Measuring Software Engineering

Jack O'Sullivan, osullj19@tcd.ie, 17331147

"To deliver a report that considers the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethics concerns surrounding this kind of analytics."

Introduction

Since there has been a software engineering discipline, there has been a need to quantify the quality of software. Up until the mid 1960's, software engineering was an entirely new field, and this need was not recognised. Mathematician Margaret Hamilton coined the term for the field at the request of NASA while they searched for a method of building the software required to land on the moon.[1] Software engineering was given a greater sense of legitimacy when the NATO Science Committee sponsored conferences on software engineering in 1968 and 1969[2].

In the late 1960's, the Software Crisis began. Many large software projects, such as IBM's OS/360, ran way over budget, costing $5 billion instead of the planned $25 million. Despite spending more than "5000 person-years" on the operating system, the project was delivered more than a year late.[3] In a more extreme case the Therac-25, a computer-controlled radiation therapy machine, resulted in 6 accidents where massive overdoses of radiation were administered to patients, resulting in death.[4] Software bugs were the cause, but the overall takeaway was that poor software engineering practices were to blame.[5]

[1] (2008, October 16). NASA Engineers and Scientists-Transforming Dreams ... - NASA. Retrieved November 8, 2019, from https://www.nasa.gov/50th/50th_magazine/scientists.html
[2] (2016, April 21). CS302: Jared King's "The History of Software" | Saylor Academy. Retrieved November 8, 2019, from https://learn.saylor.org/mod/page/view.php?id=12353
[3] (n.d.). What we can learn from the IBM System/360, the first modular .... Retrieved November 8, 2019, from https://about.sourcegraph.com/blog/the-ibm-system-360-the-first-modular-general-purpose-computer/
[4] (n.d.). An Investigation of Therac-25 Accidents - I. Retrieved November 8, 2019, from http://www.cse.msu.edu/~cse470/Public/Handouts/Therac/Therac_1.html
[5] (n.d.). Medical Devices: The Therac-25 - Nancy Leveson. Retrieved November 8, 2019, from http://sunnyday.mit.edu/papers/therac.pdf

Fred Brooks said in his 1975 book *The Mythical Man-Month* that “adding human resources to a late software project makes it later”.[6] This phrase became “Brooks’ law”, although a simplification (by his own admission), it summarises his points that (i) people added to a project need time to become productive (“ramp up time”), (ii) communication overhead increases as the number of people increases and (iii) software engineering is a less divisible task than something like cleaning a hotel.

By the late 1980’s, it was readily apparent that the methodology and practices behind software engineering needed to change. Many new technologies and engineering techniques were developed at the time. Fred Brooks said in 1986 that “there is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity.” This statement was made in his now famous paper *No Silver Bullet*.[7] In other words, despite claims made by developers of technologies such as object-oriented programming, there was no simple solution to the software engineering problem. The need to measure software and software engineering is then apparent.

Methods of measuring software engineering

Given the fact that it took 30 years to come to the realisation that developing software is not an easy problem to solve, there are many ways of (at least attempting to) measure the software engineering process.

Source lines of code (SLOC)

Likely the simplest (and one of the most widely used) method of measuring software is “source lines of code” or just “lines of code”, referring to the number of lines in a project’s codebase.[8]

[6] (n.d.). Mythical Man-Month. Retrieved November 8, 2019, from https://is.muni.cz/www/jirqa/The.Mythical.Man.Month.F.Brooks.pdf
[7] (n.d.). No Silver Bullet - WorryDream. Retrieved November 8, 2019, from http://worrydream.com/refs/Brooks-NoSilverBullet.pdf
[8] (n.d.). A SLOC Counting Standard - USC CSSE - University of .... Retrieved November 8, 2019, from https://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf

Details

Generally there are two numbers when measuring software by lines of code: physical source lines of code (physical SLOC) and logical SLOC. Physical SLOC refers simply to the number of text lines in a source code file (excluding comments). Logical SLOC's definition is less clear-cut, but generally means the number of executable statements in a program, a number of which may span one or more lines.[9] While the number of physical SLOC is comparable across many programming languages, the definition for logical SLOC can vary greatly, even for the same language (depending on the tools used). The easiest way to explain the difference is by an example:

```
// How many lines of code is this?
for (size_t i = 0; i < 100; i++)
{
    puts("Hello, world!");
}
```

The physical SLOC count for the above snippet is 4 lines: a single comment followed by 4 lines of C code. Issues with the use of physical SLOC are apparent when looking at this equivalent snippet:

```
for (size_t i = 0; i < 100; i++) puts("Hello, world!"); // How many lines?
```

Now there is only a single physical SLOC. The definition of logical SLOC thus becomes important. A common way of measuring C code logical SLOC is the number of semicolons, given that statements in C usually must end with a semicolon.[10] Counting the semicolons gives 3 logical SLOC. While counting the occurrences of a single character is as trivial as counting the number of lines in a text file, some questions are raised immediately. For example, `i++`, the increment of `i` on each iteration of the loop doesn't count towards the total, but is just as much of a statement as the call to `puts()`. It could also be argued that a for loop, being as common and well-understood as it is, should only count as a single logical SLOC. Other languages, such as Python and JavaScript, don't even require the use of semicolons (in the case of the former, semicolons are almost never used). Comparing lines of code between programming languages is also made even more difficult by the fact that high-level languages (again like Python or JavaScript) generally produce much more useful programs with far less developer-written code (because of language features or standard libraries) than "languages" like assembly.

[9] (1992, September 7). Software Size Measurement - SEI Digital Library - Carnegie .... Retrieved November 8, 2019, from https://resources.sei.cmu.edu/asset_files/TechnicalReport/1992_005_001_16082.pdf

[10] (n.d.). A SLOC Counting Standard - USC CSSE - University of .... Retrieved November 8, 2019, from https://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf

Even counting the number of executable statements can be very problematic, particularly with modern development tools. Code generation tools are common, from drag-and-drop design software (such as the Android Studio Layout Editor[11]) to transpiling (such as with TypeScript)[12]. There can also be a significant psychological effect on software engineers who are aware of the use of such metrics. For example, there is less incentive to refactor existing code for the benefits of easier development, since the resulting code might actually be smaller, not to mention the lack of recording the changing metrics over time. More ill-intentioned developers might purposefully write more verbose code (without good reason) to increase the line count.

Tools

When counting the basic metrics (physical SLOC, semicolon-based logical SLOC) a simple script or the `wc`[13] command-line tool might suffice. `cloc`[14] is a tool which recognises comments and code (to discount them from the physical line count) for many programming languages. Since counting logical SLOC is more complex, bigger proprietary tools exist for counting lines of code, such as Unified Code Count[15].

Testing and code coverage

A unit test is a testing method which tests individual pieces of a project's code ("units", such as a single function or method).[16] A good suite of unit tests, especially those with good coverage, can be indicative of a project's overall quality.

Details

Unit testing is a very common process for verifying the functionality of small, isolated pieces of code in a software project. Developers are inherently encouraged to explore potential edge cases in code. Testing makes it easier to catch bugs early in the development of a project. A typical unit test usually involves setting up some test data (test instances of objects for object oriented languages) and performing assertions comparing code results to those that have been pre-determined to be correct. For example:

[11] (n.d.). Build a UI with Layout Editor | Android Developers. Retrieved November 8, 2019, from https://developer.android.com/studio/write/layout-editor
[12] (2012, November 18). Compiling vs Transpiling - Steve Fenton. Retrieved November 8, 2019, from https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/
[13] (n.d.). wc(1) - Linux manual page - man7.org. Retrieved November 8, 2019, from http://man7.org/linux/man-pages/man1/wc.1.html
[14] (n.d.). AlDanial/cloc - GitHub. Retrieved November 8, 2019, from https://github.com/AlDanial/cloc
[15] (n.d.). USC CodeCount - Center for Systems and Software Engineering. Retrieved November 8, 2019, from http://sunset.usc.edu/research/CODECOUNT/
[16] (n.d.). Unit Test Frameworks: Tools for High-Quality Software .... Retrieved November 8, 2019, from https://books.google.com/books?id=2ksvdhhnWQsC&printsec=frontcover

```
def test_path():
    root = Node(0)
    root.right = Node(1)
    root.left = Node(2)
    root.left.right = Node(3)
    root.left.right.right = Node(4)

    assert root.right.path(root) == [Node(1), Node(0)]
    assert root.left.right.right.path(root) == \
        [Node(4), Node(3), Node(2), Node(0)]
    assert not root.right.path(root.left)
    assert root.left.right.right.path(root.left) == \
        [Node(4), Node(3), Node(2)]
```

The above code is a unit test for a method which constructs the path from a node in a binary tree to the root node in the tree.[17] The first block (highlighted red), shows the setup for the test, where a simple binary tree is created through the associated of right and left links in the tree. In the green-highlighted code, a number of assertions are made checking that the path the method constructs matches the hand-written expected result. Typically, a unit testing framework will produce a report showing the number of “passing” and “failing” tests.

Code coverage is the term given to the degree to which the test suite for a software project executes the main code in the project. 10% coverage would mean 10% of the project’s non-test code was run during testing, whereas 100% coverage would require every snippet of code in the project to be run by the test suite. Code coverage is much more useful as a metric for evaluating a software project than, for example, the number of unit tests a project contains. A large project will generally have more unit tests than a small one, which immediately makes comparisons more difficult. Depending on the programming language, environment, developer intent (e.g. purposefully splitting out simple test cases into more individual tests), etc., this number can be close to meaningless. Code coverage gives a relative indication of the usefulness of the unit tests.

[17] (n.d.). cs3012/test_tree.py at master · devplayer0/cs3012 · GitHub. Retrieved November 8, 2019, from https://github.com/devplayer0/cs3012/blob/master/tests/test_tree.py#L54

The values for coverage are usually generated from the following criteria:[18]

1. Functions: Has a given function or method in the project been executed?
2. Statements: Has a given code statement within a function been executed?
3. Branches: Have all the possible if/else branches been reached?
4. Conditions: Have all outcomes of boolean expressions been evaluated?

```
int my_function(int a, int b) {
    int result = a;
    if (a > 0 && b > 0) {
        result = b;
    }

    return result;
}
```

In a hypothetical software project containing only the above function, calling `my_function()` with any arguments in a unit test would result in 100% function coverage. Calling `my_function(1, 0)` and later `my_function(0, 1)` satisfies all coverage criteria (all statements executed, each possible branch reached and each possible boolean sub-expression has been evaluated to true and false). Modified condition coverage would not be satisfied unless all possible combinations of the boolean expression `a > 0 && b > 0` had been evaluated, e.g. with `my_function(0, 0)`, `my_function(1, 0)`, `my_function(0, 1)` and `my_function(1, 1)`.

While code coverage is definitely a better measure of software engineering quality over plain unit test counting, it is far from perfect. For example, high coverage percentage requirements set by project managers can result in developers writing tests for code which rarely causes issues instead of known problematic code to bump coverage numbers.[19]

[18] (n.d.). The Art of Software Testing - Software Engineering Research .... Retrieved November 8, 2019, from http://barbie.uta.edu/~mehra/Book1_The%20Art%20of%20Software%20Testing.pdf
[19] (2012, April 17). TestCoverage - Martin Fowler. Retrieved November 8, 2019, from https://martinfowler.com/bliki/TestCoverage.html

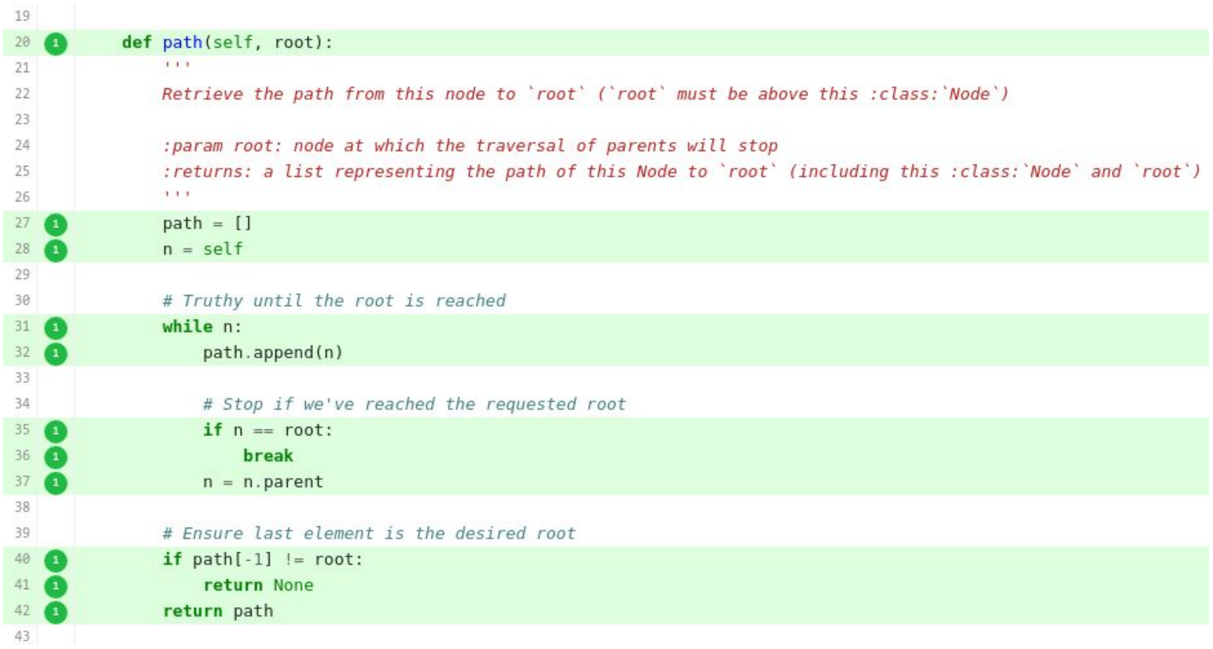
Tools

Many unit testing frameworks exist for many languages. These frameworks in particular make it easy to perform assertions comparing the results of execution to hard-coded test values. For example, JUnit is the most popular test framework for Java.[20] Methods to be run as tests need only be annotated with `@Test` for JUnit to execute them automatically and generate a test report. A number of assertion functions make it easy to see the differences between expected results and results at test time, such as `assertEquals()`, `assertTrue()` and `assertNull()`. pytest, a testing framework for Python (used in the `test_path()` example previously), does not require the use of special assert functions, but introspects assert statements at runtime and reads the comparison operators to similar effects of the special `assert*()` functions in JUnit.[21]

Code coverage tools are distinct from unit test frameworks, but very often integrate with them. JaCoCo is an example of a code coverage tool for Java. It works by instrumenting code at runtime using a "Java agent", a feature of the JVM.[22] Coverage.py is a coverage tool for Python. Regardless of the coverage tool, some kind of "coverage data" file is usually produced, which can later be analysed to produce a coverage report.

With the rising popularity of DevOps, testing, coverage, reporting and deployment are often automated with cloud-based tooling. GitHub Actions, a recently released continuous integration / continuous deployment (CI / CD) platform could be used to perform testing and code coverage automatically whenever new code is pushed to a repository.[23] Several platforms exist for the presentation of code coverage data in a useful report format. Codecov is such a platform - code coverage data can be uploaded to the platform at the end of a CI test run, resulting in an online report and a badge showing the overall code coverage percentage for use in a source code repository's README.[24] Below is an example of a Codecov report for the lowest common ancestor project (the source of the `test_path()` unit test), the green highlighted areas showing 100% coverage of that line (yellow indicating partial coverage, e.g. not full condition coverage and red indicating no coverage):

[20] (n.d.). JUnit 5. Retrieved November 8, 2019, from https://junit.org/junit5/
[21] (n.d.). The writing and reporting of assertions in tests — pytest .... Retrieved November 8, 2019, from http://doc.pytest.org/en/latest/assert.html
[22] (n.d.). JaCoCo - Java Agent - EclEmma. Retrieved November 8, 2019, from https://www.eclemma.org/jacoco/trunk/doc/agent.html
[23] (n.d.). Features • GitHub Actions · GitHub. Retrieved November 8, 2019, from https://github.com/features/actions
[24] (n.d.). Codecov. Retrieved November 8, 2019, from https://codecov.io/



```
    def path(self, root):
        '''
        Retrieve the path from this node to `root` (`root` must be above this :class:`Node`)

        :param root: node at which the traversal of parents will stop
        :returns: a list representing the path of this Node to `root` (including this :class:`Node` and `root`)
        '''
        path = []
        n = self

        # Truthy until the root is reached
        while n:
            path.append(n)

            # Stop if we've reached the requested root
            if n == root:
                break
            n = n.parent

        # Ensure last element is the desired root
        if path[-1] != root:
            return None
        return path
```

Machine learning

A rapidly growing field in recent years, it's possible that machine learning can be used to measure software engineering, given that accurately measuring software engineering is such a difficult problem.

Details

The definition of machine learning is generally the use of computers to perform tasks without being "explicitly programmed".[25] Supervised learning is one of the most common types of machine learning algorithms. Under supervised learning, a model is built based on inputs and corresponding desired outputs, or "training data". The algorithm then infers, based on the training data, what the output for an unseen input should be. This approach to machine learning would be relatively easy to implement for software engineering measurement. A large number of cherry-picked "good" metrics in more traditional metrics (such as those mentioned above) could be chosen as inputs producing high outputs, with corresponding "bad" metrics producing low outputs.

[25] (n.d.). Automated Design of Both the Topology and Sizing of Analog .... Retrieved November 8, 2019, from https://link.springer.com/chapter/10.1007%2F978-94-009-0279-4_9

Another common approach to machine learning is reinforcement learning. While the end goal of producing an output inferred from a given input is the same, no training data is used. Instead, the algorithm “explores” its environment and chooses an action based on this, along with a potential reward based on the outcome of previous actions.[26] Once the environment is established, no further manual input is required. How reinforcement might be implemented for measuring software engineering is less clear than with supervised learning. One possibility might be to employ some kind of simple code generation mechanism, running the generated code through traditional metrics (as mentioned previously) and using the results from these metrics to determine the reward.

Tools

TensorFlow is a popular example of a toolkit / library commonly used in machine learning applications.[27]

Ethics

While the usefulness of data required to produce software engineering metrics is undeniable, there are significant ethical concerns which must be acknowledged. Privacy, for example, is always an important issue, especially with the quantities of information internet companies like Google and Facebook are known to collect. Collection of most traditional metrics does not generally raise significant privacy concerns. Counting the lines of code written by an individual developer for measuring physical SLOC or logical SLOC for example does not infringe on that developer’s privacy, given that the only thing being measured is the code itself. Similarly, collecting code coverage or test results does not raise significant issues.

Less conventional methods have more potential to cause concern. For example, incorporating personal information acquired from a developer into a machine learning algorithm might be cause for alarm. This type of data could be possibly used to attempt to determine a candidate’s proficiency before hiring.

The interpretation of software engineering metrics might also be problematic. Making significant decisions affecting individual developers based on the perceived value of a particular metric may be ill-advised. Over-reliance on specific automatically generated metrics (of potentially low quality) such as SLOC or code coverage could be an example. The effects of heavily favouring low quality metrics would be compounded in the case of machine learning, along with other biases inherent within training data for supervised learning based on the authors. Given that machine learning models are essentially “black boxes”, use of such methods might be unethical.

[26] (n.d.). Reinforcement Learning: A Survey | Journal of Artificial .... Retrieved November 8, 2019, from https://www.jair.org/index.php/jair/article/view/10166
[27] (n.d.). TensorFlow.org. Retrieved November 8, 2019, from https://www.tensorflow.org/

On the other end, the potential for abuse of software metrics is possible from the developer perspective. Again, an over-reliance on simplistic metrics would be a likely cause. The classic example being the ease with which physical SLOC can be manipulated, somewhat paradoxically producing likely inferior code than with fewer lines. Similar issues with testing and code coverage were mentioned previously, an example being the fact that code coverage does not weigh one piece of code more heavily than another (even if one function is critical while another might only be executed in exceptional circumstances).