社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－SE－157 (5)
2007／9／27

ユーザビリティ改善に向けたフィードバック機構の妥当性検査

矢野 隆弘† 丸山 勝久††

現在，ソフトウェアの品質を評価するいくつかの指標が存在する．しかしながら，品質の中でもソフトウェアの使いやすさ，わかりやすさに関わるユーザビリティは定量的に評価しにくく，その評価は実際のユーザや専門家の手によって行われる．このため，従来の方法では，評価者に大きな負担がかかり，ソフトウェア開発中に気軽に実施することは難しい．本論文では，ユーザビリティの評価項目の中で，特にフィードバックに着目し，それが適切なタイミングで行われているかどうかを自動的に検査する手法を提案する．提案手法では，ウィンドウのビットマップ画像の変化をたえず監視すると同時に，オブジェクト指向ソフトウェアのメソッド呼び出しの履歴を記録する．これらの情報を対応付けることで，ソースコードにおけるフードバックに関する不備を検出し，開発者の行う修正を容易にする．

Validating feedback features for improving usability of a software

Takahiro Yano† and Katsuhisa Maruyama††

Many approaches have been proposed for evaluating various kind of quality characteristics of software. However, it is still difficult to quantitatively measure the usability of software among these characteristics. Thus, end users or experienced engineers in general investigate how usable or friendly the target software is. Unfortunately, this investigation requires much effort. This paper focuses attention on feedbacks which provide information about state of software to its users, and proposes a new mechanism for automatically validating feedbacks activated during the execution of the software. The mechanism monitors apparent changes of bitmap images the software paints and records method calls related to the observed changes. With this mechanism during usability testing, not only experienced engineers but programmers can easily detect problematic code which causes an inappropriate feedback.

1. はじめに

ソフトウェア開発において品質の向上は重要な課題であり，品質を評価するためのいくつもの指標が提案されている．品質の中でも機能性や信頼性などについては比較的定量的に評価しやすい反面，ソフトウェアの使いやすさ，わかりやすさに関わるユーザビリティについては定量的に評価しにくい．このため，ユーザビリティの品質評価では，実際のユーザによる受け入れテストやユーザビリティエンジニアなどの専門家によるあらかじめ用意された評価項目に沿った検査が一般的に行われている．ただし，このような方法では，評価者に大きな負担がかかり，ソフトウェア開発中に気軽に実施することは難しい．ここで，もし品質評価が気軽に実施できると，ソフトウェアの完成間近あるいは完成後に品質の最終的な確認ができるだけでなく，ソフトウェア開発中に品質を随時確認できることになる．よって，気軽に品質を評価できることは，ソフトウェアの品質向上を目指す上で重要な要素である．

ユーザビリティを評価する手法には，ユーザビリティテストとヒューリスティック評価法がある[1]．ユーザビリティテストは，実際のユーザにシステムの完成品，あるいは試作品などを使用してもらうことで，現実に起こりうる問題点を見つける手法である．ヒューリスティック評価法は，専門家によるガイドラインや経験則に基づいて評価を行う手法である．

本論文では，ヒューリスティック評価法で用いられる項目のうち，フィードバックに着目し，それが適切なタイミングで行われているかどうかを自動的に検査する手法を提案する．現状において，フィードバックを人手で検査している理由は，どのような情報をフィードバックとみなすのか(フィードバックの発生)，

† 立命館大学 理工学研究科
Graduate School of Science and Engineering, Ritsumeikan University
†† 立命館大学 情報理工学部
Department of Computer Science, Ritsumeikan University

適切なタイミングでフィードバックが発生しているのか(フィードバックのタイミング)を判断するための系統的な仕組みがないためである．そこで，本手法では，ウィンドウのビットマップ画像の変化をたえず監視し，画像の変化によりフィードバックに関するこれらの特性（発生とタイミング）を捉える．フィードバックを要求しない時間間隔をあらかじめ設定しておき，この時間間隔を過ぎても画像変化がない場合を，フィードバック違反と定義する．

このように画像変化を捉えることで，フィードバック違反を検出することが可能である．しかしながら，実際のソフトウェア開発では，フィードバック違反の検出だけでは不十分であり，それに対応するソースコード上の問題箇所の発見を支援することも重要である．そこで，本手法では，フィードバックの検査と同時に，試験対象ソフトウェアのメソッド呼び出しを記録しておき，それらの履歴を作成する．フィードバックを発生させるべき時刻とメソッド呼び出し履歴の時刻を対応付けることで，フィードバック違反を引き起こしたソースコード上の修正すべきメソッドの位置を絞り込むことができる．

本手法をソフトウェア開発に導入することで，開発者はフィードバックに関するユーザビリティ評価を気軽に実施することができ，さらに問題箇所を容易に修正できる可能性がある．

本論文の構成は次の通りである．2章では，ヒューリスティック評価法について述べる．3章では，フィードバックの検査手法と修正メソッドの抽出手法の詳細を説明する．最後に，4章でまとめと今後の課題を述べる．

2. ヒューリスティック評価法

ヒューリスティック評価法は，ユーザビリティエンジニアなどの専門家がシステムのインタフェースを見て，ガイドラインや経験則 (heuristics) により，その良否を判断する定性的評価である．少数の専門家によってのみ評価できるため，ユーザビリティテストに比べコストが抑えられる．しかし，一般的に用いられているユーザビリティについてのガイドラインをすべて集めると，膨大な数の項目になってしまい評価すること自体が困難になってしまう．そのため，いくつかの項目に限定したガイドラインが用いられる．具体的には Nielsen のユーザビリティ10 原則 (Ten Usability Heuristics)[1),2)]，Shneiderman の 8 つの黄金律 (Eight Golden Rules of Interface Design)[3)]，ISO 9241-10 の対話の原則 (Dialogue principles)[4)] などがある．

ヒューリスティック評価法は，このようなガイドラインに沿って評価を行うが，人手による評価であるため，評価結果が専門家の主観的な考えに依存する可能性がある．また，実際のユーザが気にならないような問題まで問題点として挙げてしまう場合もある．さらに，専門家 1 人では，十分に問題を検出できず複数人の専門家が必要であること，各々の作業時間が 1〜3 時間ほどかかってしまうということもある[1),5)]．

2.1 節では Nielsen のユーザビリティ10 原則の各項目について，2.2 節ではユーザビリティ問題と経験則の関連について述べる．

2.1 ユーザビリティ10 原則

ユーザビリティ10 原則は，ユーザインタフェースを設計する際に，適用できる一般的なガイドラインである．Nielsen は，これを経験則として，10 項目挙げている．以下で各項目について述べる．

(1) **システム状態の視認性 (Visibility of system status)**
システムは今，何をしているのか，あとどのくらいの時間がかかるのかといった情報を，ユーザにフィードバックとして返すべきである．フィードバックを返す場合でも，ユーザがフィードバックを読み取れる程度の速さにしなければならない．

(2) **システムと実世界の調和 (Match between system and the real world)**
インタフェースの用語にはシステム主導のものではなく，ユーザ主導のものでなければならない．たとえシステム内部では識別コードなどで表現しているものでも，ユーザに対して表示する場合はユーザにわかる言語，概念で表示するべきである．

(3) **ユーザコントロールと自由度 (User control and freedom)**
システムに操られているという感覚をユーザに持たせないために，いつでもキャンセルボタンなどで取り消しや終了が行えるべきである．ユーザの意思により，何の悪影響も及ぼさず直前の状態に戻れる必要がある．

(4) **一貫性と標準化 (Consistency and standards)**
同じコマンドや同じ操作によって常に同じ結果が得られるようにするべきで，画面上の同じ情報は常に同じ位置，同じ書式であるべきである．ユーザはできるだけ少ない知識でシステムを扱えるべきである．

(5) **エラーの防止 (Error prevention)**
タイプミスや操作ミスなどの危険を予防するために，入力欄ではなく選択肢を用意することや，重要な操作を行う場合には，確認のためのダイアログを表示することが必要である．

(6) **記憶しなくても，見ればわかるように (Recognition rather than recall)**
ユーザに入力を要求する場合，入力の書式や例を表示することや機能の選択肢を表示する．システムがどのような入力を要求しており，どのような機能があるのかをユーザが記憶しておく必要がないようにするべきである．

(7) **柔軟性と効率性 (Flexibility and efficiency of use)**
システムは一般的なルールさえ理解していれば，誰でも操作できるユーザインタフェースを持つべきであるが，そのシステムの熟練者に対しては，よく使う機能にアクセラレータキーなどのショートカットを設定し，より速く操作ができるようにするべきである．

(8) **美的で最小限のデザイン (Aesthetic and minimalist design)**
新しいシステムのインタフェースはユーザにとって新たに覚えなければならないものであるため，できるだけ誤解しないようにユーザの行う作業に自然に対応している必要がある．ユーザが望む情報は望む場所で得られ，余分な情報は表示しないようにするべきである．

(9) **ユーザによるエラー認識，診断，回復をサポートする (Help users recognize, diagnose, and recover from errors)**
エラーが発生した際には，エラーメッセージだけで具体的な問題の内容が理解でき，問題を解決できるような建設的な提案をするべきである．

(10) **ヘルプとマニュアル (Help and documentation)**
優れた検索機能を持ったオンラインヘルプは，印刷物のマニュアルよりもユーザが欲しい情報を得やすい．ヘルプやドキュメンテーションはインデックスなどの検索手段と問題解決のための具体的な手順が説明されていることが必要である．

2.2 ユーザビリティ問題と経験則の関連

Nielsen の実験[6] は，数多く存在するユーザビリティの経験則のうち，それぞれの経験則が実際のユーザビリティ問題をどの程度の割合だけ網羅しているかを基に順位付けしている．具体的には，既存の 11 個のプロジェクトで発見された合計 82 個の重大なユーザビリティ問題に対して，101 個の経験則のそれぞれがどれだけの問題を説明しているかを調査している．ここで，説明している問題数の割合を網羅率と呼ぶことにする．この 101 個の経験則の中には，2.1 節のユーザビリティ10 原則も含まれている．

表 1 重大なユーザビリティ問題を説明する経験則

	経験則	網羅率
1.	オブジェクトや操作が見て理解できること	22%
2.	ユーザインタフェースの一貫性	18%
3.	適切なフィードバック	17%
4.	実行可能な処理は見てわかること	12%
5.	いつでも処理を中断できること	7%
6.	ユーザになじみのある概念を使うこと	5%
7.	ユーザの入力に応答すること	5%
8.	エラーを予防すること	4%
9.	操作を容易に区別できること	2%
10.	モードのない対話	2%

表 1 はその結果である．例えば，第 1 位の「オブジェクトや操作が見て理解できること」について見ると，網羅率 22%となっており，発見された重大なユーザビリティ問題のうち 22%が，この経験則で説明できるということを示している．また，第 2 位以下の経験則については，より上位の経験則で説明されていない問題のみの割合を示している．すなわち，第 1 位と第 2 位の経験則を合わせて，重大なユーザビリティ問題のうち 40%が説明されていることになる．

ここで，これらの項目を基にユーザビリティを機械的に評価することを考える．「オブジェクトや操作が見て理解できること」や「ユーザインタフェースの一貫性」については，ユーザが判断することであり，機械的に評価することは難しい．しかし，第 3 位の「適切なフィードバック」については，出力装置に出力される情報を機械的に検出することで，フィードバックの有無を判断できる可能性は高い．また，第 7 位の「ユーザの入力に応答すること」もフィードバックに関する問題であり，第 3 位と第 7 位を合わせると網羅率 22%となる．このため，フィードバックに関するユーザビリティ問題を容易に解決することができれば，重大なユーザビリティ問題の解決に対して，低コストで大きな効果を得ることができると考えられる．

3. フィードバックに対する自動検査手法

本手法では，試験対象のソフトウェアにおいてフィードバックが必要な箇所を自動検出する．さらに，それに対応するソースコード上の問題箇所の検出も行う．

3.1節でアプローチ，3.2節で実装するシステムの概要，3.3〜3.5節で実装するシステムの詳細を述べ，その後，3.6節で実行例を示す．

3.1 アプローチ

フィードバックに関するユーザビリティ問題を解決するにあたって，以下の問題点がある．

- どのような情報をフィードバックと見なすか．
- どの程度の時間だけ応答が無ければフィードバックが必要であると判断するか．
- ソースコード上の修正箇所をどのように発見するか．

これらの問題によって，フィードバックの評価にはユーザや専門家の手に頼らざるを得ない．そこで，以下のようなアプローチによって，フィードバックに関するユーザビリティ問題を機械的に扱うことができると考えた．

- フィードバックをウィンドウのビットマップ変化であると定義する．
- フィードバックが必要な時間間隔を明確に定める．
- フィードバックの検査と同時にメソッド呼び出し履歴を記録し，フィードバックの問題箇所とソースコードの問題箇所を対応付ける．

これによって，フィードバックが必要な箇所を自動検出し，それに対応するソースコード上の問題箇所を検出することを達成する．

3.2 システム概要

全体のプロセスは**図 1** のようになる．ここで，想定する対象ソフトウェアは，Windows 上で動作する Java で作られた GUI ソフトウェアである．通常の開発工程によって作られたソフトウェアに対して，フィードバックテストとソースコード修正を行う．フィードバックテストは，対象ソフトウェアの実行，フィードバック検査，メソッド履歴作成の各プロセスから構成される．フィードバック検査では，フィードバックが必要であると考えられる箇所がないか検査する．メソッド履歴作成では，対象の Java ソフトウェアで呼び出されたメソッドの履歴を作成する．対象ソフトウェアを実行中に，フィードバック検査とメソッド履歴作成を同時に行い，それぞれの結果を用いてソースコード修正を行う．ソースコード修正では，修正メソッド抽出で修正するべきメソッド群を抽出し，ソースコード上の修正箇所を絞り込む．メソッド修正では，絞り込まれたメソッド群の中でフィードバックを返すように修正を加える．

図 1 のプロセスのうち，本研究で作成したのは，フィードバック検査，メソッド履歴作成，修正メソッド抽出である．以下でそれぞれの概要を述べる．

- **フィードバック検査**
  対象ソフトウェア上で，何らかの処理が行われている際に，フィードバックがあるかどうかの検査を行う．入力は実行中の Java ソフトウェアに対応するウィンドウのビットマップ画像である．この画像に一定時間変化がない区間があれば，その区間をフィードバックが必要な区間であると判断し，その区間の開始実時間と終了実時間を出力する．
- **メソッド履歴作成**
  実行中の Java ソフトウェアで呼び出されたメソッドの履歴を作成する．入力は Java ソフトウェアで呼び出されたメソッドに関する情報である．履歴にはメソッドが呼び出された時刻とそのメソッドを特定するための情報を出力する．
- **修正メソッド抽出**
  メソッド履歴作成では，対象の Java ソフトウェアが実行を開始してから終了するまでのメソッド呼び出しを記録しているため，修正候補となるメソッドを絞り込む必要がある．フィードバック検査とメソッド履歴作成で得られた結果を基に，修正候補となるメソッドを絞り込む．フィードバックが必要な区間内で呼び出されたメソッド群を修正候補となるメソッドであるとし，開発者に対し修正を提案する．

3.3 フィードバック検査

対象ソフトウェアに対応するウィンドウのビットマップが，変化していればフィードバックがある，変化していなければフィードバックがないと判断する．

3.3.1 節でフィードバックが必要な時間基準について定め，3.3.2 節でフィードバック検査を行う手順について述べる．3.3.3 節では出力例を示す．

3.3.1 フィードバックが必要な時間基準

フィードバックが必要となる処理時間について，Nielsen や Card らは以下のような時間基準を定めている[1),7)]．

- 0.1 秒未満
  ユーザが遅れを感じない限界であり，結果以外のフィードバックは必要ない．
- 0.1 秒から 1 秒未満
  ユーザの考えの流れが妨げられない限界であり，遅れに対する特別なフィードバックは必要ない．
- 2 秒から 10 秒未満
  大きなプログレスバーなどの表示は必要ないが，目立たない程度の処理進行を表すフィードバックが必要な場合がある．

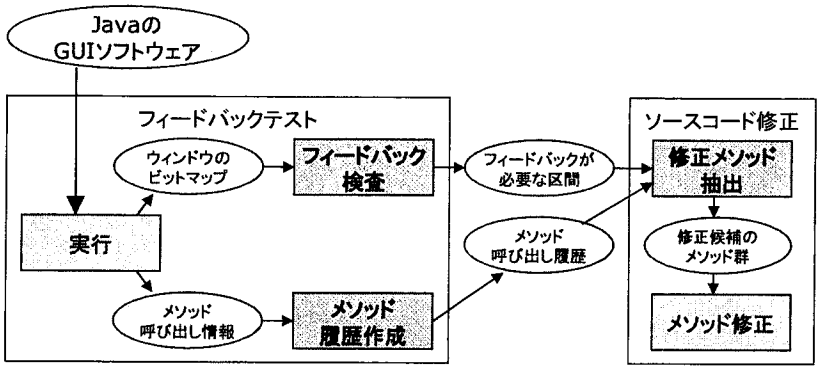

図 1　全体のプロセス

- 10 秒以上
  ユーザがシステムとの対話に集中できる限界であり，いつ処理を終えるか，どの程度進行しているかを表すプログレスバーなどのフィードバックが必要となる．

この基準より，2 秒未満ではフィードバックが必要ではなく，2 秒以上からフィードバックが必要であることから，本研究ではフィードバックが必要な処理時間を 2 秒以上とする．

3.3.2 処理手順

フィードバック検査は以下のような処理手順によって行う．

(1) **アクティブなウィンドウを取得**
本来は実行中の対象ソフトウェアに対応するウィンドウを判別し取得するべきであるが，実装上の都合により，アクティブなウィンドウを取得するにとどまった．すなわち，フィードバック検査を開始した時点でフォーカスを持つウィンドウを対象ソフトウェアに対応するウィンドウと見なす．

(2) **0.1 秒間隔でウィンドウのビットマップを取得**
0.1 秒ごとにフィードバックがあるかどうかを検査する．0.1 秒の時間間隔はユーザが遅れを感じない最長時間であり，フィードバックが必要である区間の時間測定において，0.1 秒未満の誤差は影響がないと考えた．ウィンドウのビットマップは Win32 API によって，デバイス独立ビットマップ (Device Independent Bitmap: DIB) として取得する．

(3) **連続する 2 つのビットマップを比較**
取得したビットマップと直前に取得したビットマップの連続する 2 つのビットマップのピクセルビット列をそれぞれ取得し，各ピクセルビット列が同一のものかどうか比較する．これにより，連続する 2 つのビットマップが変化しているかどうかを判断する．

(4) **連続する 2 つのビットマップが同一である時間を測定**
3.3.1 節より，連続する 2 つのビットマップが同一である時間が 2 秒以上継続すれば，その開始実時間と終了実時間を出力する．

```
...
<15:45:59.843> - <15:46:04.859>
...
```

図 2　フィードバック検査の出力例

3.3.3 出力形式

フィードバック検査を行った結果の出力形式を以下に示す．

<begin-time> - <end-time>

begin-time, *end-time* はそれぞれフィードバックが必要であると判断された区間の開始実時間と終了実時間を示している．この 1 行が対象ソフトウェアにおけるフィードバックが必要な 1 区間と対応する．**図 2** の出力例では，開始実時間が 15 時 45 分 59 秒 843 ミリ秒，終了実時間が 15 時 46 分 04 秒 859 ミリ秒であることを表している．

3.4 メソッド履歴作成

実行中の Java ソフトウェアで呼び出されたメソッドの履歴を作成する．

3.4.1 節で修正候補となるメソッドを絞り込むためのパッケージフィルタ，3.4.2 節でメソッド履歴作成を行う手順について述べる．3.4.3 節では出力例を示す．

3.4.1 パッケージフィルタ

標準クラスライブラリなども含めた，実際に呼び出されたメソッドを全て記録すると膨大な履歴が作成さ

れてしまう．また，本システムではメソッドの修正を目的とするため，標準クラスライブラリのように修正することが適切でないメソッドを記録する必要はない．そのため，修正候補となり得るメソッドを含むパッケージをあらかじめ指定してフィルタリングを行う．

3.4.2 処理手順

メソッド履歴作成は以下のような処理手順によって行う．

(1) **メソッドエントリイベントを検出**
対象となるJavaソフトウェアを実行しているJava仮想マシン上で，メソッド呼び出しが行われる際に生成されるメソッドエントリイベントを検出し，あらかじめ登録したコールバック関数を呼び出す．イベントの取得とコールバック関数の登録にはJVMTI[8]を用い，コールバック関数内で以降の処理を行う．

(2) **メソッド情報を取得**
呼び出されたメソッドのメソッド名，メソッドシグニチャ，定義されているクラスのシグニチャを取得する．

(3) **メソッド呼び出し履歴の出力**
呼び出されたメソッドが出力すべきパッケージに含まれていれば，そのメソッドが呼び出された実時間と共にそのメソッドの名前，シグニチャ，定義されているクラスのシグニチャを出力する．

3.4.3 出力形式

メソッド履歴作成を行った結果の出力形式を以下に示す．

<entry-time> class-signature
method-name method-signature

*entry-time*はメソッドが呼び出された実時間，*class-signature*は呼び出されたメソッドが定義されているクラスのシグニチャ，*method-name*は呼び出されたメソッドの名前，*method-signature*はそのメソッドのシグニチャをそれぞれ表している．**図3**の出力例の1行目では，15時45分59秒125ミリ秒に呼び出された`mouseMoved`メソッドを示しており，そのメソッドのクラスシグニチャは`Lfse/test/SwingTest;`，メソッドシグニチャは`(Ljava/awt/event/MouseEvent;)V`である．クラスシグニチャとメソッドシグニチャの表記は，Javaのクラスファイルフォーマット[9]に従っている．

3.5 修正メソッド抽出

フィードバック検査でフィードバックが必要であると判断された区間の実時間から，その時間内に呼び出されたメソッド群を抽出する．

3.5.1節で修正メソッド抽出を行う手順について述べる．3.5.2節では出力例を示す．

3.5.1 処理手順

修正メソッド抽出は以下のような処理手順によって行う．

(1) **フィードバックが必要な区間の1つを抽出**
フィードバック検査の結果から対象ソフトウェアのフィードバックが必要な区間を1つ抽出し，その実時間範囲を取得する．

(2) **修正候補となるメソッドを抽出**
取得した実時間範囲内で呼び出されているメソッドを作成したメソッド履歴から抽出する．

3.5.2 出力形式

修正メソッド抽出を行った結果の出力形式を以下に示す．

<begin-time> - <end-time
<entry-time> class-signature
method-name method-signature

1行目にフィードバックが必要であると判断された実時間範囲，以降の行に対応するメソッド呼び出し履歴が出力される．フィードバックが必要な実時間範囲の出力形式は，フィードバック検査の出力形式と同様で，メソッド呼び出し履歴の出力形式はメソッド履歴作成の出力形式と同様である．この出力を，全てのフィードバックが必要な区間について出力する．出力例を**図4**に示す．

3.6 実行例

フィードバック検査，メソッド履歴作成，修正メソッド抽出の各プロセスの実行例を**図5**に示す．実行の手順は以下の通りである．

(1) **対象のJavaソフトウェア，メソッド履歴作成プロセスを開始**
Javaソフトウェアの実行と同時に，JVMTIエージェントであるメソッド履歴作成プロセスを開始する．エージェント起動時には，履歴を作成するメソッドが含まれるパッケージを指定する．メソッド履歴作成プロセスでは，メソッドの呼び出し履歴の出力を開始する．

(2) **フィードバック検査プロセスを開始**
フィードバック検査の起動後，アクティブになっているウィンドウを対象のソフトウェアと見なす．そのウィンドウのビットマップ変化を監視し，フィードバックが必要な区間の出力を開始する．

(3) **対象のJavaソフトウェア上で作業を行う**
この作業は自動的に行うことを想定している．

```
...
<15:45:59.125> Lfse/test/SwingTest; mouseMoved(Ljava/awt/event/MouseEvent;)V
<15:45:59.140> Lfse/test/SwingTest; mouseMoved(Ljava/awt/event/MouseEvent;)V
<15:45:59.156> Lfse/test/SwingTest; mouseMoved(Ljava/awt/event/MouseEvent;)V
<15:45:59.171> Lfse/test/SwingTest; mouseMoved(Ljava/awt/event/MouseEvent;)V
<15:45:59.843> Lfse/test/SwingTest; actionPerformed(Ljava/awt/event/ActionEvent;)V
<15:46:04.859> Lfse/test/SwingTest; mouseMoved(Ljava/awt/event/MouseEvent;)V
<15:46:06.593> Lfse/test/SwingTest; mouseMoved(Ljava/awt/event/MouseEvent;)V
...
```

図 3 メソッド履歴作成の出力例

```
<15:45:59.843> - <15:46:04.859>
<15:45:59.843> Lfse/test/SwingTest; actionPerformed(Ljava/awt/event/ActionEvent;)V
<15:46:04.859> Lfse/test/SwingTest; mouseMoved(Ljava/awt/event/MouseEvent;)V
```

図 4 修正メソッド抽出の出力例

作業が終了すれば，対象の Java ソフトウェアを終了することで，フィードバック検査とメソッド履歴作成を終了する．

(4) **修正メソッド抽出プロセスを開始**
対象の Java ソフトウェアにおける作業が終了後，フィードバック検査プロセスとメソッド履歴作成プロセスによって作成されたログを用いて，修正すべきメソッドの抽出を行う．

この一連の実行によって得られた修正すべきメソッド群を用いて，開発者は実際に修正するメソッドを決定し，適切なフィードバックが行われるように修正する．

4. おわりに

本研究では従来，人の手によって評価されていたフィードバックに関するユーザビリティ問題の解決について部分的に自動化する手法について示した．本手法の実装システムを用いることによって，ユーザビリティの専門家でなくても容易にフィードバックに関する問題箇所を絞り込むことができる．これにより，専門家によるユーザビリティ評価を行う範囲を狭めることができ，評価を行うための人的，時間的コストを削減することができる．また，コスト削減によってソフトウェア開発中でも気軽に評価を行い，開発するソフトウェアの品質を向上させるための足がかりとなる．

今後の課題としては，フィードバックの有効性の判断と，開発者に対する修正のためのより良い情報の提示の 2 点が挙げられる．1 点目について，現状のシステムではフィードバックがあるかどうかのみを検査しており，フィードバックがあると判断されても，そのフィードバックが有効なものであるかどうかまで判断できていない．2 点目について，開発者に対する修正の提案では，修正候補となるメソッド群を提示しているが，その情報が本当に開発者にとって必要な情報であるかどうか判断することができない．フィードバックが必要であると判断された箇所が特定できれば，開発者はソースコード上の問題箇所をシステムに提示されるまでもなく同程度の範囲に絞り込むことができるかもしれない．そのため，問題箇所をより特定するために開発者に対して提示する情報としてより有用なものがないか検討する必要がある．

謝辞 本論文の執筆において，有益なる御教授，御示唆をいただきました立命館大学情報理工学部 山本哲男講師に深く感謝の意を表します．また，その他さまざまな面で御助言をいただきました立命館大学情報理工学部ソフトウェア基礎技術研究室の皆様に深く感謝の意を表します．

参考文献

1) ヤコブ・ニールセン：ユーザビリティエンジニアリング原論，東京電機大学出版局 (2002).
2) Nielsen, J.: Ten Usability Heuristics, `http://www.useit.com/papers/heuristic/heuristic_list.html` (2005).
3) Shneiderman, B. and Plaisant, C.: *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison Wesley (2004).
4) ISO 9241-10: Ergonomics - Office work with visual display terminals (VDTs) - Dialogue Principles (1996).
5) Constantine, L.L. and Lockwood, L. A.D.: 使

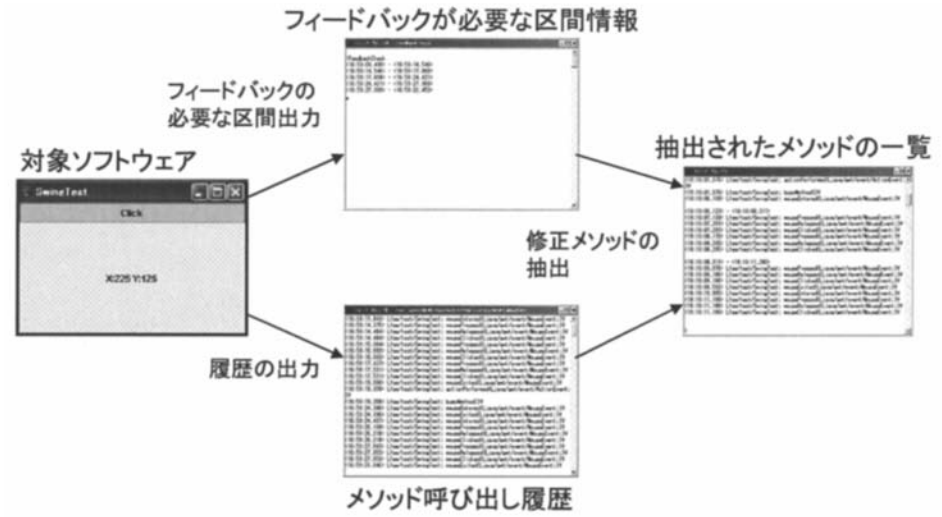

図 5 動作の様子

いやすいソフトウェア—より良いユーザインタフェースの設計を目指して—, 共立出版 (2005).

6) Nielsen, J.: Enhancing the explanatory power of usability heuristics, *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, ACM Press, pp.152–158 (1994).

7) Card, S.K., Robertson, G.G. and Mackinlay, J.D.: The information visualizer, an information workspace, *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, ACM Press, pp.181–186 (1991).

8) Sun Microsystems, Inc.: Java Virtual Machine Tool Interface (JVMTI), `http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jvmti/index.html` (2004).

9) Lindholm, T. and Yellin, F.: The Java Virtual Machine Specification Second Edition, `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html` (1999).