システム創成工学専攻
情報システムコース
Department of Systems Innovation
Advanced Information System Course

令和５年度 専攻科特別研究論文

構文木と視線移動の自動マッピング手法を用いたプログラム理解過程の分析

Program Comprehension Process Analysis used Automatic Mapping of Syntax Trees and Eye Movement

指導教員名　上野 秀剛 准教授
論文提出者名　吉岡 春彦

独立行政法人 国立高等専門学校機構
奈良工業高等専門学校 専攻科
National Institute of Technology, Nara College
Faculty of Advanced Engineering

構文木と視線移動の自動マッピング手法を用いたプログラム理解過程の分析

Program Comprehension Process Analysis
used Automatic Mapping of Syntax Trees and Eye Movement

吉岡 春彦
Yoshioka Haruhiko

独立行政法人 国立高等専門学校機構
奈良工業高等専門学校 専攻科 システム創成工学専攻 情報システムコース
大和郡山市矢田町 22 番地（〒 639-1080）
National Institute of Technology, Nara College, Faculty of Advanced Engineering
22 Yata-cho, Yamatokoriyama, Nara 639-1080, Japan

Abstract: To analyze the developers' eye movement to the source code while understanding a program, researchers need to map eye movements to the position of the source code being read. However, the eye movements are recorded as a list of the display's coordinates, hence scrolling and window movement on the editor make mapping difficult. Further, to extract the comprehension behavior for different source codes, it is necessary to consider the differences in control flow, format, identifiers, and other factors, which is time-consuming during analysis. This study analyzed eye movement during program comprehension tasks using our proposed method. Our method identifies the word being looked at and converts it into a corresponding syntax node. The experimental results show a significant difference in the syntax node focused on between the understood/not-understood participant groups. The experiment did not use the conventional eye movement analysis for individual source codes and demonstrated the effectiveness of the proposed method.

Keywords: Eye Tracking, Program Comprehension, Syntax Analysis;

関連業績リスト

1. Haruhiko Yoshioka, Hidetake Uwano, “Automatic Mapping of Syntax Trees and Eye Movement for Semantic-Based Program Comprehension Pattern Extraction,” 15th International Symposium on Advances in Technology Education (ISATE), p.p. 220–224, 2022.

2. 吉岡春彦, 上野秀剛, “プログラム理解パターン抽出のための構文木と視線移動の自動マッピング手法,” 第 29 回ソフトウェア工学の基礎ワークショップ (FOSE2022), Vol. 29, p.p. 34–42, 2022.

3. 吉岡春彦, 上野秀剛, “構文木と視線移動の自動マッピング手法を用いたプログラム理解過程の分析,” ソフトウェアエンジニアリングシンポジウム 2023 (SES2023), Vol. 2023, No. 174, p.p. 183–190, 2023.

4. Haruhiko Yoshioka, Hidetake Uwano, “An Analysis of Program Comprehension Process by Eye Movement Mapping to Syntax Trees,” 26th IEEE/ACIS International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD2023-Winter), 2023.

5. Haruhiko Yoshioka, Hidetake Uwano, “An Analysis of Program Comprehension Process by Eye Movement Mapping to Syntax Trees,” Springer’s Studies in Computational Intelligence Series (SCIS), 2023.

.

目次

1. はじめに **1**

2. 関連研究 **4**

3. 視線移動の構文木へのマッピング **6**

3.1 提案手法の概要 6

3.2 提案手法の構成 6

3.3 提案手法の利点 10

4. 実験 **13**

4.1 実験環境とタスク 13

4.2 分析 14

5. 結果と考察 **17**

6. おわりに **21**

参考文献 **24**

図目次

1.1 for 文 3
1.2 while 文 3
1.3 2 つのソースコードに対する視線移動 3

3.1 提案手法の構成 7
3.2 Task 13 のソースコード 8
3.3 Task 13 の構文木 9
3.4 構文木を使った視線移動の可視化 11

4.1 構文要素ごとの注視時間 15

5.1 各構文種類に対する注視の割合 17
5.2 Task 13 における各構文種類に対する注視の割合 20

表目次

1.1 ディスプレイにおける座標単位の視線移動 1

3.1 構文情報を用いた視線移動の出力例 8

4.1 実験に使用したタスク 14
4.2 分析で用いる構文種類 16

5.1 method1 の各要素に対する注視割合 20

1. はじめに

ソフトウェア工学の分野の 1 つに，プログラムを読んでいる間の理解過程を対象としたプログラム理解の研究がある．プログラム理解の効率的・効果的な方法を明らかにすることは開発者に効率的な読み方を教え，実装やデバッグの効率を改善する事で開発コストの削減に寄与する．これまでに多数の研究がプログラム理解の分析を目的に作業者がソースコードを読む過程を計測している [1–4]．開発者がソースコードを読む際の視線移動を異なる開発者が見ることでバグ発見効率が上がることを確認した研究もあり [5]，視線移動から有用な情報を抽出し，優れた開発者が持つ理解パターンや理解戦略を開発者に提示することは，開発効率の向上や開発者の教育に有用である．

作業者の視線を分析する研究はディスプレイに対する視線移動を計測する視線計測装置を用いることが一般的である．視線計測装置が計測する視線移動は，ディスプレイ上の座標の時系列情報として記録される．表 1.1 に視線計測装置によって記録された視線移動の例を示す．表の各行はある時刻に作業者が注視していたディスプレイ上の座標を表す．視線移動は視線計測装置の計測周期によって 1 秒間に 30〜300 点が計測され，分析の際には一定の範囲に一定時間以上留まった視線を停留として統合する場合がある．

ソースコードに対する視線移動の分析例として，Hauser ら [1] と Busjahn ら [6] はそ

表 1.1　ディスプレイにおける座標単位の視線移動

経過時間	X 座標	Y 座標
24:54.1	322	457
24:54.1	322	458
24:54.2	328	463
24:54.2	326	469
24:54.2	320	479

れぞれの研究において注視される座標位置の変化に着目し，熟練者が初心者よりも視線を上下に移動する傾向にあることを発見している．また，ソースコードの行・列と視線移動の関係に着目した研究も存在する．ソースコードを対象とした視線移動の場合，ソースコードはエディタや IDE（統合開発環境）に表示される．そのため，ウィンドウ位置の移動やソースコードのスクロール，タブの切り替えなどによって同じ座標に表示されるソースコードは変化する．視線計測を用いた研究の一部では実験用プログラムや IDE のプラグインによって視線移動と操作履歴を組み合わせ，視線移動をソースコード上の行と列に変換している [7,8]．行・列で表現された視線移動は，被験者間で操作が異なる場合でも同じ行や単語に対する注視を識別できるため，同じソースコードに対する異なる被験者の視線を比較し，特徴を抽出することができる．

一方で，異なるソースコードの場合，処理内容が同じコード片であっても制御フローや変数名，インデントなどのフォーマットが異なるため，異なる座標や行・列番号として記録される．図 1.3 に同じ処理内容を異なる制御構文で実装した 2 つのソースコード片と，それぞれに対する視線移動の例を示す．2 つのソースコード片は，いずれも 1 から 10 までの和を計算し変数 sum に格納するが，(a) は for 文を，(b) は while 文を用いている．図の円は視線の停留した箇所（停留点），線は連続した停留点のつながりを表す．2 つの視線移動はいずれも同じ処理（インデックス変数の初期化，条件式，インデックスの増加，結果の計算）を同じ順序で見ている．しかし，2 つのソースコード片は異なる構造を持つため，行列単位の視線移動は for 文に対して「左から右へ」見ているものとして，while 文に対して「上から下へ」見ているものとして記録される．一方で，プログラム理解効率の向上に有用な知見を得るためには異なる構造であっても処理内容が同一なコード片に対する視線を同一のものとして抽出することが望ましい．従来のソースコードを対象とした視線分析においてはディスプレイ座標や行・列に基づいた視線移動をソースコード上にプロットすることで可視化し，研究者が理解・分析した視線移動の内容をソースコード間で比較している．この作業はソースコードごと，作業者の視線ごとに行う必要があるため分析には多大な時間を要する．

本論文は視線計測装置が出力する座標単位の視線移動を，ソースコードから作成される構文木のノードに対する遷移に変換する手法を提案する．また，提案手法の有用性を検証するために，本手法を用いて視線移動の分析を行う．本手法はレビュー対象のソースコードを構文解析し，行列番号と対応する構文木のノードを割り出すことで行列番号に変換した視線移動と対応づける．本論文ではプログラム理解タスク時の視線移動から構文要素の

```
int sum = 0;
for (int i = 1;i <= 10;i++)
{
    sum +=i ;
}
```

図 1.1　for 文

```
int sum = 0;
int i = 1;
while (i <= 10) {
    sum += i;
    i++;
}
```

図 1.2　while 文

図 1.3　2 つのソースコードに対する視線移動

各種類に対する注視割合を求め，理解タスクの正否との関係を分析する．

2. 関連研究

視線移動の計測は初心者と熟練プログラマの違いを分析するためによく用いられる．一定の時間，一定の範囲内に視線が留まる状態を停留 (fixation) と呼び，その中心座標を停留点 (fixation point) と呼ぶ．連続した停留点は読み手が興味を持つ場所の時間変化を表すとされる．Hauser らは初心者と熟練者の間における停留点に基づいた視線移動の違いを比較した [1]．実験の結果，熟練者グループは非線形にソースコードを読む傾向にあるが，初心者は線形に読むことが分かった．

視線移動の分析に用いられる別の定義として Area of Interest (AOI) がある．AOI はその範囲内を注視した場合，同一の物を見ていると見なされる領域であり，着目の対象である画像やソースコードに対して，その一部の領域を表す矩形，または円形として定義する．同じ AOI に連続した視線が見られた場合，その AOI に対する長時間の注視として視線移動を統合する．例えばソースコードを対象とする場合，ソースコードを画像としてディスプレイ上に表示し，各メソッドや行，単語に対して手動で AOI を定義することで各 AOI に対する注視時間の比較や AOI 間の視線移動の移り変わりを分析する．Rodeghero らはメソッド宣言，メソッドの呼び出し，制御フロー，その他の 4 種類の AOI をソースコード上に定義し，それぞれに対する注視時間の長さを分析した [2]．分析の結果から熟練の Java プログラマはメソッド呼び出しと制御フローをメソッド宣言よりも長い時間見ていることが分かった．Crosby らはソースコード中の単語に定義した各 AOI に対するレビュー時間の配分を初心者と熟練者で比較し，熟練者は初心者よりもコメントを見る時間が短いことを明らかにしている [3]．Peitek らは初心者と中級者の読み方を比較するために，座標単位の視線データを行単位に変換し中級者は初心者より，より頻繁にソースコードの上の行に視線移動をすることを明らかにした [7]．

ソースコードを画像で表示する場合，スクロールやタブの切り替えは行わず，1 画面内に収まる短いソースコードやコード片を対象とした分析が多い．一方で，一部の研究ではスクロールや複数のソースコードの表示切り替えなど操作履歴を元に座標単位の視線移

動から行列を求める実験用のプログラムを作成している．オープンソースソフトウェアの 1 つである iTrace は，座標単位の視線情報をソースコードの行と列番号に変換する[*1]．iTrace は Eclipse のプラグインとして実装されており，iTrace を用いた視線移動の分析も行われている [8–10].

[*1] https://www.i-trace.org

3. 視線移動の構文木へのマッピング

3.1 提案手法の概要

提案手法は視線計測装置が出力する座標単位の視線移動をソースコードから生成される構文木のノードに対する遷移に変換する．提案手法はレビュー対象のソースコードを構文解析し，行列番号と対応する構文木のノードを割り出す．その後，視線計測装置が出力した座標単位の視線移動を，ソースコード中の行・列番号に変換し，構文木のノードと対応づける．

3.2 提案手法の構成

図 3.1 に提案手法の構成を示す．四角はシステムを構成するモジュールを表し，細い矢印は情報の流れを表す．ソースコードを読んでいる被験者の視線移動は視線計測装置によって計測される．視線計測装置は各時点における注視点をディスプレイ上の座標（例えば，X:121, Y:313）として時系列に出力する．座標 - 行/列変換モジュールは座標単位の視線移動とソースコードを入力として受け取り，ソースコード名と行・列番号 (Main.java, 行:1, 列:13) として視線を出力する．このとき，行・列番号からソースコードの単語または文字を抽出し，構文解析で得られた構文木上のノードと対応をとる．また，同じ単語，文字に対する連続した注視は 1 つに統合する．構文木/視線結合モジュールはパーサが生成した構文木と，行・列単位の視線移動を受け取り，構文ノード単位の視線移動を出力する．パーサが出力する構文解析の結果には，ソースコード中の各単語の位置を表す行・列番号と，文字数，およびその単語の構文種類が含まれる．構文木/視線結合モジュールは行・列番号に変換された視線移動を構文解析結果の各ノードが持つ行・列番号と対応付けすることで，視線移動を構文木上のノード単位に変換する．

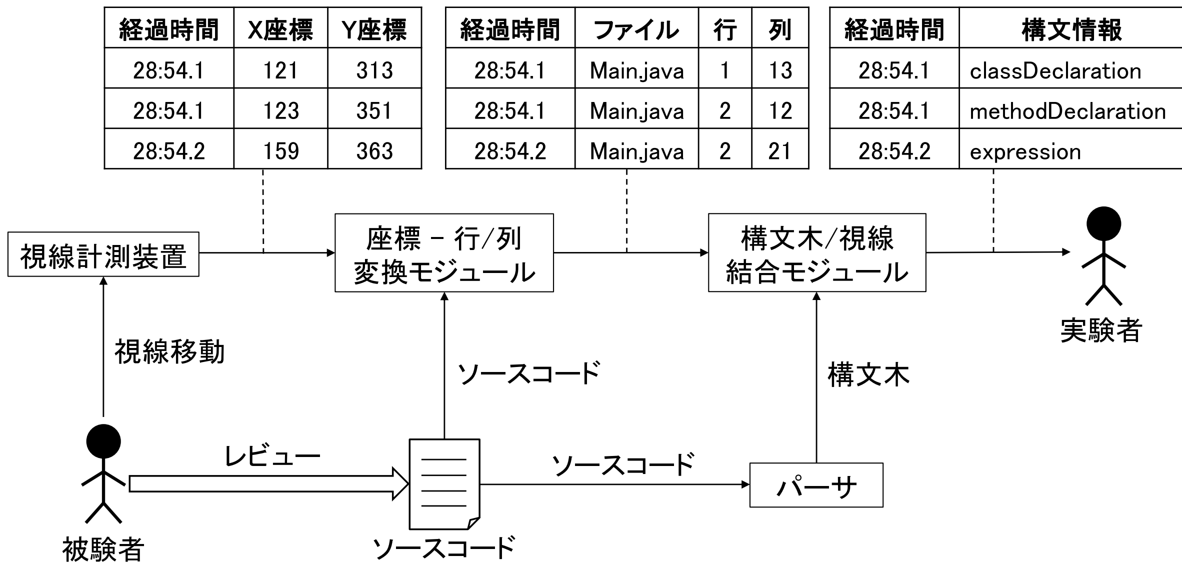

図 3.1　提案手法の構成

提案手法は Python で実装し，座標単位の視線移動から行・列番号を抽出する “座標 - 行/列変換モジュール” の実装として iTrace を用いた．iTrace は IDE が提供する API を用いて，視線座標を，ソースコード上の行・列番号にマッピングする [11]．“構文木/視線結合モジュール” の実装にはオープンソースの parser generator である，ANTLR[*1]を用い，対象言語を Java[*2]とした．Java 以外の言語についても対応するパーサを用いることで任意のプログラミング言語で記述されたソースコードに対する視線移動を分析できる．また，分析を補助するため以下の機能を実装した．

- ソースコード上に視線移動を可視化
- 構文木の各ノードに注視時間を付与し，Graphviz が対応する形式 (PNG, PDF, EPS, SVG など) で出力

表 3.1 に図 1.3(a) の視線移動を提案手法を用いて変換した例を示す．表の各行は視線の停留した単語に対応するノードの情報を表す．視線移動の列は視線が停留した単語を表すノードとその親ノードを表し，ID1 の視線が main メソッド内にある for 文の初期化式の i を見ていることを示している．

[*1] https://www.antlr.org
[*2] https://github.com/antlr/grammars-v4/tree/master/java/java

表 3.1　構文情報を用いた視線移動の出力例

ID	経過時間	視線移動
1	24:54.1	main メソッド / for 文 / 初期化式 / i
2	24:54.1	main メソッド / for 文 / 条件式 / i
3	24:54.2	main メソッド / for 文 / 変化式 / ++
4	24:54.2	main メソッド / for 文 / ブロック / sum

```
1  public class Main {
2    static int[] coin = {1, 5, 10, 50,
3                              100, 500};
4
5    public static void main(String arg[]) {
6      int x = 70;
7
8      int r = method1(x, 5);
9
10     System.out.printf("%d\n", r);
11   }
12
13   static int method1(int x, int i) {
14     if(x == 0)
15       return 1;
16
17     else if(x < 0)
18       return 0;
19
20     else if(i < 0)
21       return 0;
22
23     else {
24       return method1(x-coin[i], i)
25                       + method1(x, i-1);
26     }
27   }
28 }
```

図 3.2　Task 13 のソースコード

本稿の実験で用いたソースコードと，ソースコードから生成した構文木の例をそれぞれ図 3.2，図 3.3 に示す．構文木は method1 メソッドの宣言部に対応するノード (methodDeclaration) 以下のみを示し，2 つ目の if 文（図 3.2 の 17 行目）以降を削除している．図 3.3 の葉ノードはソースコード上の単語を表し，内側のノードは子ノードの属する構文上の要素を表す．例えば，左側にあるノード (formalParameter) は図 3.2 の 13 行目にある 2 つの単語 (int, x) から構成される仮引数を表し，13 行目の method1 メソッド (methodDeclaration) の一部である．提案手法は視線移動を停留点上にある単語と，行・

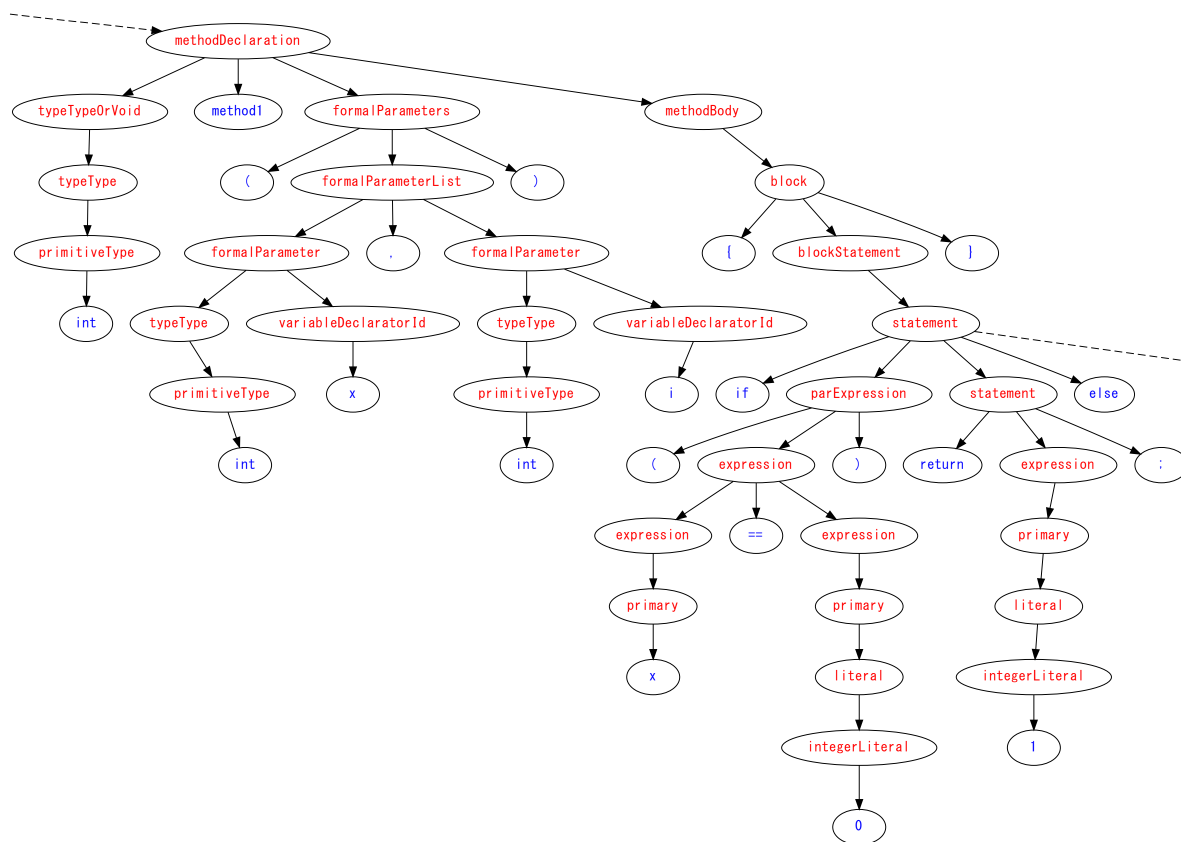

図 3.3　Task 13 の構文木

列番号，および単語が属する構文上の要素全てを組み合わせた情報に変換する．より上位の構文要素の情報を含むことで，例えば，図 3.2 の 15 行目 14 列目に対する視線を，以下の異なる粒度で捉えることができる．

- 単語 “1” に対する視線
- return 文に対する視線
- if 文に対する視線
- method1 メソッドに対する視線
- Main クラスに対する視線

提案手法は構文木のノードに対する遷移として視線移動を表現することで，ソースコードの表示位置やフォーマットによる違いを取り除く．また，視線が停留した単語が属するブロックやメソッド，クラスの情報を含む情報として出力するため，分析目

的に応じて異なる粒度で視線移動のパターンを抽出することを可能にする．例えばMethodDeclaration(メソッド宣言) や MethodBody（メソッド本体），MethodCall（メソッド呼び出し）に属する単語に対する視線移動を抽出・分析することでメソッド単位の視線移動を分析できる．また，1 つの単語に対する異なる粒度の情報をベクトルとして表現することで，単語が持つ構文上の意味に対する視線の遷移として特徴分析やパターンマイニングが行えると考えられる．

3.3　提案手法の利点

従来の視線移動の表現方法と比較して，提案手法は以下の利点がある．

- **構文上の文を単位とした分析が容易**

  従来の行・列番号で表される視線移動は単語を単位とした表現である．テキスト上の行を単位とした分析 [7] も行われているが，複数の文が含まれてる場合や，複数行で 1 文を構成することもある．そのため，視線が停留した行がどのような内容か研究者が読み取る必要がある．

  一方で提案手法は構文上の “文” に対する一連の視線として解釈可能な文字列を出力する．図 1.3(a) で可視化した視線移動を構文木上に表現した結果を図 3.4 に示す．細い点線は視線移動が停留したトークンに対応するノードの順序を示す．太い点線はトークン単位の視線移動を親ノードを単位とした形で要約した視線移動を示す．提案手法は構文上の親ノードと視線を対応づけることで，構文上の “文” に対する視線として扱うことができる．太い点線が示す視線移動は初期化式，条件式，変化式，ブロック内を順に見ていることが容易に理解できる．プログラムの基本的な単位である文を単位とした視線移動はその解釈が容易であると考えられる．また，提案手法は表 3.1 に示したように，視線が停留したトークンが属する文が文字列で出力されるため，パターンマイニングの手法を用いることで理解パターンや理解戦略の抽出が可能である．

- **異なる抽象度による視線移動の分析が可能**

  視線を文単位で要約する事と同様に，より上位の構文要素で視線を要約することで，ブロックやメソッド，クラスを単位とした視線移動の生成が可能である．従来

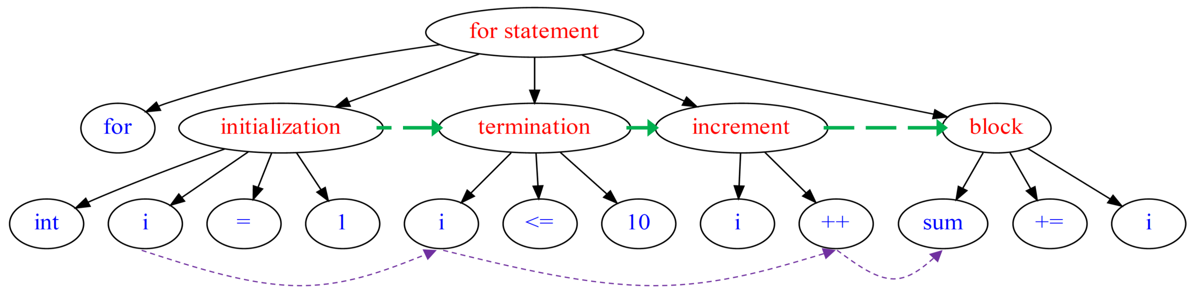

図 3.4　構文木を使った視線移動の可視化

研究においては，実験者が分析対象とする粒度で AOI を設定することで同様の分析が可能であったが，座標情報や行・列番号で個別に定義する必要があることから規模の大きいソースコードを対象とした視線分析においては手間が大きい．また，実験で計測した結果に基づいた探索的な分析が困難である．

提案手法は構文情報に基づいて，AOI の事前定義無しにプログラム理解時における理解単位を構成する文やブロック，メソッド，クラスと視線移動を関連付ける事ができる．そのため，従来分析が困難であった規模の大きなソースコードに対する，長時間にわたる視線移動に対しても分析が可能である．

- **構文情報のベクトル化によるパターンマイニングとラベル予測**

  提案手法が出力する視線情報は，視線が停留した単語と，単語が属する構文要素（文やブロック，メソッド，クラス）の双方を含む．ある単語に対する視線は，単語が持つ異なる粒度における意味に対する注視として解釈が可能である．例えば，あるメソッド内の for 文で定義されている index 変数への注視は“変数名，型の理解”，“代入値の理解”，“index を用いているループの処理回数の理解”，“メソッド引数に対応する出力値の理解”など異なる粒度の理解過程の一部を構成している可能性がある．ある注視や連続した視線移動がどのような理解過程の一部か理解することは，従来，研究者による手動で行われてきた．提案手法が出力する構文情報から，特定の注視や視線移動を特徴付けるベクトルを生成することで，共通した理解を表すパターンのマイニングや機械学習によるラベル予測が可能になる．

- **異なるソースコードを対象とした分析**

  異なるソースコードの一部に同じ処理内容（例えば配列の合計を計算）が含まれ

る場合，同一の被験者が，または異なる被験者が各ソースコードに対して同じ読み方をする可能性がある．しかし，同じ処理であっても異なるフォーマットや変数名などを用いている場合，視線の停留位置を表す行・列番号や単語からパターンを抽出することは難しい．提案手法は構文情報を含む視線移動を出力するため，同じ構文要素を持つ単語に対する視線移動を識別子や行・文字数の影響を受けずに抽出することができる．これによって，異なるソースコードに対する視線移動から共通する理解パターンの機械的な抽出が可能になる．

上記の利点はいずれも従来手法では時間を要した視線移動の分析をより短い時間で実行する点で有用であり，開発者の視線移動から理解パターンや理解戦略を抽出することを容易にする．

4. 実験

日本語と Java で記述されたプログラム課題を被験者に提示し，処理内容を理解する間の視線を計測する．被験者は奈良工業高等専門学校の学生 14 人で，年齢は 19 歳から 21 歳，全員が Java によるプログラミングの基礎講義を受講済みである．

4.1 実験環境とタスク

実験は被験者 1 名と実験者 2 名のみが居る静かな部屋で実施する．実験に使用するのは，視線計測装置，タスク提示用 PC，記録用 PC である．視線計測には Tobii 社製 tobii Eye Tracker 4C を用いる．

タスクは被験者 1 人につき 16 問を与える．実験用に作成したタスク提示ツールを用いて被験者に日本語で記述されたプログラムの仕様と，対応する Java のソースコード 1 つを提示する．各タスクでは被験者にプログラムの仕様とソースコードに加え，"6 行目が 2 回目に実行された時の a の値を答えてください"のように，ソースコードの動作を理解できているか確認するための質問を提示する．被験者はタスク中に質問への回答が分かった時点で口頭で回答する．回答内容が事前に用意した解答と一致していればソースコードの動作を理解しているとみなす．回答が一致しないか，制限時間を超過した場合，動作を理解していないとみなす．回答の正誤は被験者に伝えない．本実験ではプログラムに対する理解の有無が同程度の件数計測できるように，タスクの難易度と制限時間を調整する．タスクの難易度は低難易度と高難易度をそれぞれ 8 件用意する．低難易度は main メソッドのみからなり，1 重の繰り返し文や条件分岐で構成された理解が容易と思われるソースコードを使用する．高難易度は複数メソッドの使用や再帰構造を持ち，制限時間以内での理解が難しいと期待できる複雑なソースコードを使用する．制限時間は低難易度の理解には十分で，かつ高難易度の理解に不十分であることを予備実験によって確認し，2 分 30 秒とした．また，各タスクを提示する順番は，順序効果を考慮しカウンターバランスを行

表 4.1　実験に使用したタスク

	タスク	仕様
1	Factorial	階乗の計算
2	SearchMax	最大値検索
3	PrimeNum	素数判定
4	SearchMedian	中央値検索
5	Power	累乗計算
6	Swap	2 つの数値の入れ替え
7	Substring	指定の文字列を含むか判定
8	ReverseString	文字列を反転させる
9	TowerOfHanoi	ハノイの塔
10	NumOfRoute	経路数を求める
11	Permutation	順列を全列挙する
12	Combination	組み合わせを漸化式から求める
13	PayMoney	支払う硬貨の組み合わせを求める
14	StrCombination	文字列の組み合わせを求める
15	CloudSim	雲の移動シミュレーション
16	lcm_gcd	最小公倍数と最大公約数を求める

う．表 4.1 にタスクの一覧を示す.

4.2　分析

タスクで収集した被験者の視線データを提案手法を用いて分析する．本稿ではプログラム理解の有無と重視して着目する構文要素の関係性を明らかにするために，各構文種類に対する注視割合を求め，タスクを正解した被験者群と不正解の被験者群を比較する．タスクとして提示したソースコードに対する視線移動を提案手法により構文要素にマッピングする，

ANTLR が生成するパーサは Java のソースコードに対して 105 個の構文種類（意味ノード）と対応する単語（単語ノード）を出力する．実験で用いるタスクはそのうち 41

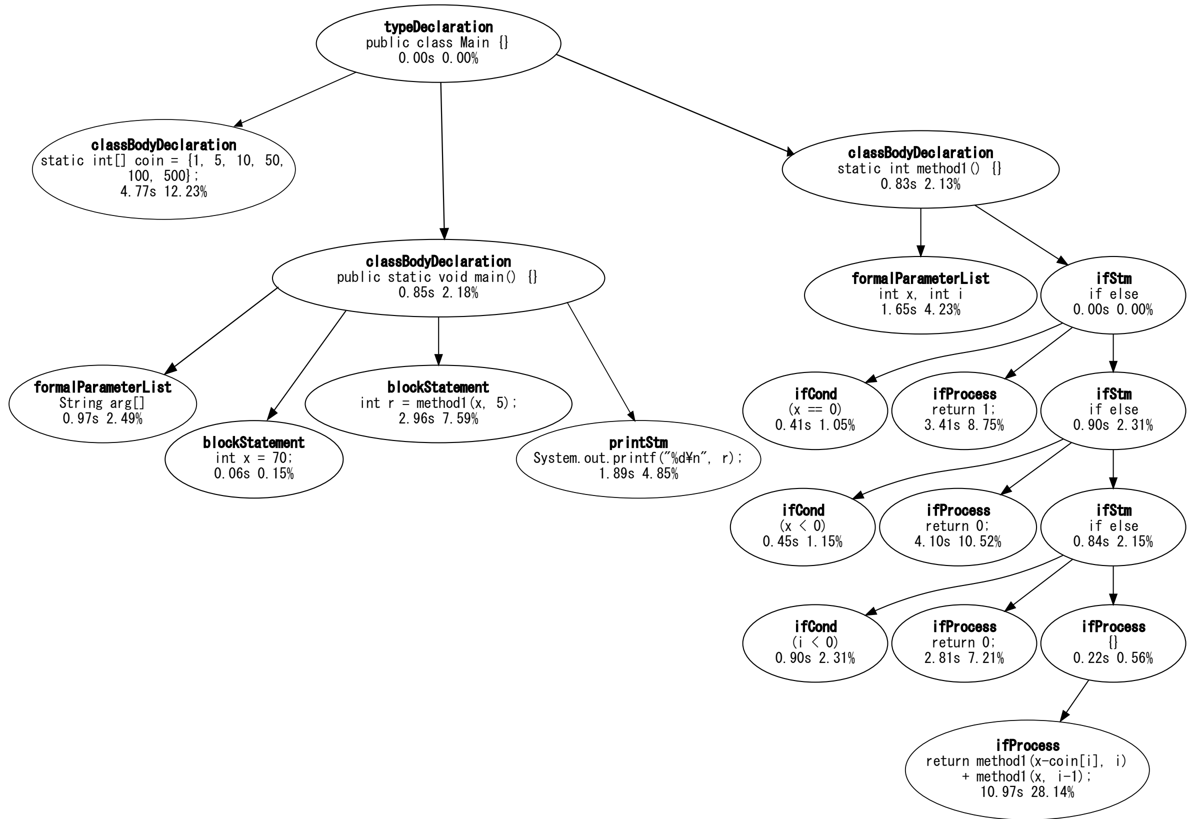

図 4.1　構文要素ごとの注視時間

種の意味ノードを含んでいる．本稿の分析では制御構造と文単位の視線移動に着目するため意味ノードを表 4.2 に示す 21 個に再定義する．再定義によって削除された構文種類の意味ノードに単語ノードが連結していた場合，単語ノードは親ノードに連結され注視時間も親ノードに加算される．図 4.1 に再定義した構文種類を用いて注視時間を構文木に表現した例を示す．各ノードの 1 行目は意味ノードの名前，2 行目はノードに対応する単語群，3 行目はノード単語群に対する注視時間の合計（秒とパーセント）を表す．親ノードの注視時間は子ノードに対する時間を含まない点に注意されたい．

表 4.2　分析で用いる構文種類

構文種類	説明
blockStatement	変数の宣言と初期化
classBodyDeclaration	フィールドとメソッドの宣言
forCond	for の条件式
forInit	for の初期化式
forProcess	for ブロックに属する statement
forStm	単語 for
forUpdate	for の変化式
formalParameterList	メソッドの仮引数
ifCond	if の条件式
ifProcess	if ブロックに属する statement
ifStm	単語 if，else
indent	インデント（空白，タブ）
newLine	空行
printStm	print 文, println 文
space	単語と単語を分離する空白
statement	代入，メソッド呼び出し，return 文
typeDeclaration	クラス宣言
whileCond	while の条件式
whileProcess	while ブロックに属する statement
whileStm	単語 while
whileUpdate	while ブロックに属する"++" または"–"を含む statement

5. 結果と考察

224 件（14 人 × 16 タスク）のプログラム理解タスクのうち，計測誤差が発生した 24 件を除いた 200 件を対象に分析を行った．

図 5.1 にプログラム理解タスクに正解した被験者と不正解だった被験者の構文種類ごとの注視時間の割合を示す．図は被験者が各タスクにおいて着目した単語への注視時間のうち各構文種類に対する注視時間の平均を表す．画面外の注視や瞬きの時間は含まれない．構文種類は 4.2 節で説明した抽象化を行った 21 種類のうち，注視の割合が高い上位 14 件を表示し，それ以外の構文種類をその他としている．

正解と不正解のいずれの場合でも space（単語間の空白）と blockStatement（変数の宣言と初期化）にそれぞれ 16.9%，10.5-10.9% ともっとも多くの視線が集まっているが，2 群の間で差は見られない．space に対する視線割合が大きい理由として，ソースコード中の文字に占める割合が大きいことが挙げられる．プログラム理解に正解したときの視線移動は以下に示す 5 つの構文種類に対する注視の割合が多かった．

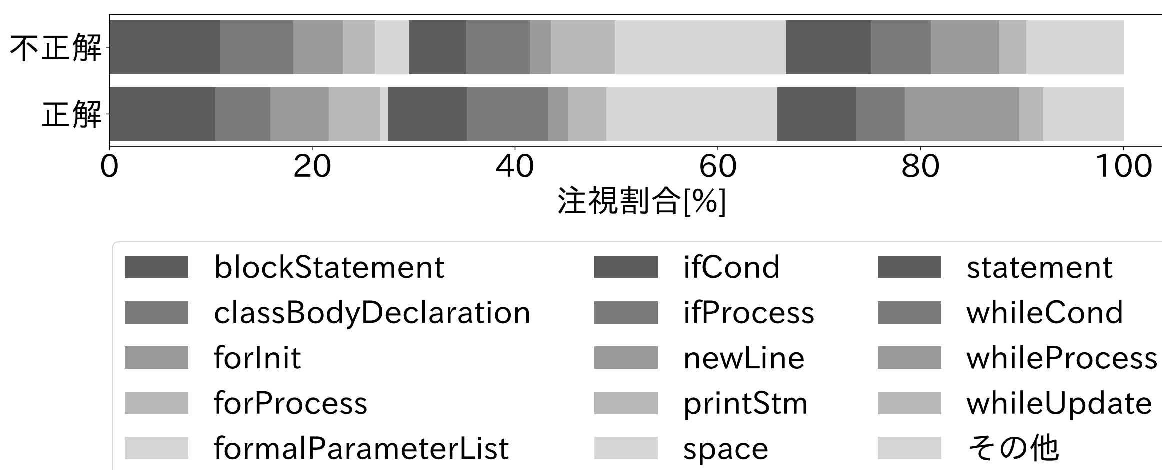

図 5.1　各構文種類に対する注視の割合

- forInit（for の初期化式）
- forProcess（for ブロック内の文）
- ifCond（if の条件式）
- ifProcess（if ブロック内の文）
- whileProcess（while ブロック内の文）

一方で，不正解のときの視線移動は以下に示す 3 つの構文種類に対する注視の割合が多い．

- classBodyDeclaration（フィールドとメソッドの宣言）
- formalParameterList（メソッドの仮引数）
- printStm（print 文）

このうち，以下に示す 3 つの構文種類で正解と不正解の平均値に有意な差 ($p < 0.05$, t 検定) が見られた．

- ifCond（正解 7.8%，不正解 5.6%）
- formalParameterList（正解 0.8%，不正解 3.4%）
- printStm（正解 3.8%，不正解 6.3%）

正解した被験者がより多く着目した構文要素はいずれもプログラムの動作を制御する制御文とその処理内容であり，提示されたソースコードからその処理内容を理解するタスクにおいては重要な要素であった可能性がある．一方で，不正解の被験者がより多く着目した構文要素はクラス宣言やメソッドの仮引数と処理結果の表示を行う print 文であり，処理に対する入力と出力の関係への着目を示唆している．

次に，1 つのタスクにおける正解／不正解間の注視割合の分布について比較を行う．同じ構文要素でもタスクによってプログラム理解の正否に対する重要度が異なるため，タスクごとに注視割合は変化すると考えられる．図 3.2 に示した Task 13 のソースコードは 6 行目で指定した金額になる硬貨の組み合わせ数を求めるプログラムである．Task 13 では 4 人の被験者が理解に成功，10 人が失敗した．図 5.2 に Task 13 における正解と不正解の構文種類ごとの注視時間の割合を示す．正解／不正解のいずれでも return 文に相当する ifProcess や coin 配列の宣言を含む classBodyDeclaration，space に対する注視の割合が大きい．タスクに正解した被験者の注視割合が高い構文要素は以下に示す 2 つの構文種類だった．

- ifProcess（正解 22.0%，不正解 14.7%）
- indent（正解 5.2%，不正解 0.9%）

一方で，不正解の被験者の注視割合が高い構文要素は以下に示す 3 つの構文種類だった．

- space（正解 21.7%，不正解 26.2%）
- ifCond（正解 4.8%，不正解 9.5%）
- formalParameterList（正解 2.9%，不正解 5.8%）

このうち，ifCond と space で正解と不正解の平均値に有意な差 ($p < 0.05$, t 検定) が見られた．

本タスクのプログラムは method1 の再帰呼び出しによって変数 x に格納された金額になる硬貨の組み合わせの数を求める．method1 の 1 つ目の引数が指定金額，2 つ目の引数が coin 配列（図 3.2 2 行目）の探索開始位置を表し，24 行目で再帰的に指定金額から探索対象の硬貨分を減じた金額に対する呼び出し（引数が x-coin[i], i）と，硬貨の探索位置を 1 つ動かした呼び出し（引数が x, i-1）を行う．

タスクに正解した被験者の注視割合が大きい ifProcess は 15,18,21,24-25 行目に対応しており，いずれも各再帰呼び出しの引数に基づいて硬貨の組み合わせとしてカウントするか判断するための重要な箇所と考えられる．反対に不正解の被験者の注視割合が有意に大きい ifCond は 14,17,20 行目の条件式に対応する．個々の要素について正解と不正解の注視割合を比較すると (表 5.1) 正解の被験者は対となる ifCond と ifProcess に対して ifProcess に対する割合が高く，不正解の被験者は同程度か ifCond に対する割合が高い．個々のタスクの内容と構文要素に対する注視割合の関係については，今後の研究で明らかにする必要がある．また，space で正解と不正解の平均値に有意な差が見られた理由として，正解の被験者はソースコードの理解に重要な箇所を集中して注視する一方で，不正解の被験者はソースコード全体を探索的に読んでおり，ソースコード中の文字に占める割合の大きい space に対する注視が増えたと考えられる．

表 5.1　method1 の各要素に対する注視割合

行	要素	コード	正解	不正解
14	ifCond	$x == 0$	3.2%	5.2%
15	ifProcess	return 1	9.9%	5.7%
17	ifCond	$x < 0$	1.4%	4.3%
18	ifProcess	return 0	3.8%	2.6%
20	ifCond	$i < 0$	2.1%	2.3%
21	ifProcess	return 0	5.1%	1.4%
24	ifProcess	return method1(...) ...	14.2%	11.5%

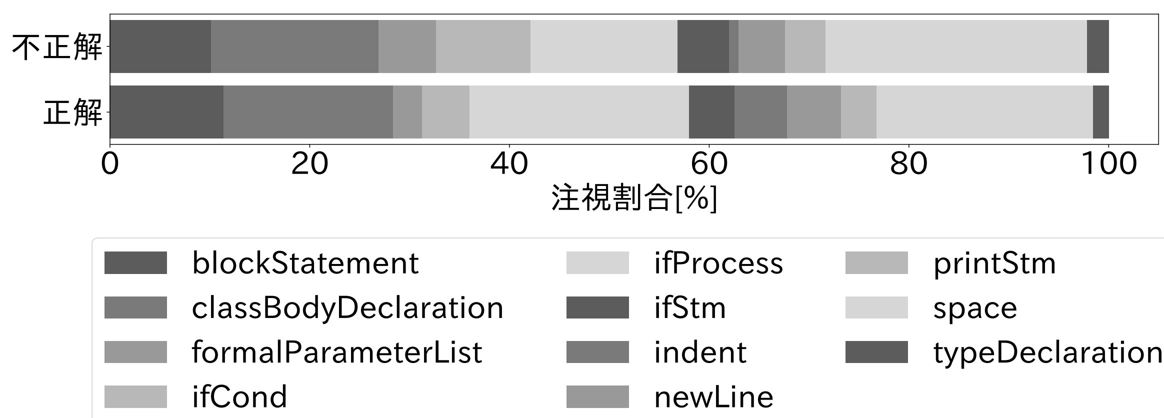

図 5.2　Task 13 における各構文種類に対する注視の割合

6. おわりに

本稿ではディスプレイ上の座標単位で記録されたプログラム理解時の視線移動を構文木のノードに対する遷移に変換する手法を用いて，プログラム理解タスクにおける視線移動を分析した．学生 14 名を被験者とした Java のソースコードの処理内容を理解する実験で得られた 200 件の視線移動のデータを対象に，構文要素ごとの注視時間の割合を求めた．実験の結果，異なるソースコードを対象としたプログラム理解タスクにおける注視割合の特徴として正解したタスクでは if の条件式を有意に多く，メソッドの仮引数と print 文を有意に少なく見ていた．また，再帰を含む複雑なタスクにおいては if 文の条件式よりも if 文内の文をより多く見ていた．本稿の結果はプログラム理解の有無と理解のために重要な構文要素の間に関係があることを示唆しており，今後の研究で因果関係や特定の構文要素を着目するよう指示することでプログラム理解を支援できるか調査する必要がある．

本稿で行った分析は各構文要素に対する注視時間を正解／不正解の被験者間で比較するために，4 章で示した定義に基づいて上位の構文要素に集約した．一方でソースコードを構成する各単語は構文木上で複数の意味と対応づけることができる．例えば，図 3.2 に示したソースコードの 14 行目にある x に対する注視は 14 行目全体から構成される if 文に対する視線としても解釈される．同様に，14 行目を含む if ブロックや method1 に対する視線としても解釈される．そのため，単語に対する注視の時間長や注視数は単語と対応する全ての構文要素に重複して算入され，注視割合の算出が困難になる．本稿では重複して算入しないように各単語の属するもっとも下位の構文要素に対する視線として計算を行った．今後，複数の構文要素に対する注視割合の算出方法を考案し，分析することは本研究の重要な発展である．

5 章で示した Task 13 における注視の割合において，有意差は見られなかったものの，indent に対する注視の割合が正解 (5.2%) と不正解 (0.9%) で差が見られた．indent は各行の左端にある複数のスペース（インデント）として定義される．インデントの有無や長さがソースコードの読みやすさに与える影響は古くから研究されており [12, 13]，現在に

おいても研究が行われている [14]．Task 13 のソースコード（図 3.2）は 2 つのメソッド（main，method1）からなり，method1 は if-elseif-else 文と再帰呼び出しから構成される，他のタスクと比較しても構造が複雑なプログラムである．本実験の結果は Task 13 のソースコードを正しく理解した被験者がプログラム全体の構造に着目した際にインデント部分に視線が停留したことを表している可能性がある．タスクごとの正解／不正解やソースコードの特徴に基づいた注視割合の詳細な分析は今後の課題である．

謝辞

本研究を進めるにあたり，多くの方々のご助力をいただきました．この場を借りて感謝の意を表します．

指導教員の上野秀剛准教授には，論文添削やアドバイス，資料作成，発表練習など様々な面でご指導をいただきました．査読教員の山口賢一教授には，中間発表や研究力向上セミナーで大変貴重な意見を賜りました．厚く御礼申し上げます．

また，発表練習に協力してくれた友人にも深く感謝申し上げます．

参考文献

[1] F. Hauser, S. Schreistter, R. Reuter, J. H. Mottok, H. Gruber, K. Holmqvist, and N. Schorr, “Code reviews in c++: Preliminary results from an eye tracking study,” in *ACM Symposium on Eye Tracking Research and Applications*, ser. ETRA ’20 Short Papers. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3379156.3391980

[2] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D’Mello, “Improving automated source code summarization via an eye-tracking study of programmers,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 390–401. [Online]. Available: https://doi.org/10.1145/2568225.2568247

[3] M. Crosby, and J. Stelovsky, “How do we read algorithms? a case study,” *Computer*, vol. 23, no. 1, pp. 25–35, 1990.

[4] I. Bertram, J. Hong, Y. Huang, W. Weimer, and Z. Sharafi, “Trustworthiness perceptions in code review: An eye-tracking study,” in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3382494.3422164

[5] R. Stein, and S. E. Brennan, “Another person’s eye gaze as a cue in solving programming problems,” in *Proceedings of the 6th International Conference on Multimodal Interfaces*, ser. ICMI ’04. New York, NY, USA: Association for Computing Machinery, 2004, p. 9–15. [Online]. Available:

https://doi.org/10.1145/1027933.1027936

[6] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, “Eye movements in code reading: Relaxing the linear order,” in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC ’15. IEEE Press, 2015, p. 255–265.

[7] N. Peitek, J. Siegmund, and S. Apel, “What drives the reading order of programmers? an eye tracking study,” in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 342–353. [Online]. Available: https://doi.org/10.1145/3387904.3389279

[8] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, “Itrace: Eye tracking infrastructure for development environments,” in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, ser. ETRA ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3204493.3208343

[9] A. Abbad-Andaloussi, T. Sorg, and B. Weber, “Estimating developers’ cognitive load at a fine-grained level using eye-tracking measures,” in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, May 2022, pp. 111–121.

[10] B. Sharif, and N. Mansoor, “Humans in empirical software engineering studies: An experience report,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2022, pp. 1286–1292.

[11] B. Sharif, C. S. Peterson, D. T. Guarnera, C. A. Bryant, Z. Buchanan, V. Zyrianov, and J. I. Maletic, “Practical eye tracking with itrace,” in *Proceedings of the 6th International Workshop on Eye Movements in Programming*, ser. EMIP ’19. IEEE Press, 2019, p. 41–42. [Online]. Available: https://doi.org/10.1109/EMIP.2019.00015

[12] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, “Program indentation and comprehensibility,” *Commun. ACM*, vol. 26, no. 11, p. 861–867, nov 1983. [Online]. Available: https://doi.org/10.1145/182.358437

[13] T. E. Kesler, R. B. Uram, F. Magareh-Abed, A. Fritzsche, C. Amport,

and H. Dunsmore, "The effect of indentation on program comprehension," *International Journal of Man-Machine Studies*, vol. 21, no. 5, pp. 415–428, 1984. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737384800681

[14] D. Oliveira, R. Santos, F. Madeiral, H. Masuhara, and F. Castor, "A systematic literature review on the impact of formatting elements on code legibility," *Journal of Systems and Software*, vol. 203, p. 111728, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001231