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.

ブレークポイント使用履歴に基づくデバッグ行動の分析

吉村 巧朗† 亀井 靖高† 上野 秀剛†† 門田 暁人† 松本 健一†

† 奈良先端科学技術大学院大学 情報科学研究科 〒 630-0192 奈良県生駒市高山町 8916–5
†† 奈良工業高等専門学校 情報工学科 〒 639-1080 奈良県大和郡山市矢田町 22
E-mail: †{takuro-y,yasuta-k,akito-m,matumoto}@is.naist.jp, ††uwano@info.nara-k.ac.jp

あらまし デバッグ作業は，作業に従事する開発者ごとに効率に大きな違いが見られる．デバッグにおける開発者の行動から効率に影響を与えている要因を明らかにできれば，教育や支援に役立てることができる．そこで本研究では、デバッガを使用したデバッグ行動について分析し，上手な人と下手な人の間にどのような差異が存在するのか明らかにすることを目的とした．そのアプローチとして，多くのデバッガが実装しているブレークポイント機能に着目し，その使用履歴よりプログラマの特徴を分析した． 150 行程度の Java プログラムを題材とした実験の結果，次のような知見が得られた．デバッグの下手な人は，連続した行にブレークポイントを設置する傾向がある．また上手い人には，ブレークポイントを用いた実行を頻繁に行う傾向がある．

キーワード デバッグ，デバッガ，ブレークポイント，プログラム理解

An Analysis on How Programmers Use Breakpoints in Debugging

Takuro YOSHIMURA†, Yasutaka KAMEI†, Hidetake UWANO††, Akito MONDEN†, and Ken-ichi MATSUMOTO†

† Graduate School of Information Science, Nara Institute of Science and Technology Takayama 8916–5, Ikoma, Nara, 630–0192 Japan
†† Infomation Engineering, Nara National Colleage of Technology Yata 22, Yamatokoriyama,Nara, 639–1080 Japan
E-mail: †{takuro-y,yasuta-k,akito-m,matumoto}@is.naist.jp, ††uwano@info.nara-k.ac.jp

Abstract Debugging performance greatly depends on the programmers'debugging skill. If factors concerning the debugging performance are extracted from programmers' actions ,it would be useful for performance improvement. Our goal is to clarify debugging skills of effective use of debuggers. To achieve this goal, we conducted a pilot experiment to analyze programmers' actions in debugging a small Java program (about 150 source lines of code) using Eclipse. As a result of analysis of 6 subjects' debugging actions, we found that expert programmers used more breakpoints than novices. Also, we found that novices tend to set breakpoints one by one to adjacent lines.

Key words debug,debugger,breakpoint,program comprehension

1. はじめに

デバッグ作業はソフトウェア開発の中でも，とりわけ手間と時間のかかる作業であり，多くの人的コストがかかる工程だと言われている. [1] そこでデバッグ作業を効率化するため，デバッガなどデバッグ作業を支援するツールが開発されていたり，デバッグ作業の一部を自動的に行う手法 [2] が提案されている．しかし，デバッグ作業の効率はその作業に従事する開発者による影響が大きいと言われている. [3]

そこで，効果的なデバッグ手法を確立することが求められている．デバッグ手法が確立できれば，それを元に初学者を教育することや，その手法に沿った支援を行うツールを開発することができる．そのために，実際にデバッグ中の開発者の行動を記録し，効果的なデバッグを行う開発者の特徴から，効果的なデバッグ戦略について知見を得ようという研究が行われている．内田ら [4] はデバッグ中のプログラマに対し定期的にインタビューを行い，各モジュールに対するプログラマの意識度合いの移り変わりを調査した．その結果，デバッグの熟練者はバグの無いモジュールを特定する能力に優れていることがわかった．高田ら [3] は，プログラマのキーストロークよりデバッグ

時の行動を計測し，デバッグ行動のモデルを提案している．また，そのモデルに従いプログラマの能力を計測するメトリクスの提案も行っている．これらの研究では，コンパイラとエディタを使用してデバッグを行うことを主眼に分析を行っている．デバッガや統合開発環境等の支援ツールの使用を特に禁止してはいないが，それらツールの影響については明示的に述べられていない．

現在，開発されるソフトウェアの規模や複雑さが増大してきている．このようなソフトウェアでは，欠陥の存在が確認されても，ソースコード中のどこでどのように欠陥が発生しているのか把握することが難しい．そのため，大規模で複雑なソフトウェアのデバッグを行う際には，支援ツールの利用が望ましい．

そこで，本研究では支援ツールの中でもよく使用されるデバッガを使用したデバッグ行動の上手な人と下手な人の間にどのような差異が存在するのか明らかにすることを目的とした．そのためにデバッガを用いたデバッグに関して二つの仮説を立て，特にデバッガの機能の一つであるブレークポイントに着目して分析を行った．ブレークポイントは，プログラムのソースコード中に設置する強制停止コードであり，デバッガを使用する際の起点となる機能であると言える．なぜならプログラマは，ブレークポイントによって処理を停止させている間に，メモリや変数の中身を確認する，その状態に至るまでの呼び出し履歴を参照するなど，デバッガが持つ様々な機能を使用することができる．

150 行程度の Java プログラムと 6 人の被験者を用いて，実験を行った．実験内容は，被験者にプログラムをデバッグさせ，その際のデバッガの使用履歴を計測するものである．デバッガの使用履歴は PSP ツールの一種である Hackystat を用いて計測した．ただし，Hackystat 自体は特にデバッグ行動を分析できるようには作られていない．そこで分析を行うため，Hackystat が記録したデバッガの使用履歴を基に，プログラマの行動を可視化するツールを作成した．このツールは，プログラマの行動を数値的またはグラフにより可視化することができる．

以降，2 章ではデバッガ効率に関する仮説を述べ，3 章では使用履歴の計測および分析に用いたツールについて述べる．4 章では実験の詳細を述べ，実験より得られた使用履歴の分析結果について 5 章で述べる．6 章では本稿のまとめと今後の課題を述べる．

2. デバッグ効率に関する仮説

H1:効率よくデバッグを行うプログラマはブレークポイントを頻繁に利用する

デバッガは，デバッグに役立つであろう様々な情報をプログラマに提示することで，デバッグを支援している．ブレークポイントはデバッガを利用する起点として使われると共に，プログラマがデバッグに必要な情報を取捨選択するためにも使われる．つまりプログラマが頻繁にブレークポイントを利用すれば，その分デバッグに有用な情報を多く獲得することができ，より容易にデバッグが行えることが期待できる．

H2:不自然なブレークポイントの利用はデバッグ効率の向上に寄与しない

前述のとおり，プログラマはブレークポイントを活用して，デバッグに役立つ情報を獲得できる．ただしブレークポイントの活用方法が不自然であると，デバッガによる支援を正常に受けられない可能性がある．よって，そのブレークポイントはプログラマに対してデバッグに有用な情報を与えられず，結果としてデバッグ効率には寄与しないと考える．

3. デバッグ行動の計測方法

前述の仮説の真偽を検証するためには，プログラマがいつ，どのようにデバッガを使用しているか計測する必要がある．そこで本稿では，Hackystat という PSP（Personal Software Process）[5] ツールによりデバッガの使用履歴を計測する．このツールは，プログラマの行動を記録するためのツールであるが，同時にデバッガが使用された時刻とその機能も計測することができる．計測できる機能はある程度制限されるものの，本稿で着目するブレークポイントの他，実行や停止など基本的なデバッガの機能に関する使用履歴を計測できる．ただし，Hackystat 自体はデバッグ行動を分析するためには作られておらず，それ単体では分析が難しい．そのため，Hackystat で計測されたデバッガの使用履歴を基に，デバッグ行動を可視化するツールを作成した．デバッガの使用履歴を計測し分析するまでの流れを図 1 に示す．以降，Hackystat と可視化ツールについて述べる．

3.1 Hackystat

Hackystat は Johnson らによって開発された PSP ツールである [6]．Hackystat は sensor と呼ばれる計測用プラグインをエディタなど各種開発ツールに導入することで，開発者がそのツールを使用する際の行動を計測することができる．例えば名前を A から B へ変更した，ファイルを開いた，プログラムを実行したなどの行動が，その行動を行った時間，用いたツール，対象のファイルなどと共に記録される．後述する実験では，統合開発環境である Eclipse を対象としているため，Eclipse に関して計測できる行動の一部を表 1 に列挙する．なお，表中の Set と Unset については，行動発生のタイミングと共にそれらの行動が行われたソースコード中の行数も計測できる．

3.2 可視化ツール

Hackystat はあくまで PSP ツールである．そのため，特にデバッグなどの行動を分析できるようには作られていない．そこ

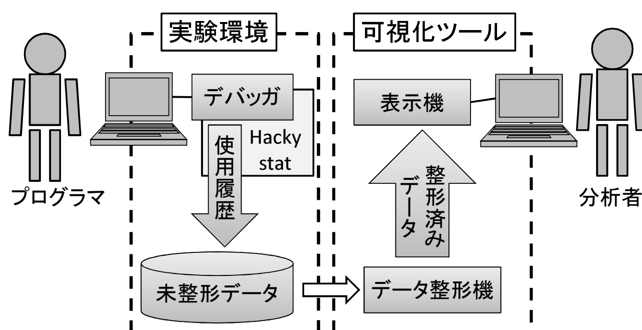

図 1 分析ツール概要図

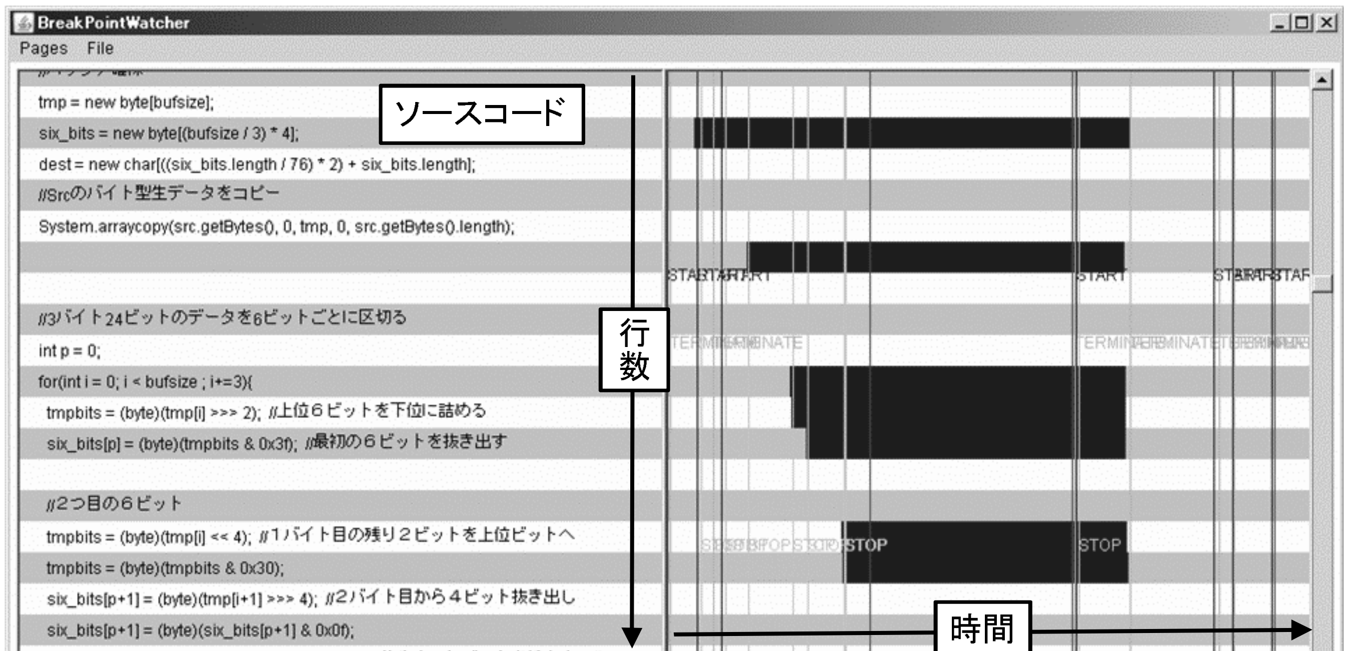

図２　表　示　機

表１　Hackystat で計測できる行動例（Eclipse）

Type	Subtype	Subsubtype	備考
Debug	BreakPoint	Set	ブレークポイントの設置
		Unset	ブレークポイントの除去
	Breakpoint †	-	処理の一時停止
	Start	-	プログラム実行開始
	StepOver	-	一行だけ実行する（関数はそのまま実行する）
	StepInto	-	一行だけ実行する（関数の中に入り実行する）
	Terminate	-	実行を強制終了する

† :誤植ではなく，一時停止はこの表記で記録される

で，Hackystat が計測したデバッガの使用履歴を基に，デバッグ行動を可視化するツールを作成した．

可視化ツールはデータ整形機と表示機からなる．まず，データ整形機が Hackystat のデータベースからデバッガの使用履歴を抽出し，表示機に入力できる形に成形する．そして，表示機が成形されたデバッガの使用履歴を基に，デバッグ行動を可視化する．このとき，仮説 H1 と仮説 H2 を検証するため，次のことを念頭に可視化を行った．

まず，デバッガの使用履歴を一目で把握できるよう，図２のような時系列グラフによりデバッグ行動を可視化した．

この画面において，左半分はデバッグ対象のソースコードを，右半分はデバッグ行動を示すグラフをそれぞれ表示している．グラフ部は，縦軸に行数，横軸に時間を取っており，縦軸の行数はソースコード表示部の行と対応している．グラフ中の横棒はブレークポイントの設置期間を示している．この設置期間は，表１中の Set が計測されてから，Set と同じ行数の Unset が計測されるまでの期間である．また Start や StepOver などその他の行動に関しては，その行動の発生タイミングをグラフ中に縦線で表している．

またグラフによる可視化以外にも，ブレークポイントの設置数やステップ実行回数など，デバッガの機能の使用頻度を表すメトリクスを計測できるようにした．具体的には以下に示すメトリクスを計測することができる．

- ブレークポイント設置総数
- 実行回数
- ステップ実行回数
- 終了回数

計測されたメトリクスは，プログラマ毎に CSV 形式の表で出力されるようにした．

4. 実　　験

この実験の目的は，プログラマによってブレークポイントの使い方にどのような違いがあるか調査することである．そのため，被験者にデバッグを行わせ，その時の行動を Hackystat と動画により計測した．この章では，実験の概要とデバッグを行わせた結果について述べる．

4.1 デバッグ対象プログラム

デバッグ対象には，Java で書かれた 150 行程度のプログラムを用いた．このプログラムは，入力されたデータを Base64 という形式にエンコード，およびデコードするプログラムである．このプログラムは，本稿の著者らにより作成された．実験に当たり，あらかじめプログラムから一通りバグを除去した後，人為的に一つのバグを混入させた．このバグ以外のバグについては，存在したとしてもデバッグ対象外とした．混入させたバグの内容は，入力データの一部が正常にエンコードされないことである．バグは，あるバイト列中にある特定の 6 ビットのデータを抽出する処理に含ませた．抽出対象の 6 ビットを抜き

表2　被　験　者

被験者	プログラミング経験年数	デバッガの使用経験
1	3年	なし
2	5年	なし
3	3年	あり
4	11年	あり
5	8年	なし
6	5年	あり

出すためのマスクが正しくないことがバグの原因である．また被験者には，ソースコードとプログラムの仕様書，及びバグ症状を提示した．バグ症状としては入力データをエンコードし，再びデコードした場合，元の入力データと異なるということを伝えた．

4.2 実験環境

デバッグには統合開発環境であるEclipseを使用した．EclipseにはHackystatのsensorを導入し，Eclipseを使用した行動を記録できるようにした．また，Eclipse以外のツールを使用した場合や，Hackystatでは計測できない詳細な行動や状態を計測可能とするため，作業画面を動画で記録した．

実験には60分という制限時間を設けた．もしこの制限時間を超過した場合は，バグの除去に失敗したとして実験を終了させた．なお，実験ではprint関数など，標準出力に値を出力させる命令の使用は禁止した．デバッガを使う代わりに標準出力に変数名や処理の到達点を表示させ，それらの出力を基にデバッグを行う方法がある．大規模なプログラムならばともかく，今回の実験で使用した規模のプログラムならばこの方法でもデバッグ可能である．しかし，デバッガを使用しないデバッグは実験の趣旨にそぐわないため，前述の措置をとった．

4.3 被験者

実験には情報系の大学院生6名を被験者に用いた．被験者は全員プログラミング経験があり，JavaもしくはC/C++言語が読めることを条件とした．なお，C言語しか経験のない被験者も存在したが，デバッグ対象のプログラム中からJava特有の仕様を極力排除し，C言語の知識のみでも十分読めるようにした．また，Javaしか経験のない被験者は存在しなかった．被験者それぞれのプログラミング経験等は表2に示す．

4.4 実験手順

次の手順で実験を行った．

手順1：実験概要及び使用環境 (Eclipse) の説明　実験の概要と実験環境について説明を行った．これは，デバッガに関する知識や経験，実験環境であるEclipseへの慣れが実験結果に影響する危険性を低減させるために行った．実験環境の説明では，デバッグ用にEclipseを使う方法と，デバッガの機能について説明した．また説明には資料を用い，実験中はその資料を閲覧できるようにした．

手順2：練習　実験環境への慣れが結果に及ぼす影響を抑えるため，被験者に練習用のプログラムをデバッグさせた．練習用プログラムは200行程度のJavaプログラムである．また，前述のデバッグ対象プログラムと同様に，一つのバグを含ませた．練習を開始する際には本実験と同様にプログラムのソースコードと仕様書，バグ症状を被験者に提示し，Eclipseを用いてデバッグを行わせた．デバッグの練習時間は20分とし，練習時間を超過するか，バグの除去を完了した時点で練習を終了とした．バグの除去が完了した場合は被験者に申告してもらい，修正方法が妥当であると確認した上で練習を終了した．なお，デバッグ中の操作や機能に関する質問は練習中に随時受け付けた．

手順3：本実験　プログラムを被験者にデバッグさせた．使用したプログラムや実験環境は前述の通りである．実験開始時に提示するものや終了条件については，デバッグの制限時間が60分であることを除き練習と同様である．実験終了後には被験者にアンケートを行い，プログラミング歴やデバッガ使用経験の有無，実験中の行動について確認した．

表3　デバッグ結果

被験者	バグ除去の成否	除去に要した時間
1	失敗	-
2	失敗	-
3	失敗	-
4	成功	58分
5	成功	39分
6	成功	20分

表4　計測されたメトリクス

被験者	1	2	3	4	5	6
ブレークポイント設置総数	4	9	14	5	2	6
実行回数	11	14	30	15	19	10
終了回数	9	14	30	15	18	10
ステップ実行回数	4	12	5	53	1	12

4.5 デバッグ結果

デバッグを行わせた結果，被験者6名中3名がバグの除去に成功した．以降，バグ除去に成功した被験者を成功者，失敗した被験者を失敗者と呼称する．被験者の成否とバグ除去に要した時間を表3に示す．また，このとき分析ツールにより計測できたメトリクスを表4に示す．

5. 分　　析

実験で得られたデバッグ時の行動履歴や計測されたメトリクスを基に，仮説の検証を行った．以降，それぞれの仮説を検証した結果について述べる．

5.1 H1：効率よくデバッグを行うプログラマはブレークポイントを頻繁に利用する

ブレークポイントは，基本的にプログラムが実行されるタイミングで機能する．よってプログラムの実行回数によりブレークポイントの利用頻度を測る．ただし，デバッグ時間が長ければ実行する機会が多くなるため，実行回数ではデバッグ時間に影響を受けてしまう．そのため，各々の被験者 $S(i|i = 1, 2, \cdots, 6)$ に対し，時間当たりの実行回数である実行頻度 $E_{freq}(i)$ を求めた．しかし，単なる実行頻度ではブレークポイントを利用しない実行も含まれてしまう．そこで，実行頻度 $E_{freq}(i)$ は，ブ

レークポイントを一個以上設置した状態での実行回数 $E_{num}(i)$ とデバッグに要した時間 $T(i)$ により，式 1 のように求めた．

$$E_{freq}(i) = \frac{E_{num}(i)}{T(i)} \quad (1)$$

これにより，ブレークポイントの利用を考慮した実行頻度を求めることができる．各被験者の実行頻度のグラフを図 3 に示す．

図 3 のうち，被験者 1，2，3 が失敗者，被験者 4，5，6 が成功者である．この図から，成功者の実行頻度は高くなる傾向が見て取れる．

被験者 4 は成功者であるが，実行頻度 $E_{freq}(i)$ は失敗者と同程度に低い．ただし被験者 4 は図 4 に示すとおり，他の被験者に比べステップ実行を非常に多用していた．ステップ実行はブレークポイントで処理を停止させた後に使用できる機能の一つである．よって，ステップ実行と通常の実行の重みや意味の違いは考慮すべきではあるが，被験者 4 は図 3 で示されている以上にブレークポイントを活用していたと言える．

また，図 3 の結果を表 3 のデバッグ時間と比べてみると，成功者の中でも早くデバッグを終えている被験者ほど，実行頻度がより高くなる傾向にあることが見て取れる．すなわち仮説の通り，より効率よくデバッグを行う被験者ほど，頻繁にブレークポイントを利用していることになる．

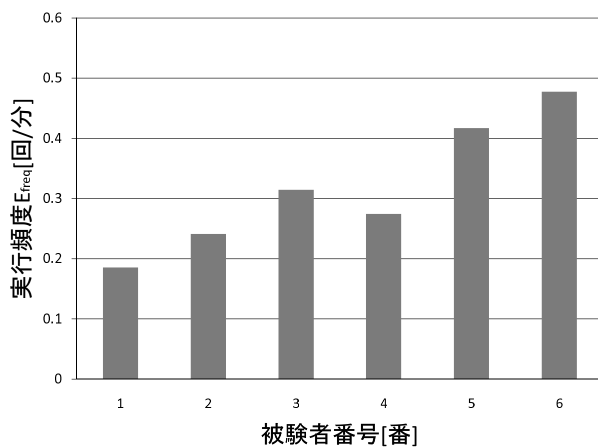

図 3　実 行 頻 度

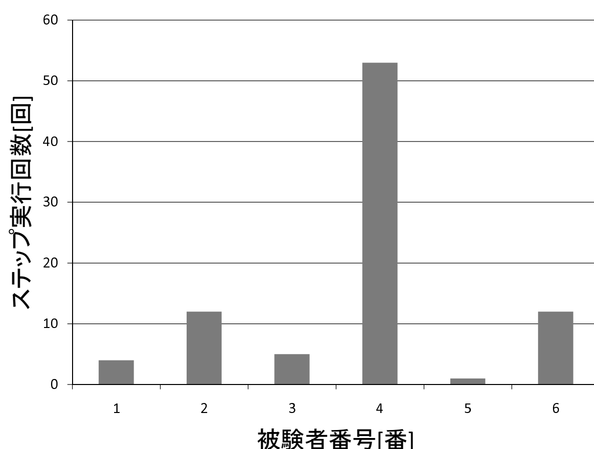

図 4　ステップ実行回数

5．2　H2:不自然なブレークポイントの利用はデバッグ効率の向上に寄与しない

可視化ツールが出力したグラフを基に，成功者と失敗者の行動について比較し，成功者と失敗者の間でデバッガの使い方に差異が見られないか検討した．その結果，失敗者はブレークポイントを連続する行に設置している傾向が見られた．そこですべての被験者について，最大何行にわたりブレークポイントを設置していたかを計測した（図 5）．

図 5 中の被験者 1，2，3 が失敗者であり，被験者 4，5，6 が成功者である．図からわかるとおり，失敗者の方は最大 2〜4 行にわたりブレークポイントを設置している．一方で，成功者はほぼ 1 行ずつしかブレークポイントを設置しておらず，多くても最大 2 行までしか連続で設置していない．一般的に，ブレークポイントはコード中のある一行に設置し，その周辺のコードのふるまいを調査する用途で使用される．そのため連続して設置されることはあまりなく，もし一行ずつプログラムの実行を停止させたいなら，ステップ実行が用いられる．

ここで，実際にどのようにブレークポイントが使われていたか確認するため，最も多く連続してブレークポイントを置いていた被験者 2 について分析する．

まず，被験者 2 がブレークポイントを設置していたソースコードを図 6 に示す．このコードは，ある 1 バイトのデータ中から特定の 6 ビットを抽出する処理である．この行動はデバッグ終盤に行われていたため，バグの位置をある程度特定した上でバグの修正方法を検討していたものと思われる．実験後のインタビューにおいても，被験者 2 はバグの位置がこの辺りであ

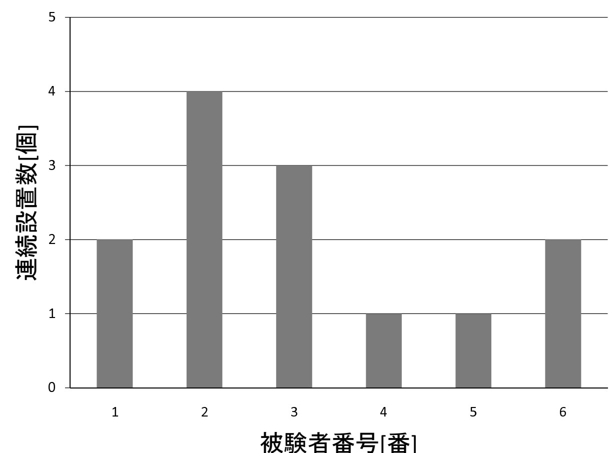

図 5　連続して設置されたブレークポイントの最大数

```
1:tmpbits = (byte)(tmpbits & 0x0c);
2:six_bits[p+2] = (byte)(tmp[i+2] >>> 6);
3:six_bits[p+2] = (byte)(six_bits[p+2] & 0x03);
4:six_bits[p+2] = (byte)(tmpbits | six_bits[p+2]);
```

図 6　ブレークポイントが設置されていたコード（被験者 2）

ると特定し，修正方法を検討していたことも確認できた．すなわち，被験者 2 はこの処理を精査しバグの修正方法を探るため，集中してブレークポイントを設置したのだと推察できる．しかし，この用途でブレークポイントを使用する場合，ブレークポイントを一つだけ設置し，ステップ実行を用いる方法が一般的である．

6. まとめと今後の課題

本研究ではデバッガを使用したデバッグ行動について，上手な人と下手な人の間にどのような差異が存在するのか明らかにすることを目的とした．その上で特にデバッガの機能の一つであるブレークポイントに着目し，それに関する二つの仮説を立てた．仮説を検証するため，150 行程度の Java プログラムを題材とした実験の結果，次のような知見が得られた．デバッグの下手な人は，連続した行にブレークポイントを設置する傾向があった．また上手い人には，ブレークポイントを用いた実行を頻繁に行う傾向があった．ただしこれは，6 人という非常に少ない被験者の下で得られた結果である．そのため，さらに被験者を追加して実験を行い，この特徴が常に見られるか検討する必要がある．また，今回の分析ではステップ実行の影響度合いなど，十分に分析が行われていない要素が残った．そのため，他の特徴が見られないか分析を行う余地がある．

謝　辞

本研究の実験に参加してくださった奈良先端科学技術大学院大学情報科学研究科ソフトウェア工学講座，並びにソフトウェア基礎学講座の皆様に感謝いたします．また，本研究の一部は，文部科学省「次世代 IT 基盤構築のための研究開発」の委託に基づいて行われました．

文　献

[1] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. A general framework for debugging. *IEEE Software*, Vol. 8, No. 3, pp. 14–20, 1991.

[2] Cemal Yilmaz and Clay Williams. An automated model-based debugging approach. In *Proc. IEEE/ACM International Conference on Automated Software Engineering* (ASE2007), pp. 174–183, 2007.

[3] 高田義広, 鳥居宏次. プログラマのデバッグ能力をキーストロークから測定する方法. 電子情報通信学会論文誌 D-I, Vol. J77-D-I, No. 9, pp. 646–655, 1994.

[4] Shinji Uchida, Akito Monden, Hajimu Iida, Kenichi Matsumoto, and Hideo Kudoh. Analysis of program reading process in software debugging based on multiple-view analysis model. In *Proc. 6th Joint Workshop on System Development* (JWSD2003), pp. CD–ROM Edition, 2003. Nagoya.

[5] Watts S. Humphrey. *PSP:A Self-Improvement Process for Software Engineers.* Addison Wesley, 2005.

[6] Philip M.Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, and William E. J. Doane. Beyond the personal software process: metrics collection and analysis for the differently disciplined. In *Proc. 25th International Conference on Software Engineering* (ICSE2003), pp. 641–646, 2003.