به نام خدا

مستند پروژه پایانی درس کامپایلر

دکتر پارسا

زمستان 1402

اعضای گروه :

علیرضا اسلامی خواه 99521064

مهدی قضاوی 99522014

علی سلطانی 99521343

احمدرضا طهماسبی 99521424

سبا رضی 99521316

وحید محمدی 99522077

در مرحله اول با توجه به مقاله باید پیدا میشد که چه چیزهایی میتوانند فاکتورهای سنجش کیفیت یک کد نرم افزاری باشند. با کمی جست و جو در مقاله داده شده در فاز اول به این تیجه رسیدیم که در صفحه 5 مقاله میتوان به این فاکتورها دسترسی پیدا کرد.

Table 6. A comparison of criteria of the four quality model and ISO/IEC 2510

factors	McCall	Boehm	Dromey	FURPS	ISO/IEC25010
Correctness	*				
Integrity	*				
Usability	*			*	*
Efficiency	*	*	*		*
Flexibility	*	*			
Testability	*				*
Maintainability	*				*
Reliability	*	*	*	*	*
Portability	*	*	*		*
Reusability	*		*		
Interoperability	*		*		
Human engineering		*	*		
Understandability		*			*
Modifiability		*			*
Functionality				*	*
Performance			*	*	
Supportability				*	
Security					*
17	11	7	7	5	8

Extended Al-Outaish [17]

سپس به دنبال راه های ارزیابی این ویژگی ها برای سنجش دقیق نرم افزار رفتیم و متوجه شدیم که عمده پارامتر های استفاده شده در این سنجش ها از طریق متریک هایمختلف قابل اندازه گیری است پس اول اینها را پیاده سازی کردیم.

بخش های مختلف پروژه :

بخش پیاده سازی شده توسط **علیرضا اسلامی خواه** :

- **Halstead**

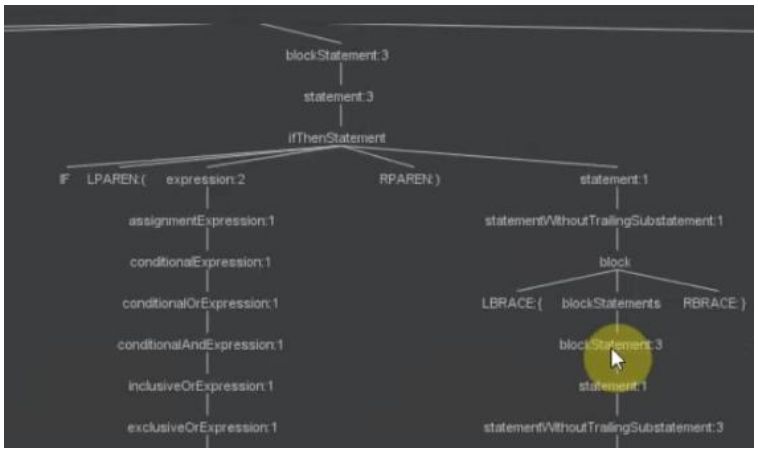
متریک هالستد یک مجموعه از معیارهاست که توسط موریس هالستد (Maurice Halstead) در دهه 1970 برای اندازه‌گیری پیچیدگی کد منبع برنامه نویسی ارائه شد. این معیارها به منظور ارزیابی جنبه‌های مختلف پیچیدگی کد و هزینه‌های پیاده‌سازی برنامه استفاده می‌شوند.

معیارهای هالستد بر مبنای تعداد اجزای مختلف کد منبع محاسبه می‌شوند. این اجزا به دو دسته تقسیم می‌شوند: اجزای حاصل‌ضرب (Operands) و عملگرها (Operators).

1. عملگرها (Operators): عبارت‌هایی هستند که انجام یک عملیات را نشان می‌دهند. مثال‌هایی از عملگرها عبارتند از `+`, `-`, `*`, `/` و غیره.

2. اجزای حاصل‌ضرب (Operands): مقادیری هستند که تحت عملیات‌های انجام شده تغییر می‌کنند. مثال‌هایی از اجزای حاصل‌ضرب می‌توانند متغیرها، ثابت‌ها و مقادیر متغیرها باشند.

معیار اندازه گیری اپراتورها و اپرندها بر اساس درخت اولیه parser میباشد که با توجه به آن جدول هارا پر میکنیم. مثلا در این شکل بچه اول را اپراتور در نظر گرفته و بچه سوم و چهارم را اپرند.



The measure of vocabulary: $n = n_1 + n_2$

Program length: $N = N_1 + N_2$

Program volume: $V = N \log_2 n$

Program level: $L = \frac{V^*}{V}$

Where

n_1 = the number of unique operators

n_2 = the number of unique operand

N_1 = the total number of operators

N_2 = the total number of operands

Christensen et al. [21] have taken the idea further and produced a metrics called difficulty. V^* is the minimal program volume assuming the minimal set of operands and operators for the implementation of given algorithm:

Program effort: E = $\frac{V^*}{L}$

Difficulty of implementation: D = $\frac{n_1 N_2}{2n_2}$

Programming time in seconds: T = $\frac{E}{S}$

Difficulty: $n_1/2 + N_2/n_2$

سپس آنها را با توجه به ویژگی که میخواهیم تعریف کنیم استفاده میکنیم.

به عنوان مثال قطعه کد زیر نشان دهنده اینست که چگونه برای ساختار شرط operand ها و operator ها را جدا کرده ایم.

```
169             self.add_to_operands(right)
170     def exitIfThenStatement(self, ctx:JavaParser.IfThenStatementContext):
171         if ctx.getChildCount() == 1:
172             pass
173         if (ctx.getChildCount() > 1):
174             iff = ctx.getChild(0).getText()
175             self.add_to_operators(iff)
176         #----------
177         self.graphh.create_node_if_else()
```

و در آخر هم خروجی دلخواه را گرفتیم:

```
Halstead's Software Metrics
------------------------------------------------------------------------
The operators table is : {'void': 1, '+': 1, '=': 4, 'int': 3, '*': 1, '>': 1, 'if': 1, '<': 1, '++': 1, '+=': 1, 'for': 1, 'public': 1, 'static': 1}
The operands table is : {'main': 1, '2': 1, '3': 1, 'a': 2, '2+3': 1, 'a=2+3': 1, '4': 1, '5': 1, 'b': 4, '4*5': 1, 'b=4*5': 1, '0': 2, 'i': 4, 'i=0': 1, '10': 1}
N1 = 13
N2 = 15
n1 = 11
n2 = 11
Vocabulary = 28
Program length = 28
Length = 28
Calculated program length = 76.10749561002055
The formula for Voulume is : Volume = (N1 + N2) * log2(n1 + n2)
Volume = 124.86408532184433
The formula for Difficulty is : (n1/2) * (N2/n2)
Difficulty = 7.499999999999999
The formula for Effort is : Effort = Difficulty * Volume
Effort = 936.4806399138323
The formula for Time is : Time = Effort / 18
Time = 52.02670221743513
The formula for Size is : Size = Effort / 3000
Size = 0.31216021330461075
```

در اینجا دو جدول operator و operands داریم هر اپراتور یا اپرند رو با توجه به فرکانس تکرار آن نشان میدهد و در آخر مقادیر unique یا یکتا را از آنها استخراج میکند و سپس در فرمول های مختلف میگذارد.

در مرحله آخر باید دو ویژگی flexibility و complexity از کد بررسی میشد.

که متریک های آنها بدین صورت است :

COMPLEXITY :

Cyclomatic Complexity Metric : CC = E – N + 2P

Halsted Volume Metric : HV = N/2* LOG2(N/L)* L/V

McCabe's Cyclomatic Complexity (MCC): MCC = E-N+ 2

Maintainability Index (MI) Metric = MI = 171 -5.2LOG2(HV)-0.23*CC -16.2LOG2(LOC)

Lines of Code Metric: ZIGMA LOCi

خروجی نهایی :

```
--------------------------------------------------------------------------
--------------------------------------------------------------------------


The Software Quality Factor is: *Complexity*
The Cyclomatic Complexity Metric is: 3
The Halsted Volume Metric is: 124.86408532184433
The McCabe's Cyclomatic Complexity (MCC): 3
The Maintainability Index (MI) Metric is: 130.18919263280625
The Lines of Code Metric is: 15
--------------------------------------------------------------------------
--------------------------------------------------------------------------
```

Flexibility :

Cyclomatic Complexity Metric : CC = E – N + 2P

Lines of Code Metric: ZIGMA LOCi

Maintainability Index (MI) Metric = MI = 171 -5.2LOG2(HV)-0.23*CC -16.2LOG2(LOC)

McCabe's Cyclomatic Complexity (MCC): MCC = E-N+ 2

خروجی نهایی :

```
--------------------------------------------------------------------------


The Software Quality Factor is: *Flexibility*
The Cyclomatic Complexity Metric is: 3
The Lines of Code Metric is: 15
The Maintainability Index (MI) Metric is: 130.18919263280625
The McCabe's Cyclomatic Complexity (MCC): 3
--------------------------------------------------------------------------
--------------------------------------------------------------------------
```

بخش پیاده سازی شده توسط **سبا رضی:**

- **Ejiogu**

متریک نرم‌افزاری Ejiogu یک روش طراحی شده برای ارزیابی پیچیدگی ساختاری یک برنامه نرم‌افزاری است. پیچیدگی ساختاری اهمیت زیادی دارد زیرا بر کیفیت و قابلیت نگهداری نرم‌افزار تأثیر می‌گذارد. ساختارهای پیچیده می‌توانند منجر به نرم‌افزاری شوند که دشوار در درک، تغییر و اشکال‌زدایی است، در حالی که ساختارهای ساده‌تر معمولاً قابلیت استفاده کاربران و محافظت‌پذیری بیشتری دارند.

در سطح بالا، رویکرد ایجیوگو از ساختار کد نرم‌افزار با استفاده از نمایش درختی تجزیه و تحلیل می‌کند که ساختارهای زبانی مختلف (مانند دستورات if-else، حلقه‌ها، فراخوانی‌های متد و غیره) به عنوان گره‌ها در این درخت نمایش داده می‌شوند. این معیارها بر روی چندین جنبه کلیدی تمرکز دارند:

(H): اینجا ارتفاع به عمق‌ترین سطح تو درونی در ساختار اشاره دارد. یک حلقه تو درونی به عنوان مثال ارتفاع درخت را افزایش می‌دهد. ارتفاع‌های بلندتر (با ارتفاع بیشتر) معمولاً نشانه منطق تودرتوی پیچیده‌تری هستند که ممکن است دشوار در پیگیری و نگهداری باشد.

(Rt): این بطور معمول یک اندازه‌گیری از عرض در سطح گره ریشه است. این نشان می‌دهد چند گره مستقیماً از ریشه شاخه می‌زنند. یک عدد دوقلو بالا ممکن است نشانه پیچیدگی باشد زیرا به این معناست که گره ریشه مستقیماً به بسیاری از گره‌های دیگر متصل است.

(M): موناد‌ها یا گره‌های برگ، نقاط پایانی درخت هستند - آن‌ها دیگر شاخه‌ای از خودشان ندارند. در اصطلاح نرم‌افزاری، آن‌ها اغلب نقاط عملیات نهایی یا نقاط انتهایی در یک دنباله دستورات را نشان می‌دهند. تک‌گرایی شمارش این گره‌ها است.

در فرمول $Sc = H \times Rt \times M$ معیارهای ایجیوگو پیچیدگی ساختاری را با ترکیب این سه بعد محاسبه می‌کند. این نتیجه یک نشانه مقداری از اینکه چقدر طراحی نرم‌افزار پیچیده است ارائه

می‌دهد. ساختارهای بیشترین تو درونی، تعداد بیشتری از شاخه‌های اولیه و تعداد بیشتری از نقاط پایانی همگی به افزایش پیچیدگی ساختاری کمک می‌کنند.

علاوه بر این، اندازه‌گیری جداگانه اندازه نرم‌افزار را نشان می‌دهد و تعداد کل اجزای بخش‌های منبع کد نرم‌افزار است. فرمول داده شده S = تعداد کل گره‌ها - 1 که '1'، گره ریشه را نشان می‌دهد. به عبارت دیگر، این معیار به شما ایده‌ای از اینکه نرم‌افزار چقدر بزرگ است خارج از پیچیدگی آن را می‌دهد. حتی اگر ساختار نسبتاً صاف باشد (آن‌قدر که خیلی تو درونی نباشد) تعداد بی‌شمار اجزا می‌تواند به پیچیدگی و چالش آن مشارکت کند.

درک پیچیدگی ساختاری و اندازه‌گیری آن در چندین حوزه توسعه و نگهداری نرم‌افزار کمک می‌کند:

- **قابلیت استفاده**: اگر پیچیدگی کمتر باشد، ممکن است برای کاربران راحت‌تر باشد تا با نرم‌افزار تعامل کنند زیرا معمولاً جریان‌ها و تعاملات ساده‌تری دارد.
- **خوانایی**: نرم‌افزاری که ساختاری ساده‌تر دارد، معمولاً راحت‌تر قابل خواندن و درک است، که آن را برای توسعه‌دهندگان جدید یا برای افرادی که کدها را نگهداری می‌کنند، قابل دسترس‌تر می‌کند.
- **قابل‌تغییری**: اجزاء ساده‌تر و کم‌تر اتصالی معمولاً اسان‌تر قابل تغییر هستند بدون ایجاد عواقب غیرمنتظره در دیگر نقاط نرم‌افزار، که در یک محیط توسعه چابک و تکراری حیاتی است.

با اختصاص ارزش‌های عددی به این مفاهیم، متریک‌های ایجیوگو ابزارهای مفیدی را برای ارزیابی و بهینه‌سازی پیچیدگی و اندازه برنامه‌های نرم‌افزاری به توسعه‌دهندگان ارائه می‌دهد.

خروجی کد با توجه به توضیحات کد :

```
------------------------------------------------------------------------
Ejiogu's Software Metrics
------------------------------------------------------------------------
H: the height of the deepest nested node : 43
Rt: the Twin number of the root: 1
M: the Monadicity 30
Sc = H × Rt × M
Sc: Structural Complexity: 1290
S = total umber of nodes - 1
S: Software Size: 261

Process finished with exit code 0
```

فاکتور structural complexity:

پیچیدگی ساختاری در زمینه نرم‌افزار به توده‌یافتگی و پیونددهی اجزا و ماژول‌های مختلف در یک پایگاه کد اشاره دارد. این معیار نشان‌دهنده این است که چقدر دشوار است کدام نسامتل کد شبکه‌ای باشد، تیبای در نهادن و نگهداری یک سیستم نرم‌افزاری به دلیل ارتباطات و تعاملات بین بخش‌های آن است.

پیچیدگی ساختاری معمولاً از حضور ساختارهای منظم یا مرتبط با همیشه، تعداد بزرگ ماژول‌های وابسته یا ترکیبی از گردان وابسته با داخل با کد، به وجود می‌آید. این پیچیدگی‌ها می‌تواند باعث شود که برای توسعه‌دهندگان دشوار باشد تا کد را درک کنند که منجر به دشواری در دیباگ کردن، اضافه کردن ویژگی‌های جدید یا انجام تغییرات بدون اثرات جانبی ناخواسته شود.

اندازه‌گیری پیچیدگی ساختاری مشتمل بر ارزیابی عواملی مانند درهم‌تنیدگی ساختارهای کنترلی (حلقه‌ها، شرطی‌ها)، تعداد وابستگی‌ها بین ماژول‌ها یا اجزا و جریان کلی داده و کنترل در سیستم است. انواع معیارها و روش‌های محاسباتی، مانند متریک‌های نرم‌افزاری ایجیوگو، هدف دارند تا این جنبه‌های پیچیدگی ساختاری را به طور کمی اندازه‌گیری نمایند.

پیچیدگی ساختاری بالا می‌تواند منجر به تأثیرات منفی متعددی بر توسعه و نگهداری نرم‌افزار شود، شامل افزایش خطاها، طولانی‌شدن زمان توسعه، افزایش هزینه‌های نگهداری و کاهش

کیفیت کلی نرم‌افزار. حل مسائل پیچیدگی ساختاری معمولاً شامل بازسازی کد برای ساده‌سازی ساختار آن، کاهش وابستگی‌ها و بهبود طراحی کلی برای ساخت سیستم قابل فهم‌تر و قابل نگهداری است.

فاکتور maintainability:

بازسازی و نگهداری آسان آن اطلاعات داده می‌شود. قابلیت نگهداری یک ویژگی مهم در زمینه توسعه نرم‌افزار است زیرا هر چه یک سیستم نرم‌افزاری قابل نگهداری‌تر باشد، تغییرات درخواستی، اصلاحات و به‌روزرسانی‌های لازم برای سازمان به راحتی قابل انجام خواهند بود، و هزینه و زمان مورد نیاز برای این تغییرات کاهش خواهد یافت.

بخش پیاده سازی شده توسط **مهدی قضاوی**:

- **4.9: متریک‌های خوانایی (Readability Metrics):**

معیار Readability توسط والستون و فلیکس با عنوان نرخ مستندات معرفی شد، که با فرمول زیر نمایش داده می‌شود:

$$D = 49 * (L^{1.01})$$

که در آن D تعداد صفحات مستند یا همان نرخ LOC بوده و L تعداد ۱۰۰۰ خط کد است.

این معیار نشان می‌دهد که میزان مستندات مورد نیاز، به نسبت اندازه کد است.

با افزایش تعداد خطوط کد(L) ، این معیار نشان می‌دهد که افزایش به نسبتی در تعداد مستندات (D) ضروری است تا قابلیت خواندن و درک بهتری از کد حفظ شود.

4.9 Readability metrics

Walston and Felix [31] defined a ratio of document pages to LOC as:

$$D = 49L^{1.01}$$

Where

D= number of pages of document
L = number of 1000 lines of code.

برای پیاده‌سازی این متریک در کد، در فایل Listener تعریف‌شده باعنوان CustomMahdiListener.py، از متغیر self.line_counts برای شمارش تعداد خطوط کد ورودی استفاده می‌کنیم. سپس برای محاسبه این مقدار، نیاز داریم متد exitCompilationUnit() از کلاس JavaParserListener را Override کنیم و در آن، تعداد خطوط را حساب کنیم:

```
def exitCompilationUnit(self, ctx: JavaParser.CompilationUnitContext):
    start_line = ctx.start.line
    stop_line = ctx.stop.line

    # Calculate the number of lines
    lines_in_context = stop_line - start_line + 1
    self.line_count += lines_in_context
```

سپس این نرخ را بصورت زیر محاسبه می‌کنیم:

```
    "------------------------\nReadability's Software Metrics \n------------
# Walston & Felix ratio formula of document pages as: D = 49 * L, where:
# D = Number of pages of document and L = Number of 1000 lines of code
L = self.line_count / 1000

D = 49 * (L ** 1.01)
```

در مرحله آخر، نیاز داریم دو فاکتور Testability و Integrity از کد را پیاده‌سازی و بررسی کنیم.

برای اندازه‌گیری فاکتور Testability، از دو متریک Cyclomatic Complexity(CC) و Lines of Code(LOC) استفاده می‌کنیم.

```
E = self.graph_listener.graphh.edges_numbers
N = len(self.graph_listener.graphh.nodes)
P = 1  # for a single program
V = E - N + 2 * P
```

متریک CC با فرمول زیر اندازه‌گیری می‌شود:

$$V(G) = E - N + 2P$$

که درآن، E تعداد یال‌ها در گراف Control Flow، N تعداد گره‌ها و P تعداد اجزای متصل(Connected Components) در گراف است. برای این متریک، هرچه مقدار آن بیشتر باشد، به این معنی‌ست که پیچیدگی کد بیشتربوده که منجر به کاهش Testability خواهدشد.

برای متریک LOC نیز از متغیر self.lines_count که قبلا حساب کردیم استفاده می‌کنیم.

برای اندازه گیری فاکتور Integrity، از دو متریک Dependency Cyclomatic Complexity(DCC) و متریک Maintainability Index(MI) استفاده می‌کنیم.

مقدار متریک DCC از فرمول زیر محاسبه می‌شود:

$$V(G) = E / N + 1$$

```
E = self.graph_listener.graphh.edges_numbers
N = len(self.graph_listener.graphh.nodes)
DCC = E / N + 1
```

این متریک، پیچیدگی کدرا براساس Module Dependencyهای آن اندازه می‌گیرد و هرچه مقدار آن بیش‌تر باشد، پیچیدگی بیش‌تر است.

متریک MI را از رابطه زیر بدست می‌آوریم:

$$MI = 171 - 5.2 \times \ln(HalsteadVolume) - 0.23 \times (CyclomaticComplexity) - 16.2 \times \ln(LinesofCode)$$

```
# calculate Maintainability Index (MI):
# MI = 171 - 5.2 × ln(Halstead Volume) - 0.23 × (Cyclomatic Complexity) - 16.2 × ln(LOC)
halstead_volume = (self.graph_listener.N1_operators + self.graph_listener.N2_operands) * (
    math.log(self.graph_listener.n1_uniquieoperators + self.graph_listener.n2_uniqueoperands, 2))

CC = self.graph_listener.graphh.edges_numbers - self.graph_listener.graphh.last_node_numbers + 2

MI = 171 - 5.2 * math.log(halstead_volume, 2) - 0.23 * CC - math.log(self.line_count, 2)
```

که برای این رابطه، تنها نیاز به محاسبه HalsteadVolume داریم که از فرمول زیر بدست می‌اوریم:

$$Volume = (N+n) \times \log(N+n)$$

خروجی این فاکتورها به‌ازای فایل ورودی مشخص‌شده در فایل main.py بصورت زیر خواهد بود:

```
-------------------------
Readability's Software Metrics
-----------------------------
Metrics's formula: D = 49 * (L ** 1.01)
Number of code lines = 15
L: Number of 1000 lines of code = 0.015
E: Number of pages of the document = 0.7047713672443691
--------------------------------------------------
The First Software Quality Factor is *Testability*
Cyclomatic Complexity of code = 3
Number of Lines in the code = 15
--------------------------------------------------
The Second Software Quality Factor is *Integrity*
Dependency Cyclomatic Complexity of code = 2.111111111111111
The Maintainability Index (MI) Metric is: 130.18919263280625
```

بخش پیاده سازی شده توسط **علی سلطانی** :

معیارهای اطلاعاتی هنری و کافورا معیارهایی هستند که برای ارزیابی کیفیت نرم‌افزارها استفاده می‌شوند. این معیارها بر مبنای اطلاعاتی که در نرم‌افزار وجود دارد، ارائه می‌شوند و به ما کمک می‌کنند تا اطلاعاتی درباره عملکرد و ویژگی‌های نرم‌افزار به دست آوریم. به طور کلی، معیارهای اطلاعاتی هنری و کافورا بر اساس اطلاعات موجود در متن برنامه، مثل تعداد خطوط کد، تعداد توابع، تعداد کلاس‌ها و موارد مشابه، به عنوان معیارهای ارزیابی استفاده می‌شوند.

یکی از معیارهای ارزیابی ارائه شده توسط هنری و کافورا، "پیچیدگی جریان اطلاعات" (IFC) است که میزان اطلاعاتی که به داخل و خارج از یک رویه جریان دارد را توصیف می‌کند. این معیار از جریان اطلاعات بین رویه‌ها استفاده می‌کند تا پیچیدگی جریان داده در یک برنامه را نشان دهد. فرمول این معیار به صورت زیر است:

$$IFC = Length \times (Fan\text{-}in \times Fan\text{-}out)^2$$

در اینجا،

Fan in تعداد جریان‌های محلی ورودی به یک رویه به علاوه تعداد ساختارهای داده سراسری است که یک رویه اطلاعات را بازیابی می‌کند.

Fan out تعداد جریان‌های محلی خروجی از یک رویه به علاوه تعداد ساختارهای داده سراسری است که یک رویه اطلاعات را به‌روزرسانی می‌کند.

Length تعداد خطوط کد منبع در رویه است. در این محاسبه، توضیحات تعبیه‌شده نیز شمرده می‌شوند، اما توضیحاتی که قبل از شروع کد اجرایی قرار دارند، شمرده نمی‌شوند.

با استفاده از این معیارها، می‌توانیم به صورت کمی و دقیق، پیچیدگی جریان اطلاعات در نرم‌افزارها را اندازه‌گیری کرده و ارزیابی کنیم.

فاکتور های مرتبط :

- **Modifiability**

اصلاح‌پذیری در کد به میزانی اشاره دارد که چقدر تغییرات به نرم‌افزار اعمال می‌شوند بدون این که خطاهای جدیدی ایجاد شود یا عملکرد آن تحت تاثیر قرار گیرد. وقتی ما اصلاح‌پذیری را در زمینه پیچیدگی جریان اطلاعات هنری و کافورا و معیارهای اطلاعاتی کافورا در نظر می‌گیریم، می‌توانیم بررسی کنیم که چگونه این معیارها تاثیرگذاری دارند و کدامیک میزان مد نظر بر اصلاح‌پذیری را دارند.

1. پیچیدگی جریان اطلاعات (IFC):

- تاثیر بر اصلاح‌پذیری: مقادیر بالای IFC نشان‌دهنده جریان داده‌های پیچیده درون رویه‌ها است. وقتی کد با مقادیر بالای IFC تغییر داده می‌شود، توسعه‌دهندگان ممکن است با چالش‌هایی در فهم و ردیابی وابستگی‌های داده‌ای روبرو شوند که این امر موجب افزایش خطر ایجاد خطاها در زمان اصلاحات می‌شود.

- مثال: اگر یک رویه مقادیر بالایی از fan-in و fan-out داشته باشد که نشانگر حجم قابل توجهی از جریان داده است، اصلاح آن رویه ممکن است نیاز به فهم و به‌روزرسانی چندین وابستگی داده داشته باشد، که این موضوع موجب افزایش خطرات غیرمنتظره در زمان اصلاحات می‌شود.

2. معیارهای اطلاعاتی کافورا:

- تاثیر بر اصلاح‌پذیری: معیارهای اطلاعاتی کافورا، که شامل عواملی مانند طول برنامه، fan-in و fan-out هستند، نشان می‌دهند که بخش‌های کدی معینی چقدر پیچیده هستند و احتمالاً سختی بیشتری برای اصلاح دارند.

- مثال: یک رویه با طول برنامه و fan-in/fan-out بالا ممکن است دارای تعداد زیادی وابستگی و تعامل داده باشد که این موضوع باعث می‌شود اصلاح آن بدون ایجاد اثرات جانبی غیرمنتظره دشوار شود.

به طور خلاصه، مقادیر بالای IFC و معیارهای اطلاعاتی کافورا می‌توانند نشانگر ساختارهای پیچیده کد با جریان داده پیچیده باشند که این موضوع می‌تواند اصلاح‌پذیری را مختل کند. وقتی هدف بهبود اصلاح‌پذیری است، توسعه‌دهندگان ممکن است نیاز به بازطراحی یا بازسازی مناطق پیچیده داشته باشند تا وابستگی‌های داده را ساده‌تر کرده و خطر ایجاد خطاها در زمان اصلاحات را کاهش دهند.
همچنین این معیار به تعداد خطوط کد، عمیق ترین نود درخت و سایز نرم افزار هم سنجیده میشود، که همه موارد پیاده سازی شده در انتها توضیح داده میشوند.

- **Usability**

کاربرپذیری در کد به میزانی اشاره دارد که چقدر این کد برای استفاده و درک توسط توسعه‌دهندگان و سایر افراد قابل فهم و استفاده است. زمانی که ما این موضوع را در زمینه پیچیدگی جریان اطلاعات هنری و معیارهای اطلاعاتی کافورا بررسی می‌کنیم، می‌توانیم ببینیم که چگونه این معیارها بر کاربرپذیری کد تاثیر می‌گذارند.

1. پیچیدگی جریان اطلاعات (IFC):

- تاثیر بر کاربرپذیری: مقادیر بالای IFC ممکن است به این معنا باشد که داخل کد جریان داده‌های پیچیده‌ای وجود دارد که ممکن است برای توسعه‌دهندگان سخت به نظر برسد. این موضوع می‌تواند باعث کاهش کاربرپذیری کد و افزایش زمان و هزینه توسعه شود.

- مثال: وقتی یک رویه با IFC بالا دارای تعداد زیادی از وابستگی‌های داده است، توسعه‌دهندگان ممکن است درک و تجزیه و تحلیل کد را دشوار بیابند که این موضوع می‌تواند به تأخیر در توسعه کد و افزایش احتمال خطاها منجر شود.

2. معیارهای اطلاعاتی کافورا:

- تاثیر بر کاربرپذیری:معیارهای اطلاعاتی کافورا نشان می‌دهند که چقدر ساختار کد ساده یا پیچیده است. این موارد می‌توانند به عنوان نشانگرهایی برای کاربرپذیری کد عمل کنند.

- مثال: یک رویه با طول برنامه بالا و fan-in/fan-out بالا ممکن است برای توسعه‌دهندگان دشوار باشد که منجر به کاهش کاربرپذیری کد شود. به عنوان مثال، افزایش تعداد خطوط کد ممکن است باعث کاهش قابلیت فهم کد و افزایش پیچیدگی شود.

به طور خلاصه، مقادیر بالای IFC و معیارهای اطلاعاتی کافورا ممکن است به پیچیدگی جریان داده و ساختار پیچیده کد اشاره کنند که این موارد می‌توانند به کاهش کاربرپذیری کد منجر شوند. وقتی که هدف بهبود کاربرپذیری است، توسعه‌دهندگان ممکن است نیاز به بهینه‌سازی و ساده‌سازی ساختار کد داشته باشند تا به کاربران کمک کنند که به راحتی کد را درک و تغییر دهند.

همچنین calculate_comment_ratio ، تعداد کل نود ها و پیچیدگی نرم افزار و حجم آن هم در این فاکتور موثر اند که در ادامه نحوه پیاده سازی آن ها گفته میشود.

پیاده سازی:

برای معیار IFC نیاز هست که به پارامتر های تابع ها و متغیر هایی که تعریف میشوند دسترسی داشته تا fan in را به دست آوریم، بنابراین در لیسنر هنگام ورود به این توابع این کار را انجام میدهیم، به این صورت که در زیر آمده است :

```
def enterVariableDeclarator(self, ctx:JavaParser.VariableDeclaratorContext):
    if self.inMethod == True :
        self.variables.append( ctx.variableDeclaratorId().Identifier())
        self.Fan = self.Fan + 1
        self.Fout = self.Fout + 1

def enterFormalParameter(self, ctx:JavaParser.FormalParameterContext):
    self.variables.append( ctx.variableDeclaratorId().Identifier())
    self.Fan = self.Fan + 1

def enterMethodDeclaration(self, ctx:JavaParser.MethodDeclarationContext):
    self.inMethod = True
```

حال برای fan out به متغیر های تغییر پیدا کرده به غیر از داخل متود نیاز داریم که کافی هست تشخیص دهیم ایا داخلی هست یا نه، اگر نه fan out را اضافه کنیم.

در نهایت طول تابع را به دست آورده و مقدار IFC طبق فرمول گفته شده به دست می آید.

برای دو فاکتور گفته شده چندین متریک لازم بود از جمله پیچیدگی نرم افزا و از آن جایی که این ویژگی قبلا پیاده سازی شده بود به صورت ضمنی پیاده سازی کردم.

حال مابقی متریک ها را به ترتیب نحوه پیاده ساطیشان را نشان میدهم :

متریک لاین اف کد :

```
def exitCompilationUnit(self, ctx: JavaParser.CompilationUnitContext):
    start_line = ctx.start.line
    stop_line = ctx.stop.line
    # Calculate the number of lines
    lines_in_context = stop_line - start_line + 1
```

مقدار اول و اخر تعداد کد هارا از هم کم کرده به دست می آوریم.

متریک کامن ریشیو :

```python
def calculate_comment_ratio(file_path):
    total_lines = 0
    comment_lines = 0

    with open(file_path, 'r') as file:
        for line in file:
            total_lines += 1
            line = line.strip()
            if line.startswith('#') or line.startswith('//') or line.startswith('/*') or line.startswit
                    '*') or line.startswith('*/'):
                comment_lines += 1

    if total_lines == 0:
        return 0
    else:
        return comment_lines / total_lines
```

تعداد خطوط کامنت دار را به دست آورده و بر تعداد کل خطوط تقسیم میکنیم.

برای متود تعداد نود به ازای ورود به هر رول یکی به آن اضافه میکنیم.

برای تخمین عمق به هنگام ورود یکی اضافه کرده و به هنگام خروج یکی کم میکنیم و ماکسیموم این مقدار را میگیرم. تمرین درس بوده هست :

```python
def enterEveryRule(self, ctx):
    self.totalNodes += 1
    self.currentDepth += 1

    # Update max depth if needed
    self.maxDepth = max(self.maxDepth, self.currentDepth)

    # Check if the current node is monadic
    if not ctx.children or len(ctx.children) == 1:
        self.monadicity += 1

    # Calculate twin number for non-root nodes
    if self.currentDepth > 0:
        parent = ctx.parentCtx
        if parent is not None:
            self.twinNumber = max(self.twinNumber, parent.getChildCount())
```

حال به ازای ورودی دلخواه این دو متریک و این دو فاکتور را میسنجیم :

بخش متریک ها:

```
------------------------------------------------------------------------
Henry and Kafura's Information Software Metrics
------------------------------------------------------------------------
the number of procedures : 2
the formula for IFC is : Length * (Fin * Fout)^2
print the each procedure IFC :
main: 8064.0
test: 8064.0
the total IFC is : 16128.0
------------------------------------------------------------------------
Ejiogus's Software Metrics
------------------------------------------------------------------------
Structural Complexity (S): 216216
Software Size: 613
Height of Deepest Node (H): 44
Twin Number of the Root (Rt): 9
Monadicity (M): 546
Total Number of Nodes: 614
```

بخش فاکتور ها:

```
********** modifiability **********
related metrics :
Structural Complexity (S): 216216
IFC : 16128.0
Software Size: 613
Height of Deepest Node (H): 44
Lines of code is :31
********** usability **********
IFC : 16128.0
Structural Complexity (S): 216216
Software Size: 613
Height of Deepest Node (H): 44
Total Number of Nodes: 614
calculate_comment_ratio : 0.03225806451612903
```

بخش پیاده سازی شده توسط **احمدرضا طهماسبی**:

سنجه‌های مک‌کب، همچنین به عنوان سنجه‌های cyclomatic complexity شناخته می‌شوند که توسط توماس ج. مک‌کب استفاده می‌شوند به منظور اندازه‌گیری پیچیدگی یک جریان کنترل برنامه. سنجه اصلی معرفی شده توسط مک‌کب، پیچیدگی چرخشی (Cyclomatic Complexity)یا (CC است. این سنجه تعداد مسیرهای مستقل خطی از طریق کد منبع برنامه را اندازه‌گیری می‌کند.

فرمول محاسبه پیچیدگی چرخشی به صورت زیر است:

$$CC = E - N + 2P$$

E تعداد یال‌ها در نمودار جریان کنترل است.

N تعداد گره‌ها در نمودار جریان کنترل است.

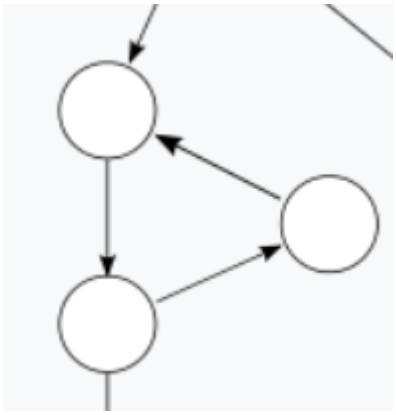
P تعداد مؤلفه‌های متصل (مناطق) در نمودار جریان کنترل است.

هر چه پیچیدگی چرخشی بیشتر باشد، احتمالاً کد پیچیده‌تری داریم. یک پیچیدگی بالا نشان‌دهنده وجود مسیرهای بیشتر احتمالی در کد است که باعث مشکلات در فهم، آزمون و نگهداری می‌شود.

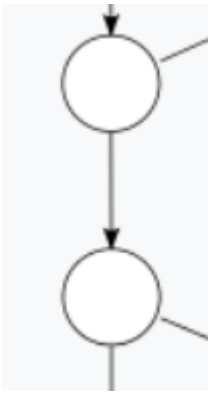
پیچیدگی چرخشی مک‌کب به عنوان یک سنجه مفید در تجزیه و تحلیل نگهداری کد و توسعه نرم‌افزار به عنوان یک راهنمای مقدار آزمون برنامه استفاده می‌شود. ارزیابی و استفاده از این سنجه باید به همراه سایر سنجه‌ها و ارزیابی‌های کیفی صورت گیرد تا یک فهم جامع از کیفیت و نگهداری کد حاصل شود.

نحوه کار کردن با این متریک به این صورت اسا که از درخت پارسر شروع به ساخت درخت AST می کنیم به این صورت که :

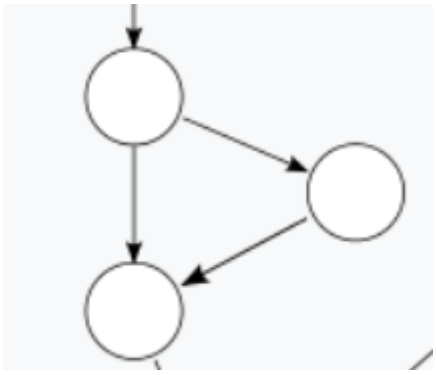
پیاده سازی فور لوپ ها :



پیاده سازی assignment ها :



پیاده سازی if statement ها :



به همین ترتیب نود های مورد نظر را با توجه به ترتیب گفته شده جایگذاری می کنیم. در نهایت درخت را طوری تحلیل می کنیم که مقادیر فاکتور های گفته شده بدست می آیند.

برای مثال کد پیاده سازی for :

```
def create_node_for_while(self):
    first = node()
    second = node()
    third = node()
    if self.current_node == None:
        self.current_node = first
    else:
        self.current_node.next = first
        self.edges_numbers = self.edges_numbers + 1
    #----------
    first.number = self.last_node_numbers + 1
    self.last_node_numbers = self.last_node_numbers + 1
    first.next = second
    self.edges_numbers = self.edges_numbers + 1
    self.nodes.append(first)
    #----------
    second.number = self.last_node_numbers + 1
    self.last_node_numbers = self.last_node_numbers + 1
    second.next_divert = third
    self.edges_numbers = self.edges_numbers + 1
    self.nodes.append(second)
    #----------
    third.number = self.last_node_numbers + 1
    self.last_node_numbers = self.last_node_numbers + 1
    third.next = first
    self.edges_numbers = self.edges_numbers + 1
    self.nodes.append(third)
    #----------
    second.next = None
    self.current_node = second
```

با توجه به کد های بالا می بینیم که به طور دستی نود ها را جایگذاری کرده ایم.

در مرحله آخر باید دو ویژگی usability و Testability از کد بررسی میشد.

که متریک های آن به این صورت هست :

Usability :

Cyclomatic Complexity Metric : CC = E – N + 2P

Halsted Volume Metric : HV = N/2* LOG2(N/L)* L/V

McCabe’s Cyclomatic Complexity (MCC): MCC = E-N+ 2

Maintainability Index (MI) Metric = MI = 171 -5.2LOG2(HV)-0.23*CC -16.2LOG2(LOC)

Lines of Code Metric: ZIGMA LOCi

Testability:

Cyclomatic Complexity Metric : CC = E – N + 2P

Lines of Code Metric: ZIGMA LOCi

Maintainability Index (MI) Metric = MI = 171 -5.2LOG2(HV)-0.23*CC -16.2LOG2(LOC)

McCabe's Cyclomatic Complexity (MCC): MCC = E-N+ 2

بخش پیاده سازی شده توسط **وحید محمدی**:

Albrecht’s Function Point Method

این روش بر اساس پنج متریک منطقی قابل شناسایی توسط کاربر است که به دو نوع تابع داده و سه نوع تابع تراکنشی تقسیم می شوند. این توابع عبارتند از:

ورودی خارجی (EI): اینها فرآیندهای ابتدایی هستند که در آن داده های مشتق شده از خارج به داخل از مرز عبور می کنند.

خروجی خارجی (EO): اینها فرآیندهای ابتدایی هستند که در آن داده های مشتق شده از داخل مرز به خارج عبور می کنند.

درخواست های خارجی (EQ): اینها فرآیندهای ابتدایی با اجزای ورودی و خروجی هستند که منجر به بازیابی داده ها از یک یا چند فایل منطقی داخلی و فایل های رابط خارجی می شود.

فایل منطقی داخلی (ILF): اینها گروه های قابل شناسایی کاربر از داده های منطقی مرتبط هستند که کاملاً در محدوده برنامه ها قرار دارند و از طریق ورودی های خارجی نگهداری می شوند.

فایل رابط خارجی (EIF): اینها گروه های قابل شناسایی کاربر از داده های منطقی مرتبط هستند که فقط برای مقاصد مرجع استفاده می شوند و کاملاً خارج از سیستم قرار دارند.

هر یک از این عناصر با شمارش عناصر مشخصه آن، مانند ارجاعات فایل یا فیلدهای منطقی، کمی و وزن می شوند. اعداد حاصل (FP تعدیل نشده) در مجموعه توابع اضافه شده، تغییر یافته یا حذف شده گروه بندی می شوند و با ضریب تنظیم ارزش (VAF) ترکیب می شوند تا عدد نهایی FP به دست آید.

هر یک از این پنج متریک شامل سه وزن میشوند که به صورت زیر است:

	LOW	AVERAGE	COMPLEXITY
EI	3	4	6
EO	4	5	7
ILF	5	7	10
EIF	7	10	15
EQ	3	4	6

که با استفاده از ۵ متریک و وزن های آنها میتوان مقدار function counts را بدست آورد که به صورت زیر است:

$$FC = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} \times x_{ij}$$

برای ضریب ارزش ۱۴ پارامتر وجود دارد که مقداری بین ۰ تا ۵ دارند که بسته به نوع برنامه باید تعیین شود که به صورت زیر VAF حساب میشود:

$$VAF = 0.65 + 0.01 \sum_{i=1}^{14} c_i$$

در اخر مقدار نهایی متریک نقطه تابعی به صورت زیر در می آید:

$$FP = FC \times VAF$$

و خروجی برنامه هم به صورت زیر است:

Albrecht's Function Points Metrics	
Metric	Number of Component
EI	4
EO	3
EQ	1
ILF	1
EIF	1

General System Characteristics	
Feature	Scale
Data Communication	2.974372171979583
Distributed Function	4.704470423871255
Heavily Used	3.087550648316848
Configuration	2.973769167649282
Transaction Rate	1.2120075568157274
Online Data Entry	0.12544646457029462
End User Efficiency	2.0250967066368237
Online Update	3.8253051230015074
Complex Processing	0.5039930246555624
Reusability	3.72760917044862
Installation Ease	3.8845304820380107
Operational Ease	1.0001186651462053
Multiple Sites	2.362533590109312
Facilitation of Change	0.7549577303678884
Value Adjustments Factor (VAF)	0.9816176092560693
Function Counts (FC)	167
Function Points (FP)	163.93014074576357

منابع :

McCabe, T. J. (1976). "A Complexity Measure". In Proceedings of the 2nd International Conference on Software Engineering (ICSE '76).

Microsoft. (2005). "How to: Calculate Maintainability Index." MSDN - Microsoft. (2005). "How to: Calculate Maintainability Index." MSDN - Maintainability Index.

Chidamber, S. R., & Kemerer, C. F. (1994). "A Metrics Suite for Object-Oriented Design". IEEE Transactions on Software Engineering, 20(6), 476–493.

- **Source:** Sommerville, I. (2011). Software engineering (9th ed.). Boston: Pearson Education.

- **Source:** Pressman, R. S. (2010). Software engineering: A practitioner's approach (7th ed.). Boston: McGraw-Hill.

- **Source:** Basili, V. R., & Perricone, G. V. (1984). Software complexity measures and program development productivity. IEEE Transactions on Software Engineering, SE-10(5), 432-442.

- **Source:** DeMarco, T., & Lister, T. (1985). Peopleware: Productive projects and teams. New York: Dorset House.

- **Source:** Fenton, N., & Pfleeger, S. L. (1997). Software metrics: A rigorous and practical approach (2nd ed.). London: PWS Publishing.

- **Source:** Boehm, B. W. (1981). Software engineering economics. Englewood Cliffs, NJ: Prentice-Hall.

- **Source:** Fan, X., & Zhang, L. (2000). A study on information flow complexity and software maintainability. Journal of Systems and Software, 52(1), 1-12.

- **Source:** Li, H., & Yang, Y. (2006). Measuring information flow complexity in object-oriented software. Journal of Software Maintenance and Evolution: Research and Practice, 18(1), 1-17.

- **Source:** McCabe, T. J. (1976). Complexity measure for software. IEEE Transactions on Software Engineering, 2(4), 308-320.

- **Source:** Henry, S. M., & Kafura, D. (1981). Software complexity measures and maintenance productivity. IEEE Transactions on Software Engineering, 7(4), 486-496.

پایان