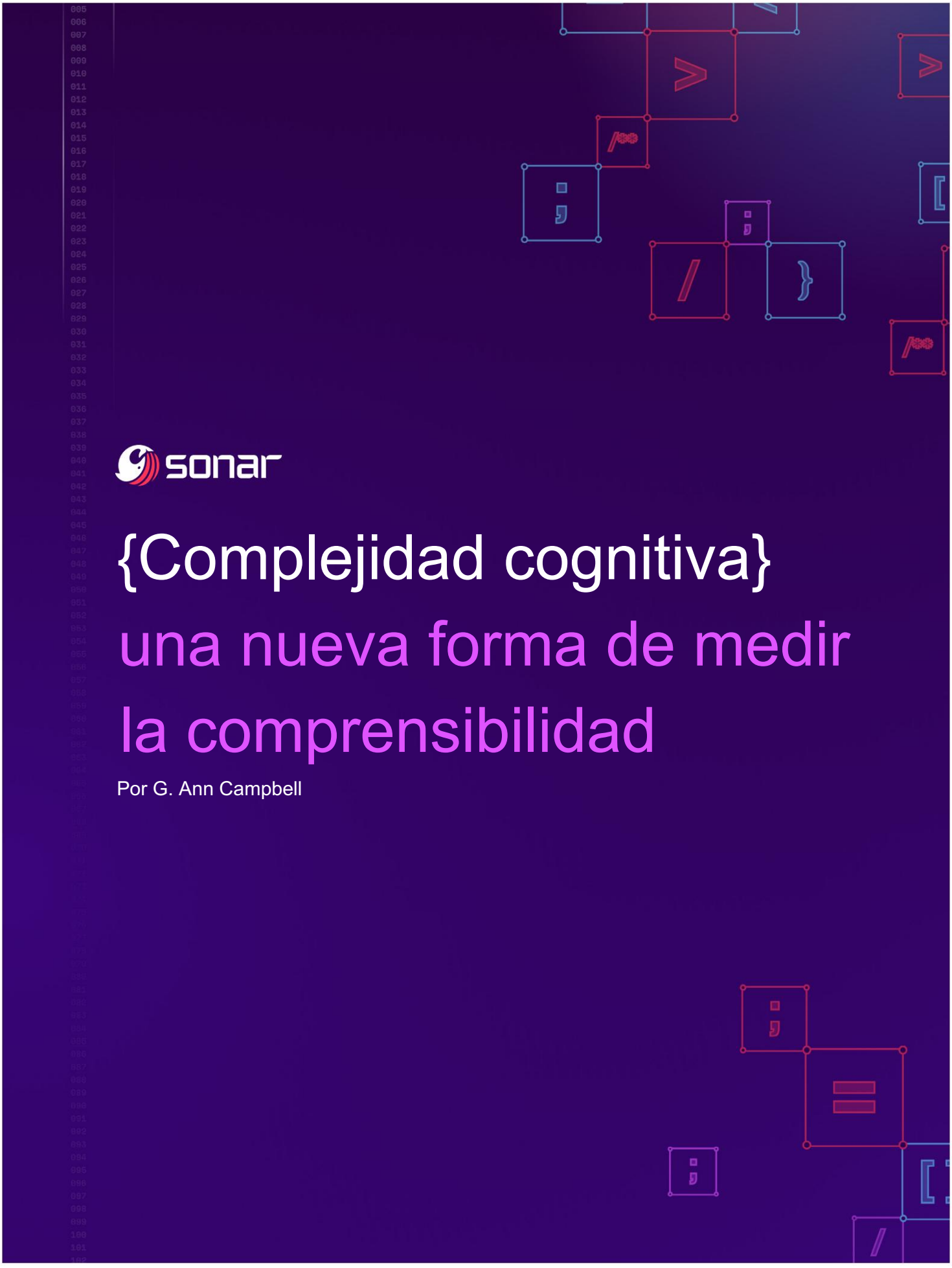

{Complejidad cognitiva}

una nueva forma de medir la comprensibilidad

Por G. Ann Campbell

29 de agosto de 2023, versión 1.7

Abstracto

La complejidad ciclomática se formuló inicialmente como una medida de la "capacidad de prueba y mantenimiento" del flujo de control de un módulo. Si bien destaca en medir lo primero, su modelo matemático subyacente no es satisfactorio a la hora de producir un valor que mida lo segundo. Este documento técnico describe una nueva métrica que rompe con el uso de modelos matemáticos para evaluar código para remediar las deficiencias de Cyclomatic Complexity y producir una medición que refleje con mayor precisión la dificultad relativa de comprensión y, por lo tanto, de mantenimiento de métodos, clases y aplicaciones.

Una nota sobre terminología

Si bien la Complejidad Cognitiva es una métrica neutral al lenguaje que se aplica igualmente a archivos y clases, y a métodos, procedimientos, funciones, etc., la Complejidad de Objetos
Los términos orientados “clase” y “método” se utilizan por conveniencia.

Tabla de contenido

Introducción

La Complejidad Ciclomática de Thomas J. McCabe ha sido durante mucho tiempo el estándar de facto para medir la complejidad del flujo de control de un método. Originalmente estaba destinado a “identificar módulos de software que serán difíciles de probar o mantener”[1], pero si bien calcula con precisión el número mínimo de casos de prueba necesarios para cubrir completamente un método, no es una medida satisfactoria de comprensibilidad. Esto se debe a que los métodos con igual Complejidad Ciclomática no necesariamente presentan la misma dificultad para quien los mantiene, lo que lleva a la sensación de que la medición "llora el lobo" al sobrevalorar algunas estructuras, mientras que subvalora otras.

Al mismo tiempo, la Complejidad Ciclomática ya no es integral. Formulado en un entorno Fortran en 1976, no incluye estructuras de lenguaje modernas como try/catch y lambdas.

Y finalmente, debido a que cada método tiene una puntuación mínima de Complejidad Ciclomática de uno, es imposible saber si una clase determinada con una Complejidad Ciclomática agregada alta es una clase de dominio grande y de fácil mantenimiento, o una clase pequeña con un flujo de control complejo. Más allá del nivel de clase, es ampliamente reconocido que las puntuaciones de Complejidad Ciclomática de las aplicaciones se correlacionan con sus líneas de código totales. En otras palabras, la Complejidad Ciclomática es de poca utilidad por encima del nivel del método.

Como remedio para estos problemas, la Complejidad Cognitiva se ha formulado para abordar las estructuras del lenguaje moderno y producir valores que sean significativos a nivel de clase y aplicación. Más importante aún, se aparta de la práctica de evaluar código basado en modelos matemáticos para que pueda producir evaluaciones del flujo de control que correspondan a las intuiciones de los programadores sobre el esfuerzo mental o cognitivo requerido para comprender esos flujos.

Una ilustración del problema

Es útil comenzar la discusión sobre la Complejidad Cognitiva con un ejemplo del problema que está diseñado para abordar. Los dos métodos siguientes tienen la misma complejidad ciclomática, pero son sorprendentemente diferentes en términos de comprensibilidad.

```
int sumaDePrimes(int máx) { int total = 0;                                  // +1

    SALIDA: for (int i = 1; i <= max; ++i) { // +1
        for (int j = 2; j < i; ++j) { if (i % j == 0) { continuar           // +1
            FUERA;                                                          // +1

            }
        }
        total += yo;
    }
    total de retorno;
}                                       // Complejidad ciclomática 4
```

```
Cadena getWords(int número) { // +1
        cambiar (número) {
            caso 1:                                                         // +1
                devolver "uno";
            caso 2:                                                         // +1
                devolver “una pareja”;
            caso 3:                                                         // +1
                devolver “algunos”;
            por defecto:
                devolver “lotes”;
        }
    }           // Complejidad ciclomática 4
```

El modelo matemático subyacente a Cyclomatic Complexity otorga a estos dos métodos el mismo peso, pero es intuitivamente obvio que el flujo de control de sumOfPrimes es más difícil de entender que el de getWords. Esta es la razón por la que Cognitive Complexity abandona el uso de modelos matemáticos para evaluar el flujo de control en favor de un conjunto de reglas simples para convertir la intuición del programador en números.

Criterios básicos y metodología.

Una puntuación de Complejidad Cognitiva se evalúa según tres reglas básicas:

1. Ignore las estructuras que permiten simplificar la lectura de varias declaraciones. en uno
2. Incrementar (agregar uno) por cada interrupción en el flujo lineal del código.
3. Incremento cuando se anidan estructuras rompedoras de flujo.

Además, una puntuación de complejidad se compone de cuatro tipos diferentes de incrementos:

A. Anidamiento: evaluado para anidar estructuras de flujo de control una dentro de otra

B. Estructural: evaluado en estructuras de control de flujo que están sujetas a un anidamiento. incremento, y que aumentan el recuento de anidamiento

C. Fundamental: evaluado en declaraciones no sujetas a un incremento de anidación

D. Híbrido: evaluado en estructuras de flujo de control que no están sujetas a anidamiento. incremento, pero que sí aumentan el recuento de anidamiento

Si bien el tipo de incremento no hace ninguna diferencia en las matemáticas (cada incremento suma uno a la puntuación final), hacer una distinción entre las categorías de características que se cuentan hace que sea más fácil comprender dónde se aplican y dónde no se aplican los incrementos de anidamiento.

Estas reglas y los principios detrás de ellas se detallan con más detalle a continuación. secciones.

ignorar la taquigrafía

Un principio rector en la formulación de la Complejidad Cognitiva ha sido que debería incentivar buenas prácticas de codificación. Es decir, debería ignorar o descartar características que hacen que el código sea más legible.

La estructura del método en sí es un buen ejemplo. Dividir el código en métodos le permite condensar múltiples declaraciones en una única llamada con un nombre evocador, es decir, "abreviarla". Por tanto, la Complejidad Cognitiva no aumenta durante métodos.

Cognitive Complexity también ignora los operadores de fusión nula que se encuentran en muchos lenguajes, nuevamente porque permiten reducir varias líneas de código en una sola. Por ejemplo, los dos ejemplos de código siguientes hacen lo mismo:

```
MiObj miObj = nulo;
si (a! = nulo) {
    miObj = a.miObj;
}
```

```
MiObj miObj = a?.miObj;
```

El significado de la versión de la izquierda tarda un momento en procesarse, mientras que la versión de la derecha queda inmediatamente clara una vez que se comprende la sintaxis de fusión nula. Por esa razón, Cognitive Complexity ignora los operadores de fusión nula.

Incremento por rupturas en el flujo lineal

Otro principio rector en la formulación de la Complejidad Cognitiva es que las estructuras que rompen el flujo lineal normal del código de arriba a abajo y de izquierda a derecha requieren que sus mantenedores trabajen más duro para comprender ese código. En reconocimiento a este esfuerzo adicional, Cognitive Complexity evalúa incrementos estructurales para:

- Estructuras de bucle: for, while, do while, ...
- Condicionales: operadores ternarios, if, #if, #ifdef, ...

Evalúa incrementos híbridos para:

- más si, elif, más, …

No se evalúa ningún incremento de anidamiento para estas estructuras porque el costo mental ya se pagó al leer el if.

Estos objetivos de incremento les resultarán familiares a quienes estén acostumbrados a la Complejidad Ciclomática. Además, la Complejidad Cognitiva también aumenta para:

capturas

Una captura representa una especie de rama en el flujo de control tanto como un if.
Por lo tanto, cada cláusula de captura da como resultado un incremento estructural de la Complejidad Cognitiva. Tenga en cuenta que una captura solo agrega un punto a la puntuación de Complejidad cognitiva, sin importar cuántos tipos de excepción se detecten. Los bloques try y finalmente se ignoran por completo.

interruptores

Un interruptor y todos sus casos combinados incurren en un único incremento estructural.

En Complejidad ciclomática, un interruptor se trata como un análogo de una cadena if-else if . Es decir, cada caso en el cambio provoca un incremento porque
provoca una bifurcación en el modelo matemático del flujo de control.

Pero desde el punto de vista de un mantenedor, un interruptor (que compara una sola variable con un conjunto de valores literales nombrados explícitamente) es mucho más fácil de entender que una cadena if-else if porque esta última puede hacer cualquier número de comparaciones, usando cualquier número. de variables y valores.

En resumen, una cadena if-else if debe leerse con atención, mientras que un interruptor a menudo se puede captar de un vistazo.

Secuencias de operadores lógicos.

Por razones similares, la Complejidad Cognitiva no aumenta para cada operador lógico binario. En cambio, evalúa un incremento fundamental para cada secuencia de operadores lógicos binarios. Por ejemplo, considere los siguientes pares:

```
a && b
a B C D

un || b
un || segundo || c || d
```

Comprender la segunda línea de cada par no es mucho más difícil que entender la primera. Por otro lado, hay una marcada diferencia en el esfuerzo por comprender las dos líneas siguientes:

```
a B C D
un || b && c || d
```

Debido a que las expresiones booleanas se vuelven más difíciles de entender con operadores mixtos, la complejidad cognitiva aumenta para cada nueva secuencia de operadores similares. Por ejemplo:

```
si (un                          // +1 para `si`
        && antes de Cristo      // +1
        || d || e&&f)           // +1
                                // +1

si (un                          // +1 para `si`
        &&                      // +1
        !(antes de Cristo))     // +1
```

Si bien Cognitive Complexity ofrece un "descuento" para operadores similares en relación con Complejidad ciclomática, incrementa para todas las secuencias de operadores booleanos binarios, como los de asignaciones de variables, invocaciones de métodos y retornos. declaraciones.

recursividad

A diferencia de la Complejidad Ciclomática, la Complejidad Cognitiva agrega un incremento fundamental para cada método en un ciclo de recursividad, ya sea directo o indirecto.
Hay dos motivaciones para esta decisión. En primer lugar, la recursividad representa una especie de "metabucle" y la complejidad cognitiva aumenta los bucles. En segundo lugar, la complejidad cognitiva consiste en estimar la dificultad relativa de comprender el flujo de control de un método, e incluso algunos programadores experimentados encuentran difícil la recursividad. comprender.

Salta a etiquetas

goto agrega un incremento fundamental a la Complejidad Cognitiva, al igual que romper o continuar a una etiqueta y otros saltos de varios niveles, como romper o continuar a un número que se encuentra en algunos idiomas. Pero debido a un regreso anticipado

A menudo puede hacer que el código sea mucho más claro, ningún otro salto o salida anticipada causa un incremento.

Incremento para estructuras de interrupción de flujo anidadas

Parece intuitivamente obvio que una serie lineal de cinco estructuras si y para sería más fácil de entender que esas mismas cinco estructuras anidadas sucesivamente, independientemente del número de rutas de ejecución a través de cada serie.
Debido a que dicho anidamiento aumenta las demandas mentales para comprender el código, la Complejidad Cognitiva evalúa un incremento de anidamiento.

Específicamente, cada vez que una estructura que provoca un incremento estructural o híbrido se anida dentro de otra estructura similar, se agrega un incremento de anidamiento para cada nivel de anidamiento. Por ejemplo, en el siguiente ejemplo, no hay ningún incremento de anidamiento para el método en sí ni para el intento porque ninguna de las estructuras da como resultado un incremento estructural o híbrido:

```
anular mi método () {
    intentar {
        if (condición1) { for (int i =                    // +1
            0; i < 10; i++) { while (condición2) { … }    // +2 (anidamiento=1)
                                                          // +3 (anidamiento=2)
            }
        }
    } captura (ExcepType1 | ExcepType2 e) { // +1
        si (condición2) {...}                             // +2 (anidamiento=1)
    }
}                                                         // Complejidad cognitiva 9
```

Sin embargo, las estructuras if, for, while y catch están sujetas a incrementos tanto estructurales como de anidamiento.

Además, si bien los métodos de nivel superior se ignoran y no hay ningún incremento estructural para lambdas, métodos anidados y características similares, dichos métodos sí incrementan el nivel de anidamiento cuando se anidan dentro de otras estructuras similares a métodos:

```
anular miMétodo2 () {
    Ejecutable r = () -> {                    // +0 (pero el nivel de anidamiento ahora es 1)
        si (condición1) {...}};               // +2 (anidamiento=1)

}                                             // Complejidad cognitiva 2


#si DEBUG                                     // +1 para si
anula myMethod2 () {                          // +0 (el nivel de anidamiento sigue siendo 0)
    Ejecutable r = () -> {                    // +0 (pero el nivel de anidamiento ahora es 1)
        si (condición1) {...}};               // +3 (anidamiento=2)

                                              // Complejidad cognitiva 4
} #terminara si
```

Las implicaciones

Cognitive Complexity se formuló con el objetivo principal de calcular puntuaciones de métodos que reflejen con mayor precisión la relativa comprensibilidad de los métodos, y con objetivos secundarios de abordar construcciones del lenguaje moderno y producir métricas que sean valiosas por encima del nivel del método. Es evidente que se ha logrado el objetivo de abordar las construcciones del lenguaje moderno. Los otros dos objetivos se examinan a continuación.

Puntuaciones de complejidad intuitivamente "correctas"

Esta discusión comenzó con un par de métodos con igual complejidad ciclomática pero decididamente desigual comprensibilidad. Ahora es el momento de reexaminar esos métodos y calcular sus puntuaciones de Complejidad Cognitiva:

```
int sumaDePrimes(int máx) { int total = 0;

    SALIDA: for (int i = 1; i <= max; ++i) { // +1
        for (int j = 2; j < i; ++j) { if (i % j == 0) { continuar // +2
            FUERA; // +3

            } // +1
        }
        total += yo;
    }
    total de retorno;
}                                   // Complejidad cognitiva 7
```

```
Cadena getWords(int número) {
        interruptor (número) {caso 1:                   // +1

            devolver "uno";
        caso 2:
            devolver “una pareja”;
        caso 3:
            devolver “algunos”;
        por defecto:
            devolver “lotes”;
    }
}           // Complejidad cognitiva 1
```

El algoritmo de Complejidad Cognitiva otorga a estos dos métodos puntuaciones marcadamente diferentes, que reflejan mucho más su relativa comprensibilidad.

Métricas que son valiosas por encima del nivel del método

Además, debido a que la Complejidad Cognitiva no aumenta para la estructura del método, los números agregados resultan útiles. Ahora puede distinguir entre una clase de dominio (una con una gran cantidad de captadores y definidores simples) y una que contiene un flujo de control complejo simplemente comparando sus valores métricos. La Complejidad Cognitiva se convierte así en una herramienta para medir la comprensibilidad relativa de clases y aplicaciones.

Conclusión

Los procesos de escribir y mantener código son procesos humanos. Sus resultados deben adherirse a modelos matemáticos, pero no encajan en los modelos matemáticos mismos. Por eso los modelos matemáticos son inadecuados para evaluar el esfuerzo que requieren.

Cognitive Complexity rompe con la práctica de utilizar modelos matemáticos para evaluar la mantenibilidad del software. Parte de los precedentes establecidos por Cyclomatic Complexity, pero utiliza el juicio humano para evaluar cómo se deben contar las estructuras y decidir qué se debe agregar al modelo en su conjunto. Como un
Como resultado, produce puntuaciones de complejidad del método que a los programadores les parecen evaluaciones relativas de comprensibilidad más justas que las que estaban disponibles con modelos anteriores. Además, debido a que la Complejidad Cognitiva no cobra ningún “costo de entrada” por un método, produce evaluaciones relativas más justas no sólo a nivel de método, sino también a nivel de clase y aplicación.

Referencias

[1] Thomas J. McCabe, "Una medida de complejidad", IEEE Transactions on Software Engineering, vol. SE-2, núm. 4, diciembre de 1976

Apéndice A: Compensación Usos

La Complejidad Cognitiva está diseñada para ser una medida independiente del idioma, pero no se puede ignorar que diferentes idiomas ofrecen características diferentes. Por ejemplo, no existe otra estructura en COBOL, y hasta hace poco JavaScript carecía de una estructura similar a una clase. Desafortunadamente, esos déficits no impiden que los desarrolladores necesiten esas estructuras o intenten construir algo análogo con las herramientas disponibles. En tales casos, una aplicación estricta de las reglas de Complejidad Cognitiva daría como resultado puntuaciones desproporcionadamente altas.

Por esa razón, y para no penalizar el uso de una lengua sobre otra, se pueden hacer excepciones para los déficits lingüísticos, es decir, estructuras que se utilizan comúnmente y se esperan en la mayoría de las lenguas modernas, pero que faltan en la lengua en cuestión, como Falta COBOL else if.

Por otro lado, cuando un lenguaje innova para introducir una característica, como la capacidad de Java 7 para detectar múltiples tipos de excepciones a la vez, la falta de esa innovación en otros lenguajes no debe considerarse un déficit y, por lo tanto, no debe haber ninguna excepción.

Esto implica que si capturar múltiples tipos de excepciones a la vez se convierte en una característica del lenguaje comúnmente esperada, se podría agregar una excepción para cláusulas de captura "adicionales" en lenguajes que no ofrecen esa capacidad. Esta posibilidad no está excluida, pero las evaluaciones sobre si agregar o no tales excepciones futuras deberían pecar del lado del conservadurismo. Es decir, las nuevas excepciones deberían llegar lentamente.

Por otro lado, si una versión futura del estándar COBOL agrega un “si no” estructura, la tendencia debería ser eliminar la excepción COBOL "de lo contrario... si" (descrita a continuación) tan pronto como sea práctico.

Hasta la fecha se han identificado tres excepciones:

COBOL: Falta más si

Para COBOL, que carece de una estructura else if , un if como única declaración en una cláusula else no incurre en una penalización por anidamiento. Además, no hay ningún incremento para el resto en sí. Es decir, un else seguido inmediatamente de un if se trata como un else if, aunque sintácticamente no lo es.

Por ejemplo:

```
Si condición1                    // +1 estructura, +0 para anidamiento
    ...
DEMÁS
    Si condición2                // +1 estructura, +0 para anidamiento
        ...
    DEMÁS
        Si condición3            // +1 estructura, +0 para anidamiento
            declaración1
            Condición IF4 // +1 estructura, +1 para anidamiento
                ...
            TERMINARA SI
        TERMINARA SI
    TERMINARA SI
TERMINARA SI.
```

JavaScript: estructuras de clases faltantes

A pesar de la reciente incorporación de clases a JavaScript mediante la especificación ECMAScript 6, la característica aún no se ha adoptado ampliamente. De hecho, muchos marcos populares requieren el uso continuo del lenguaje compensador: el uso de una función externa como sustituto para crear una especie de espacio de nombres o clase falsa. Para no penalizar a los usuarios de JavaScript, estas funciones externas se ignoran cuando se utilizan puramente como mecanismo declarativo, es decir, cuando contienen sólo declaraciones en el nivel superior.

Sin embargo, la presencia en el nivel superior de una función (es decir, no anidada dentro de una subfunción) de declaraciones sujetas a incrementos estructurales indica algo más que un uso puramente declarativo. En consecuencia, dichas funciones deberían recibir un tratamiento estándar.

Por ejemplo:

```
función(...) { var foo;                          // declarativo; ignorado


    bar.myFun = function(…) { // anidamiento = 0
        si(condición) { // +1
            ...
        }
    }
}                                                // complejidad total = 1


función(...) { var foo;                          // no declarativo; no ignorado

    si (condición) {                             // +1; incremento estructural de alto nivel
        ...
    }

    bar.myFun = función(…) { // anidamiento = 1
        si(condición) { // +2
            ...
        }
    }
}                                                // complejidad total = 3
```

Python: decoradores

El lenguaje decorador de Python permite agregar comportamiento adicional a una función existente sin modificar la función en sí. Esta adición se logra con el uso de funciones anidadas en el decorador que proporciona el comportamiento adicional.
Para no penalizar a los programadores de Python por el uso de una característica común de su lenguaje, se agregó una excepción. Sin embargo, se ha intentado definir la excepción de forma estricta. Específicamente, para ser elegible para la excepción, una función puede contener solo una función anidada y una declaración de devolución.

Por ejemplo:

```
def a_decorador(a, b):
    def interior(func): si # anidamiento = 0
        condición: # +1
            imprimir(b)
        función()
    volver interior # total = 1

def not_a_decorador(a, b):
    mi_var = a*b
    def interior(func): si # anidamiento = 1
        condición: # +1 estructura, +1 anidamiento
            imprimir(b)
        función()
    volver interior # total = 2

def decorador_generador(a):
    generador def (función):
        def decorador(func): # anidamiento = 0
            si condición: # +1
                imprimir(b)
            función de retorno()
        decorador de regreso
    generador de retorno # total = 1
```

Apéndice B: Especificaciones

El propósito de esta sección es brindar una enumeración concisa de las estructuras y circunstancias que incrementan la Complejidad Cognitiva, sujeta a las excepciones enumeradas en el Apéndice A. Se pretende que sea una lista completa sin ser exhaustiva en el lenguaje. Es decir, si un idioma tiene una ortografía atípica para una palabra clave, como elif para else if, su omisión aquí no pretende omitirla de la especificación.

1. Incrementos

Hay un incremento para cada uno de los siguientes:

- si, si no, si, si no, operador ternario
- cambiar
- para, para cada uno
- mientras, hacer mientras
- atrapar
- ir a ETIQUETA, romper ETIQUETA, continuar ETIQUETA, romper NÚMERO, continuar NÚMERO
- secuencias de operadores lógicos binarios
- cada método en un ciclo de recursividad

B2. Nivel de anidación

Las siguientes estructuras incrementan el nivel de anidamiento:

- si, si no, si, si no, operador ternario
- cambiar
- para, para cada uno
- mientras, hacer mientras
- atrapar
- métodos anidados y estructuras similares a métodos como lambdas

B3. Incrementos de anidamiento

Las siguientes estructuras reciben un incremento de anidamiento proporcional a su profundidad anidada dentro de las estructuras B2:

- si, operador ternario
- cambiar
- para, para cada uno
- mientras, hacer mientras
- atrapar

Apéndice C: Ejemplos

Desde org.sonar.java.resolve.JavaSymbol.java en el analizador SonarJava:

```
@Anulable
Método privadoJavaSymbol overriddenSymbolFrom(ClassJavaType classType) {
    if (classType.isUnknown()) { return                                    // +1
        Símbolos.unknownMethodSymbol;
    }

    booleano desconocidoEncontrado = falso;
    Lista<JavaSymbol> símbolos = classType.getSymbol().members().lookup(name);
    for (JavaSymbol overrideSymbol: símbolos) { if                         // +1
        (overrideSymbol.isKind(JavaSymbol.MTH) &&!overrideSymbol.isStatic())  // +2 (anidamiento = 1)
                        {                                                  // +1

            MethodJavaSymbol métodoJavaSymbol = (MethodJavaSymbol)overrideSymbol;
            if (canOverride(methodJavaSymbol)) { // +3 (anidamiento = 2)
                Anulación booleana = checkOverridingParameters(métodoJavaSymbol,
                                        tipo de clase);
                if (anulando == nulo) { if (!unknownFound) {               // +4 (anidamiento = 3)
                                                                           // +5 (anidamiento = 4)
                        desconocidoEncontrado = verdadero;
                    }
                } más si (anular) {                                        // +1
                    método de retornoJavaSymbol;
                }
            }
        }
    }

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

    devolver nulo;
}                                                                          // complejidad total = 19
```

De com.persistit.TimelyResource.java en sonar-persistit:

```
addVersion privada vacía (entrada de entrada final, txn de transacción final)
        lanza PersistitInterruptedException, RollbackException {
    índice de transacción final ti = _persistit.getTransactionIndex();
    mientras (verdadero) { // +1
        intentar {
            sincronizado (esto) {
                if (primero! = nulo) { if                                   // +2 (anidamiento = 1)
                    (frst.getVersion() > entrada.getVersion()) {            // +3 (anidamiento = 2)
                        lanzar nueva RollbackException();
                    }
                    si (txn.isActive()) {para                               // +3 (anidamiento = 2)
                                                                            // +4 (anidamiento = 3)
                            (Entrada e = primera; e != nula; e = e.getPrevious()) {
                        versión larga final = e.getVersion();
                        final largo depende = ti.wwDependency(versión,
                                txn.getTransactionStatus(), 0);
                        if (depende == TIMED_OUT) { lanzar nueva            // +5 (anidamiento = 4)
                            WWRetryException (versión);
                        }
                        if (¡depende!= 0 &&                                 // +5 (anidamiento = 4)
                                depende!= ABORTADO) { throw new             // +1
                            RollbackException();
                        }
                    }
                }
            }
            entrada.setPrevious(primero);
            primero = entrada;
            romper;
        }
    } captura (final WWRetryException re) {                                 // +2 (anidamiento = 1)
        intentar {
            final largo depende = _persistit.getTransactionIndex()
                    .wwDependency(re.getVersionHandle(),txn.getTransactionStatus(),
                    SharedResource.DEFAULT_MAX_WAIT_TIME);
            if (¡depende!= 0 &&                                             // +3 (anidamiento = 2)
                    depende!= ABORTADO) { throw new                         // +1
                RollbackException();
            }
        } captura (excepción interrumpida final, es decir) {                // +3 (anidamiento = 2)
            lanzar una nueva PersistitInterruptedException (es decir);
        }
    } captura (excepción interrumpida final, es decir) {                    // +2 (anidamiento = 1)
        lanzar una nueva PersistitInterruptedException (es decir);
    }
}                                                                           // complejidad total = 35
```

Desde org.sonar.api.utils.WildcardPattern.java en SonarQube:

```
Cadena estática privada toRegexp(String antPattern,
                Separador de directorio de cadena) {

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

    mientras (yo <antPattern.length()) {                                     // +1

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

                if (i + 2 < antPattern.length() &&                           // +4 (anidamiento = 3) // +1
                        isSlash(antPattern.charAt(i + 2))) { sb.append("(?:.*")

                            .append(escapedDirectorySeparator).append("|)");
                    yo += 2;
                } más                                                        // +1
                    { sb.append(".*");
                    yo += 1;
                }
            } else                                                           // +1
                { sb.append("[^").append(escapedDirectorySeparator).append("]*?");
            }
        } else if (ch == '?')                                               // +1
            { sb.append("[^").append(escapedDirectorySeparator).append("]");
        } más si (isSlash(ch)) {                                            // +1
            sb.append(escapedDirectorySeparator);
        } más                                                               // +1
            { sb.append(ch);
        }
        yo ++;
    }

    sb.append('$');
    devolver sb.toString();
}                                                                           // complejidad total = 20
```

Desde model.js en YUI

```
guardar: función (opciones, devolución de llamada) {
    var self = esto;

    if (tipo de opciones === 'función') {                                    // +1
        devolución de llamada = opciones;
        opciones = {};
    }

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

    self._validate(self.toJSON(), función (err) {
        if (err)                                                             // +2 (anidamiento = 1)
            { devolución de llamada && devolución de llamada.call(nulo,      // +1
            err); devolver;
        }

        self.sync(self.isNew()? 'crear': 'actualizar', opciones, función (err, respuesta) {   // +2 (anidamiento = 1)

             var fachada = {
                    opciones: opciones,
                    respuesta: respuesta
                },
            analizado;

            if (err)                                                         // +3 (anidamiento = 2)
                { fachada.error = err;
                fachada.src = 'guardar';
                self.fire(EVT_ERROR, fachada);

            } else { if (!                                                   // +1
                self._saveEvent) { self._saveEvent =                         // +4 (anidamiento = 3)
                    self.publish(EVT_SAVE, {
                        prevenible: falso
                    });
                }

                si (respuesta) {                                             // +4 (anidamiento = 3)
                    analizado = fachada.parsed = self._parse(respuesta);
                    self.setAttrs(analizado, opciones);
                }

                self.cambiado = {};
                self.fire(EVT_SAVE, fachada);
            }

            devolución de llamada && callback.apply(nulo, argumentos); });   // +1

    });
    regresar a uno mismo;
}                                                                            // complejidad total = 20
```

Registro de cambios

Versión 1.1

6 de febrero de 2017

- Actualice la sección sobre recursividad para incluir la recursividad indirecta.
- Agregue el tipo de incremento “Híbrido” y utilícelo para aclarar el manejo de else y else if; no están sujetos a un incremento de anidamiento, pero sí aumentan el nivel de anidamiento.
- Aclarar que la Complejidad Cognitiva sólo se ocupa de los operadores booleanos binarios.
- Corrija getWords para que realmente tenga una Complejidad Ciclomática de 4.
- Agregar Apéndice A: Usos compensatorios.
- Actualizar los derechos de autor.
- Iniciar registro de cambios.

Versión 1.2

19 de abril de 2017

- Ajustes y correcciones textuales, como el uso de la palabra “comprensibilidad” en lugar de “mantenibilidad”.
- Agregar explicación de por qué un híbrido El incremento se evalúa en otra cosa si y más en lugar de un estructural incremento.
- Agregar Apéndice B: Especificaciones.
- Agregar Apéndice C: Ejemplos.

Versión 1.3

15 de marzo de 2018

- Ampliar el Apéndice A para incluir un uso de compensación para los decoradores de Python.
- Actualizar los derechos de autor.

Versión 1.4

10 de septiembre de 2018

- Aclarar qué tipo de método el anidamiento incrementa el anidamiento nivel.

Versión 1.5

5 de abril de 2021

- Hacer explícito que todos los usos multinivel de romper y continuar recibir un incremento fundamental.
- Nueva portada y pie de página, además de correcciones de errores tipográficos y otras actualizaciones menores visuales y de legibilidad.
- Actualizar los derechos de autor.

Versión 1.6

9 julio 2021

- Eliminar el título del autor

Versión 1.7

29 agosto 2023

- Actualizaciones de la marca
- Ajustes de diseño

www.sonarsource.com

La solución líder en la industria de Sonar permite a los desarrolladores y organizaciones alcanzar el estado de Código Limpio. Sus soluciones comerciales y de código abierto (SonarLint, SonarCloud y SonarQube) admiten más de 30 lenguajes de programación, marcos y tecnología de infraestructura. Con la confianza de más de 400.000 organizaciones en todo el mundo, Sonar se considera fundamental para ofrecer un mejor software.