社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE

プログラマの視線を用いたレビュープロセスの分析

上野 秀剛† 中道 上† 井垣 宏† 門田 暁人†
中村 匡秀† 松本 健一†

†奈良先端科学技術大学院大学 〒630-0192 奈良県生駒市高山町 8916-5
E-mail: †{hideta-u, noboru-n, hiro-iga, akito-m, masa-n, matumoto}@is.naist.jp

あらまし ソフトウェアレビューにおいて，従来，レビュー手法の違いがバグ検出効率に与える影響について数多く研究されているが，より大きな要因である作業者の個人差（人的要因）についてはほとんど研究されていない．本稿では，コードレビューにおける人的要因を明らかにすることを目的とし，レビュー作業者の視線の動きを分析した．分析の結果，バグ検出効率に影響すると思われる視線の動きのパターンをいくつか明らかにした．

キーワード レビュー，ソースコード，バグ，視線

Analysis of Review Process using Programmers' Eye Movement

Hidetake UWANO† Noboru NAKAMICHI† Hiroshi IGAKI† Akito MONDEN†
Masahide NAKAMURA† Ken-ichi MATSUMOTO†

†Nara Institute of Science and Technology 8916-5 Takayama, Ikoma-shi, Nara, 630-0192 Japan
E-mail: †{hideta-u, noboru-n, hiro-iga, akito-m, masa-n, matumoto}@is.naist.jp

Abstract Although there exists a lot of studies to evaluate the effectiveness (fault detection rate) of software review techniques, human factors in review process, which are larger factor than the review techniques, are rarely studied. This paper aims to clarify the human factors in code review through an analysis of eye movement of review engineers. As a result of analysis, we found some patterns of eye movement that seem to affect the effectiveness of code review.

Keyword Review, Source Code, Bug, Eye Movement

1. はじめに

ソフトウェアの開発コストの削減，および品質の向上のためにはバグを早期に取り除くことが重要である[1]．その手段として，仕様書のレビューやコードレビューが多くの開発現場で実施されている．近年では，Checklist-Based Reading (CBR)や Usage-Based Reading (UBR)などさまざまなレビュー手法が提案されており，ソフトウェア工学研究の一つの流行となっている[5]．これらレビュー手法の違いによるレビュー効率（単位時間あたりのバグ発見率やバグ発見時間など）への影響を分析する反復実験(Replicated Experiment)も数多く行われている[3][8]．

しかし，レビュー効率に与える影響は，レビュー手法の違いよりも個人差（人的要因）のほうが大きく，手法間のレビュー効率の差が 1.2～1.5 倍であるのに対し，個人差は 3 倍以上に及ぶ(2.1 節)．この個人差は，仕様書やプログラムコードの読解能力や理解戦略といった，個人の技術や経験の差であると考えられるが，従来，個人差に着目したレビュー効率の研究はほとんど行われていない．

本稿では，コードレビューの効率に影響する人的要因を明らかにすることを目的として，特に，レビュー作業者の視線の動きに着目した分析を行う．一般に，何らかの作業を遂行しようとする人間の判断や意図は，視線の動きに現れるといわれており[7]，レビューにおける人的要因の分析においても視線の計測が有効であ

ると期待される．

我々は，レビュー作業者の視線の動きを詳細に分析するために，計算機ディスプレイに表示されたプログラムコード上の作業者の視線の位置を，行単位で定量的に計測するシステム Eye Mark Analyzer (EMA)を開発した．EMA は，市販の視線追跡装置（NAC 社製 EMR-NC）の出力と，ディプレイ上のプログラムの位置，フォントの大きさ，スクロール行数などから，被験者が注目するソースコード行番号をリアルタイムに算出し，記録できる．

本稿では，この EMA を用いた分析のアプローチについて紹介するとともに，小規模なプログラムを用いて行ったコードレビューの分析実験について報告する．

以降，レビューと視線それぞれの既存研究について述べた後に（2 章），本研究でのアプローチとツール EMA について述べる(3 章)．そして，EMA を用いた実験とその分析結果について述べ(4，5 章)，最後にまとめと今後の課題について述べる(6 章)．

2. 関連研究

2.1. コードレビュー

レビューとは，ソフトウェア開発過程において作成されるコードや設計仕様等のプロダクトの内容について，プロダクト作成者と他の開発者などの利害関係者間で議論し，バグを取り除くためのプロセスである．本稿では，そのレビューの中でも特にコードレビュー時の作業者の行動に注目する．コードレビューとは，プログラムコード中に含まれるフォールトを発見するためのレビュープロセスである．

従来，いくつかのレビュー手法が提案されている．開発作業の前にテストケースを作成し，それに基づいて成果物をレビューする Test Case Based Reading (TCBR)，チェックリストを用いた CBR，ユースケースに基づく UBR，ユーザやテスト担当者などの観点別に行う Perspective-Based Reading (PBR)などある[3][8]．

これらの手法の違いによるレビュー効率の差を実験的に評価する研究が盛んに行われており，世界中で 100 本を越える論文が発表されている[5]．CBR は，ソフトウェア業界でもっともよく使われている手法であるが，Adhoc 法（特定の手順や制約を設けずに自由にバグを探す方法）と同程度のレビュー効率しかないことが示されている．一方，UBR，TCBR，PBR は，CBR や Adhoc と比べてレビュー効率が良いという結果が数多く示されている[5]．

しかし，レビュー効率に与える影響は，レビュー手法の違いよりも個人差のほうが遥かに大きい．例えば，[8]では，バグ発見率（発見した欠陥数／総欠陥数）の差は，手法間（CBR と UBR の間）では 1.2～1.5 倍程度であるのに対し，CBR，UBR いずれの手法においても個人差は 3 倍以上であった．このように，個人差がレビュー効率に与える影響は非常に大きいにも関わらず，個人差に着目した研究は従来ほとんど行われていない．

2.2. 視線追跡による意図の推測

視線追跡は，被験者の視線とディスプレイとの交点「注視点」を視線計測装置によって計測することにより行われる．注視点を測定する際には，細かい測定誤差や同じ場所を見ていても目が常に動いているという性質(固視微動)の影響を考慮する必要がある．そのため，一定の時間以上，一定の面積の範囲内に注視点が留まっていた場合に，被験者が視ていると判断する領域（停留点）を定義する．本稿における停留点は 50ms 以上，30 ピクセル以内の領域としている．

一般に，何らかの作業を遂行しようとする人間の判断や意図は，視線の動きに現れるといわれており[7]，様々な分野で視線の動きの分析が行われている．[4]では，英語例文検索システムのユーザの視線の移動の軌跡から，ユーザの「迷い」を検出している．

また，[2][6]では，デバッグにおける人的要因を明らかにすることを目的として，プログラムコード上のデバッグ作業者の視線の動きを計測し，作業者がバグ位置を絞り込む様子を分析している．しかし，デバッグは，１つの故障の症状に基づいてその原因となるバグの位置を絞り込む作業であるのに対し，コードレビューは仕様書とプログラムコードの不一致をできるだけ数多く，かつ，網羅的に探す作業であり，必要とされる技術や経験は異なる．そのため，[2][6]で得られた知見は，レビューにほとんど当てはまらない．

3. アプローチ

3.1. 視線計測によるレビュー行動の分析

2.1 節で述べたように，レビューを行う被験者間には，レビュー手法が同一の場合であっても，レビュー効率に大きな差が存在する．この差は，プログラムコードの理解順序や内容等の，被験者ごとのレビュー行動の違いから発生していると考えられる．そこで本稿では，レビュー時の被験者の視線を計測することで，被験者ごとのレビュー行動の分析を行う．

計算機ディスプレイ上の視線の移動の分析は，[2][4][6]などでも行われているが，その粒度は十分に詳細であるとはいえない．[2]では，ソースコードのモジュール（関数）単位の分析であり，[4]は，画面をいくつかに分割した領域単位の分析である．また，[6]では，ディスプレイ上の視線の移動を繰り返し再生するシステムを用いているが，定量的な分析は行っていない．

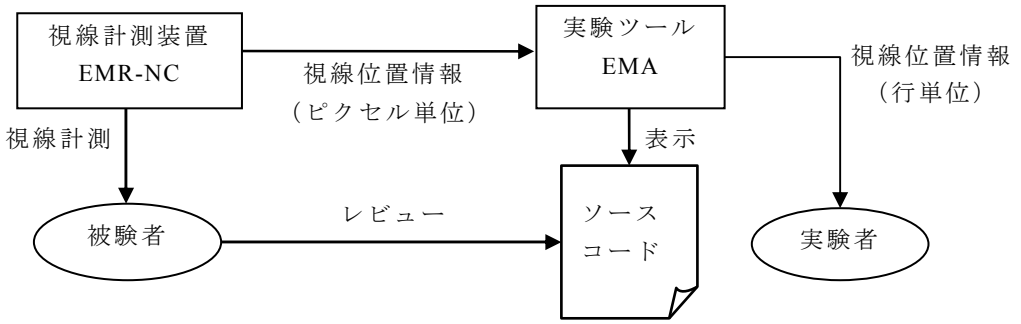

図 1 実験環境の概要

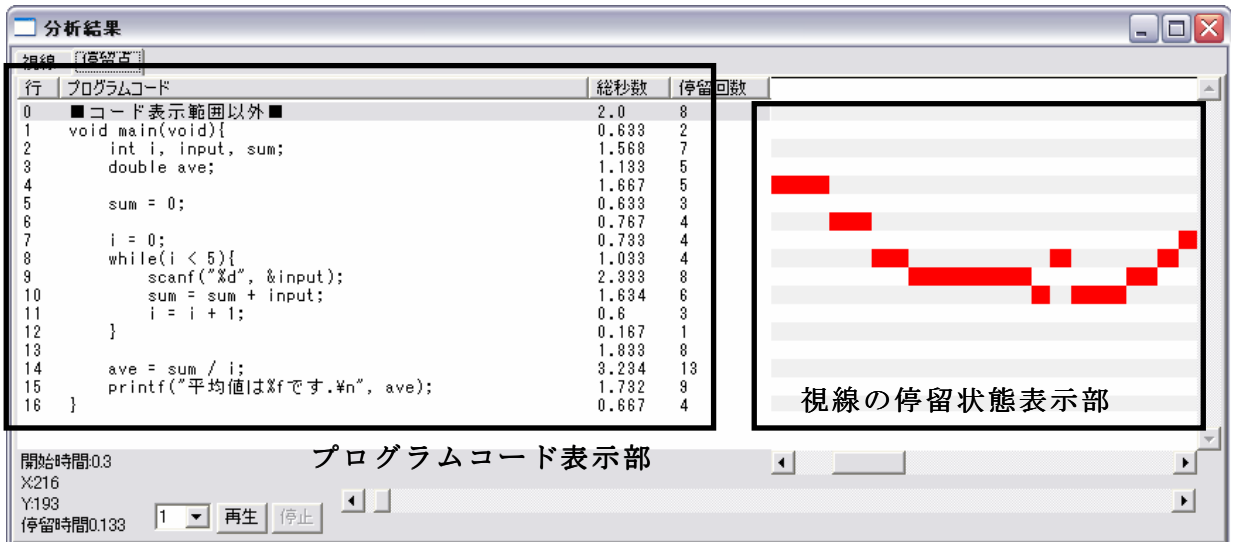

図 2 ＥＭＡの実行画面

レビュー作業は，プログラムコードを 1 行ずつ読み進めたり，特定の行へジャンプするという行動を含むため，プログラムコード上の作業者の視線の位置を，行単位で記録するシステムが必要となる．

次節では，行単位での被験者の視線を計測するために開発した Eye Mark Analyzer について説明する．

3.2. Eye Mark Analyzer

プログラムコード上の視線の動きを分析するためのツール「EMA：Eye Mark Analyzer」を開発した．図 1 に EMA をとりまく実験環境の概要を示す．EMA は Java で開発され，SWT(Standard Widget Kit)による GUI を備えている．また Communication API を用いることで RS-232C インタフェースを介して視線計測装置と通信を行っている．プログラムのクラス数は 34 クラスである．

EMA の主な機能は以下の 3 つである．

(1) 視線計測装置が出力する画面上の座標情報を（スクロール行数を考慮して）プログラムコード上の行番号に対応させる変換機能

(2) 記録した視線の動きを画面上に表示する再生機能

(3) 視線情報の分析のために下記の 3 種類の情報を出力する出力機能

- 被験者が注目した行番号とその時刻の履歴
- 行ごとの注視時間の合計
- 視線の停留が起きた行とその時刻の履歴

図 2 に EMA の実行画面の例を示す．画面左側にレビューの対象となったプログラムコード，右側に各行の停留状態の遷移を示すグラフを表示している．

4. 実験

本章ではレビュー時におけるバグを特定する過程を明らかにするために EMA を用いて行った実験について述べる．

4.1. 対象プログラム

被験者に表示するプログラムは C 言語で書かれた 12～23 行のソースコード 6 個である．各プログラムの仕様は「５つの入力の合計を出力する」のような簡単なものであり，事前に口頭で伝えた．各プログラムにはバグが 1 つだけ埋め込まれており，被験者にはその旨を事前に伝えた．

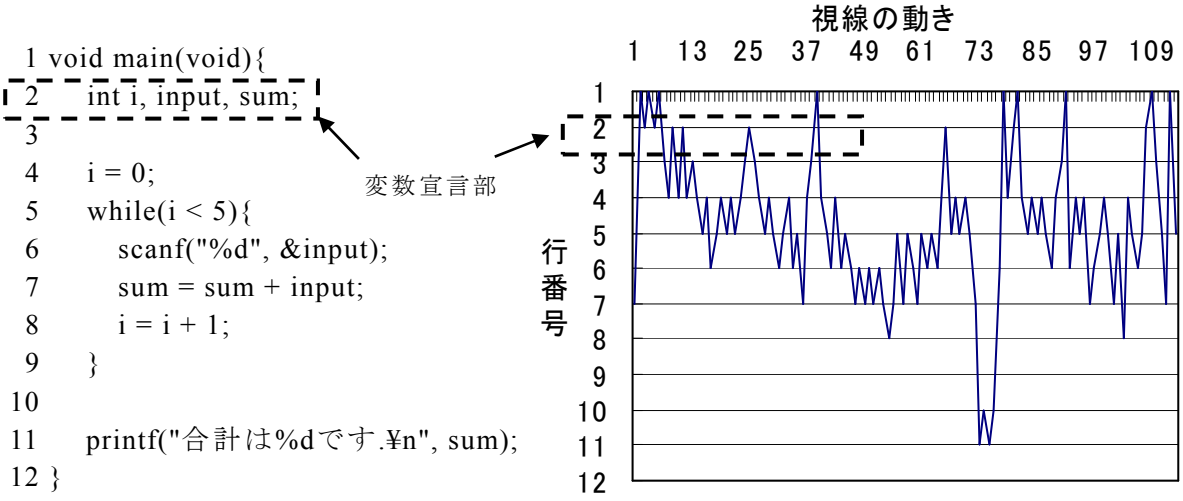

図 3 変数初期化および代入の確認

4.2. レビュー

前述のプログラムに対して被験者にはレビューを行ってもらい，バグを探してもらった．プログラムはディスプレイ上に表示されており，レビュー時の視線を視線計測装置により記録した．被験者は C 言語の仕様およびレビューするプログラムの仕様についてレビュー開始前，またはレビュー中に実験者に質問することができた．レビューの際，チェックリストなどを用いないアドホックレビューを行ってもらった．

レビュー中，被験者はバグを見つけた場合一度作業を中断し，バグについて実験者に口頭で伝えた．実験者からはその正否をその場で伝えられ，バグが正しく発見できていなかった場合，再び作業に復帰してもらった．なお，各問題につき作業時間の合計が 5 分を超えた場合，それ以降は被験者の任意で終了することができるよう設定した．

4.3. インタビュー

レビューが終了するごとに被験者へのインタビューを行った．インタビューは (1) プログラムコードのみを提示したものと，(2) プログラムコードに加えて実験で取得した視線情報を次節で述べる実験ツールで画面上に表示し，それを参照しながらの 2 回行った．それぞれのインタビューではレビュー時に考えていたことについて被験者に尋ねた．特に視線情報を提示しながらのインタビューでは視線情報を元にそのとき何を考えていたかを尋ねた．

4.4. 被験者

実験に参加してもらった被験者は大学院生 5 名でプログラミングの経験が 3～4 年，C 言語の経験が 2～4 年であった．また，全員が少なくとも 1 度はレビュー経験があり，1 人は業務経験者であった．

表1 インタビューで得られた回答の一例

コードのみ	コードと視線情報
• ループの回数を見た • scanf の書き方が正しいかを確認した	• 変数のとり方を確認した • ループの条件式は見ていない • sum の計算が怪しいと感じた • printf は気にしなかった

4.5. 実験環境

被験者の視線情報を取得するための視線計測装置として NAC 社製の非接触型アイカメラ EMR-NC を用いた．またコードの表示には 21 インチの液晶ディスプレイを用い，解像度は 1024×768 とした．

4.6. レビュー結果

全 30 試行のうち有効な視線データが取得できたのは 27 試行であった．またそのうち 3 試行がバグを特定できないまま終了された．インタビューを含めた一人当たりの実験時間は 58～90 分であった．

表 1 に同じプログラムのレビュー後に行ったインタビューで得られた回答例を示す．被験者はコードと視線情報を見たときのほうが，コードだけを見せたときよりもより細かくレビュー中の思考について思い出すことができていた．

5. 分析

実験により取得した視線情報について分析を行った．なお，被験者ごとのレビュー速度の違いを排除するために各被験者がレビューにかかった平均時間で各問題に要した時間を割っている．実験結果を(1)結果全体について，(2)バグ発見が遅い結果について，(3)バグを発見できなかった結果についての 3 点に着目し分析

```
 1 int makeSum(int max){
 2     int i, sum;
 3     sum = 0;
 4
 5     i = 0;
 6     while(i < max){
 7         sum = sum + i;
 8         i = i + 1;
 9     }
10     return sum;
11 }
12
13 void main(void)
14 {
15     int input, sum;
16
17     scanf("%d",&input);
18     sum = makeSum(input);
19     printf("合計は%dです.¥n", sum);
20 }
```

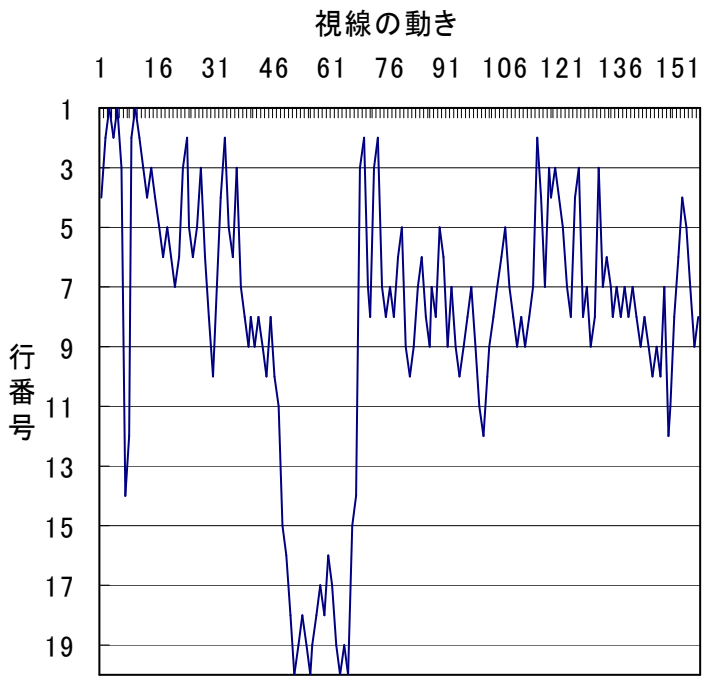

図 4 コードのスキャン

した結果，４つの特徴的な動作を確認することができた．

5.1. 結果全体に見られた動き

特徴１：変数初期化および代入の確認

ある変数が初期値代入以外の処理に始めて利用された際，変数の宣言部，あるいは変数への初期値代入を行っている行に戻るという動作が見られた．図 3 に視線の動きの例を示す．図の左側は被験者に提示したプログラムコード(実験時には行番号は無いものを提示した), 右側は被験者の視線の動きについて時間を圧縮して表示したものである．図 3 の例では 4,6,7 行目で新たな変数が使われた際，2 行目の変数宣言部へと戻っている．有効なデータのうち 56.3%(63／112)が 3 秒以内に宣言部または初期値代入の行を見ていた．また最終的には 92.0%の変数について変数宣言または初期化の行が確認されていた．

特徴２：コードのスキャン

レビューに要した時間の内，初めの 30%の間にコード全体の 72.8%を見ていた．すなわち，レビューの初めの段階でコード全体を眺めた後（以後，このような動きをスキャンと呼ぶ）他の行を見ていた．この動作はレビュー後のインタビューで，多くの被験者が「まず初めに全体を見て・・」と言っている事からも確認された．また，スキャン時には結果を表示する printf 文をほとんど見ていないこともわかった．図 4 にコードをスキャンする視線の例を示す．この例では初めに 2 つある関数のヘッダーを見た後にそれぞれの関数を読んでいる視線の動きが見られた．

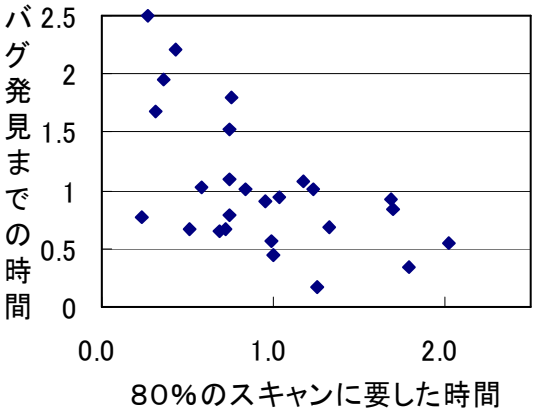

図 5 スキャンの割合

5.2. バグ発見が遅かった結果の動き

特徴３：短いスキャン時間

レビュー時間に占めるスキャンの割合が低かった．図 5 にコードの８０％をスキャンするまでに要した時間と，バグを発見するまでに要した時間の相関を示す．どちらの時間も被験者ごとの読む速度の違いを考慮するために各被験者の平均レビュー時間で各問題のレビュー時間を割っている．平均よりもスキャンの時間が短かった時，バグを発見するまでの時間がのびる傾向が見られた．

5.3. バグを発見できなかった結果の動き

特徴４：特定の行への集中

```
 1 void main(void){
 2       int i, input, sum;
 3       double ave;
 4
 5       sum = 0;
 6
 7       i = 0;
 8       while(i < 5){
 9            scanf("%d", &input);
10            sum = sum + input;
11            i = i + 1;
12       }                              バグのある行
13
14       ave = sum / i;
15       printf("平均値は%fです.¥n", ave);
16 }
```

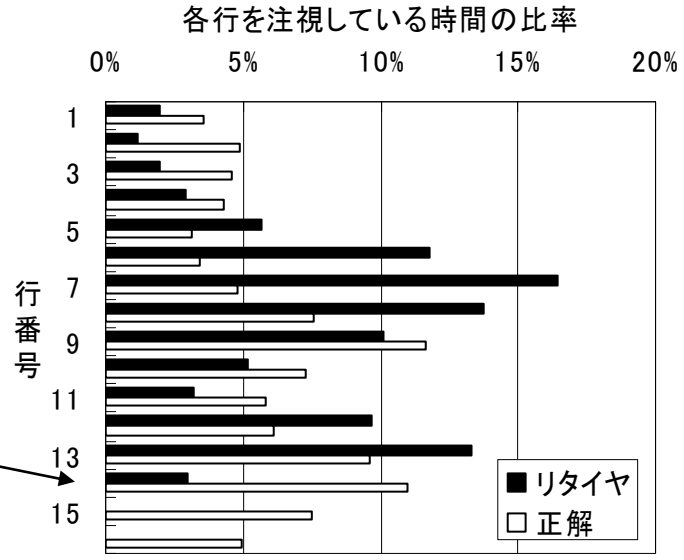

図 6 特定の行への集中

バグを発見することができずにリタイヤした例では視線がバグ以外の行に集中している傾向が見られた．図 6 にバグを発見することができずリタイヤした例とバグを見つけることのできた例それぞれの各行へ視線が留まった時間を示す．この例ではリタイヤした被験者はバグのある 14 行目ではなく 7，8 行目に視線が集中していた．この被験者はレビュー中に while 文が終了した時点で変数 i が 4 になっているため，後の処理に影響が出るという指摘ミスをしていた．7,8 行では変数 i に関する処理を行っているため，視線が集中していたと考えられる．

6. おわりに

本稿ではレビュー実施者がコードレビューによってバグを特定する行動の分析を目的とし，視線計測装置を用いた実験について述べた．

開発したツール EMA を用いた実験の結果，コードレビュー中の視線をレビュー実施者に見せることでレビュー中の思考についてより思い出せることがわかった他，以下の特徴的な視線の動きが明らかになった．

- 変数初期化および代入の確認
- コードのスキャン
- 短いスキャン時間
- 特定の行への集中

多くの変数において初期化および代入の確認という動作が見られた．この際レビュー実施者は変数の型や初期値を記憶していると思われる．したがって変数を注視している際に型や初期値などを変数の付近に表示するといったレビュー支援方法が考えられる．またコード全体を見るスキャンの時間が短い場合，バグ発見までの時間が延びる傾向にあることがわかった．これはスキャンによってプログラム全体の構造を理解することができずにどこに問題があるのかを特定しにくかったものと思われる．

今回の実験ではプログラムの規模が 20 行前後と小さく，より大規模なプログラムのレビューについても同様の結果が得られるか不明である．これについては今後より大きなプログラムコードについても実験を行い明らかにしたい．

文　献

[1] Boehm, B.: Software Engineering Economics, Prentice Hall, 1981.

[2] 内田真司, 工藤英男, 門田暁人, "定期的なインタビューを取り入れたデバッグプロセスの実験と分析", ソフトウェアシンポジウム'98 論文集, pp. 53-58, June 1998.

[3] 野中誠, “設計・ソースコードを対象とした個人レビュー手法の比較実験”, 情報処理学会研究報告, SE-146-4, Nov. 2004.

[4] 高木啓伸, “視線の移動パターンに基づくユーザの迷いの検出”, 情報処理学会論文誌, Vol.41, No.5, May 2000.

[5] Laitenberger, O., Rombach, D. (Session Chairs), “Software Inspections, Reviews & Walkthroughs,” ICSE2002 IMPACT Presentations, 2002.

[6] 門田暁人, 高田義広, 鳥居宏次, "視線追跡装置を用いたデバッグプロセスの観察実験," 信学技報, ソフトウェアサイエンス, SS96-5, pp.1-8, July 1996.

[7] 苧阪良二, 中溝幸夫, 古賀一男, “眼球運動の実験心理学”, 名古屋大学出版社, 1993.

[8] Thelin, T., Andersson, C., Runeson, P., Dzamashvili-Fogelstrom, N, “A replicated experiment of usage-based and checklist-based reading,” Proc. 10th Int’l Symposium on Software Metrics, 2004.