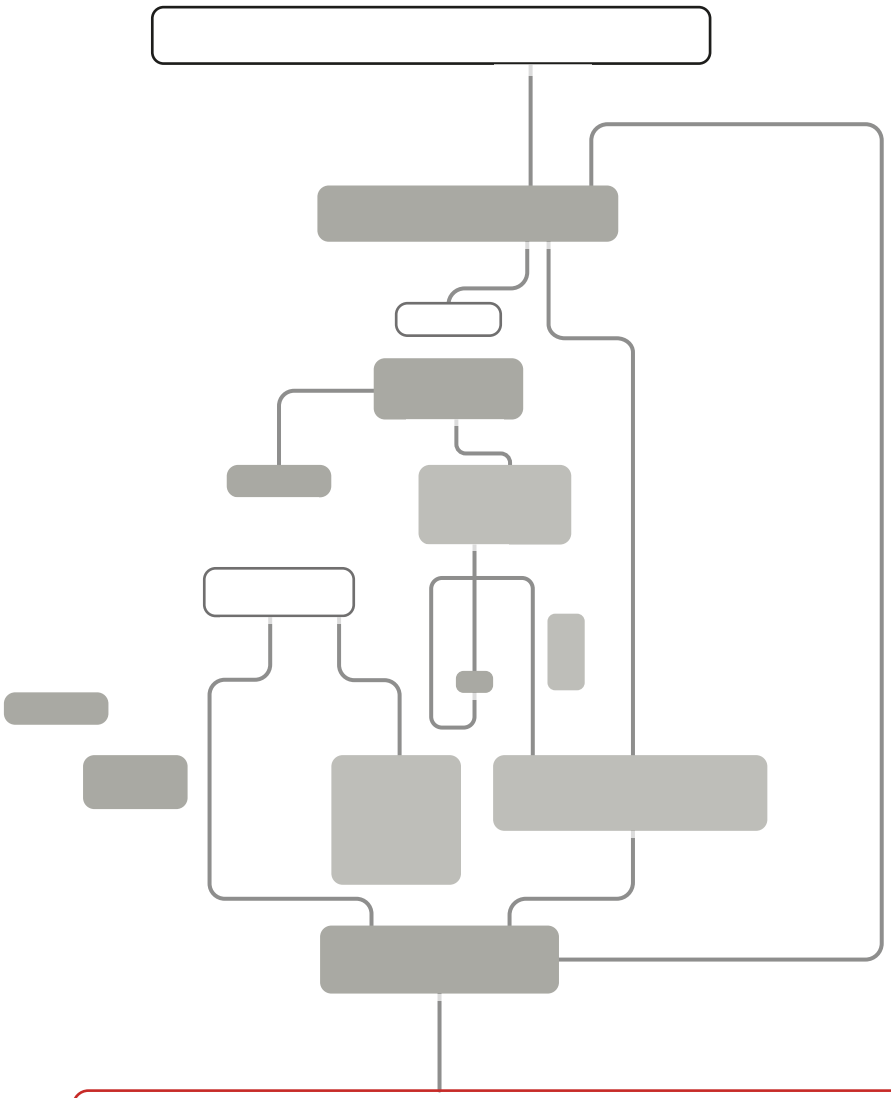
sonarsource



COGNITIVE COMPLEXITY

A new way of measuring understandability

By G. Ann Campbell, SonarSource SA

5 April 2021, Version 1.5

Abstract

Cyclomatic Complexity was initially formulated as a measurement of the “testability and maintainability” of the control flow of a module. While it excels at measuring the former, its underlying mathematical model is unsatisfactory at producing a value that measures the latter. This white paper describes a new metric that breaks from the use of mathematical models to evaluate code in order to remedy Cyclomatic Complexity’s shortcomings and produce a measurement that more accurately reflects the relative difficulty of understanding, and therefore of maintaining methods, classes, and applications.

A note on terminology

While Cognitive Complexity is a language-neutral metric that applies equally to files and classes, and to methods, procedures, functions, and so on, the Object-Oriented terms “class” and “method” are used for convenience.

Table of Contents

Introduction

Thomas J. McCabe's Cyclomatic Complexity has long been the de facto standard for measuring the complexity of a method's control flow. It was originally intended "to identify software modules that will be difficult to test or maintain"[1], but while it accurately calculates the minimum number of test cases required to fully cover a method, it is not a satisfactory measure of understandability. This is because methods with equal Cyclomatic Complexity do not necessarily present equal difficulty to the maintainer, leading to a sense that the measurement "cries wolf" by over-valuing some structures, while under-valuing others.

At the same time, Cyclomatic Complexity is no longer comprehensive. Formulated in a Fortran environment in 1976, it doesn't include modern language structures like `try`/`catch`, and lambdas.

And finally, because each method has a minimum Cyclomatic Complexity score of one, it is impossible to know whether any given class with a high aggregate Cyclomatic Complexity is a large, easily maintained domain class, or a small class with a complex control flow. Beyond the class level, it is widely acknowledged that the Cyclomatic Complexity scores of applications correlate to their lines of code totals. In other words, Cyclomatic Complexity is of little use above the method level.

As a remedy for these problems, Cognitive Complexity has been formulated to address modern language structures, and to produce values that are meaningful at the class and application levels. More importantly, it departs from the practice of evaluating code based on mathematical models so that it can yield assessments of control flow that correspond to programmers' intuitions about the mental, or *cognitive* effort required to understand those flows.

An illustration of the problem

It is useful to begin the discussion of Cognitive Complexity with an example of the problem it is designed to address. The two following methods have equal Cyclomatic Complexity, but are strikingly different in terms of understandability.

```
int sumOfPrimes(int max) {                  // +1
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) {     // +1
    for (int j = 2; j < i; ++j) {           // +1
      if (i % j == 0) {                     // +1
        continue OUT;
      }
    }
    total += i;
  }
  return total;
}                   // Cyclomatic Complexity 4
```

```
String getWords(int number) { // +1
    switch (number) {
      case 1:                 // +1
        return "one";
      case 2:                 // +1
        return "a couple";
      case 3:                 // +1
        return "a few";
      default:
        return "lots";
    }
  }      // Cyclomatic Complexity 4
```

The mathematical model underlying Cyclomatic Complexity gives these two methods equal weight, yet it is intuitively obvious that the control flow of `sumOfPrimes` is more difficult to understand than that of `getWords`. This is why Cognitive Complexity abandons the use of mathematical models for assessing control flow in favor of a set of simple rules for turning programmer intuition into numbers.

Basic criteria and methodology

A Cognitive Complexity score is assessed according to three basic rules:

1. Ignore structures that allow multiple statements to be readably shorthanded into one
2. Increment (add one) for each break in the linear flow of the code
3. Increment when flow-breaking structures are nested

Additionally, a complexity score is made up of four different types of increments:

A. Nesting - assessed for nesting control flow structures inside each other
B. Structural - assessed on control flow structures that are subject to a nesting increment, and that increase the nesting count
C. Fundamental - assessed on statements not subject to a nesting increment
D. Hybrid - assessed on control flow structures that are not subject to a nesting increment, but which do increase the nesting count

While the type of an increment makes no difference in the math - each increment adds one to the final score - making a distinction among the categories of features being counted makes it easier to understand where nesting increments do and do not apply.

These rules and the principles behind them are further detailed in the following sections.

Ignore shorthand

A guiding principle in the formulation of Cognitive Complexity has been that it should incent good coding practices. That is, it should either ignore or discount features that make code more readable.

The method structure itself is a prime example. Breaking code into methods allows you to condense multiple statements into a single, evocatively named call, i.e. to “shorthand” it. Thus, Cognitive Complexity does not increment for methods.

Cognitive Complexity also ignores the null-coalescing operators found in many languages, again because they allow short-handing multiple lines of code into one. For example, both of the following code samples do the same thing:

```
MyObj myObj = null;
if (a != null) {
  myObj = a.myObj;
}
```

```
MyObj myObj = a?.myObj;
```

The meaning of the version on the left takes a moment to process, while the version on the right is immediately clear once you understand the null-coalescing syntax. For that reason, Cognitive Complexity ignores null-coalescing operators.

Increment for breaks in the linear flow

Another guiding principle in the formulation of Cognitive Complexity is that structures that break code’s normal linear flow from top to bottom, left to right require maintainers to work harder to understand that code. In acknowledgement of this extra effort, Cognitive Complexity assesses structural increments for:

- Loop structures: `for`, `while`, `do while`, ...
- Conditionals: ternary operators, `if`, `#if`, `#ifdef`, ...

It assesses hybrid increments for:

- `else if`, `elif`, `else`, ...

No nesting increment is assessed for these structures because the mental cost has already been paid when reading the `if`.

These increment targets will seem familiar to those who are used to Cyclomatic Complexity. In addition, Cognitive Complexity also increments for:

Catches

A `catch` represents a kind of branch in the control flow just as much as an `if`. Therefore, each `catch` clause results in a structural increment to Cognitive Complexity. Note that a `catch` only adds one point to the Cognitive Complexity score, no matter how many exception types are caught. `try` and `finally` blocks are ignored altogether.

Switches

A `switch` and all its `case`s combined incurs a single structural increment.

Under Cyclomatic Complexity, a `switch` is treated as an analog to an `if-else if` chain. That is, each `case` in the `switch` causes an increment because it causes a branch in the mathematical model of the control flow.

But from a maintainer's point of view, a `switch` - which compares a single variable to an explicitly named set of literal values - is much easier to understand than an `if-else if` chain because the latter may make any number of comparisons, using any number of variables and values.

In short, an `if-else if` chain must be read carefully, while a `switch` can often be taken in at a glance.

Sequences of logical operators

For similar reasons, Cognitive Complexity does not increment for each binary logical operator. Instead, it assesses a fundamental increment for each *sequence* of binary logical operators. For instance, consider the following pairs:

```
a && b
a && b && c && d

a || b
a || b || c || d
```

Understanding the second line in each pair isn't that much harder than understanding the first. On the other hand, there is a marked difference in the effort to understand the following two lines:

```
a && b && c && d
a || b && c || d
```

Because boolean expressions become more difficult to understand with mixed operators, Cognitive complexity increments for each new sequence of like operators. For instance:

```
if (a                          // +1 for `if`
    && b && c                  // +1
    || d || e                  // +1
    && f)                      // +1

if (a                          // +1 for `if`
    &&                         // +1
    !(b && c))                 // +1
```

While Cognitive Complexity offers a “discount” for like operators relative to Cyclomatic Complexity, it does increment for *all* sequences of binary boolean operators such as those in variable assignments, method invocations, and return statements.

Recursion

Unlike Cyclomatic Complexity, Cognitive Complexity adds a fundamental increment for each method in a recursion cycle, whether direct or indirect. There are two motivations for this decision. First, recursion represents a kind of “meta-loop”, and Cognitive Complexity increments for loops. Second, Cognitive Complexity is about estimating the relative difficulty of understanding the control flow of a method, and even some seasoned programmers find recursion difficult to understand.

Jumps to labels

`goto` adds a fundamental increment to Cognitive Complexity, as do `break` or `continue` to a label and other multi-level jumps such as the `break` or `continue` to a number found in some languages. But because an early `return` can often make code much clearer, no other jumps or early exits cause an increment.

Increment for nested flow-break structures

It seems intuitively obvious that a linear series of five `if` and `for` structures would be easier to understand than that same five structures successively nested, regardless of the number of execution paths through each series. Because such nesting increases the mental demands to understand the code, Cognitive Complexity assesses a nesting increment for it.

Specifically, each time a structure that causes a structural or hybrid increment is nested inside another such structure, a nesting increment is added for each level of nesting. For

instance, in the following example, there is no nesting increment for the method itself or for the `try` because neither structure results in either a structural or a hybrid increment:

```
void myMethod () {
  try {
    if (condition1) {                    // +1
      for (int i = 0; i < 10; i++) {     // +2 (nesting=1)
        while (condition2) { … }         // +3 (nesting=2)
      }
    }
  } catch (ExcepType1 | ExcepType2 e) { // +1
    if (condition2) { … }               // +2 (nesting=1)
  }
}                                      // Cognitive Complexity 9
```

However, the `if`, `for`, `while`, and `catch` structures are all subject to both structural and nesting increments.

Additionally, while top-level methods are ignored, and there is no structural increment for lambdas, nested methods, and similar features, such methods do increment the nesting level when nested inside other method-like structures:

```
void myMethod2 () {
  Runnable r = () -> {           // +0 (but nesting level is now 1)
    if (condition1) { … }        // +2 (nesting=1)
  };
}                                // Cognitive Complexity 2


#if DEBUG                        // +1 for if
void myMethod2 () {              // +0 (nesting level is still 0)
  Runnable r = () -> {           // +0 (but nesting level is now 1)
    if (condition1) { … }        // +3 (nesting=2)
  };
}                                // Cognitive Complexity 4
#endif
```

The implications

Cognitive Complexity was formulated with the primary goal of calculating method scores that more accurately reflect methods' relative understandability, and with secondary goals of addressing modern language constructs and producing metrics that are valuable above the method level. Demonstrably, the goal of addressing modern language constructs has been achieved. The other two goals are examined below.

Intuitively 'right' complexity scores

This discussion began with a pair of methods with equal Cyclomatic Complexity but decidedly unequal understandability. Now it is time to re-examine those methods and calculate their Cognitive Complexity scores:

```
int sumOfPrimes(int max) {
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) { // +1
    for (int j = 2; j < i; ++j) {       // +2
      if (i % j == 0) {                 // +3
        continue OUT;                   // +1
      }
    }
    total += i;
  }
  return total;
}                  // Cognitive Complexity 7
```

```
String getWords(int number) {
    switch (number) {          // +1
      case 1:
        return "one";
      case 2:
        return "a couple";
      case 3:
        return "a few";
      default:
        return "lots";
    }
  }      // Cognitive Complexity 1
```

The Cognitive Complexity algorithm gives these two methods markedly different scores, ones that are far more reflective of their relative understandability.

Metrics that are valuable above the method level

Further, because Cognitive Complexity does not increment for the method structure, aggregate numbers become useful. Now you can tell the difference between a domain class - one with a large number of simple getters and setters - and one that contains a complex control flow by simply comparing their metric values. Cognitive Complexity thus becomes a tool for measuring the relative understandability of classes and applications.

Conclusion

The processes of writing and maintaining code are human processes. Their outputs must adhere to mathematical models, but they do not fit into mathematical models themselves. This is why mathematical models are inadequate to assess the effort they require.

Cognitive Complexity breaks from the practice of using mathematical models to assess software maintainability. It starts from the precedents set by Cyclomatic Complexity, but uses human judgment to assess how structures should be counted, and to decide what should be added to the model as a whole. As a result, it yields method complexity scores which strike programmers as fairer relative assessments of understandability than have been available with previous models. Further, because Cognitive Complexity charges no “cost of entry” for a method, it produces those fairer relative assessments not just at the method level, but also at the class and application levels.

References

[1] Thomas J. McCabe, “A Complexity Measure”, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976

Appendix A: Compensating Usages

Cognitive Complexity is designed to be a language-agnostic measurement, but it cannot be ignored that different languages offer different features. For instance, there is no `else if` structure in COBOL, and until recently JavaScript lacked a class-like structure. Unfortunately, those deficits don't prevent developers from needing those structures or from trying to construct something analogous with the tools at hand. In such cases, a strict application of Cognitive Complexity's rules would result in disproportionately high scores.

For that reason, and in order not to penalize the use of one language over another, exceptions may be made for language deficits, i.e. structures which are commonly used, and expected in most modern languages, but missing from the language under consideration, such as COBOL's missing `else if`.

On the other hand, when a language innovates to introduce a feature, such as Java 7's ability to catch multiple exception types at once, the lack of that innovation in other languages should not be considered a deficit, and thus there should be no exception.

This implies that if catching multiple exception types at once becomes a commonly-expected language feature, an exception might be added for "extra" `catch` clauses in languages that do not offer the ability. This possibility is not excluded, but evaluations of whether or not to add such future exceptions should err on the side of conservatism. That is, new exceptions should come slowly.

On the other hand, if a future version of the COBOL standard adds an "else if" structure, the tendency should be to drop the COBOL "`else … if`" exception (described below) as soon as is practical.

To date, three exceptions have been identified:

COBOL: Missing `else if`

For COBOL, which lacks an `else if` structure, an `if` as the only statement in an `else` clause does not incur a nesting penalty. Additionally, there is no increment for the `else` itself. That is, an `else` followed immediately by an `if` is treated as an `else if`, even though syntactically it is not.

For example:

```
IF condition1          // +1 structure, +0 for nesting
  ...
ELSE
  IF condition2        // +1 structure, +0 for nesting
    ...
  ELSE
    IF condition3      // +1 structure, +0 for nesting
      statement1
      IF condition4    // +1 structure, +1 for nesting
        ...
      END-IF
    END-IF
  ENDIF
ENDIF.
```

JavaScript: Missing class structures

Despite the recent addition of classes to JavaScript by the ECMAScript 6 specification, the feature is not yet widely adopted. In fact, many popular frameworks require the continued use of the compensating idiom: the use of an outer function as a stand-in to create a kind of namespace or faux class. So as not to penalize JavaScript users, such outer functions are ignored when they are used purely as a declarative mechanism, that is when they contain only declarations at the top level.

However, the presence at the top level of a function (i.e. not nested inside a sub-function) of statements subject to structural increments indicates something other than a pure declarative usage. Consequently, such functions should receive a standard treatment.

For example:

```
function(...) {                 // declarative; ignored
  var foo;

  bar.myFun = function(…) {     // nesting = 0
    if(condition) {             // +1
      ...
    }
  }
}                               // total complexity = 1


function(...) {                 // non-declarative; not ignored
  var foo;
  if (condition) {              // +1; top-level structural increment
    ...
  }

  bar.myFun = function(…) {     // nesting = 1
    if(condition) {             // +2
      ...
    }
  }
}                               // total complexity = 3
```

Python: Decorators

Python's decorator idiom allows additional behavior to be added to an existing function without modifying the function itself. This addition is accomplished with the use of nested functions in the decorator providing the additional behavior. In order not to penalize Python coders for the use of a common feature of their language, an exception has been added. However, an attempt has been made to define the exception narrowly. Specifically, to be eligible for the exception, a function may contain only a nested function and a return statement.

For example:

```
def a_decorator(a, b):
  def inner(func):          # nesting = 0
    if condition:           # +1
      print(b)
    func()
  return inner              # total = 1

def not_a_decorator(a, b):
  my_var = a*b
  def inner(func):          # nesting = 1
    if condition:           # +1 structure, +1 nesting
      print(b)
    func()
  return inner              # total = 2

def decorator_generator(a):
  def generator(func):
    def decorator(func):    # nesting = 0
      if condition:         # +1
        print(b)
      return func()
    return decorator
  return generator          # total = 1
```

Appendix B: Specification

The purpose of this section is to give a concise enumeration of the structures and circumstances that increment Cognitive Complexity, subject to the exceptions listed in *Appendix A*. This is meant to be a comprehensive listing without being language-exhaustive. That is, if a language has an atypical spelling for a keyword, such as `elif` for `else if`, its omission here is not intended to omit it from the specification.

B1. Increments

There is an increment for each of the following:

- `if`, `else if`, `else`, ternary operator
- `switch`
- `for`, `foreach`
- `while`, `do while`
- `catch`
- `goto LABEL`, `break LABEL`, `continue LABEL`, `break NUMBER`, `continue NUMBER`
- sequences of binary logical operators
- each method in a recursion cycle

B2. Nesting level

The following structures increment the nesting level:

- `if`, `else if`, `else`, ternary operator
- `switch`
- `for`, `foreach`
- `while`, `do while`
- `catch`
- nested methods and method-like structures such as lambdas

B3. Nesting increments

The following structures receive a nesting increment commensurate with their nested depth inside *B2* structures:

- `if`, ternary operator
- `switch`
- `for`, `foreach`
- `while`, `do while`
- `catch`

Appendix C: Examples

From `org.sonar.java.resolve.JavaSymbol.java` in the SonarJava analyzer:

```
@Nullable
private MethodJavaSymbol overriddenSymbolFrom(ClassJavaType classType) {
  if (classType.isUnknown()) {                                  // +1
    return Symbols.unknownMethodSymbol;
  }

  boolean unknownFound = false;
  List<JavaSymbol> symbols = classType.getSymbol().members().lookup(name);
  for (JavaSymbol overrideSymbol : symbols) {                   // +1
    if (overrideSymbol.isKind(JavaSymbol.MTH)                   // +2 (nesting = 1)
          && !overrideSymbol.isStatic()) {                      // +1

      MethodJavaSymbol methodJavaSymbol = (MethodJavaSymbol)overrideSymbol;
      if (canOverride(methodJavaSymbol)) {                      // +3 (nesting = 2)
        Boolean overriding = checkOverridingParameters(methodJavaSymbol,
                  classType);
        if (overriding == null) {                               // +4 (nesting = 3)
          if (!unknownFound) {                                  // +5 (nesting = 4)
            unknownFound = true;
          }
        } else if (overriding) {                                // +1
          return methodJavaSymbol;
        }
      }
    }
  }

  if (unknownFound) {                                           // +1
    return Symbols.unknownMethodSymbol;
  }

  return null;
}                                       // total complexity = 19
```

From `com.persistit.TimelyResource.java` in sonar-persistit:

```
private void addVersion(final Entry entry, final Transaction txn)
      throws PersistitInterruptedException, RollbackException {
  final TransactionIndex ti = _persistit.getTransactionIndex();
  while (true) {                                          // +1
    try {
      synchronized (this) {
        if (frst != null) {                               // +2 (nesting = 1)
          if (frst.getVersion() > entry.getVersion()) {   // +3 (nesting = 2)
            throw new RollbackException();
          }
          if (txn.isActive()) {                           // +3 (nesting = 2)
            for                                           // +4 (nesting = 3)
                (Entry e = frst; e != null; e = e.getPrevious()) {
              final long version = e.getVersion();
              final long depends = ti.wwDependency(version,
                   txn.getTransactionStatus(), 0);
              if (depends == TIMED_OUT) {                 // +5 (nesting = 4)
                throw new WWRetryException(version);
              }
              if (depends != 0                            // +5 (nesting = 4)
                  && depends != ABORTED) {                // +1
                throw new RollbackException();
              }
            }
          }
        }
        entry.setPrevious(frst);
        frst = entry;
        break;
      }
    } catch (final WWRetryException re) {                 // +2 (nesting = 1)
      try {
        final long depends = _persistit.getTransactionIndex()
            .wwDependency(re.getVersionHandle(),txn.getTransactionStatus(),
                 SharedResource.DEFAULT_MAX_WAIT_TIME);
        if (depends != 0                                  // +3 (nesting = 2)
            && depends != ABORTED) {                      // +1
          throw new RollbackException();
        }
      } catch (final InterruptedException ie) {           // +3 (nesting = 2)
        throw new PersistitInterruptedException(ie);
      }
    } catch (final InterruptedException ie) {             // +2 (nesting = 1)
      throw new PersistitInterruptedException(ie);
    }
  }
}                                       // total complexity = 35
```

From `org.sonar.api.utils.WildcardPattern.java` in SonarQube:

```
private static String toRegexp(String antPattern,
      String directorySeparator) {

  final String escapedDirectorySeparator = '\\' + directorySeparator;
  final StringBuilder sb = new StringBuilder(antPattern.length());
  sb.append('^');
  int i = antPattern.startsWith("/") ||                  // +1
      antPattern.startsWith("\\") ? 1 : 0;               // +1

  while (i < antPattern.length()) {                      // +1

    final char ch = antPattern.charAt(i);
    if (SPECIAL_CHARS.indexOf(ch) != -1) {               // +2 (nesting = 1)
      sb.append('\\').append(ch);
    } else if (ch == '*') {                              // +1
      if (i + 1 < antPattern.length()                    // +3 (nesting = 2)
            && antPattern.charAt(i + 1) == '*') {        // +1

        if (i + 2 < antPattern.length()                  // +4 (nesting = 3)
            && isSlash(antPattern.charAt(i + 2))) {      // +1
          sb.append("(?:.*")
            .append(escapedDirectorySeparator).append("|)");
          i += 2;
        } else {                                         // +1
          sb.append(".*");
          i += 1;
        }
      } else {                                           // +1
        sb.append("[^").append(escapedDirectorySeparator).append("]*?");
      }
    } else if (ch == '?') {                              // +1
      sb.append("[^").append(escapedDirectorySeparator).append("]");
    } else if (isSlash(ch)) {                            // +1
      sb.append(escapedDirectorySeparator);
    } else {                                             // +1
      sb.append(ch);
    }
    i++;
  }

  sb.append('$');
  return sb.toString();
}                                    // total complexity = 20
```

From `model.js` in YUI

```
save: function (options, callback) {
  var self = this;

  if (typeof options === 'function') {                    // +1
    callback = options;
    options  = {};
  }

  options || (options = {});                              // +1

  self._validate(self.toJSON(), function (err) {
    if (err) {                                            // +2 (nesting = 1)
      callback && callback.call(null, err);               // +1
      return;
    }

    self.sync(self.isNew() ? 'create' : 'update',         // +2 (nesting = 1)
      options, function (err, response) {
      var facade = {
          options : options,
          response: response
        },
      parsed;

      if (err) {                                          // +3 (nesting = 2)
        facade.error = err;
        facade.src   = 'save';
        self.fire(EVT_ERROR, facade);

      } else {                                            // +1
        if (!self._saveEvent) {                           // +4 (nesting = 3)
          self._saveEvent = self.publish(EVT_SAVE, {
            preventable: false
          });
        }

        if (response) {                                   // +4 (nesting = 3)
          parsed = facade.parsed = self._parse(response);
          self.setAttrs(parsed, options);
        }

        self.changed = {};
        self.fire(EVT_SAVE, facade);
      }

      callback && callback.apply(null, arguments);        // +1
    });
  });
  return self;
}                                    // total complexity = 20
```

Change log

Version 1.1
6 February 2017

- Update the section on recursion to include indirect recursion.
- Add the “Hybrid” increment type and use it to clarify the handling of `else` and `else if`; they are not subject to a nesting increment, but do increase the nesting level.
- Clarify that Cognitive Complexity is only concerned with binary boolean operators.
- Correct `getWords` to truly have a Cyclomatic Complexity of 4.
- Add Appendix A: Compensating Usages.
- Update the copyright.
- Initiate Change log.

Version 1.2
19 April 2017

- Textual adjustments and corrections, such as the use of the word "understandability" instead of "maintainability".
- Add explanation for why a hybrid increment is assessed on `else if` and `else` instead of a structural increment.
- Add Appendix B: Specification.
- Add Appendix C: Examples.

Version 1.3
15 March 2018

- Extend Appendix A to include a compensating usage for Python decorators.
- Update the copyright.

Version 1.4
10 Sept 2018

- Clarify what type of method nesting increments the nesting level.

Version 1.5
5 April 2021

- Make explicit that all multi-level uses of `break` and `continue` receive a fundamental increment.
- New cover page and footer, plus typo corrections and other minor visual and readability updates.
- Update the copyright.

Version 1.6
9 July 2021

- Remove author's title.