sonar

# {Когнитивная сложность}

## новый способ измерения понятности кода

Автор: G. Ann Campbell

# Оглавление

**29 августа 2023, версия 1.7**

# Аннотация

Цикломатическая сложность изначально была сформулирована как мера измерения «тестируемости и удобства сопровождения» потока управления программного модуля. Хотя она превосходно справляется с определением значения первого параметра, лежащая в ее основе математическая модель не позволяет добиться приемлемой точности для измерения второго. В этом докладе описывается новая метрика, которая отказывается от использования математических моделей для оценки программного кода, чтобы исправить недостатки цикломатического подхода и более точно определить относительную сложность понимания и, следовательно, сложность доработки методов, классов и приложений.

Примечание касательно используемой терминологии: хотя когнитивная сложность – это независимая от языка метрика, которая в равной степени применима как к файлам и классам, так и к методам, процедурам, функциям и т. д., в тексте данного доклада для удобства используются термины ООП «класс» и «метод».

# Вступление

Цикломатическая сложность, разработанная Томасом Дж. Маккейбом, уже давно стала де-факто стандартом для количественного выражения сложности потока управления методов приложения. Первоначально этот подход предназначался для «идентификации программных модулей, которые будет сложно тестировать или сопровождать» [1], и хотя он помогает точно рассчитать минимальное количество тестовых примеров, необходимых для полного покрытия метода, но не позволяет с приемлемой точностью оценить степень понятности кода. Это связано с тем, что методы с одинаковой цикломатической сложностью не обязательно в равной мере сложны для разработчика, занимающегося их сопровождением, что приводит к ощущению «ложной тревоги» – рассчитанная цикломатическая сложность одних управляющих конструкций кажется завышенной, а других – заниженной.

В то же цикломатическая сложность больше не является всеобъемлющей метрикой. Она была разработана в 1976 году для языка Фортран и не включает конструкции современных языков программирования – такие, как **try/catch** и лямбда-выражения.

И, наконец, поскольку минимальный показатель цикломатической сложности любого из методов равен единице, невозможно выяснить, является ли какой-либо класс с высокой совокупной цикломатической сложностью *большим и простым в сопровождении классом предметной области* или же *небольшим классом со сложным потоком управления*. Широко известно, что если перейти от классов на уровень выше, то окажется, что показатель цикломатической сложности приложения коррелирует с общим числом строк его кода. Другими словами, применимость цикломатической сложности ограничена уровнем методов.

В качестве способа решения описанных выше проблем была сформулирована концепция когнитивной сложности, которая учитывает конструкции современных языков программирования и позволяет проводить оценку на уровне классов и приложений. Что ещё более важно, эта концепция отходит от практики оценки кода на основе математических моделей, чтобы количественно измерить степень умственных или когнитивных усилий, необходимых для понимания потока управления.

## Иллюстрация проблемы

Будет полезно начать обсуждение метрики когнитивной сложности с рассмотрения примера проблемы, для решения которой она предназначена. Два следующих метода имеют одинаковую цикломатическую сложность, но разительно отличаются с точки зрения простоты восприятия.

```
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;
}                    // Цикломатическая сложность = 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";
    }
}      // Когнитивная сложность = 4
```

Используя математическую модель, лежащую в основе цикломатической сложности, мы вычислили для обоих методов равный «вес». Однако интуитивно понятно, что поток управления метода **sumOfPrimes** сложнее осознать, чем поток управления метода **getWords**. Вот почему при разработке метрики когнитивной сложности был произведен отказ от использования математических моделей для оценки потока управления в пользу набора простых правил, количественно выражающих интуицию программиста.

## Основные критерии и методология

Оценка уровня когнитивной сложности основывается на трёх основных правилах:

**1**. Игнорируйте конструкции, позволяющие объединить несколько операторов в один без потери читабельности.
**2**. Инкрементируйте метрику сложности для каждого нарушения непрерывности потока управления в «линейном» (без вложенных конструкций) коде.
**3**. Увеличивайте метрику сложности за каждую вложенную конструкцию, нарушающую непрерывность потока управления, на 1 + n, где n – уровень вложенности.

Кроме того, приращения когнитивной сложности разделяются на 4 типа:

**A**. Вложенные (N) – рассматриваются для операторов управления, вложенных друг в друга.
**B**. Структурированные (S) – рассматриваются для операторов управления, которые увеличивают уровень вложенности потока управления (см. п. 3 выше).
**C**. Фундаментальные (F) – рассматриваются для выражений, не увеличивающих уровень вложенности.
**D**. Гибридные (H) – рассматриваются для операторов управления, не нарушающих поток управления, но увеличивающих уровень вложенности.

Хотя тип приращения не учитывается в расчётах (каждое из них добавляет +1 к финальной оценке), формулировка различий между ними облегчает понимание того, в каких случаях вложенные конструкции влияют на когнитивную сложность, а в каких нет.

Данные правила и лежащие в их основе принципы подробно описаны в следующих разделах.

## Игнорируйте «сокращения»

Одним из основных принципов, использованных при разработке метрики когнитивной сложности, является создание стимула для программистов применять хорошие практики кодирования. То есть при расчёте сложности следует игнорировать всё, что делает код более читабельным.

Ярким примером является использование методов. Разбиение кода на отдельные фрагменты позволяет объединить несколько операторов в единственный объект с запоминающимся именем, т. е. «сократить» общий объём линейного кода. Таким образом, методы не увеличивают когнитивную сложность.

При определении когнитивной сложности также игнорируются встречающиеся во многих языках операторы нулевого слияния, которые позволяют объединить несколько строк кода в одну. Например, два следующих примера равнозначны с точки зрения получаемого результата:

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

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

Левый пример (в котором используется оператор опциональной последовательности) требует некоторого времени на понимание, а правый понятен сразу, если вы знаете синтаксис соответствующего оператора. По этой причине при расчёте метрики когнитивной сложности подобные операторы игнорируются.

# Инкрементируйте метрику сложности за каждый «разрыв» линейного потока управления

Другой базовый принцип когнитивной сложности заключатся в том, что операторы, которые нарушают обычный линейный поток управления (то есть «сверху вниз, слева направо»), требуют дополнительных усилий для понимания. Для учёта этих дополнительных усилий значение метрики когнитивной сложности должно быть увеличено на +1 для каждого:

- оператора цикла (**for**, **while**, **do while** и т. п.)
- условного оператора (тернарного оператора, **if, #if, #ifdef** и т. п.)

Условные операторы могут включать в себя вложенные, обуславливающие приращения когнитивной сложности типа H:

- **else if, elif, else** и т. п.

Они не увеличивают когнитивную сложность, потому что умственные усилия уже были произведены при понимании первого условного оператора в конструкции (**if** и т. п.).

Всё описанное выше знакомо для тех, кто привык к метрике цикломатической сложности. В дополнение к этому значение метрики когнитивной сложности увеличивают перечисленные в следующих подпунктах операторы.

### Оператор catch

Оператор **catch** представляет собой ветвление потока управления – такое же, как и оператор **if**. Таким образом, каждый оператор **catch** приводит к увеличению когнитивной сложности типа S. Обратите внимание, что каждый такой оператор увеличивает значение метрики когнитивной сложности на +1 независимо от того, сколько типов исключений в нем обрабатывается. Блоки **try** и **final** при оценке сложности игнорируются.

### Оператор switch

Оператор **switch** приводит к увеличению когнитивной сложности типа S. Каждый такой оператор увеличивает значение метрики когнитивной сложности на +1 независимо от числа входящих в его состав меток **case**.

В концепции цикломатической сложности оператор **switch** рассматривается как аналог конструкции **if ... else if**. То есть каждая метка **case** увеличивает метрику цикломатической сложности, потому что создаёт новую ветвь в рамках математической модели потока управления.

Но с точки зрения программиста, сопровождающего код, оператор **switch**, в котором значение переменной сравнивается с явно указанным набором литералов, понять гораздо легче, чем конструкцию **if ... else if**, в состав которой может входить произвольное количество сравнений, состоящих из произвольного числа переменных и значений.

В общем, в конструкцию **if ... else if** нужно внимательно вчитываться, а оператор **switch** часто можно понять с первого взгляда.

## Последовательность логических операторов

По тем же причинам отдельный бинарный логический оператор не увеличивает значение метрики когнитивной сложности. Вместо этого каждая последовательность подобных операторов приводит к увеличению когнитивной сложности типа F. В качестве примера рассмотрим два двухстрочных блока кода:

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

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

Понять вторую строку в каждом из блоков ненамного сложнее, чем первую. С другой стороны, для понимания каждой из следующих двух строк требуются заметно разные усилия:

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

Поскольку чем больше в выражении разных логических операторов, тем сложнее его понять, то значение метрики когнитивной сложности увеличивается с каждой новой «последовательностью» логических операторов:

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

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

Хотя по сравнению с концепцией цикломатической сложности здесь делается «скидка», каждая новая последовательность бинарных логических операторов (в том числе в выражениях присваивания переменных, вызовах методов и операторах **return**) увеличивает значение метрики когнитивной сложности на 1.

### Рекурсия

В отличие от цикломатической сложности, метрика когнитивной сложности увеличивается на 1 (тип F) за каждый прямо или косвенно рекурсивно вызываемый метод. У этого решения есть две причины. Во-первых, рекурсия представляет собой своего рода «метацикл», а циклы увеличивают когнитивную сложность кода. Во-вторых, когнитивная сложность связана с оценкой относительной трудности понимания потока управления метода – а даже некоторые опытные программисты считают, что рекурсию понять сложно.

### Метки и переходы

Оператор **goto** приводит к увеличению когнитивной сложности типа F, равно как операторы **break** и **continue**, используемые для перехода к символьным и (в некоторых языках) числовым меткам. Поскольку оператор **return**, используемый в начале метода, может сделать код намного понятнее, то он и аналогичные ему операторы не приводят к увеличению к когнитивной сложности.

## Увеличивайте метрику сложности за каждую вложенную конструкцию, нарушающую непрерывность потока управления

Интуитивно ясно, что пять линейно размещенных конструкций **if** и **for** понять легче, чем те же конструкции, вложенные друг в друга (независимо от общего количества путей потока управления для каждого из случаев). Поскольку вложенность увеличивает умственные усилия, требуемые для понимания кода, то для оценки её вклада используется приращение метрики когнитивной сложности типа N.

В частности, когда конструкция, вызывающая приращение когнитивной сложности типа S или H, вложена в другую конструкцию, то значение метрики когнитивной сложности увеличивается на +1 и еще на +1 за каждый уровень вложенности. В следующем примере метод или оператор **try** не создают приращение когнитивной типа N, потому что они не вызывают приращения типа S или H:

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

Однако операторы **if**, **for**, **while** и **catch** вызывают как структурные (S), так и вложенные (N) приращения сложности.

Кроме того, методы верхнего уровня игнорируются при оценке сложности; лямбда-выражения, вложенные методы и другие подобные объекты не добавляют приращения сложности (S), но учитываются при оценке уровня вложенности других конструкций:

```
void myMethod2 () {
  Runnable r = () -> {          // +0 (но вложенность теперь = 1)
    if (condition1) { … }       // +2 (вложенность = 1)
  };
}                               // Когнитивная сложность = 2


#if DEBUG                 // +1 за 'if'
void myMethod2 () {       // +0 (вложенность пока что = 0)
  Runnable r = () -> {    // +0 (но вложенность теперь = 1)
    if (condition1) { … }// +3 (вложенность = 2)
  };
}                               // Когнитивная сложность = 4
#endif
```

# Результаты

Основной целью разработки концепции когнитивной сложности являлось создание методики оценки, которая достаточно точно отражала бы относительную понятность методов. К второстепенным целями относилась поддержка конструкций современных языков программирования и возможность оценки не только методов, но и более высокоуровневых объектов (классов и приложений). Очевидно, что поддержка конструкций современных языков была достигнута. Степень достижения двух других целей рассматривается ниже.

## Интуитивно «верная» оценка сложности

В начале доклада были рассмотрены два метода с одинаковой цикломатической сложностью, но явно отличающиеся уровнем понятности. Пришло время вернуться к этим методам и рассчитать для них метрику когнитивной сложности.

```
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;
}                    // Когнитивная сложность = 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";
    }
}      // Когнитивная сложность = 1
```

Полученные с помощью алгоритма когнитивной сложности оценки существенно различаются, что гораздо лучше отражает их относительную понятность.

## Оценка высокоуровневых объектов

Кроме того, поскольку сам факт наличия в коде метода не увеличивает метрику когнитивной сложности, то суммирование метрик отдельных методов класса становится показательным (в отличие от цикломатической сложности). Теперь вы можете определить разницу между классом предметной области (с большим количеством простых геттеров и сеттеров) и классом, включающим в себя сложный поток управления, просто сравнивания значения их метрик. Таким образом, когнитивная сложность становится инструментом измерения относительной понятности классов и приложений.

# Заключение

Написание и сопровождение кода – это процессы человеческой деятельности. Результаты этих процессов должны соответствовать математическим моделям, но сами процессы этими моделями не опишешь. Вот почему с помощью них невозможно выразить степень понятности кода.

Концепция когнитивной сложности отказывается от практики использования математических моделей для оценки сопровождаемости ПО. Она базируется на прецедентах, сформулированных в процессе разработки метрики цикломатической сложности, но использует «человеческие» суждения для определения конструкций, которые вносят вклад в сложность кода, и принципов подсчёта этого вклада. В результате, полученная метрика кажется программистам более справедливой оценкой относительной понятности кода по сравнению с уже существующими моделями. Кроме того, поскольку в рамках когнитивной сложности сам факт выделения фрагмента кода в метод не приводит к увеличению метрики сложности, то она предоставляет более справедливые оценки не только на уровне методов, но и на уровне классов и приложений.

# Список использованной литературы

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

# Приложение А: Исключительные случаи

Когнитивная сложность разрабатывалась как метрика, не зависящая от языка программирования, но нельзя игнорировать тот факт, что разные языки предлагают разработчику разный функционал. Например, в COBOL отсутствует поддержка оператора **else if**, а в JavaScript до недавнего времени не было классов. К сожалению, эти недостатки языков не отменяют потребностей разработчиков, что приводит к попыткам создать что-то подобное с помощью подручных инструментов. В таких случаях строгое следование правилам оценки когнитивной сложности приводит к непропорционально завышенным результатам.

По этой причине, и чтобы не вводить «штрафы» одним языкам по сравнению с другими, для идиом, компенсирующих недостатки языка (т. е. замены конструкций, которые широко используются и ожидаются в большинстве современных языков, но отсутствуют в рассматриваемом языке – например, **else if** в COBOL) могут сделаны «скидки» при определении метрики когнитивной сложности.

С другой стороны, если в каком-то языке есть уникальные нововведения (например, в Java 7 появилась возможность перехватывать сразу несколько типов исключений), то отсутствие этих нововведений в других языках не следует считать их недостатком и, соответственно, в данном случае «скидок» не предусмотрено.

В свою очередь это означает, что если перехват нескольких типов исключений становится ожидаемым функционалом языка, то для «дополнительных» операторов **catch** в тех языках, в которых такой функционал отсутствует, могут быть сделаны «скидки» при оценке когнитивной сложности. Но решение о том, делать эти «скидки» или нет, должно быть принято с консервативных позиций и представлений о развитии языков программирования. То есть такие новые «исключительные случаи» должны появляться медленно.

Если же в будущей версии стандарта языка COBOL будет добавлен оператор **else if**, то попытка заменить его каким-то набором конструкций больше не будет считаться исключительным случаем. На сегодняшний день определено 3 исключительных случая, которые описаны в следующих подпунктах.

### COBOL: отсутствие else if

В языке COBOL отсутствует оператор **else if**, поэтому если внутри **else** используется **if** (и только он), то это не считается увеличением уровня вложенности. Кроме того, сам факт использования оператора **else** не приводит к увеличению метрики когнитивной сложности. То есть **else**, за которым следует **if**, расценивается как **else if**, хотя синтаксически это не так.

Пример:

```
IF condition1         // +1 (S), +0 для вложенности
  ...
ELSE
  IF condition2       // +1 (S), +0 для вложенности
    ...
  ELSE
    IF condition3     // +1 (S), +0 для вложенности
      statement1
      IF condition4   // +1 (S), +1 для вложенности
        ...
      END-IF
    END-IF
  ENDIF
ENDIF.
```

### JavaScript: отсутствие классов

Несмотря на недавнее добавление классов в JavaScript в спецификации ECMAScript 6, этот функционал еще не получил широкого распространения. Фактически, многие популярные фреймворки требуют постоянного использования следующей идиомы: создания внешней (outer) функции для конструирования чего-то вроде пространства имен или аналога класса. Чтобы не «штрафовать» разработчиков, использующих JavaScript, такие внешние функции игнорируются при расчёте метрики когнитивной сложности, если они применяются исключительно для объявления объектов в теле самой функции (top-level).

Однако наличие в теле функции (т. е. не во вложенных в нее подфункциях) операторов, обеспечивающих увеличение когнитивной сложности типа S (структурного) указывает, что функция используется не только для объявления объектов и, следовательно, должна учитываться при расчёте метрики.

Пример:

```
function(...) {                // только объявление; игнорируется
  var foo;

  bar.myFun = function(…) {    // вложенность = 0
    if(condition) {            // +1
      ...
    }
  }
}                              // сложность = 1


function(...) {                // не только объявление; учитывается
  var foo;
  if (condition) {             // +1 (S)
    ...
  }

  bar.myFun = function(…) {    // вложенность = 1
    if(condition) {            // +2
      ...
    }
  }
}                              // сложность = 3
```

### Python: декораторы

Концепция декораторов, реализованная в языке Python, позволяет добавлять дополнительное поведение к функциям без изменения их исходного кода. Дополнительное поведение добавляется за счет функций, указываемых в декораторе. Чтобы не «штрафовать» разработчиков, использующих Python, за эксплуатацию общей особенности из языка – декораторы считаются исключительным случаем и не учитываются при расчёте метрики когнитивной сложности. Однако данный исключительный случай ограничен следующим критерием: функция должна включать в себя только вложенную функцию (определяемую декоратором) и оператор возврата.

Пример:

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

def not_a_decorator(a, b):
  my_var = a*b
  def inner(func):          # вложенность = 1
    if condition:           # +1 (S), +1 для вложенности
      print(b)
    func()
  return inner              # сложность = 2

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

# Приложение B: Спецификация

Цель этого раздела – привести перечисление операторов и обстоятельств, увеличивающих когнитивную сложность (за исключением случаев, описанных в приложении A). Список является полным и не учитывает особенности наименования операторов в конкретных языках (например, если в языке используется обозначение **elif** вместо **else if** – то его отсутствие в списке не подразумевает, что для него нужно делать исключение).

1. Операторы и конструкции, приводящие к инкременту метрики когнитивной сложности

- **if**, **else if**, **else**, тернарный оператор;
- **for**, **foreach**;
- **while**, **do while**;
- **catch**;
- **goto LABEL, break LABEL, continue LABEL, break NUMBER, continue NUMBER**;
- последовательность логических операторов;
- каждый рекурсивно вызываемый метод.

2. Операторы и конструкции, приводящие к инкременту уровня вложенности

- **if**, **else if, else**, тернарный оператор;
- **for, foreach**;
- **while, do while;**
- **catch**;
- вложенные методы и объекты, похожие на методы (например, лямбда-функции).

3. Операторы и конструкции, приводящие к увеличению метрики когнитивной сложности, соразмерному их уровню вложенности в конструкции из пп. 2 (см. примеры далее).

- **if**, тернарный оператор;
- **for**, **for each**;
- **while**, **do while**;
- **catch**.

# Приложение C: Примеры расчёта метрики когнитивной сложности

Из **org.sonar.java.resolve.JavaSymbol.java** в анализаторе **SonarJava**:

```
@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 (вложенность = 1)
              && !overrideSymbol.isStatic()) {                  // +1

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

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

  return null;
}                                                               // сложность = 19
```

Из **com.persistit.TimelyResource.java** в **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 (вложенность = 1)
          if (frst.getVersion() > entry.getVersion()) {           // +3 (вложенность = 2)
            throw new RollbackException();
          }
          if (txn.isActive()) {                                    // +3 (вложенность = 2)
            for                                                    // +4 (вложенность = 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 (вложенность = 4)
                throw new WWRetryException(version);
              }
              if (depends != 0                                    // +5 (вложенность = 4)
                     && depends != ABORTED) {                     // +1
                throw new RollbackException();
              }
            }
          }
        }
        entry.setPrevious(frst);
        frst = entry;
        break;
      }
    } catch (final WWRetryException re) {                         // +2 (вложенность = 1)
      try {
        final long depends = _persistit.getTransactionIndex()
              .wwDependency(re.getVersionHandle(),txn.getTransactionStatus(),
              SharedResource.DEFAULT_MAX_WAIT_TIME);
        if (depends != 0                                          // +3 (вложенность = 2)
              && depends != ABORTED) {                            // +1
          throw new RollbackException();
        }
      } catch (final InterruptedException ie) {                   // +3 (вложенность = 2)
        throw new PersistitInterruptedException(ie);
      }
    } catch (final InterruptedException ie) {                     // +2 (вложенность = 1)
      throw new PersistitInterruptedException(ie);
    }
  }
}                                                                 // сложность = 35
```

Из **org.sonar.api.utils.WildcardPattern.java** в **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 (вложенность = 1)
      sb.append('\\').append(ch);
    } else if (ch == '*') {                                // +1
      if (i + 1 < antPattern.length()                      // +3 (вложенность = 2)
              && antPattern.charAt(i + 1) == '*') {        // +1

        if (i + 2 < antPattern.length()                    // +4 (вложенность = 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();
}                                                          // сложность = 20
```

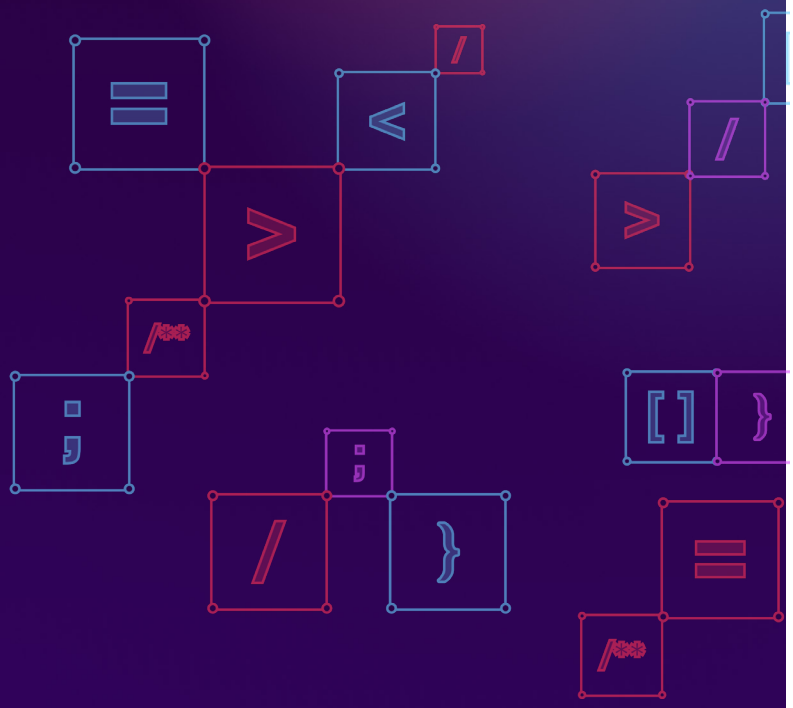
Из **model.js** в **YUI**:

```
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 (вложенность = 1)
      sb.append('\\').append(ch);
    } else if (ch == '*') {                                    // +1
      if (i + 1 < antPattern.length()                          // +3 (вложенность = 2)
              && antPattern.charAt(i + 1) == '*') {            // +1

        if (i + 2 < antPattern.length()                        // +4 (вложенность = 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();
}                                                              // сложность = 20
```

# История изменений

| Версия | Дата | Список изменений |
|---|---|---|
| 1.1 | 06.02.2017 | 1. В разделе о рекурсии добавлено упоминание косвенной рекурсии<br>2. Добавлено описание приращений гибридного типа (H) и пояснено их применение для операторов **else** и **else if**. Они не увеличивают метрику когнитивной сложности, но увеличивают уровень вложенности<br>3. Добавлены уточнение в пункт Последовательность логических операторов<br>4. Исправлен метод **getWords** из пункта Иллюстрация проблемы<br>5. Добавлен пункт Приложение A: Исключительные случаи<br>6. Обновлены авторские права[1]<br>7. Добавлена история изменений |
| 1.2 | 19.04.2017 | 1. Изменения в используемой терминологии<br>2. В пункт Увеличивайте метрику сложности за каждую вложенную конструкцию, нарушающую непрерывность потока управления добавлены пояснения по различиям приращений гибридного типа (H) и структурного типа (S)<br>3. Добавлен пункт Приложение B: Спецификация<br>4. Добавлен пункт Приложение C: Примеры расчёта метрики когнитивной сложности |
| 1.3 | 15.03.2018 | 1. В приложение A добавлены декораторы Python<br>2. Обновлены авторские права |
| 1.4 | 10.09.2018 | 1. Добавлено уточнение, какие типы вложенности методов увеличивает уровень вложенности |
| 1.5 | 05.04.2021 | 1. Уточнено, что все многоуровневые варианты операторов **break** и **continue** приводят к приращению сложности фундаментального типа (F)<br>2. Изменена титульная страница и нижний колонтитул. Исправлен ряд опечаток. Внесены мелкие изменения, связанные с версткой[1]<br>3. Обновлены авторские права |
| 1.6 | 09.06.2021 | 1. Удалено авторское заглавие |
| 1.7 | 29.08.2023 | 1. Оформление документа приведено к соответствию с брендбуком компании[1]<br>2. Внесены изменения, связанные с версткой[1] |

[1] Не учитывается в данном переводе.



www.sonarsource.com

Лидирующее в отрасли решение от Sonar позволяет разработчикам и компаниям достичь состояния "чистого кода".
Ее комммерческие решения с открытым исходным кодом - SonarLint, SonarCloud и SonarCube - поддерживают более 30 языков программирования, фреймворков и инфраструктурных технологий.
Решениям от Sonar доверяют более 400 000 компаний по всему миру, и эти решения считаются неотъемлемой частью процессов создания лучшего ПО.

© 2008-2023, SonarSource S.A, Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource SA. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.