COGNITIVE COMPLEXITY

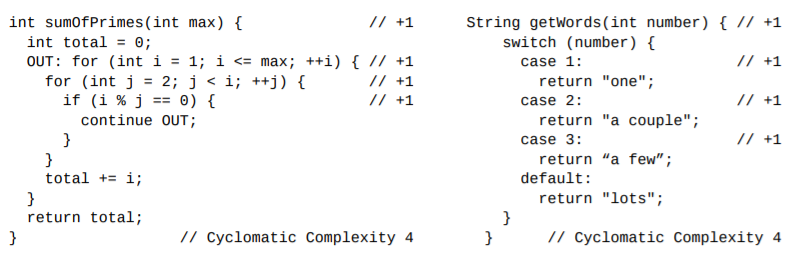
1. Giới thiệu

Thomas J.McCabe’s Cyclomatic Complexity từ lâu đã trở thành tiêu chuẩn thực tế cho việc đánh giá độ phức tập của luồng điều khiển của phương thức. Nó là ẩn dụ độc đáo “để nhận dạng mô đun phần mềm sẽ khó để kiểm thử hoặc duy trì”, nhưng trong khi nó được tính chính xác đến con số nhỏ nhất của các trường hợp kiểm thử yêu cầu một phương thức hoàn chỉnh, nó không phải thước đo thỏa đáng về sự dễ hiểu. Đây là vì phương thức với Cyclomatic Complexity bằng nhau không cần thiết thể hiện bằng độ khó của người duy trì code, dẫn đến một cảm giác rằng đo lường tiếng của con sói, chỉ lấy những cấu trúc cao, trong khi đánh giá thấp những cái khác.

Cùng một thời điểm, Cyclomatic Complexity không còn toàn diện. Được đề ra trong Forten năm 1976, nó không bao gồm cấu trúc ngôn ngữ hiện đại như try/catch và lamda.

Và cuối cùng, bởi vì mỗi phương thức có một điểm Cyclomatic Complexity nhỏ nhất. nó không thể biết điểm tổng Cyclomatic Complexity nào là lớn.

1. Đặt vấn đề: Hai phương thức dưới đây đều có cùng Cyclomatic Complexity, nhưng có sự khác nhau rất dễ nhận thấy đó là sự dễ hiểu.



Mô hình toán học cơ bản của Cyclomatic Complexity cho chúng ta hai phương thức có độ phức tạp như nhau, nhưng rõ ràng là trực giác cho thấy luồng điều khiển của phương thức sumOfPrimes khó để hiểu hơn getWords. Đó là lý do Cognitive Complexity từ bỏ việc sử dụng mô hình toán học cho đánh giá luồng điều khiển có lợi cho một tập hợp các quy tắc đơn giản để biến trực giác của lập trình viên thành số.

1. Tiêu chí và phương pháp cơ bản

Một điểm Cognitive Complexity được đánh giá dựa vào ba quy luật sau:

1. Bỏ qua cấu trúc mà cho phép nhiều câu lệnh viết nhanh thành một.
2. Tăng (cộng 1) cho mỗi lần ngắt trong luồng tuyến tính của code.
3. Tăng khi cấu trúc ngắn luồng được lồng nhau.

Ngoài ra, một điểm phức tạp được tạo ra bởi 4 loại tăng khác nhau:

1. Lồng (Nesting) – được đánh giá cho cấu trúc luồng điều khiển lồng ở bên trong nhau
2. Cấu trúc(Structural) – được đánh giá trên cấu trúc luồng điều khiển mà chịu sự gia tăng của lồng và điều đó làm tăng số lượng lồng.
3. Nền tảng(Fundamental) – được đánh giá trên câu lệnh không chịu sự gia tăng lồng
4. Hỗn hợp(Hybrid) – được đánh giá trên cấu trúc luồng điều khiển mà không chịu sự gia tăng của lồng, nhưng làm gia tăng số lượng lồng.

Trong khi loại của một sự gia tăng không có sự khác biệt trong toán học – mỗi sự gia tăng thêm 1 vào số điểm cuối cùng – làm một sự phân biệt lớn giữa các mục của tính năng được đếm làm nó dễ hơn để hiểu nơi mà lồng tăng và không được thực hiện.

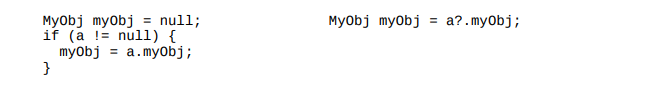
Những quy luật và nguyên lý sau đó sẽ được đề cập chi tiết trong các phần sau.

1. Bỏ qua tốc ký (cách viết nhanh gọn)

Một nguyên lý hướng dẫn trong cách diễn đạt của Cognitive Complexity nó khuyến khích thực hành cách viết code tốt. Đó là nên bỏ qua hoặc giảm các tính năng viết code dễ đọc hơn.

Cấu trúc phương thức chính là ví dụ điển hình. Ngắt code thành phương thức cho phép bạn gộp lại nhiều dòng lệnh thành một dòng duy nhất, có tên gọi rõ ràng, đó là shorthand.

Cognitive Complexity cũng bỏ qua những toán tử kết hợp với null được tìm thấy trong nhiều ngôn ngữ, một lần nữa bởi vì họ cho phép viết gọn nhiều dòng lệnh thành một. Ví dụ, hai đoạn code dưới đây đều làm cùng một việc:



Ý nghĩa của đoạn code bên trái cần có một khoảnh khắc để xử lý, trong khi đó đoạn code bên phải là rõ ràng ngay lập tức một khi bạn hiểu cú pháp của toán tử điều kiện null.

1. Tăng vì ngắt luồng tuyến tính

Trong cách diễn đạt của một nguyên lý hướng dẫn khác của Cognitive Complexity là cấu trúc đó ngắt luồng đi bình thường của code từ trên xuống dưới, từ trái qua phải, yêu cầu người bảo trì phải làm việc chăm chỉ hơn để hiểu được code. Nhìn thấy sự tốn khá nhiều công sức này, Cognitive Complexity đánh giá sự giá tăng cấu trúc cho:

* Cấu trúc lặp (Loop structures): for, while, do while,...
* Câu điều kiện (Conditionals): toán tử điều kiện (ternary operator), if, #if, #ifdef,...

Nó đánh giá gia tăng hỗn hợp cho:

* Else if, elif, else,...

Không gia tăng lồng nào được tính cho những cấu trúc này bởi vì những chi phí tin thần đã được tính trong khi đọc những dòng if rồi.

Mục tiêu để tăng độ phức tạp sẽ quen thuộc với những ai đã từng biết về Cyclomatic Complexity. Ngoài ra, Cognitive Complexity cũng tăng vì những cái sau đây:

1. Catches

Một catch đại điện một nhánh trong luồng điều khiển giống như if. Vì thế, mỗi kết quả của lệnh catch trong một cấu trúc gia tăng đến Cognitive Complexity. Ghi nhớ catch chỉ cộng 1 điểm Cognitive Complexity, không cần biết là có báo nhiêu loại exception được bắt. Các khối try và finally đều bị bỏ qua.

1. Switches

Một switch và tất cả các trường hợp kết hợp phát sinh một gia tăng cấu trúc duy nhất.

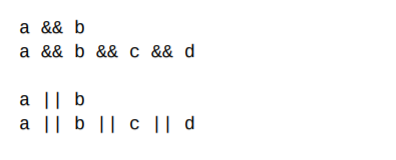
Trong Cyclomatic Complexity, một switch được xem như một dấu hiệu đến chuỗi if else if. Đó là mỗi trường hợp trong switch sẽ làm tăng thêm 1 bởi vị nó làm một nhánh trong mô hình toán học của luồng điều khiển.

Nhưng từ cái nhìn của người bảo trì, một switch – sẽ so sánh một biến duy nhất với từng giá trị cụ thể rõ ràng trong một tập hợp - và như vậy thì dễ để hiểu hơn một chuỗi if-else if bởi vì sau này có thể có thêm bất kỳ số lượng so sánh, sử dụng bất kỳ số lượng biến và giá trị.

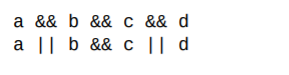
Nói ngắn gọn, một chuỗi if-else if phải được đọc cẩn thận, trong khi đó switch có thể chỉ cần liếc cái là hiểu.

1. Trình tự các toán tử logic

Cũng giống như những lý do trên, Cognitive Complexity không tăng cho mỗi toán tử logic nhị phân. Ví dụ, xem xét những cặp toán tử dưới đây:



Hiểu được dòng thứ hai của mỗi cặp không khó hơn hiểu những dòng đầu của mỗi cặp. Tuy nhiên, có một sự khác biệt được đánh dấu trong sự nổ lực để hiểu hai dòng dưới đây:



Bởi vì cách diễn đạt boolean trở nên hơi khó để hiểu hơn với hỗn hợp các toán tử, Cognitive complexity tăng cho mỗi tuần tự mới của các toán tử giống nhau. Ví dụ:

Trong khi Cognitive Complexity đề nghị “giảm” cho toán tử giống nhau liên quan đến Cyclomatic Complexity, nó được tăng cho mỗi trình tự của toán tử nhị phân như các toán tử trong các phép gán biến, các lệnh gọi phương thức và các câu lệnh trả về.

1. Đệ quy

Không giống như Cyclomatic Complexity, Cognitive Complexity thêm một sự tăng cơ bản cho mỗi phương thức trong vòng đệ quy, một cách trực tiếp hoặc gián tiếp. Có hai động lực cho quyết định này. Đầu tiên, đệ quy đại diện cho một loại “meta-loop” có nghĩ là vòng lặp lớn, và Cognitive Complexity tính tăng cho những vòng lặp. Thứ hai, Cognitive Complexity là về ước tính độ khó để hiểu của luồng điều khiển của phương thức, và cũng một số lập trình viên dày dạn kinh nghiệm thấy đệ quy khó để hiểu.

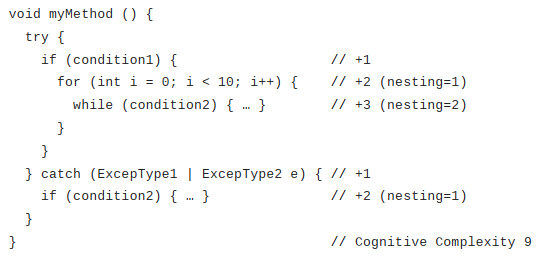
1. Nhảy đến nhãn (Jumps to labels)

Goto, và break hoặ clà continue là những nhãn sẽ bị cộng thêm điểm Cognitive Complexity. Nhưng bởi vì một return trả về sớm có thể thường làm code sạch sẽ hơn, không có bước nhảy nào khác hoặc sớm tồn tại gây ra sự gia tăng.

1. Tăng cho cấu trúc ngắt luồng lồng

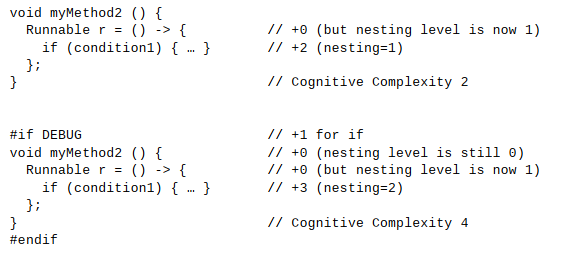
Có vẻ như rõ ràng trực quan rằng chuỗi tuyến tính của năm cấu trúc if và for sẽ dễ hiểu hơn năm cấu trúc lồng nhau, bất kể số lần thực thi mỗi chuỗi. Bởi vì những chuỗi lồng lệnh này yêu cầu tinh thần để hiểu code, Cognitive Complexity tính độ tăng luôn cho mỗi cái đó.

Đặc biệt, mỗi lần cấu trúc gây một sự tăng có cấu trúc hoặc tăng hỗn hợp là được lồng như trong mỗi cấu trúc khác, một gia tăng lồng nhau được thêm chỗ mỗi mức lồng.Ví dụ như trong đoạn code dưới đây, trong có gia tăng lồng cho phương thức trong nó hoặc cho try bởi vì không có cấu trúc dẫn đến gia tăng cấu trúc hay kết cấu.



Tuy nhiên, cấu trúc if, for, while và catch là những đối tượng cho gia tăng có cấu trúc và lồng.

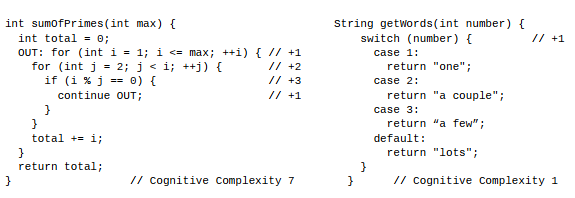
Ngoài ra, trong khi phương thức mức cao nhất bị bỏ qua, và không có gia tăng cấu trúc nào cho lambda, phương thức lồng, và những tính năng tương tự, như những phương thức tăng với mức lồng khi lồng trong một phương thức như cấu trúc dưới đây:



1. Ý nghĩa

Cognitive Complexity được trình bày với mục đích chính là tính toán điểm của phương thức mà điểm càng chính xác phương thức càng dễ hiểu, mục đích thứ hai là giải quyết cấu trúc ngôn ngữ hiện đại và tạo ra các số liệu có giá trị trong phương thức. Nói một cách rõ ràng, mục đích là giải quyết cấu trúc ngôn ngữ hiện đại đã đạt được. Còn hai mục đích được kiểm tra ở dưới đây:

1. Một cách trực quan về sự đúng đắn của điểm phức tạp

Cuộc thảo luận này bắt đầu với một cặp phương thức có Cyclomatic Complexity bằng nhau nhưng độ dễ hiểu thì rõ ràng là không như nhau. Bây giờ là lúc kiểm tra lại những phương thức này và tính toán điểm Cognitive Complexity:  


Thuật toán Cognitive Complexity cho hai phương thức rõ ràng điểm khác nhau, một số phản ánh lên sự dễ hiểu của mình.

1. Số liệu có giá trị trên phương thức

Hơn thế nữa, bởi vì Cognitive Complexity không tăng cho cấu trúc phương thức, tổng các con số thì trở nên có ích. Bây giờ bạn có thể nói về sự khác biệt giữa lớp miền (domain class) – một cái với số lượng lớn getter và setter – và một cái có những luồng điều khiển phức tạp bởi sự so sánh đơn giản của tập giá trị. Cognitive Complexity do đó trở thành một công cụ cho việc đo độ tương đối dễ hiểu của lớp và ứng dụng.

1. Kết luận

Qúa trình viết và vảo trì code là tiến trình của con người. Đầu ra của họ phải bám chặt đến mô hình toán học, nhưng họ không đáp ứng mô hình toán học của chính họ. Đó là lý do mô hình toàn học không đủ khả năng để đánh giá những nỗ lực và yêu cầu.

Cognitive Complexity ngắt từ sự luyện tập sử dụng các mô hình toán học để xác định khả năng duy trì của phần mềm. Nó bắt đầu từ tiền lệ được thiết lập bởi Cyclomatic Complexity, nhưng sử dụng sự đánh giá của con người đế đánh giá cấu trúc nên được đếm, và để quyết dịnh cái gì nên được thêm vào mô hình cũng như tất cả. Kết quả là, nó mang lại điểm phức tạp của phương thức mà làm cho các lập trình viên đánh giá tương đối công bằng hơn so với các mô hình trước đó. Ngoài ra, bởi vì Cognitive Complexity không tốn “phí đầu vào” cho một phương thức, nó thực hiện đánh giá công bằng không những đối với phương thức mà còn đối với lớp và cái mức của ứng dụng.

# Phụ lục A: Lợi ích bù trừ

Cognitive Complexity được thiết kế trở thành một phép đo bất kỳ ngôn ngữ, nhưng nó không thể bỏ qua những tính năng khác biết của những ngôn ngữ khác nhau. Ví dụ, sẽ không có cấu trúc else if trong COBOL, và cho đến gần đây Javascript rò rĩ một cấu trúc giống như class. Rất tiếc, những thâm hụt đó không ngăn chặn những developers từ những cấu trúc cần thiết và từ việc cố gắng xây dựng một cái gì đó tương tự với các công cụ trong tay. Giống như một số trường hợp, một ứng dụng chuẩn xác với các quy luật của Cognitive Complexity sẽ dẫn đến kết quả một số điểm số cao không cân xứng.

Vì lý do đó, và để không gây khó khăn khi sử dụng một ngôn ngữ khác, các trường hợp ngoại lệ có thể được viết cho ngôn ngữ thiếu hụt. Đó là cấu trúc được sử dụng phổ biến, và được mong đợi trong nhiều ngôn ngữ hiện đại, nhưng thiếu từ ngôn ngữ đang xem xét, như là COBOL’s thiếu else if.

Mặt khác, khi một ngôn ngữ đổi mới giới thiệu một tính năng như Java 7 có khả năng bắt được nhiều loại ngoại lệ cùng một lúc, không có sự đổi mới trong những ngôn ngữ nên được xem xét một thiếu hụt, và do đó không nên có ngoại lệ.

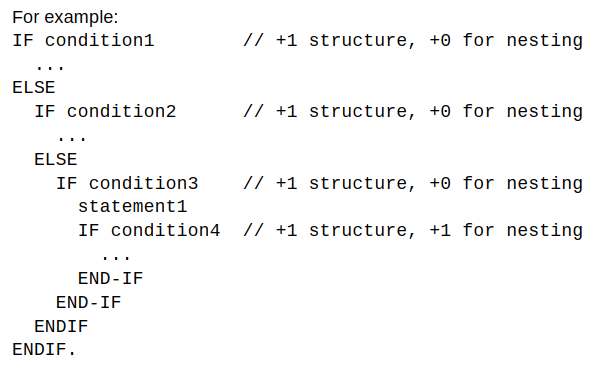
Ẩn ý này nếu bắt được nhiều loại ngoại lệ cùng một lúc trở thành một tính năng của ngôn ngữ phổ biến được mong chờ, một ngoại lệ có thể được thêm cho “nhiều” mệnh đề catch trong nhiều ngôn ngữ mà không có khả năng cung cấp. Cơ hội này không được loại trừ, nhưng các đánh giá về việc có thêm các ngoại lệ hay không nên đứng về phía Bảo thủ. Đó là những ngoại lệ mới nên một cách chậm chạp.

Mặt khác, nếu một phiên bản tương lai của tiêu chuẩn COBOL thêm một cấu trúc “else if”, xu hướng nên bị giảm ngoại lệ COBOL “else ... if” (được miêu tả dưới đây) ngay khi thực tế.

Đến hiện tại, ba ngoại lệ được phát hiện:

## COBOL: Thiếu else if

Với COBOL, thiếu cấu trúc *else if* , một *if* là một câu duy nhất trong mệnh đề *else* không phải chịu một hình phạt lồng. Ngoài ra, không có tăng cho bất kỳ *else* chính nó. Đó là, một *else* theo sau ngay lập tức là một *if*  được xem nhưng là một *else if,* mặc dù cú pháp thì không đúng.



## Javascript: Thiếu cấu trúc class

Mặc dù những class được thêm gần đây trong yêu cầu Javascript bởi ECMA Script 6, tính năng chưa được chấp nhận rộng rãi. Thực tế, nhiều framework phổ biến yêu cầu tiếp tục sử dụng thành ngữ bù trừ: sử dụng một outer function độc lập để tạo một loại namespace hoặc lớp sai. Để không phạt người dùng Javascript,

## Python: Trang trí