220

Code Metrics

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley **#doc220**

In this section we look at different types of metrics that we can use to judge the quality of our code and its design. We have already been using some metrics, such as test coverage and number of lines of code in our exercises - what else could we measure?

Coupling Metrics

Afferent coupling (Ca) : A class's **afferent** couplings is a measure of how many other classes use the specific class - a measure of the class's **responsibility**.

Efferent coupling (Ce) : A class's **efferent** couplings is a measure of how many different classes are used by the specific class - a measure of the class's **independence**.

#doc220

There are two different types of coupling: afferent end efferent. Both will appear to different degrees in different parts of any system. A system with no coupling at all would not do anything, as it would just be a collection of independent, unconnected parts.

Afferent coupling measures the number of dependencies that *arrive* at a particular class (or module).
Efferent coupling measures the number of dependencies that leave (or *exit*) from a particular class (or module).

What are **Ca** and **Ce** for collaborators in the **Template Method Pattern?**

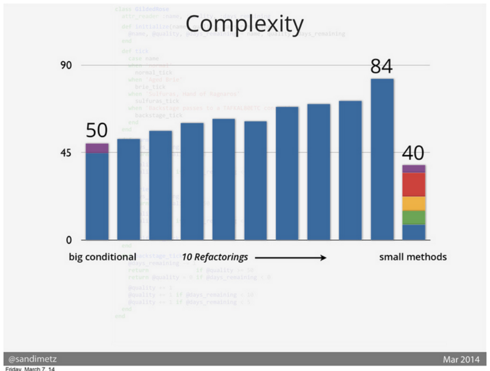
?

#doc220

What are **Ca** and **Ce** for collaborators in the **Strategy Pattern?**

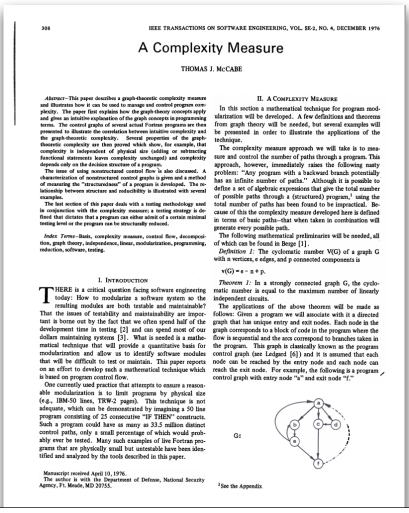
#doc220

Sandi Metz' metrics…



In her talk "All The Little Things" Sandi Metz measures the complexity at each step of an extended refactoring, showing how it starts of getting worse, but eventually gets much better. She uses Flog, which we will discuss later.

Cyclomatic Complexity (McCabe)



308 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO. 4, DECEMBER 1976

A Complexity Measure

THOMAS J. McCABE

Abstract–This paper describes a graph-theoretic complexity measure and illustrates how it can be used to manage and control program complexity. The paper first explains how the graph-theory concepts apply and gives an intuitive explanation of the graph concepts in programming terms. The control graphs of several actual Fortran programs are then presented to illustrate the correlation between intuitive complexity and the graph-theoretic complexity. Several properties of the graph-theoretic complexity are then proved which show, for example, that complexity is independent of physical size (adding or subtracting functional statements leaves complexity unchanged) and complexity depends only on the decision structure of a program.

The issue of using nonstructured control flow is also discussed. A characterization of nonstructured control graphs is given and a method of measuring the "structuredness" of a program is developed. The relationship between structure and reducibility is illustrated with several examples.

The last section of this paper deals with a testing methodology used in conjunction with the complexity measure; a testing strategy is defined that dictates that a program can either admit of a certain minimal testing level or the program can be structurally reduced.

Index Terms–Basis, complexity measure, control flow, decomposition, graph theory, independence, linear, modularization, programming, reduction, software, testing.

I. Introduction

THERE is a critical question facing software engineering today: How to modularize a software system so the resulting modules are both testable and maintainable? That the issues of testability and maintainability are important is borne out by the fact that we often spend half of the development time in testing [2] and can spend most of our dollars maintaining systems [3]. What is needed is a mathematical technique that will provide a quantitative basis for modularization and allow us to identify software modules that will be difficult to test or maintain. This paper reports on an effort to develop such a mathematical technique which is based on program control flow.

One currently used practice that attempts to ensure a reasonable modularization is to limit programs by physical size (e.g., IBM-50 lines, TRW-2 pages). This technique is not adequate, which can be demonstrated by imagining a 50 line program consisting of 25 consecutive "IF THEN" constructs. Such a program could have as many as 33.5 million distinct control paths, only a small percentage of which would probably ever be tested. Many such examples of live Fortran programs that are physically small but untestable have been identified and analyzed by the tools described in this paper.

Manuscript received April 10, 1976.
The author is with the Department of Defense, National Security Agency, Ft. Meade, MD 20755.

II. A Complexity Measure

In this section a mathematical technique for program modularization will be developed. A few definitions and theorems from graph theory will be needed, but several examples will be presented in order to illustrate the applications of the technique.

The complexity measure approach we will take is to measure and control the number of paths through a program. This approach, however, immediately raises the following nasty problem: "Any program with a backward branch potentially has an infinite number of paths." Although it is possible to define a set of algebraic expressions that give the total number of possible paths through a (structured) program,[1] using the total number of paths has been found to be impractical. Because of this the complexity measure developed here is defined in terms of basic paths–that when taken in combination will generate every possible path.

The following mathematical preliminaries will be needed, all of which can be found in Berge [1].

Definition 1: The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is

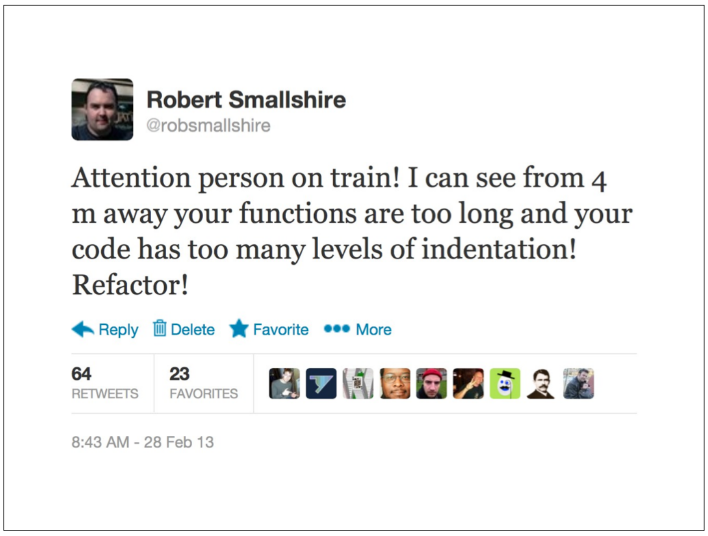
$$v(G) = e - n + p.$$

Theorem 1: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

The applications of the above theorem will be made as follows: Given a program we will associate with it a directed graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is classically known as the program control graph (see Ledgard [6]) and it is assumed that each node can be reached by the entry node and each node can reach the exit node. For example, the following is a program control graph with entry node "a" and exit node "f."

[1]See the Appendix

One of the original complexity metrics is Cyclomatic Complexity, as described by McCabe in his 1976 paper. This counts nodes and edges in the control flow graph of a programme, trying to count the possible different executions that could happen. McCabe complexity gives a lower bound on the number of tests required for a particular unit.

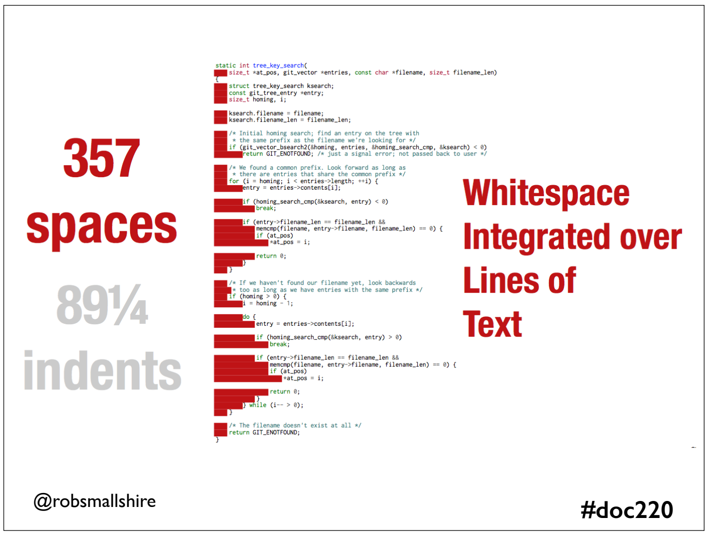


Robert Smallshire
@robsmallshire

Attention person on train! I can see from 4 m away your functions are too long and your code has too many levels of indentation! Refactor!

8:43 AM - 28 Feb 13

McCabe complexity can be quite difficult, or computationally expensive, to calculate. Robert Smallshire introduced a simpler measure which is quicker to compute, but dives a similar indication.



357 spaces

89¼ indents

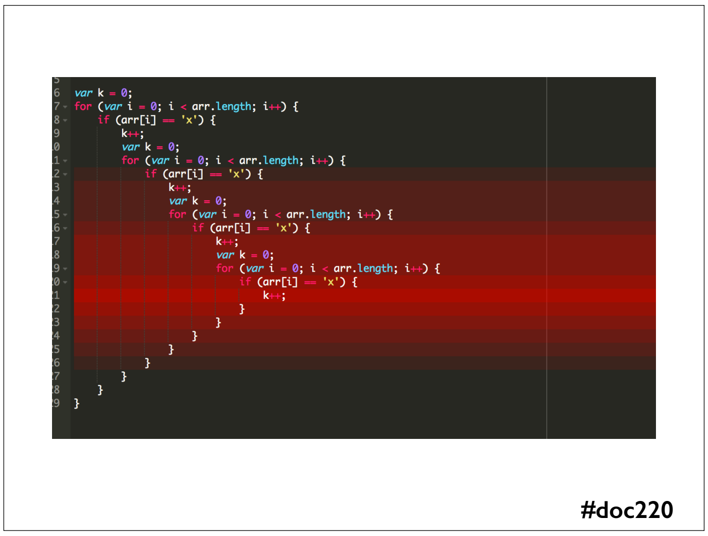
```
static int tree_key_search(
    size_t *at_pos, git_vector *entries, const char *filename, size_t filename_len)
{
    struct tree_key_search ksearch;
    const git_tree_entry *entry;
    size_t homing, i;

    ksearch.filename = filename;
    ksearch.filename_len = filename_len;

    /* Initial homing search; find an entry on the tree with
     * the same prefix as the filename we're looking for */
    if (git_vector_bsearch2(&homing, entries, &homing_search_cmp, &ksearch) < 0)
        return GIT_ENOTFOUND; /* just a signal error; not passed back to user */

    /* We found a common prefix. Look forward as long as
     * there are entries that share the common prefix */
    for (i = homing; i < entries->length; ++i) {
        entry = entries->contents[i];

        if (homing_search_cmp(&ksearch, entry) < 0)
            break;

        if (entry->filename_len == filename_len &&
            memcmp(filename, entry->filename, filename_len) == 0) {
            if (at_pos)
                *at_pos = i;

            return 0;
        }
    }

    /* If we haven't found our filename yet, look backwards
     * too as long as we have entries with the same prefix */
    if (homing > 0) {
        i = homing - 1;

        do {
            entry = entries->contents[i];

            if (homing_search_cmp(&ksearch, entry) > 0)
                break;

            if (entry->filename_len == filename_len &&
                memcmp(filename, entry->filename, filename_len) == 0) {
                if (at_pos)
                    *at_pos = i;

                return 0;
            }
        } while (i-- > 0);
    }

    /* The filename doesn't exist at all */
    return GIT_ENOTFOUND;
}
```

Whitespace Integrated over Lines of Text

@robsmallshire

#doc220

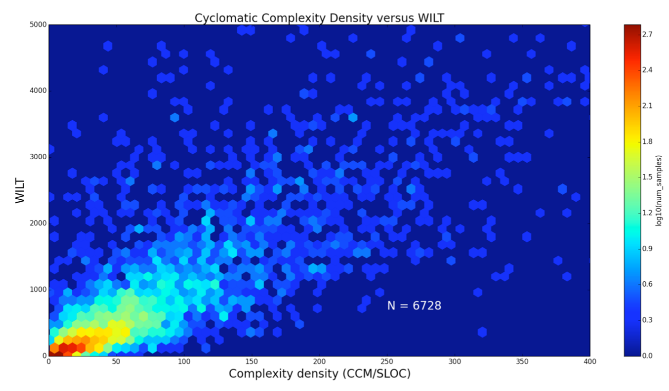
WILT is the Whitespace Integrated over Lines of Text for a given piece of code. By integrating over the indented area, we get a measure of complexity.



```
var k = 0;
for (var i = 0; i < arr.length; i++) {
    if (arr[i] == 'x') {
        k++;
        var k = 0;
        for (var i = 0; i < arr.length; i++) {
            if (arr[i] == 'x') {
                k++;
                var k = 0;
                for (var i = 0; i < arr.length; i++) {
                    if (arr[i] == 'x') {
                        k++;
                        var k = 0;
                        for (var i = 0; i < arr.length; i++) {
                            if (arr[i] == 'x') {
                                k++;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

#doc220

Former Imperial student Shravan Jadhav built a tool that calculates WILT on every key press in a code editor and visualises the results.



@robsmallshire

#doc220

This graph shows that there is a strong correlation between WILT and traditionally calculated McCabe complexity.

ABC Metrics

Assignments
Branches
Conditions

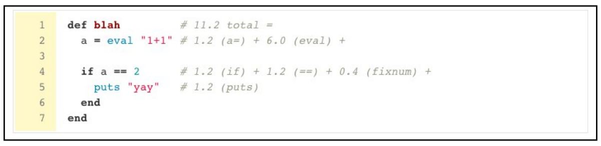
#doc220

An ABC metric counts the numbers of occurrences of different types of programming element, and uses the total as a measure of complexity. Typically they count any occurrences of:

- Assignment -- an explicit transfer of data into a variable, e.g. = *= /= %= += <<= >>= &= |= ^= >>>= ++ --
- Branch -- an explicit forward program branch out of scope -- a function call, class method call, or new operator
- Condition -- a logical/Boolean test, == != <= >= < > else case default try catch ? and unary conditionals.

http://c2.com/cgi/wiki?AbcMetric

Flog



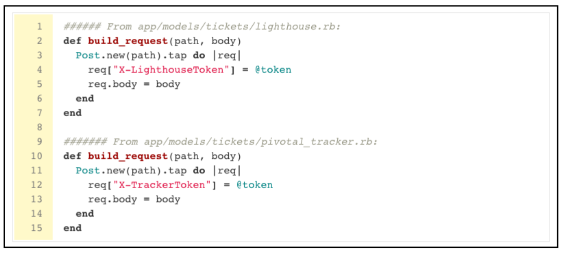
```
def blah           # 11.2 total =
  a = eval "1+1" # 1.2 (a=) + 6.0 (eval) +

  if a == 2        # 1.2 (if) + 1.2 (==) + 0.4 (fixnum) +
    puts "yay"     # 1.2 (puts)
  end
end
```

An ABC metric with some specialist knowledge of Ruby

#doc220

Flog is an ABC metric that is tailored to Ruby with a little extra knowledge of the language. For example, here it adds 6 points for the occurrence of a call to eval(). It is also weighted so that rather than just counting, each counted element has a particular value that adds to the total.

Flay



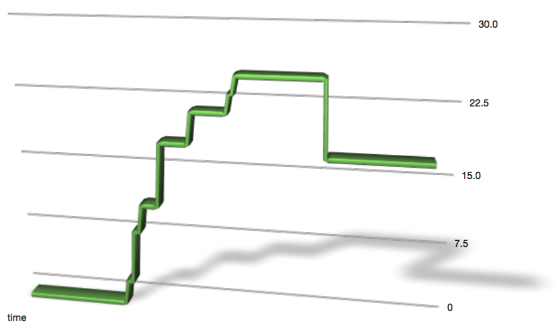
```
###### From app/models/tickets/lighthouse.rb:
def build_request(path, body)
  Post.new(path).tap do |req|
    req["X-LighthouseToken"] = @token
    req.body = body
  end
end

####### From app/models/tickets/pivotal_tracker.rb:
def build_request(path, body)
  Post.new(path).tap do |req|
    req["X-TrackerToken"] = @token
    req.body = body
  end
end
```

Flay detects code *clones* - i.e. duplication

#doc220

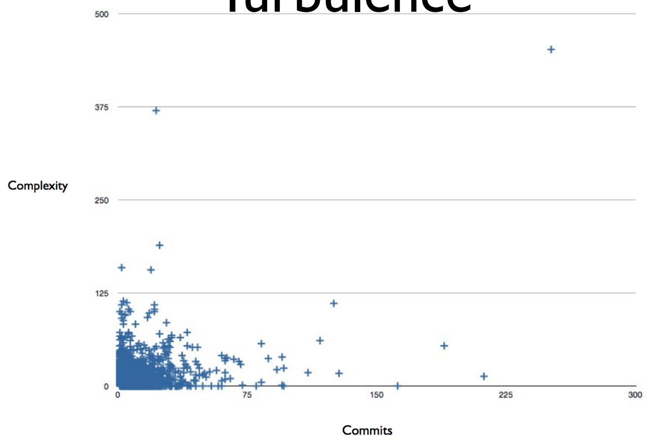
As well as measuring complexity, we can use tools to try to identify duplication in the code. A companion to Flog is Flay, which uses a structural matching algorithm to compare the ASTs of different pieces of code to determine whether they are clones.

Lifelines



Version control systems give us the possibility of uncovering historical data about how code changes over time. This graph shows the changes in the complexity of a method (or some metric that proxies for complexity, such as WILT or even just method length), over time - with a graph point for every commit in the version control system. Can you spot different types of change, just by looking at the lifeline?

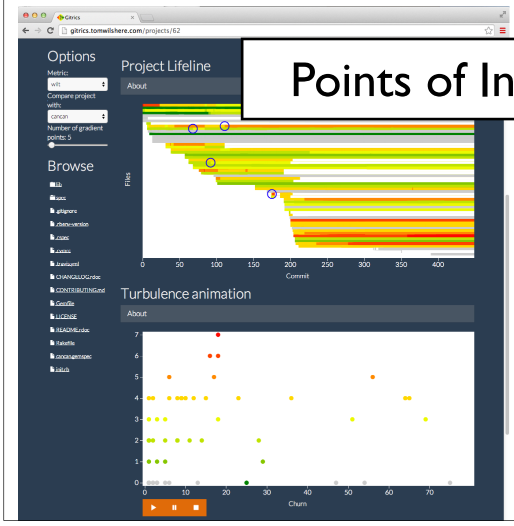
Turbulence



This graph shows something a bit more sophisticated - it plots the number of commits that have been made to each file (i.e. how much does this file get changed - known as *churn*), against the complexity of the code in that file.

Think about the four quadrants: a) simple classes they rarely change, b) complex classes that rarely change, c) simple classes that change often, d) complex classes that change often. What type of code might be in each of these quadrants? Which of them would you consider good/bad?

Points of Interest

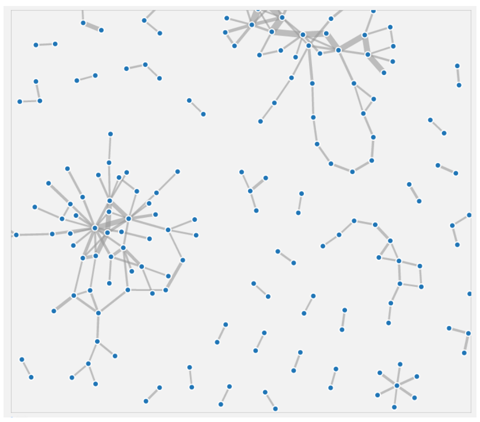


@tomwilshire

#doc220

We can also get an interesting picture of the way that codebases evolve by looking at metrics at each revision in the version control repository, and graphing them over time. There is a growing trend for research in Software Archaeology. Former Imperial student Tom Wilshere built Gitrics, a tool to visualise the changes in a project's code and code quality over time. http://gitrics.tomwilshire.com

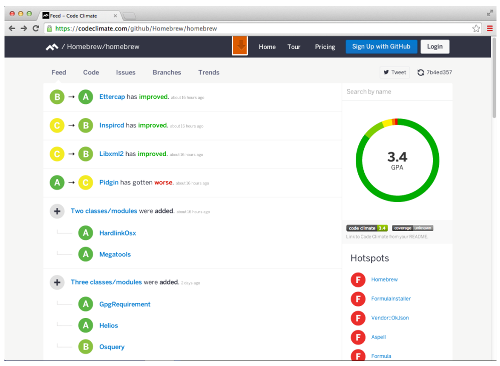
Files changed in same commit



A different type of information that we can find from the version control is what things change together (for example, what things get committed together). This might give us a different kind of structural or behavioural relationship that isn't obvious from the static view of the code.

The diagram on the slide was generated by some tooling written by Dmity Kandalov - this one is a fragrant of a visualisation of the codebase for Maven. http://dkandalov.github.io/code-history-mining/Maven.html

Code Climate



B → A Ettercap has improved.

C → B Inspircd has improved.

C → B Libxml2 has improved.

A → C Pidgin has gotten worse.

Two classes/modules were added.

A HardlinkOsx

A Megatools

Three classes/modules were added.

A GpgRequirement

A Helios

B Osquery

F Homebrew

F FormulaInstaller

F Vendor::OkJson

F Aspell

F Formula

#doc220

Services are now appearing that put together a number of these tools and metrics into one analysis package. For example Code Climate https://codeclimate.com/