultaneously an end and start node

The last thing to mention is a generic interface called ”Metric”. This is subclassed by all metric calculators, and simply takes a graph as input and verifies that all children redefine the ”calculate” method.

I should also mention why we actually build the graph, as opposed to counting the number of loops and if statements. Not all metrics are the same, and at the time of writing I was not certain how all programs would look at the control graph. Therefore, rather than using a shortcut, I constructed it so that all metrics could gather whatever data they needed without me needing to write multiple version of ”Graph”. A good example of where this paid of the Data Flow metric; it uses the control flow graph to help calculate the data flow numbers, by stepping back through data nodes to see where variables were first defined.

5.2 Program Output

Figure 2 shows the calculate metrics of each of the scripts. These metrics were calculated via a program which will be included with the follow work to this paper, or when requested. NCD in this table represents a comparison between the given project, and Alpha. We used Alpha as a based project, and calculated the NCD in relation to that project. Interestingly, for 2 identical programs we ended with a wildly different NCD.

5.3 Survey Results

We had a total 5 of people complete the survey, and got the reults listed in figures 3 and 4.

Figure 3: Survey Results – Relative

Respondent	rel_alpha	rel_beta	rel_bravo	rel_charlie	rel_delta
1	3	1	2	5	4
2	1	4	3	5	2
3	2	3	1	4	5
4	1	5	2	4	3
Mode	1	#N/A	2	5	#N/A
Median	1.5	3.5	2	4.5	3.5
Mean	1.75	3.25	2	4.5	3.5

Figure 4: Survey Results – Absolute

Respondent	abs_alpha	abs_beta	abs_bravo	abs_charlie	abs_delta
1	30	10	15	50	45
2	5	13	11	20	10
3	16	32	8	64	100
4	20	50	25	40	35
Mode	#N/A	#N/A	#N/A	#N/A	#N/A
Median	18	22.5	13	45	40
Mean	17.75	26.25	14.75	43.5	47.5

Figure 5: Survey Results – Responder Gender and Ethnicity

Respondent	gender	ethnicity
1	male	white
2	female	white
3	male	white
4	female	african american

6 Conclusion

We can clearly see a general consensus that charlie is the most complex project, and alpha the least complex. There is some disagreement as to how complex delta was; 3 of the respondents listed at as fairly complex (and in most cases, much more complex then the less complex project), except for respondant 2. However, this respondent commented that the reason for the lower ranking was that it didn't rely on "hidden magic" in the draw triangle block. This raises an interesting discussion around our program, which did not delve into the blocks when calculating complexity. It did this for 2 reasons:

1. Not all blocks are written in CSnap, so we would need to understand Javascript
2. It is reasonable to expect that you won't have access to all code all the time; the use of libraries complicates and

obsuficates much code in real life.

By not evaluating hidden blocks, we can focus in on the code the user sees, which we believe to be more important in terms of measuring user contribution (but might lead to inaccurate data-flow results).

If we take each projects relative score from each respondant and sum them, we can order projects (from least to most complex):

1. Bravo (7)
2. Alpha (8)
3. Beta (13)
4. Delta (14)
5. Charlie (18)

There is only a very small correlation with Lines of Code as a metric; 2/5 scores line up.

Cyclomatic complexity and data flow are much better; 5/5 (alpha and bravo tied, but they were also really close according to user ratings). This suggests that these two ratings are more accurate than LOC.

Comparing to NCD is a bit more tricky, as we need to see how similar they are. There is no accurate way to do this based on the survey, and would require a much larger testing group as it would be much more subjective.

The absolute ranking provide an interesting insight if look at them as functions of ”distance”. Most users didn't utilize the entire range (1-100), but instead stuck with a range of 1-50 or less. We can say that, on average, the least complex project (accoring to the relative ranking, bravo), is 3.5 times less complex than the most complex project (charlie). This may be scewed by responder bias; we can clearly see that responder who went up to 100 would have gladly went up to 128 if we had let them.

Based on this small sample size, we can't make too many obersvations. However, it does help to clarify the problem, and demonstrate how a larger expirement may be conducted for future works.

On this small amount of data, I don't want to make any speculation on correlations between gender and/or ethnicity and responses. However, I suspect there would be an interesting correlation there based on the responses given; previous works (?) have found that females want a more complete picture than males; in our results, the two female participants ranked delta lower than the males consistently, which was the only project that didn't use ”magic”.

References

[1] W3Techs. (Apr. 14, 2015). Usage of operating systems for websites, [Online]. Available: http://w3techs.com/technologies/overview/operating_system/all.

[2] MITRE. (Dec. 3, 2013). Cve-2014-0160, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[3] Z. Whittaker. (Apr. 8, 2014). Heartbleed bug affects yahoo, okcupid sites; users face losing passwords, [Online]. Available: http://www.zdnet.com/article/heartbleed-bug-affects-yahoo-okcupid-sites-users-face-losing-passwords/.

[4] J. Brodkin. (Apr. 24, 2014). Tech giants, chastened by heartbleed, finally agree to fund openssl, [Online]. Available: http://arstechnica.com/information-technology/2014/04/tech-giants-chastened-by-heartbleed-finally-agree-to-fund-openssl/.

[5] T. J. McCabe, "A complexity measure," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. SE-2, no. 4, pp. 308–320, 1976.

[6] E. I. Oviedo, "Control flow, data flow and program complexity," in *Software engineering metrics I*, McGraw-Hill, Inc., 1993, pp. 52–65.

[7] M. H. Halstead, "Elements of software science," 1977.

[8] Collection, *Software Engineering Metrics Volume 1: Measures and Validations*, M. Shepperd, Ed. 1993, ISBN: 0-07-707410-6.

[9] E. J. Weyuker, "Evaluating software complexity measures," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,*, vol. 14, no. 9, pp. 1357–1365, 1988.

[10] H. Tao and Y. Chen, "Complexity measure based on program slicing and its validation," *Wuhan University Journal of Natural Sciences*, vol. 19, no. 6, pp. 512–518, 2014.

[11] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, 1988.

[12] R. Cilibrasi and P. M. Vitányi, "Clustering by compression," *Information Theory, IEEE Transactions on*, vol. 51, no. 4, pp. 1523–1545, 2005.

[13] A. V. Aho, J. D. Ullman, and author, *Principles of compiler design*. Addison-Wesley Reading, Mass., 1977, vol. 21.

Figure 6: Sample Project Alpha

Figure 7: Sample Project Beta

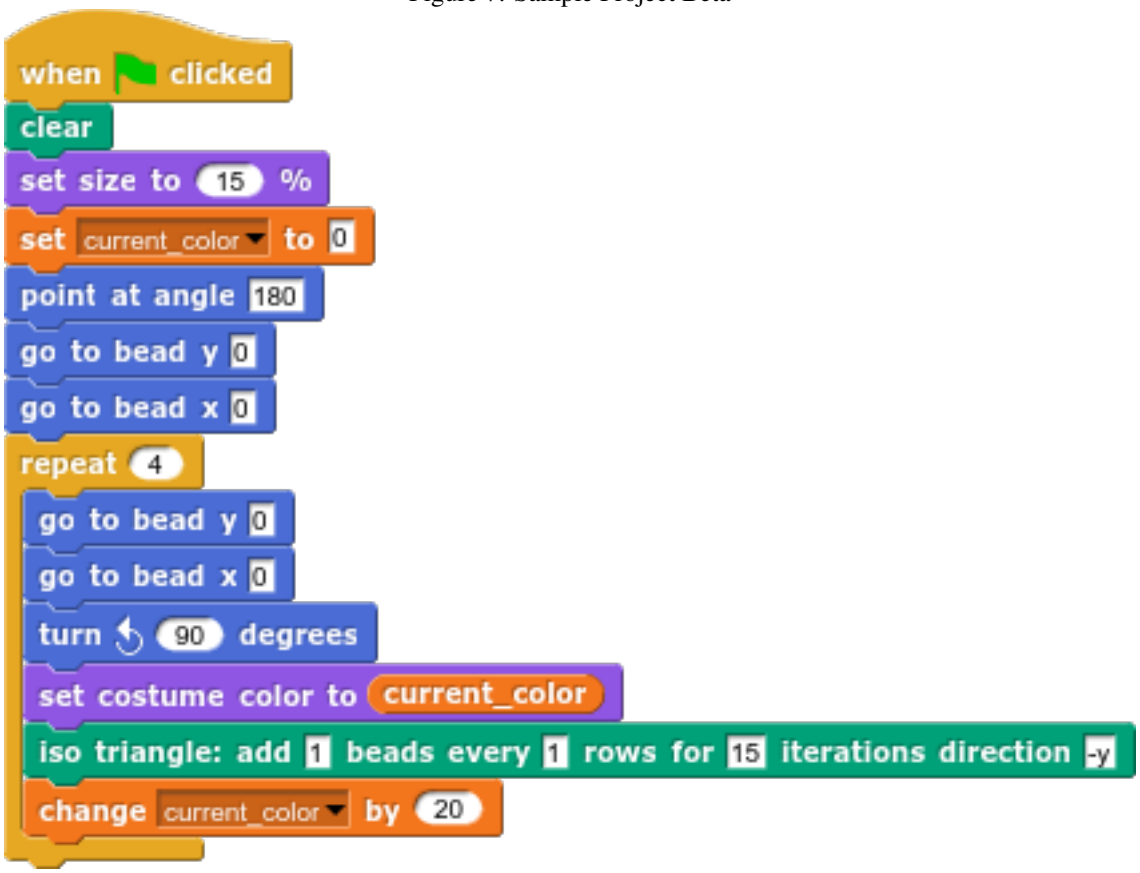

Figure 8: Sample Project Bravo

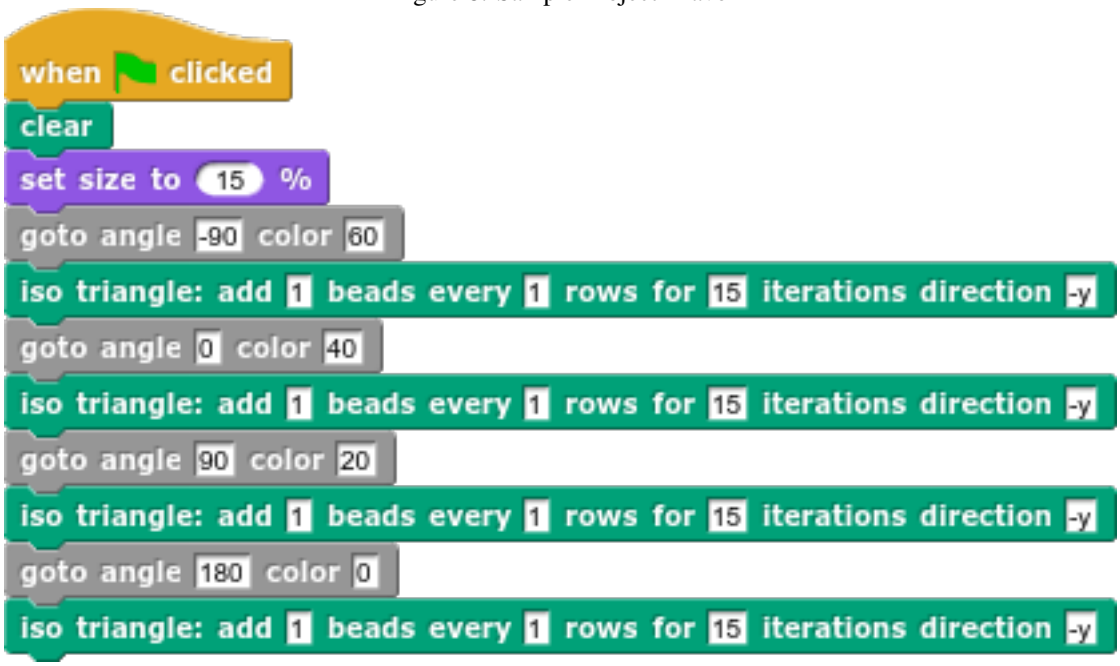

Figure 9: Sample Project Charlie
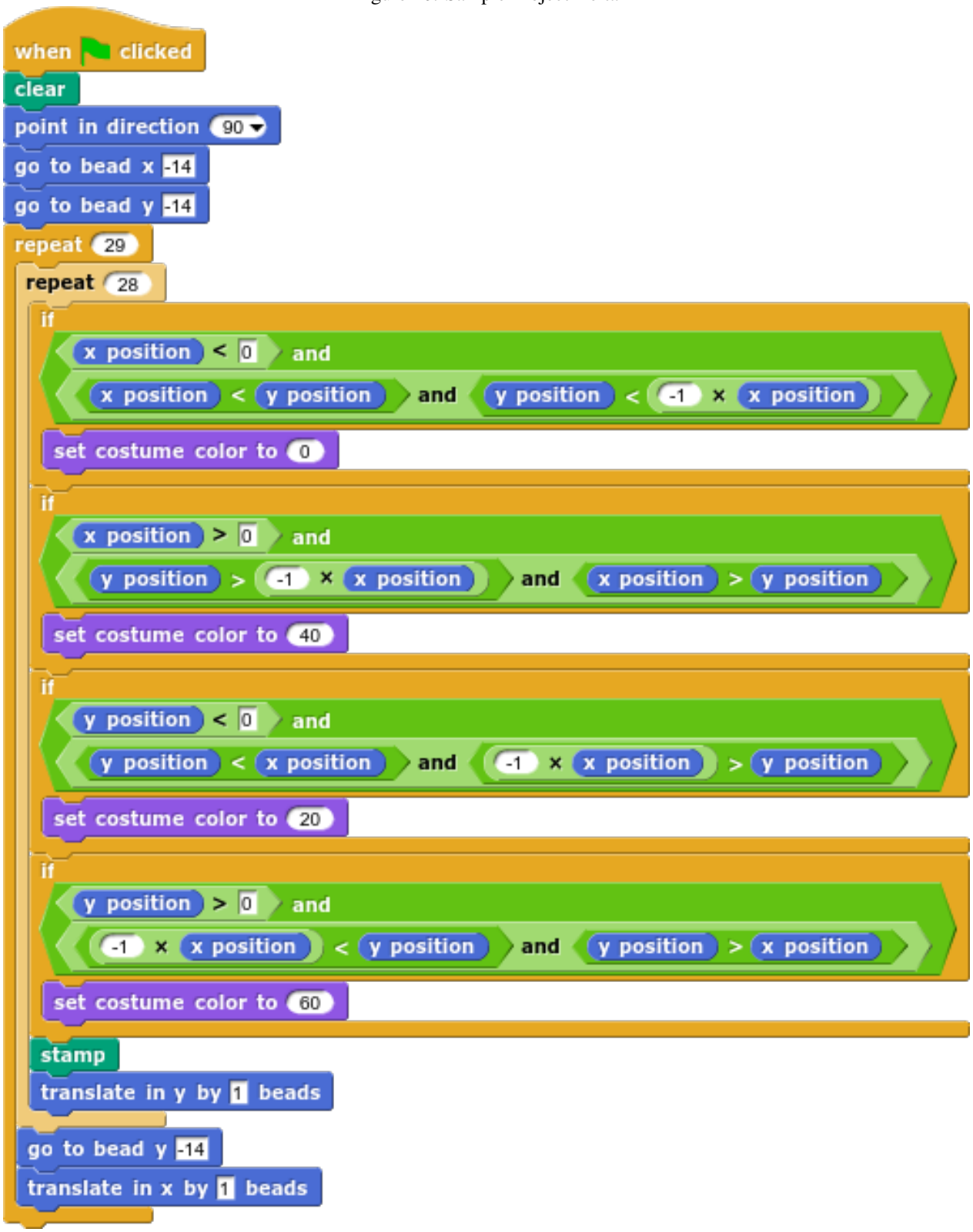
Figure 10: Sample Project Delta