情報処理学会論文誌　プログラミング　Vol. 2　No. 3　1–19 (July 2009)

ORCA：実行トレースと画面変化の対応を可視化することによるGUIプログラム理解支援システム

佐藤竜也[†1]　志築文太郎[†1]　田中二郎[†1]

ソフトウェア開発者が，既存のプログラムに対して保守やコード再利用，機能追加をするためには，プログラムの特定の機能を理解する必要がある．GUI プログラムの場合，機能は GUI への操作によって引き起こされ，操作に応じてソースコードを実行し画面表示を更新する．そのため GUI プログラムの機能を理解するためには，GUI への操作と操作によって引き起こされた実行されたソースコードおよび画面変化を対応付けることが必要である．しかし，GUI への操作とソースコード上に分散している実行部分と画面変化を容易に結び付ける手段が存在しないことから，この対応付けを行うことが困難である．我々はこの対応付けを行えるようにするために，GUI プログラムの実行情報を可視化する．提案する可視化手法では，GUI への操作に対する実行のトレース情報をソースコード全体の縮小表示の上に重畳表示し，操作前後での画面のスナップショットとあわせて表示する．このような表示を行うことで，1 画面上で操作と画面変化，実行されたソースコードを対応付けることが可能になる．我々は提案手法に従った Java の GUI プログラムの理解支援システム ORCA を作成した．ORCA では提案手法に従った表示に加えて，実行のトレースを順に追うことができるように，基本表示上へのポップアップ表示と基本表示上の特定の部分を魚眼ズームするズーミング表示を提供する．本論文では，提案手法と ORCA について述べ，Java の知識を持つ学生を対象にして行った ORCA の評価実験の結果をもとに手法の有効性を示す．

ORCA: A Support System for Understanding GUI Programs by Visualizing Execution Traces Synchronized with Screen Transitions

Tatsuya Sato,[†1] Buntarou Shizuki[†1] and Jiro Tanaka[†1]

To maintain, reuse, and add a functionality of an existing program, the software developer needs to understand a specific functionality in the program. In the case of a GUI functionality, according to operations to the GUI, the source code is executed and then the screen is updated. Therefore, to understand a GUI functionality, the developer needs to comprehend the correspondence between the operation, the screen change and the code executed by the operation. However, it is difficult to comprehend the correspondence. We propose a visualization technique which represents the correspondence between the screens before and after the operation, as well as the traces of the source code executed by the operation. They are represented as highlights which are superimposed on the entire source code. The representation enables the developer to comprehend a correspondence between an operation, screen change and code execution as one picture. We developed ORCA which is based on our visualization technique. ORCA is the visualization system for understanding Java GUI programs. To help the developer to trace execution, the system provides pop-up representation which puts fragments of the code on the basic representation, and zooming representation which enlarges the focused part of the basic representation.

In this paper, we describe our visualization technique and ORCA, and then show effectiveness of ORCA based on the result of user study using ORCA.

1. はじめに

既存のプログラムを保守，再利用，拡張するには，対象プログラムを理解する必要がある．プログラムの特定の機能を理解するためには，その機能の入力と出力を明らかにしたうえで，その実装を把握する必要がある．プログラムが製造される際，仕様書や設計書等の文書が作成されていれば，それらは理解の手助けとなる．しかし，ユーザの入力によって動的に振る舞うプログラムの場合には，その動的な挙動を，静的な媒体である文書のみから把握することは困難である．

Graphical User Interface を持つプログラム（以降，GUI プログラム）はその代表例である．GUI プログラムは，マウスやキーボードによる操作に応じて，動的に画面表示を更新する．この特徴から，GUI プログラムを理解するためには，以下の作業が必要である．

操作・実行されたソースコード・画面変化の対応付け　GUI プログラムでは，GUI への操作が行われるたびにイベントが発生し，ソースコード中に記述されたイベントハンドラがそのイベントに対する処理を行う．また多くの GUI プログラムの機能は表示の更新をともなう．表示の更新はイベントハンドラの一処理である．このことから，GUI へ

†1 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba

の入力および画面出力とそれぞれの間を取り持つソースコードの 3 者の間には密接な関係があることが分かる．たとえば，ドローイングツールプログラムにおいて，オブジェクト追加ボタンを押すとキャンバス上に図形オブジェクトが配置されるとする．ボタンを押すという入力がされたとき，まず入力に対応するイベントハンドラが呼ばれ，その中から図形を描画する処理を呼び出すことになる．そのためボタン押下から図形描画までの実行の流れを理解する際には，まず入力操作と出力画面をとらえたうえで，入出力それぞれに対応して実行されるソースコードを把握する必要がある．

複数ファイル上に点在するソースコード実行部分の抽出　一般的に，GUI ツールキットはモデル・ビュー・コントローラ（MVC）アーキテクチャに基づいて作られる．そのため，GUI ツールキットを用いた GUI プログラムも MVC の役割ごとに分けて実装される．さらに，GUI プログラムはオブジェクト指向に従っている．それゆえ，それら MVC の役割は複数のファイルに分けて記述されることが多い．たとえば，ドローイングツールプログラムでは「描画処理部」，「図形オブジェクトモデル」，「ユーザ操作処理部」という役割がファイルに分けて実装され，図形の描画機能等はそれぞれの実装部分を横断的に利用する．そのため機能の理解時には，その機能を実装しているソースコード部分を複数ファイルから抽出する必要が発生する．この作業は先に述べたソースコードの把握を行いにくくする．

複数イベントをともなう機能の実装部分の把握　GUI プログラムの機能には，マウスのドラッグのように連続した複数のイベント（以降，複数イベント）をともなうものがある．複数イベントをともなう機能には，一連のイベントすべてが関係する場合と，一連のイベント中に発生する特定のイベントのみが関係する場合の 2 通りがある．ドローイングツールを題材にそれぞれの例を考える．前者の例としては，描画された図形オブジェクトをドラッグすることによるオブジェクトの移動やサイズ変更が考えられる．このような機能の理解時においては，マウスプレス，ドラッグ，リリースといったイベントの処理をイベントの発行の順序どおりに把握する必要がある．また後者の例としては図形オブジェクトのスナッピング機能が考えられる．スナッピング機能とは，操作中の図形がグリッドや他の図形に吸い寄せられる動作のことであり，ユーザが図形の整列を行いたい場合に有効である．一般に，スナッピング機能は，マウス操作中にオブジェクトがある領域に入る等の一定の条件を満たしたときに発生する．ゆえに，この機能を理解するためには，連続的なマウス操作中の画面変化と，ソースコード中のその機能に関連した実装部分を抽出しなければならない．

プログラムの動的な挙動を理解するための手段として，デバッガおよび動的実行可視化ツール等の，動的解析ツールがある．しかし，先にあげたような GUI プログラム特有の理解作業を既存の動的解析ツールで行うことは困難である．

デバッガのブレークポイント機能やステップトレース機能を利用すると，実行を段階的にトレースすることが可能である．これによって，操作とソースコード実行部分の対応関係の確認をある程度行うことができる．しかし，デバッガではプログラムへの操作を中断しながら解析を行わなければならないため，ドラッグ中に同一のイベントが繰り返し発行される中で表示が変化するような場面での解析は困難である．

またプログラム中に printf 文等のデバッグコードを挿入するという方法がある．一般にデバッグコードはソースコード内に直接記述する必要があるが，アスペクト指向[1] を利用すればデバッグコードとソースコードを別々に記述することも可能である．デバッグコードを埋め込むことで，プログラムの実行を中断することなく実行をトレースすることができるが，ソースコード実行部分の抽出作業に関する支援はない．

プログラムの動的実行情報の可視化を行う研究もされている．まず GUI プログラムを対象とした実行情報の可視化の研究として柏村ら[2] や久永ら[3] のものがある．これらは GUI への操作と画面，実行されたソースコードの対応付けを支援するが，ソースコード実行部分の抽出作業に関する支援が不十分である．またソースコード実行部分の可視化を行う研究として Seesoft [4] や Reiss らの研究[5),6] 等があるが，これらの手法では GUI への操作や画面変化の様子と実行されたソースコードを対応付けることは困難である．

そこで我々は，GUI への操作，操作によって引き起こされたソースコード実行部分，および画面変化の 3 者を 1 画面上で可視化することによって，上記 3 つの GUI プログラム特有の理解作業を支援する．今回，理解の対象とするプログラムは Java を使って書かれたものとした．これは Java が GUI プログラムの記述言語として広く普及しており，オブジェクト指向言語を使った GUI プログラミングに則っているためである．

本論文ではまず，我々が研究を行ってきた理解支援システム ORCA [7)–9)] の設計と実装について述べ，新たに行った ORCA の被験者による評価実験の結果を元に手法の有効性を示す．最後に関連研究と議論をもとに本研究についてまとめる．

2. 可視化手法の設計

1 章に述べた GUI プログラム理解に必要な 3 つの作業を支援するために以下のような可視化を行う．

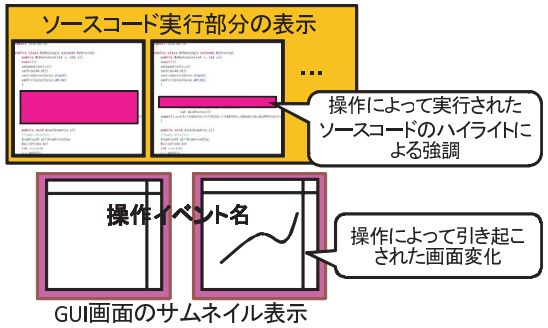

図 1　操作・実行されたソースコード・画面変化の対応の提示
Fig. 1　Representation of correspondence between executed source code parts and the screen change.

2.1　操作・実行されたソースコード・画面変化の対応の提示

GUI への操作・操作によって実行されたソースコード・画面変化の 3 者の対応関係の把握を支援するために，これら 3 つの実行情報をあわせて可視化する．**図 1** に可視化の模式図を示す．まず可視化の単位を操作ごととし，実行情報を操作ごとにまとめて閲覧可能にする．このために，GUI への操作が行われたときに，各操作の前と，イベントから始まる一連の関数呼び出しが終了した後の画面をキャプチャし，キャプチャした画面のサムネイルのペアを並べて表示する（図 1 下部）．図中に示すように，操作のトリガとなった関数名を操作イベント名として，サムネイルペアの上に重畳表示する．サムネイルペアとあわせて，操作によって実行されたソースコードを強調表示する（図 1 上部）．

上記の可視化は，理解の対象となる GUI プログラムを実行しながら行う．すなわち，対象プログラムの GUI へ操作が行われると，その操作に対する実行情報を即座に可視化する．ひととおりの更新が終了した後は可視化結果を実行情報を示す静的なグラフとして閲覧することができる．さらに，それぞれのイベントの可視化結果は履歴として保存し，再閲覧可能にする．ユーザが操作の流れを追いやすくし，かつ必要な部分の可視化結果を再閲覧可能とするために，対象プログラムの起動時から現在までの画面のサムネイルを時系列順に並べて提示する．ユーザはサムネイルを選択すれば，その時点での可視化結果を閲覧することができる．このように，可視化結果の履歴を保存しアクセス可能にすることによって，特に複数イベントをともなう機能の理解作業を支援する．

2.2　点在するソースコード実行部分の包括的表現

複数ファイル上に点在するソースコード実行部分の抽出を支援するために，ソースコード

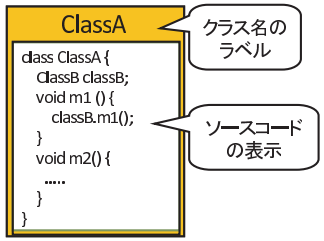

図 2　クラス定義
Fig. 2　Representation of a class definition.

実行部分を包括的に眺めることができるような表現で可視化する．それを実現するために，プログラムの可視化手法の 1 つであるソースコードベースの可視化を用いる．この手法では，UML [⋆1] に代表されるような図による記述をするのではなく，ソースコードそのものを活かした表示を行う．

ソースコードベースでの可視化システムの代表的な研究として Seesoft [4] がある．Seesoft では，ソースコード全体を 1 画面上に縮小表示する．その際，ソースコードはファイルごとに区切って表示し，ソースコードの各行は 1 本の線として表現される．ソースコードの更新履歴や実行履歴の統計的な解析結果を，解析結果に基づいて線を色分けすることによってユーザに提示する．

本手法においても Seesoft と同様にソースコード全体を 1 画面上に表示する可視化表現を用いる．これにより，ソースコードの包括的な表現の中から実行部分を抽出できるようにする．さらに，本手法でもまたソースコードをファイルごとに区切って表示する．ただし，本研究では対象プログラムが Java のコーディングの慣例に従ってなされていることを前提として，各ファイルに 1 個のクラス定義があるものとする．**図 2** にクラス定義の表現を示す．クラス定義はクラス名（ファイル名）のラベルとソースコードそのものによって表現する．この表現を用いて，プログラム内のすべてのクラス定義を表示領域の大きさに収まるように 1 画面上に敷き詰めて縮小表示する．

このソースコード全体の縮小表示に対して，以下のような実行情報をあわせて可視化することによって，実行情報の包括的な閲覧を可能とする．

実行されたソースコード行　GUI プログラムの機能を実現している部分は，GUI への入力

⋆1 http://uml.org/

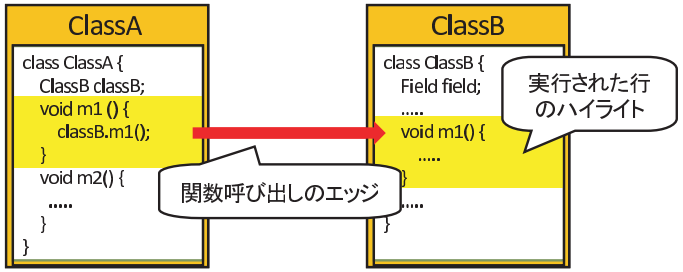

図 3　動的解析情報の表現
Fig. 3　Representation of dynamically analyzed information.

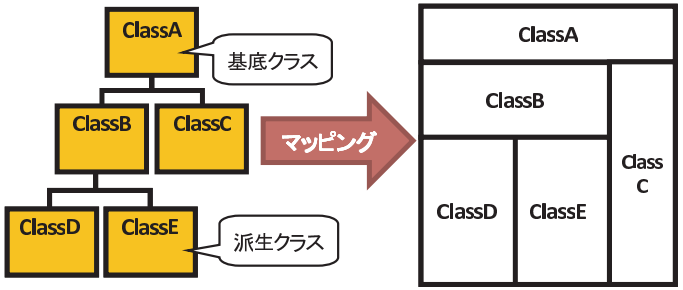

図 4　クラス階層表現
Fig. 4　Representation of class hierarchy.

により実行されたソースコード行と部分的に一致する．そのため，実行されたソースコード行を把握することは重要である．実行されたソースコード行は，**図 3** のように，クラス定義内の実行されたソースコード行を色づけによって強調して表示する．

関数呼び出し　実行されたソースコード行は複数のクラスの関数内に点在しており，それらが互いに呼び出しあっている．そのため，関数呼び出し情報を把握することによって，実行時呼び出しの順序関係と結び付きを知ることができる．

関数呼び出しを示すために，呼び出された関数間に矢印によるエッジ付けを行う．図 3 に例を示す．図中では **ClassA** の **m1()** から **ClassB** の **m1()** への関数呼び出しが行われる様子を可視化している．このように関数呼び出しがあったときには，図中の矢印のようにエッジ付けする．

クラス階層　GUI プログラムを構成する GUI 部品や描画対象は，継承を使って実装されることが多い．たとえば，ドローイングツールで描画対象である図形オブジェクトを複数種類定義する場合，図形に共通する属性や動作を持つ抽象クラスを作り，このクラスを継承することで図形オブジェクトを定義する．このような状況で GUI プログラムを理解するためには，クラス階層を把握することが望まれる．

クラス階層を表現するために，各クラス定義をクラスの親子関係を表す木構造に基づいて配置する．多重継承のように木構造で表現しきれない構造は，クラス表現間にエッジを描いて補う．前述のとおり，ソースコード全体は 1 画面上に収めて表示する．そのため，表示領域に対して，木構造に従ってすべてのクラス定義を敷き詰める必要がある．本研究では，上に親，下に子がくるような一般的な木構造表現に基づいた独自の手法を用いてクラス定義の敷き詰めを行う．**図 4** に木構造の表示を示す．図左のような **ClassA** を基底クラスとする派生クラス群の木構造に対して，図右のように上に親が下に子がくるような木構造表現に従った割当てを行う．もしもプログラム中に複数個のクラス階層構造があった場合には，このような木構造を表現した領域を横に並べて表示する．なお，敷き詰め手法の詳細については，後述のズーミング表示が関係するため，後ほど紹介する．

注目度に基づくズーミング

本可視化手法では，ソースコード全体を 1 画面上に収めるように表示する．しかし，ソースコードのクラス数と行数が増えるに従って，次第にソースコード表示が縮小されていくことになる．このため，ソースコード自体を読むことが困難になるだけでなく，ソースコード上に重畳表示される実行されたソースコード行や関数呼び出しのエッジも閲覧しにくくなる．

そこで我々は，クラス階層表現の注目している部分のソースコードを見えるように拡大するズーミング機能を用意する．具体的には，閲覧している時点で実行された関数呼び出しが注目部分であると考え，呼び出し元と先のクラスと関数呼び出しのエッジを拡大表示する．

ズーミングに際して，ソースコードの全体表示やクラスの配置を損なうことは，プログラム構造の理解を妨げてしまう可能性がある．そのため，ソースコードの可視化結果の概観を維持しながら，ズーミングを行う．本手法では，クラスは親子関係の木構造で表現されているので，ズーミング手法の 1 つである Fisheye 表示[10] のうち木構造に特化したものを用いる．Fisheye 表示は Furnas によって提案された手法で，この手法を用いることで概観を維持しながら注目部分を拡大表示することができる．一般に Fisheye 表示では，ある一定の閾値を下回るような重要度が低い要素は間引きして，表示を行わないことがある．本可視化シ

ステムではつねにソースコード全体を表示するため，表示の間引きは行わない．今回用いた Fisheye 表示では，重要度が高い，すなわち注目しているクラスほど，大きな表示領域を割り当てる．重要度は注目している関数呼び出しとその前後の呼び出しに関係するクラスであるほど高いものとした．さらにそれらのクラスに親子関係が近いクラスにも高い重要度を割り当てるものとした．

以上をふまえたうえで，クラス階層の木構造の具体的な敷き詰め方法について述べる．本研究におけるクラス階層の敷き詰め表現は，木構造を空間に敷き詰めて表示する手法である TreeMap [11] をクラス階層の表現に適するように修正したものである．TreeMap では木構造を入れ子構造と見なして，領域分割を再帰的に繰り返すことで，領域割当てを行う．それゆえ，この手法は木構造の葉となる要素を表示するために特に有効である．しかし，TreeMap では途中の階層は外枠として扱うので，今回のクラス階層構造のように，途中の階層でも情報を表示する場合には不向きである．そこで本研究では各要素に十分な表示領域を与えるように TreeMap に変更を施し，上に親，下に子がくるような一般的な木構造表現に基づいた敷き詰めを行う（図 4 参照）．この敷き詰めを行う際に表示領域を各クラスが持つ重要度に従って定めることで，ズーミング表示を実現する．**図 5** にズーミングをしている様子を示す．今，**ClassC** が最も注目したいクラスであるとする．左図は通常時の木構造の割当てですべてのクラスに対して均等に領域が割り当てられている．それに対して右図では **ClassC** に対してより大きな表示領域が割り当てられている．

また各クラスのソースコード行についてもズーミングを行う．1 つのクラスを表示する領域は限られているので，行数が多いクラスの可読性の低下を防ぐためである．現在は，実行されているソースコード行に近い行ほど，重要度が高いものと見なした Fisheye 表示を行っている．

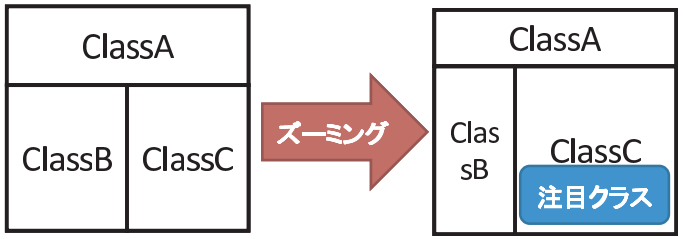

図 5　注目度に基づくズーミング
Fig. 5　Zooming presentation based on importance.

3. GUI プログラム理解支援システム ORCA

設計した可視化手法に基づいたシステム ORCA（Operation Reaction Code Analyzer）の実装を行った．システムの概観を**図 6** (a) に示す．システムはコントロール部，グラフ表

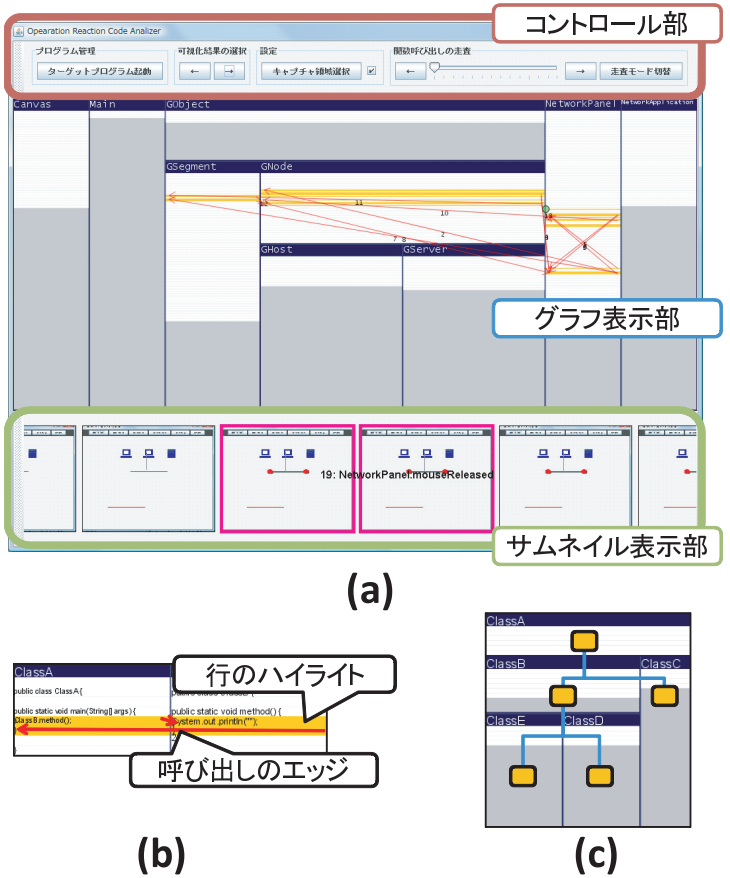

図 6　GUI プログラム理解支援システム ORCA．(a) 概観，(b) 呼び出しエッジと強調表示，(c) クラス階層表示
Fig. 6　ORCA system. (a) Overview, (b) Representation of call edge and line highlights, (c) Representation of class hierarchy.

示部，サムネイル表示部によって構成される．

グラフ表示部ではソースコードおよび実行情報を可視化する（図 6 (a) グラフ表示部参照）．このようにソースコード全体が 1 画面上に収まるように縮小して表示する．ここで各実行情報の表示について詳細に説明する．実行されたソースコード行と関数呼び出しは図 6 (b) のように表示される．関数呼び出しの表示は，呼び出し回数が多くなった場合や複雑になった場合に，ソースコード行の強調表示による実行部分の把握を阻害してしまうことがある．その問題に対処するため，ORCA では関数呼び出し表示の ON/OFF 切替え機能を用意した．図 6 (c) は図 4 のクラス階層を実際にマッピングしている様子である．2 章で述べた表示方法に従い，領域がそれぞれ割り当てられていることが分かる．

サムネイル表示部には対象プログラムの GUI の画面変化をサムネイルとして並べて表示する（図 6 (a) サムネイル表示部参照）．中央のサムネイルペアを囲っている強調枠は現在注目している画面変化であることを示す．グラフ表示部には注目している画面変化が発生した時点での実行情報が表示される．

ここで現在の ORCA における操作・実行されたソースコード・画面変化の対応の提示方法について詳細に説明する．Java の GUI プログラムでは GUI への操作が行われると，入力イベント（マウスイベント等）以外の内部イベント（再描画イベント等）があわせて発行されることがある．このような場面を理解するためには，それぞれのイベントごとに実行されたソースコードを把握する必要がある．そのため現在の ORCA では，これらの内部イベントを入力イベントとは別の 1 つの操作であると見なして，別々に可視化をすることにしている．また，現在の ORCA では，対象プログラムを起動した時点からイベントが発行された順に番号がカウントされ，その番号がイベント名の表示に付加される．このイベント番号は開発者が画面とイベント名からシーンを特定にするための手助けとなる．

開発者は本システムへの操作として，対象プログラムの制御，可視化結果の選択，関数呼び出しの走査を行うことができる（図 6 (a) コントロール部参照）．対象プログラムの制御とは，対象プログラムの起動・再開，一時停止，停止である．可視化結果の選択を行うと，閲覧している可視化情報が前後の GUI 操作時のものに切り替わる．関数呼び出しの走査は可視化情報ごとの関数呼び出しのエッジをたどる機能である．ステップバック，ステップフォワードボタンを押下するとステップが切り替わり，その時点で行われた関数呼び出しに注目した関数走査表示が行われる．

図 6 (a) で示されるシステムの概観は**図 7** のネットワーク構造図エディタを理解対象のプログラムとして起動した場合の表示結果である．本ネットワーク構造図エディタのソース

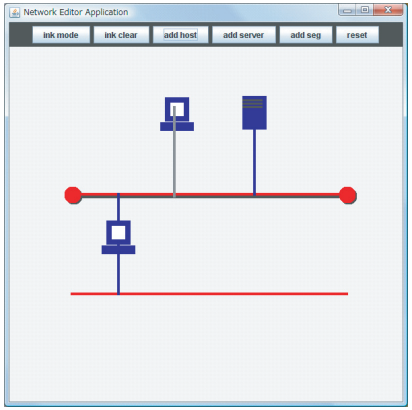

図 7　対象とする GUI プログラム（ネットワーク構造図エディタ）
Fig. 7　Target GUI program (Network configuration diagram editor).

コードは，785 行である．また，プログラムは 9 個のクラスから構成されており，クラス階層の深さは最大 3 である．開発者はこのプログラムについて理解したい場合には，このプログラムの GUI を直接操作しながら可視化を行う．ORCA ではプログラムへの操作が行われるたびに実行情報を自動的に取得するため，GUI への直接操作をしている間に ORCA 上で別の操作を行う必要はない．一方，デバッガを用いる場合には，対象プログラムへの操作とデバッガへの操作とが干渉する．たとえば，ドラッグ操作を対象プログラムに対して行いながらデバッガを操作することはできない．そのため，デバッガでは同一イベントの連続的な発行の様子を解析することが困難だったが，ORCA ではそのようなイベントも解析することができる．

可視化結果に対する関数呼び出しの走査機能

ORCA は，操作ごとの可視化結果に対して，ソースコードの実行情報を関数呼び出しごとにソースコードを順々にたどりながら読み進めることができる機能（関数走査機能）を提供する．具体的には，開発者がある時点における 1 つの関数呼び出しに注目したときに，その呼び出し元と先のソースコードを詳細表示する．このように関数呼び出しに関わるソースコードを順に連続的に表示することで，可視化結果における処理の流れを追うことができる．ソースコードの詳細表示方法には，2 章で述べたズーミング表示を用いる方法とソースコードの断片をポップアップとして表示する方法の 2 種類を用意した．以降は，これら 2

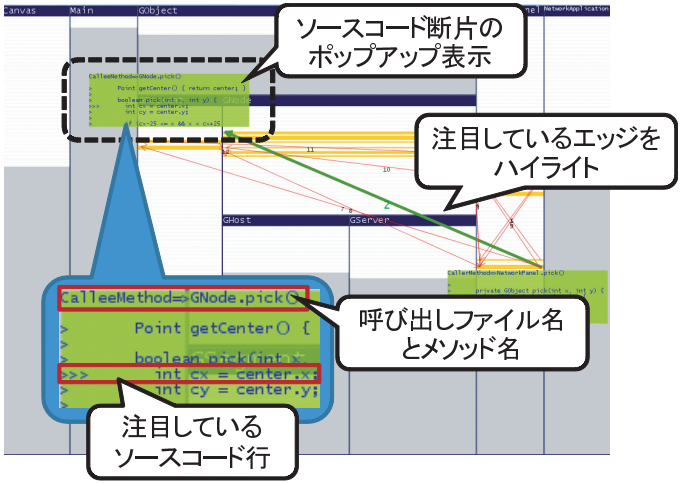

図 8　ポップアップ走査
Fig. 8　Pop-up scanning.

つの表示方法を用いた関数走査をそれぞれズーミング走査，ポップアップ走査と呼ぶ．

ポップアップ走査では，可視化結果のグラフ表示の上で，注目している関数呼び出しのエッジを強調表示し，さらに呼び出し元と先のソースコード断片をポップアップとして表示する．ソースコード断片には，呼び出しが起こったクラス名とメソッド名，呼び出しが発生した行付近のソースコードを表示する．ソースコード断片に表示されるソースコードは数行程度であるが，ソースコード断片の上でマウスホイールを動かすことによって表示行を前後に移動して内容を確認することができる．**図 8** にポップアップ走査を行っている様子を示す．このように，注目している関数呼び出しのエッジが強調され，その呼び出し元と先がポップアップとして表示される．現在，ポップアップの表示には半透明単色のパネルを使っている．半透明色を用いることでグラフ表示上の実行されたソースコードの強調表示の閉塞を軽減することができる．ポップアップ表示上でグラフ表示の強調表示の配色を反映させず単色にした理由は，色が複雑に重なり合うことで可視性が低下することを防ぐためである．ポップアップ表示ではグラフ表示の概観をそのままに関数呼び出しの様子を順々にたどることができるというメリットがある．

ズーミング走査を行っている様子を**図 9** に示す．今，**クラス A** の関数から**クラス B** の

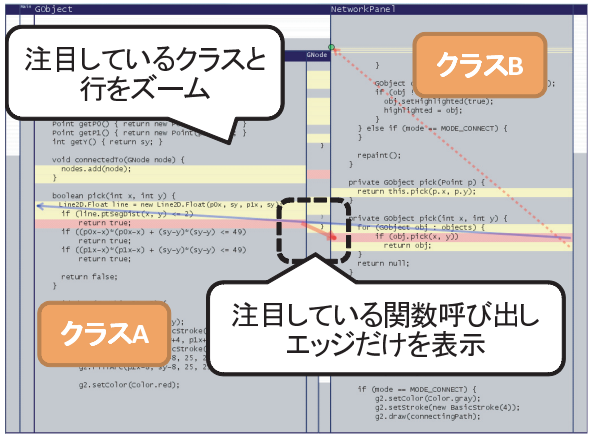

図 9　ズーミング走査
Fig. 9　Zooming scanning.

関数が呼び出されている．つまりこの場面では**クラス A**（図の左部）と**クラス B**（図中の右部）が注目すべきクラスであるため，これらのクラスが拡大表示されている．画面中央の矢印が注目している関数呼び出しのエッジである．各ソースコードも注目時に実行された行の周辺のみがズーミング表示されている．ORCA 上でズーミング走査を連続的に行っている様子については文献 8) を参照されたい．

ズーミングに際して，今回 ORCA が用いた木構造敷き詰めアルゴリズムは以下のとおりである．

(1)　親クラスとすべての子クラス部分木の重要度の比で領域を縦に分割する．
(2)　各子クラス部分木の重要度の比で下領域を横に分割する．
(3)　以下は各子クラス部分木に対して，再帰的に割当てを繰り返す．

図 10 を使って上記のアルゴリズムを図説する．今 **ClassA** は **ClassB** と **ClassC** の親クラスである．ここで図左のクラス表示内の数字は各クラスの重要度を表す．**ClassC** の重要度が最も高いことから，このクラスが今最も注目したいクラスであることが分かる．まずは手順 1 の縦方向への分割をする．**ClassA** の重要度は 1，すべての子クラス（**ClassB**, **ClassC**）の重要度は 3（= 1 + 2）なので，領域を 1 : 3 で分割する．次に手順 2 の横方向への分割をする．**ClassB** と **ClassC** の重要度から，下領域を 1 : 2 で分割する．図右はこのようにして分割された領域である．今はクラス階層が浅いため，手順 3 は行わなかった

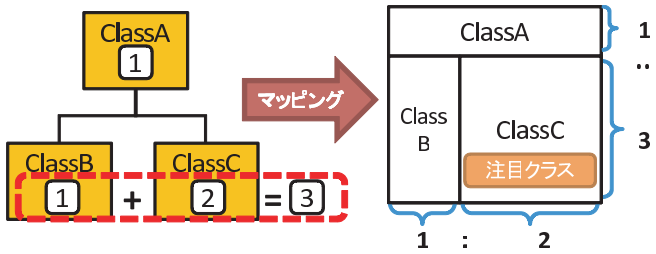

図 10　木構造敷き詰めアルゴリズム
Fig. 10　The allocation algorithm for the display region with the tree structure.

が，クラス階層が深くなった場合にはこれらを再帰的に行う．ここで **ClassC** の領域に着目すると，他の領域よりも大きな領域が割り当てられていることが確認できる．このように本アルゴリズムを用いることで，重要度が高いクラスに対してより大きな領域を割り当てることができる．なお，ズーミング操作を実現するための重要度の決定法を次に示す．ある関数呼び出しに注目しているとき，まず，注目すべきクラスに親子関係が近いクラスほど高い重要度を割り当てる．現実装では，注目すべきクラスの重要度を 0，子孫クラスの方向に 1 親等増えるごとに重要度を −0.5 し，先祖クラスの方向に 1 親等増えるごとに重要度を −1 している．加えて，一連の関数呼び出しにおいて用いられたクラスも見やすくするために，注目している関数呼び出しの前後において利用されたクラスに対しても重要度を追加する．

ポップアップ走査では関数呼び出しの起こった行の実行しか確認できないが，ズーミング走査ではそれ以外の行における実行の有無についても確認しながらソースコードを読み進めることができるというメリットがある．

Eclipse との連動機能

ORCA は統合開発環境（以下，IDE）である Eclipse [*1] のプラグインとして実装されており，Eclipse との統合が図られている．可視化システムを IDE に統合することにより，従来の IDE が提供するプログラム静的情報やデバッガを使った理解に加えて，GUI プログラムを操作しながらの動的解析による理解を行うことが可能となる．また，ORCA では変数情報の提示をしていないが，IDE が提供する情報から変数情報を取得できるようになる．さらに，IDE にはエディタも備わっているため，可視化システムを利用した理解作業後あるいは作業中に，即座にプログラムの編集作業にとりかかることができる．このようにして，統合により可視化システムと IDE がそれぞれ提供する機能を補完しあうことが可能なため，本環境を利用することで理解作業の向上が期待できる．

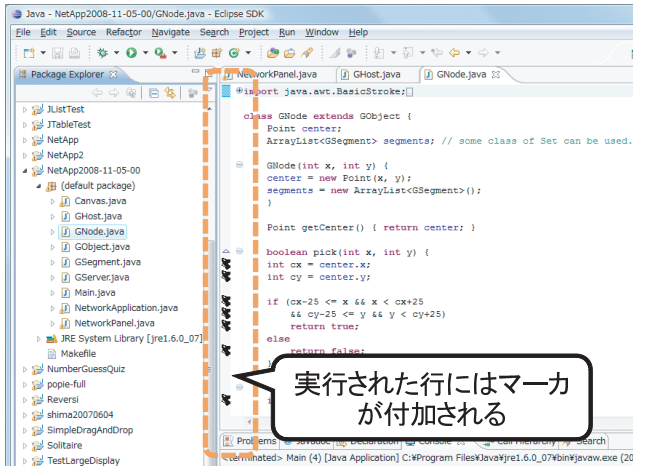

図 11　Eclipse エディタ上へのマーカ表示機能
Fig. 11　Display of marker on Eclipse editor.

Eclipse プラグインとしての機能は，まず ORCA から Eclipse エディタ上へのソースコードジャンプがある．ORCA のグラフ表示部に示されるソースコード行をクリックすると，Eclipse のソースコード編集画面上において，クリックされたクラスのソースコードエディタが開かれ，クリックしたソースコード行が表示される．

また Eclipse エディタ上のソースコードに対してもソースコードの実行が可視化される．**図 11** のように，ORCA 上で強調表示された行に対応して，Eclipse エディタ上のソースコードの左横にマーカが配置される．これにより Eclipse エディタ上でも操作ごとのソースコード実行箇所が分かる．

このように Eclipse と ORCA では連動機能を実現しているため，双方が提供する情報が対応付けやすくなっている．

4. 利用シナリオ

本章では理解作業の例として，図 7 に示したネットワーク構造図エディタを改良する目的

⋆1 http://www.eclipse.org/

で ORCA を利用した理解作業を示す．このネットワーク構成図エディタは，ホストやサーバのアイコンを追加し，それらとバスの間にエッジをつなぎながらネットワーク構成図を作成することが可能なプログラムである（図 7 参照）．ホストやサーバからバスに向かって垂直に右ドラッグを行うことでエッジ付けを行うことができる．アイコンを左ドラッグをするとアイコンの配置を移動することができるが，アイコンとバスの間にエッジが付いている場合には，バスに垂直にエッジがつながっているという制約条件内でのみ移動可能となる．また，バスにマウスカーソルを合わせている間には，視覚的フィードバックとしてバスがやや太く表示される．ただし，現在のネットワーク構成図エディタの仕様では，バスが数ピクセル程度の線でしか表現されないために，バス上にカーソルを残しながらマウスをリリースすることが難しいという問題がある．そこで，バスの視覚的フィードバックを大きくし，さらに広げたフィードバックの上でマウスリリースを行ったときにもエッジが結ばれるように機能変更したい．これがこの機能を理解する目的である．

そこで，ホストとバスのエッジ付け機能を理解することを考える．ORCA を利用したこの理解作業の手順は次のようになる．

まず，理解対象の機能に関するクラスを抽出する．これは ORCA と対象プログラムの起動後，対象プログラムの GUI 上でエッジ付け操作を行い，その可視化結果を観察すればよい．ここでは，はじめにホストにカーソルを合わせて右ドラッグを開始し，その後バスにカーソルを合わせて視覚的フィードバックの変化が起こったことを確認してからマウスをリリースする．以上の操作に対して，マウス操作に関わるイベントがマウスプレス→マウスドラッグの数回繰返し→マウスリリースの順に発行され，各イベントの間で再描画イベントが発行されるので，これらのイベントそれぞれに対して可視化結果が得られる（たとえばマウスリリース時に図 6 (a) が得られる）．これらの可視化結果のグラフ表示部において強調表示されているソースコード行を観察すれば，これらの各イベントで使われているクラスが抽出できる．実際には **NetworkPanel**，**GObject**，**GNode**，**GSegment**，**GHost**，**GServer** という 6 クラスが実行されていたことが分かる．

この時点で機能に関わるクラスを絞ることができたので，次にこれらのクラスの役割や関係についての理解を深める．イベントの開始点はすべて **NetworkPanel** であったことから，このクラスに一連の処理のイベントハンドラが実装されていることが把握できる．また ORCA のクラス階層表示から，**NetworkPanel** 以外のクラスはすべて **GObject** を基底クラスとした親子関係にあることが分かる（図 6 (a) 参照）．クラス名と階層構造から，**GObject** は画面に配置されるホストやバスを表すオブジェクトを示し，その子クラスが具体的なオブジェクトを表していると推測できる．またホストとサーバの共通機能がクラスとして抽象化されていることも推測できる．

次に，実行された関数を順に追うことによって処理内容を詳細に理解する作業を行う．この作業は ORCA のポップアップおよびズーミング走査，Eclipse との連動機能を活用して行う．これらを使ってソースコードを解析すると，マウスプレス時に開始座標を記録しておき，マウスリリース時に開始座標と終了座標上にある **GObject** がそれぞれ **GNode** と **GSegment** だったとき，それぞれの間にエッジをつなぐ処理が行われることが把握できる．また，一連の処理の中で，マウスがある位置に **GObject** があるどうかを調べるために，**GObject** の子クラスでオーバライドされている包含判定関数が呼び出されていることも分かる．終了座標における包含判定で **GSegment** の包含判定関数が実行されて真を返していたことから，終了座標位置でカーソルが置かれていたバスは **GSegment** に対応していることが理解できる．以上から今回の改良では，**GSegment** の包含判定関数の実装を変えればよいことが分かる．

最後に描画処理の理解作業を行う．包含判定領域の実装を変える必要があることは分かったが，上記のマウス操作で実行されたソースコード行にはバスの視覚的フィードバックを描画する処理が見つからなかったからである．バスの視覚的フィードバックを実装している部分を明らかにするためには，バスに対してフィードバックが表示されている場合といない場合の再描画イベントに対する可視化結果を比較してやればよい．これには，マウスドラッグ中の画面サムネイルの中から，視覚的フィードバックが切り替わるシーンを探し出し，切替わり前と後での可視化結果を比較する．実際にそれぞれの実行されたソースコード部分を比較すると，**GSegment** 内で実行されたソースコード部分に違いが見られる．さらにズーミング走査を利用して詳細にソースコードをたどると，**GSegment** に描画関数が実装されていて，包含判定の真偽によって定まる内部変数の値に従って視覚的フィードバックの描画内容を決めていることが分かる．

以上により今回の改良では，**GSegment** 内で実装されている包含判定関数と描画関数をそれぞれ修正すればよいことが分かった．このように ORCA を用いることで，GUI プログラムにおいて必須である 3 つの理解作業を容易に行うことができる．

5. 実　　装

本章では実装に利用した技術について詳細に説明する．

Java プログラムの動作を解析するには各クラスが持つメソッドやフィールド，クラス間

の関連といった静的情報，およびプログラム実行中のメソッド呼び出しやフィールドの変化といった動的情報を取得する必要がある．特に動的情報は実行時に取得し，後で参照できるように保存する．静的情報は Eclipse が提供する Java 開発環境 JDT（Java Development Tools）の API を利用し取得する．動的情報を取得するために JavaVM の実行を明示的に制御し，各クラスの情報を観察することができる JDI（Java Debug Interface）API[⋆1]を使用した．

サムネイルの取得には，Java の AWT パッケージの Robot クラスが提供する画面キャプチャ機能を利用した．

システムは，まずソースコードから静的なクラス構成を取得してシステムの概観を表示する．その後，JDI を用いて解析対象となる GUI プログラムを立ち上げる．開発者がプログラムを操作することによりイベントが発生すると，JDI によってその場で動的にプログラムの実行を解析し，実行情報を表示に反映させる．またグラフ表示部では取得した動的情報をもとに実行されたソースコードを可視化する．

ズーミング表示のレンダリングには Piccolo.Java1.2 を利用した[12]．Piccolo はズーミングインタフェースの構築をサポートするツールキットである．

今回提案・実装した手法では，対象 GUI プログラムが数千行程度の規模であれば，システムを無理なく稼働させながら解析を行うことが可能である．

6. 被験者による評価実験

我々の作成したシステムの有効性を検証するために，ユーザスタディを行った．実験を行うにあたって，プログラム可視化ツールのユーザスタディを行っている研究である SHriMP [13)–15)] や ClassBlueprint [16]，CARE [17] の実験設計や評価方法を参考にした．

6.1 使用する理解支援ツール

実験では以下の 3 つの理解支援ツールを比較対象とした．

- Eclipse
- ORCA（画面サムネイル表示なし）
- ORCA（画面サムネイル表示あり）

これまで研究がなされてきた動的実行をともなう理解支援ツールのほとんどは，現在利用することができないか Java プログラムを対象としていない．そのため，我々の提案手法の比較対象としては無償で利用可能な統合開発環境である Eclipse を用いることにした．

また画面変化と実行されたソースコードをあわせて表示するという本手法の有効性を確かめるために，実験には ORCA のサムネイル表示がないものを用意した．各被験者にはこれらの 3 つのツールを利用してプログラム理解作業を行ってもらった．

⋆1 http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html

表 1　実験に使用したプログラム
Table 1　Programs used in the experiment.

プログラム名	行数	ファイル数	最大クラス階層
ドローイングツール	1,445	16	2
ネットワーク構成図エディタ	785	9	3
ドラッグ&ドロッププログラム	323	6	2

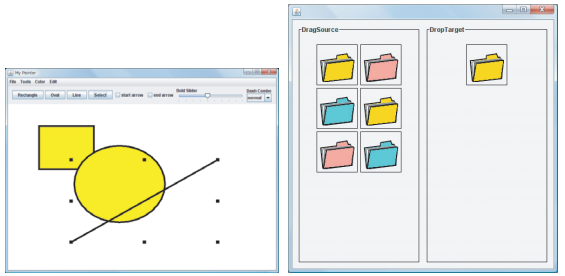

図 12　対象とする GUI プログラム（左）ドローイングツール，（右）ドラッグ&ドロッププログラム
Fig. 12　Target GUI program.
(Left) Drawing tool, (Right) Drag and drop program.

6.2 理解の対象となる GUI プログラム

今回の実験において，理解対象とするプログラムは GUI を備えること，オブジェクト指向設計に従って実装されていることを前提とした．特に GUI に関しては，ユーザ操作に対して視覚的なフィードバックを返すような機能を有していることを条件とした．

1 つのプログラムだけを理解の対象とすると，1 つのツールを利用した時点でプログラムの構造を理解してしまう．そのため，今回は異なる 3 つのプログラムを用意し，被験者ごとにこれらのプログラムと使用するツールの組合せを，実験全体で偏りがないように割り当てた．実験においてツールを使用する順番や理解対象とするプログラムの順番についてはランダマイズして割り当てることでバランスをとった．

理解の対象とするプログラムは**表 1** に示すとおりである．ドローイングツールは**図 12** 左

に示すようなプログラムである．描画図形選択ボタンをクリックして描画モード切替えを行い，キャンバスをドラッグすることで図形の描画や移動をすることができる．ネットワーク構成図エディタは，ホストやサーバとバスの間にエッジをつないでネットワーク構成図を作成するプログラムである．利用シナリオでの理解に使用したプログラムであるため詳細については 4 章を参照されたい．ドラッグ&ドロッププログラムでは，単純なドラッグ&ドロップ処理のみを実装している．画面は 2 つの領域から構成され，左側の領域に置かれているオブジェクトをドラッグして右側の領域にドロップすると，ドラッグしたオブジェクトを左側の領域に移動またはコピーすることができる（図 12 右参照）．表 1 に示すように，今回の実験では使用するプログラムの規模は大・中・小の 3 種類になるように設定している．

6.3　出題した設問

被験者には各対象プログラムに対する理解作業を行ってもらった．理解作業はこちらから出題する設問を制限時間内で解くというものである．設問は以下の形式に従ったものをプログラムごとに用意した．

設問 1　プログラムに対して X 操作を行った場合に，実行されるファイルを列挙せよ．
設問 2　プログラムに対して X 操作を行った場合に，呼び出されるメソッドの順序を答えよ．
設問 3　X 処理と Y 処理におけるフローの違いはどこか説明せよ．

すべての設問は，GUI プログラムへの操作をともなう機能の理解を問うものである．またすべての設問は複数イベントをともなうものであり，その結果から ORCA が複数イベントをともなう機能の理解に有効であるかを検証する．設問 1 は機能の実装部分を操作に結び付けて理解する作業を行うものである．また設問 2 はさらに処理の流れの理解を問うものである．これら 2 つの設問では，ORCA の表示を用いて，操作と実行されたソースコードを結び付けることが可能であるかを検証する．また設問 1 では，機能の実装部分を把握するため，ORCA でソースコード実行部分を 1 画面上に表示することが有効であるかもあわせて検証する．設問 3 は処理の詳細を問う内容になっており，2 つのシーンを比較する作業を必要とする．さらに設問 3 では操作間に視覚的なフィードバックが起こるような問題を用意した．このように設定した設問 3 の結果から ORCA の画面サムネイル表示が状況の同定に有効に働くかを検証する．

具体的な設問の内容として，ネットワーク構成図エディタに対する設問を抜粋して以下に掲載する（実際には，前提条件や操作内容の詳細な記載等が併記されており，設問の曖昧さをなくすような設問となっている）．

設問 1　ホストからバスに向かって右ドラッグしてエッジをつないだ場合に，実行されるファイルを列挙せよ．
設問 2　ホストの追加ボタンを押下した場合に，呼び出されるメソッドの順序を答えよ．
設問 3　サーバにマウスを載せているときの描画処理と載せていないときの描画処理におけるフローの違いはどこか説明せよ．

また，被験者には設問を解答するにあたっての諸注意として以下のようなアナウンスをした．

- 各プログラムにつき解答の制限時間は 30 分とする．
- 制限時間内になるべく多くの設問に解答すること（解答には部分点がつく）．
- すべての設問に手をつけるように各設問につき最低 5 分以上の解答時間を確保する（ただし，5 分以内に解答が完了した場合は除く）．
- 設問はどの順番で解いてもかまわない，解答途中で別の問題に移ってもよい．

6.4　被　験　者

実験には 23 から 25 歳までのコンピュータサイエンスを専攻する学部生および大学院生 9 人を雇用した．すべての被験者は授業や独学での Java プログラミング経験および GUI プログラムの作成経験があった．

6.5　実験の手順

各実験は以下の手順に従って遂行した．

(1)　実験概要の説明（5 分）
(2)　事前アンケートの実施（5 分）
(3)　各ツールの説明（30 分）
(4)　練習タスクによるトレーニング（30 分）
(5)　各ツールを利用したプログラム理解作業（30 分 × 3 ツール）
(6)　事後アンケートの実施（10 分）
(7)　インタビュー（10 分）

実験にかかる総時間は 3 時間程度である．

6.6　実 験 環 境

実験にはデスクトップ PC（Intel（R）Core（TM）2 Duo 3.00 GHz，メモリ 2 GB，OS Windows Vista SP1）にデュアルディスプレイ（それぞれ 22 インチワイド・解像度 1680×1050，縦置き 20 インチ・解像度 1024×1028）を接続して用いた．今回使用する理解支援ツールは，いずれもより広い画面領域の上で利用したほうが快適であると考えられる．逆に小さな作業領域では，ユーザに余計なストレスを与える，集中力を切らしてしまう

といった恐れがある．デュアルディスプレイはそれほど一般的ではないが，それらの懸念に対処するため実験に導入することにした．デュアルディスプレイの使用方法の違いによる影響を抑えるため，被験者には，22 インチディスプレイに理解対象となるプログラムを配置し，20 インチディスプレイに各理解支援ツールを配置するように指示をした．

実験には発話思考法を用いた．被験者にはツール上での操作の意図や実験中の気づきについて発話してもらうようにあらかじめ依頼し，その様子は被験者に許可をとりビデオカメラに収めた．

また，実験中に操作方法がすぐに参照できるようにツールの操作マニュアルを渡し，実験中に操作方法や設問内容に関して疑問が生じた場合には適宜質問を投げかけてもらった．なお，操作マニュアルについては事前に読んだうえで実験に臨んでもらった．

6.7　各ツールの説明と練習タスクによるトレーニング

被験者には本実験の問題を解いてもらう前に，ORCA，Eclipse それぞれについて，タスクをこなすのに十分な機能サブセットをレクチャした．レクチャは簡単なプログラムに対する理解作業を実演しながら行った．

その後，練習問題として「数字当てゲームプログラム」の理解タスクをこなしてもらった．数字当てゲームプログラムの規模は行数 344，ファイル数 9，最大クラス階層 1 である．プログラムの理解には Eclipse と ORCA の両ツールを使ってもらった．その際に，各設問についてまず片方のツールを用いた解答をし，もう一方のツールを用いて，先の解答の正当性を検証するという形式をとった．練習問題は特に制限時間を設けず，すべての設問に解答し終えるまで作業を行ってもらった．被験者が操作方法に迷っている場合には助言をし，より早くツールに慣れてもらうように努めた．

6.8　アンケート

実験開始前に被験者のプログラミング経験に関するアンケートを行った．ふだん使用しているプログラミング言語や環境についても尋ねた．

3 つのツールすべてに対するプログラム理解作業が終了した後で，ORCA の有効性を評価するアンケートを行ってもらった．ORCA が提供する各機能がプログラムの理解に役に立ったかどうかを，肯定，やや肯定，やや否定，否定の 4 段階で評価してもらい，その理由について自由記述を行ってもらった．評価してもらった項目は以下のとおりである．

項目 1　GUI 操作に合わせてソースコード中の実行部分が強調される表示
項目 2　クラス階層表示
項目 3　GUI 画面のサムネイル表示
項目 4　ポップアップ表示による関数走査機能
項目 5　ズーミング表示による関数走査機能
項目 6　Eclipse エディタ上へのマーカ表示
項目 7　ORCA 全体

また最後に ORCA を実際のプログラム理解作業に使用したいかどうかを尋ねた．

アンケート回答終了後，アンケート結果についての確認とコメントを得るために被験者へのインタビューを行った．

7.　被験者による評価実験の結果

7.1　設問解答の分析

各設問はあらかじめ用意した採点基準に従って，0〜1（1 が満点）の間でスコアをつけた．採点作業は著者自身が行った．**表 2** に結果を示す．

解答の採点結果に対して，使用したツールを因子とした分散分析を行ったところ，設問全体で見ると画面サムネイル表示ありの ORCA は Eclipse よりも良い結果が得られた（$p < 0.01$）．さらに画面サムネイル表示なしの ORCA も Eclipse よりも良い結果が得られた（$p < 0.01$）．しかし ORCA の画面サムネイル表示があるものとないものの間には有意差は見られなかった（$p = 0.498$）．

次に解答時間について言及する．Eclipse を用いたとき，大部分の被験者が 30 分以内にすべての設問についての解答を完了することができなかった．一方，ORCA を用いた場合

表 2　実験の採点結果
Table 2　Task results.

ツール	設問の種類	平均	標準偏差
Eclipse	設問 1	0.444	0.232
	設問 2	0.600	0.325
	設問 3	0.556	0.497
	設問全体	0.533	0.229
ORCA（サムネイル表示なし）	設問 1	1.000	0.000
	設問 2	0.951	0.092
	設問 3	0.750	0.236
	設問全体	0.900	0.075
ORCA（サムネイル表示あり）	設問 1	0.988	0.035
	設問 2	0.926	0.070
	設問 3	0.931	0.157
	設問全体	0.948	0.052

には大部分の被験者が制限時間内ですべての設問を解き終えることができた．平均の実験完了時間は画面サムネイル表示ありの場合には 26 分 30 秒，なしの場合には 26 分 59 秒であった．

次に各設問について考察する．設問 1 と設問 2 については，ORCA と Eclipse の間に有意差がみられた（$p < 0.01$）．Eclipse の場合には操作に対応するソースコードを発見するのに時間がかかる傾向にあり，列挙すべき情報の抜け漏れが目立った．ビデオを分析して確認したところ，情報の抜け漏れは多くの場合，静的な情報だけに頼ったことやデバッガ使用時にトレースのステップを飛ばすことが原因で発生していた．これらの結果から ORCA が操作と実行部分の結び付けを行い，実行されたソースコード部分を抽出するうえで十分有効であることが実証できた．

設問 3 では，表示の異なる 2 つのシーンを解析しなければならないこと，操作間に視覚的なフィードバックが変化するような機能を設問としたことから，ORCA の画面サムネイル表示がある場合とない場合では大きな差が出ると予想していた．実際の結果を見ると，画面サムネイル表示ありの ORCA を用いた場合の平均スコア（0.931）は，画面サムネイル表示なしの ORCA を用いた場合の平均スコア（0.750）と比べて高かった．しかし今回は被験者のサンプル数が少なかったこともあり，統計的な有意差には至らなかった（$p = 0.261$）．ビデオ分析をしたところ，1 人の被験者は，画面サムネイル表示のない ORCA を利用している際に，誤ったシーンの解析を行い続けていた．これは連続操作間に発行された各イベントにより実行されたソースコードと，画面の変化とが正しくマッピングできていなかったために生じたものと思われる．もし画面サムネイル表示があったとしたら，サムネイル情報から正しいマッピングを行うことができたと予想される．また数人の被験者は設問 3 において，ORCA の可視化結果とその時点の実行状況の間でのマッピングができずに何度も GUI への操作を繰り返した．さらに，ある被験者は状況と可視化結果のマッピング作業が不十分なまま解答を進めたために解答を誤っていた．このように，画面サムネイル表示の有無は少なからず GUI プログラムの理解作業に影響を与えていた．

Eclipse において何人かの被験者は，マウスのドラッグ操作のように一連の複数イベントを把握する必要があった場合にも，一部のイベントのみしか収集することができなかった．また同一のイベントが連続して発行している中で変化が起こる条件を見つけ出すような問題では，デバッガを利用した把握が難しく，被験者はソースコードを静的に読み進める中から該当部分を見つけ出す必要があった．これらの事実および，Eclipse と ORCA でのスコアの比較から，ORCA は複数イベントをともなう機能の把握にも十分有効であると考えられる．

表 3　アンケートの結果
Table 3　Questionnaire results.

アンケート項目	平均値	標準偏差
項目 1（GUI 操作に合わせてソースコード中の実行部分が強調される表示）	3.56	0.53
項目 2（クラス階層表示）	2.44	0.88
項目 3（GUI 画面のサムネイル表示）	3.56	0.73
項目 4（ポップアップ表示による関数走査機能）	3.63	0.74
項目 5（ズーミング表示による関数走査機能）	3.50	0.76
項目 6（Eclipse エディタ上へのマーカ表示）	2.63	1.30
項目 7（ORCA 全体）	3.56	0.53

また対象プログラムを因子として分散分析を行ったところ，プログラム間での有意差はみられなかった（$p = 0.699$）．そのためプログラム規模の大小にかかわらず，上記の結果が得られると予想される．

事前アンケートでは 5 人がふだんからデバッガを利用し，4 人がふだんあまりデバッガを利用しないと答えた．デバッガの使用経験は，Eclipse を利用してプログラムを理解するうえで差がみられると予想した．しかしながら，デバッガの使用経験の有無を因子とした分散分析の結果からも統計的な差はみられなかった（$p = 0.677$）．

7.2　アンケート結果の分析

事後アンケートの結果を**表 3** に示す．アンケートは 1〜4 の間でスコアを付けた．スコアが高いほど肯定的な評価である．以下に各アンケート項目ごとの結果に対する考察を述べる．

項目 1（GUI 操作に合わせてソースコード中の実行部分が強調される表示）

この項目は肯定的な評価を得た．多くの被験者が理解作業を開始するポイントを絞るためや処理の大まかな流れをつかむために役立つと答えた．

項目 2（クラス階層表示）

この項目は平均的に低い評価を得た．この評価は，今回の実験では対象 GUI プログラムのクラス階層が少なくクラス階層に大きな関わりのある設問はなかったことに原因があるものと思われる．この原因をインタビューにおいて確かめたところ，問題解答中にクラス階層表示について意識する必要がなかったため，低めの評価をせざるをえなかったというのが被験者全員の意見だった．また，評価こそ低いものの否定的な意見は特になく，“実際の利用時には役に立ちそう” という意見もあった．

項目 3（GUI 画面のサムネイル表示）

この項目に関しては 8 人の被験者が理解に役立ったと答え，一方で 1 人は役に立たなかっ

たと答えた．肯定的な意見では，“サムネイルがない場合にどの可視化結果がどの操作に対応しているものかまったく分からなかった”，“注意深く画面変化を見ていなくても，自動的に画面変化を記録してくれるので便利である”，“イベントが多く発行されるときに便利だった” という声があった．このことから画面サムネイルの表示が操作とソースコードを結び付けるうえで有益に働いていると考えられる．また “サムネイルが並べてあることで遷移の様子が視覚的に分かりやすい” という意見があった．このことから，画面表示の切替わりが早く肉眼では精密に遷移をたどりきれない場合等にも，サムネイル表示は有効に働いていると考えられる．一方，否定的な評価をした被験者は “イベント番号を確認しながら理解作業を進めていけば，サムネイルがなくても支障はない” とコメントした．また肯定的な評価をした被験者からあがった不満として “サムネイルがイベントが発行されるたびに更新されるのが少し直観的でない気がした．画面変化がないときにもサムネイルが増えていくと混乱する” という意見があった．まず前者の意見に関して言及する．今回の実験ではマウスの連続操作中に画面変化がともなう機能の理解作業を行う設問を含んでいたものの，マウスを滞留するとイベント番号を確認することが可能であった．しかし，たとえば，スナッピング機能の理解作業では上記の方法は困難である．スナッピングは連続的なドラッグ入力の処理中に実行される．スナッピングの処理が行われた時点でのドラッグイベントでイベント発行を止めようとしても，ドラッグ中の場合には即座に次のイベントが発行されてしまうため，イベント番号の確認を行うことは困難である．次に後者の意見に関して言及する．画面変化ごとに区切って表示することはユーザにとって直感的である一方，イベントごとに区切ることも行単位で処理を理解するときに役立つ．そのため今後は，現実装のサムネイルを同一の画面状態を持つイベント群をグルーピングしたものとし，ユーザがそのグルーピングを解除するとイベントごとのサムネイルが現れるような機能を実装することでこの問題の対処を行いたいと考えている．

項目 4（ポップアップ表示による関数走査機能）と項目 5（ズーミング表示による関数走査機能）

ポップアップ走査とズーミング走査は，それぞれ肯定的な評価を得た．実験中に各被験者は好みだったどちらかの表示を重点的に使用する傾向にあった．ポップアップ走査では，全体を俯瞰しながら関数呼び出しを走査できるという点で良い評価を得た．ある被験者は “Eclipse のデバッガを使ったステップ実行に比べて見やすかった” と述べた．一方で，表示の大きさや表示内容についての不満がいくつかあがっており，表示方法に改善の余地があることが分かった．現在のポップアップは単色で表示され，実行された行と実行されなかった行を区別しない．実験中に 1 人の被験者がポップアップされたソースコードを見て，実際には実行されていない行が実行されているものと勘違いをしてしまい，問題解答を誤るということがあった．また，“ポップアップされたソースコードの断片だけを見ても詳細な解析を行うには不十分である” という意見もあがった．これはポップアップ走査の表示上で実行された行と実行されなかった行を区別していないことが原因であると考えられる．以上をふまえると，ポップアップの表示上でも実行された行を確認することができれば，情報の質を落とさずに関数走査を行うことが可能になると思われる．

ズーミング走査では，前後の呼び出しも分かる点やソースコードの詳細を見ながら各行が実行されたかどうかを確認できる点で肯定的な評価を得た．一方，“ステップを切り替えるたびに起こる表示の変化が大きいので混乱してしまい，関数呼び出しをたどりにくかった” という意見があった．そのため，“切替え時におけるアニメーションをよりリッチにしてほしい” という要望があがった．

項目 6（Eclipse エディタ上へのマーカ表示）

この項目に対しては否定的な意見が多かった．“ソースコードの閲覧は ORCA が提供するズーミング走査とポップアップ走査だけで十分だった”，“関数呼び出しのエッジが表示されていないため，処理の流れが追いにくかった” というのが主な意見である．ある意味ではこれは ORCA が提供する表示がソースコードを理解するうえで十分に機能するということを表す結果であるととらえることができるため，それほど悲観的に考える必要はない．また，今回の実験では ORCA を使用する場合に Eclipse が有している各機能（関数呼び出し階層表示等）を併用することを禁止した．実際には ORCA と Eclipse それぞれの機能を併用しながら理解作業を行うことも想定できる．その際にはマーカ表示がより効果を発揮するものと考えている．そのためむしろ，色づけによるソースコード行の強調表示や関数呼び出しのエッジ付けといった ORCA の表示が提供する実行情報すべてを Eclipse エディタ上で実現することにより，ORCA 自体を完全に Eclipse に統合することが今後目指していく方向の 1 つであると考えられる．

項目 7（ORCA 全体）

ORCA 全体に関してはすべての被験者が肯定的な評価をし，実際の理解作業に ORCA を使用したいと答えた．特に “理解作業の初期時にざっと理解するのに役立つ”，“把握が必要な部分を特定するのに役立つ” という意見が多かった．

表 4　実行時間の測定結果
Table 4　Measurement results of the execution time.

プログラムの種類	方法 1 の実行時間	方法 2 の実行時間	方法 3 の実行時間	Robot により入力されたイベント数	方法 1 と方法 3 の実行時間比	方法 3 の入力イベント 1 つあたりの実行時間
オセロゲーム	1.60	3,598.49	8,734.33	2	5,473.07	2,736.54
フィボナッチ数列	0.37	1,666.81	4,391.14	2	11,824.94	5,912.47
円周率	2.10	7,710.50	20,240.65	2	9,657.41	4,828.71
ポップアップメニュー	144.78	3,010.61	3,808.54	96	26.31	39.67
ドローイングツール	22.25	2,275.55	2,983.83	39	134.09	76.51
ネットワーク構造図エディタ	61.73	248.32	496.82	9	8.05	55.20

※ 実行時間の単位はミリ秒

8. 性能評価実験

我々は ORCA の実行時間におけるオーバヘッドを調べるために，以下に示す 3 種類の実行方法について計測した．

方法 1　通常どおりプログラムを実行

方法 2　実行情報（ステップ実行と画面情報）を取得しながらプログラムを実行

方法 3　実行情報の取得と同時に ORCA による可視化をしながらプログラムを実行

対象プログラムには，計算ロジックによる処理を中心としたプログラム（以降，計算中心プログラム）3 種類と GUI への操作入力処理を中心としたプログラム（以降，GUI 中心プログラム）3 種類を用意した．これらのプログラムは Swing により実装されている．実験環境には被験者実験に用いた PC（Intel（R）Core（TM）2 Duo 3.00 GHz，メモリ 2 GB，OS Windows Vista SP1）を利用した．実行時間の計測には java.lang.System クラスの nanoTime メソッドを利用し，GUI への操作を処理する各イベントリスナの実行開始から終了までの時間を測った．たとえば，マウスドラッグ操作に対して描画を更新するようなプログラムの場合には，関数 **mousePressed()**，**mouseDragged()**，**mouseReleased()** と **paintConponent()** が実行されるごとに処理時間を計測し，それらすべての合計値が実行時間となる．GUI への操作入力には，Java の awt パッケージの Robot クラスを使用してネイティブの入力イベントを発行する実験用プログラムを別途作成した．Swing 上で処理されるマウスドラッグイベントや再描画イベントが省略されることを防ぐために，入力イベントを発行する時間間隔は十分に空けた．

計算中心プログラムを対象プログラムとした測定について述べる．コンピュータ対戦型オセロゲームプログラムでは，プレイヤが盤上に駒を置いてから，コンピュータが次の一手を計算し盤上に駒を置くまでの時間の計測を行った．コンピュータが次の手を計算するロジックは，盤上で最も多く相手の駒を返すことのできる位置を探すというもので，多重ループ処理を用いて実装されている．フィボナッチ数列を算出するプログラムでは，15 番目のフィボナッチ数を求めるまでの時間を計測した．このプログラムは，再帰処理を用いて実装されている．また，逆正接関数の Tayler 級数を利用して円周率を計算するプログラムでは，第 10,000 項までの部分和を用いて円周率を求めるまでの実行時間を計測した．後者の 2 つのプログラムは，ボタンのみが配置されたウィンドウによって構成され，ボタンを押すと計算が開始される．

GUI 中心プログラムを対象プログラムとした測定について述べる．まず柏村らによる実行トレースの性能評価[2] で用いられたポップアップメニュープログラムを模したものを用意した．このプログラムはウィンドウ内の任意の点をクリックすると，2 つのアイテムを持つポップアップメニューが開かれ，いずれかのアイテムをクリックするとそのアイテムを選択したことが画面に表示されるというものである．今回は 2 つのアイテムを交互に選択する操作を 8 回繰り返すまでにかかった実行時間を計測した．これに加えて，被験者実験に用いたプログラムであるドローイングツールとネットワーク構造図エディタについても計測を行った．前者についてはキャンバスをドラッグして 3 個の矩形を描画する間の実行時間を計測した．後者ではボタンを押下するとアイコンが追加されるという機能を有しているため，ボタンを押してホストアイコンを追加するという操作を 3 回行った際の実行時間を計測した．

上記の実験についての測定結果は**表 4** のようになった．この結果から，可視化を行った際の実行時間は，計算中心プログラムを実行した場合では通常の数千倍以上，GUI 中心プログラムを実行した場合では 100 倍程度以下となり，GUI 中心プログラムを実行したときのほうが実行時間の差が少なくなることが分かった．この理由には，GUI 中心プログラム

では描画処理等により 1 ステップあたりにライブラリに任せる処理量が多いことがあげられる．そのため，GUI ツールキットに任せる処理が多ければ多いほど，この実行時間の差はより少なくなると予想される．また，可視化を行った際の実行時間と実行情報の取得のみを行った際の実行時間比は数倍程度であった．

なお，今回の GUI 中心プログラムの計測において，ORCA における実行時間は Robot による 1 つの入力イベントあたりで 100 ms 以下であった．実際，理解作業を行う場合には，GUI プログラムへの操作を必要以上に速く行う必要はない．そのため，今回の実験に用いた GUI 中心プログラムのように簡単な処理を行うプログラムを理解する場合，このオーバヘッドは理解作業に影響を与えない．また，実行時間はマシンの性能に大きく依存すると考えられるため，マシンの性能向上がオーバヘッドの減少に直結すると期待できる．

9. 関 連 研 究

まず GUI プログラムの実行情報を可視化する研究について述べる．

柏村らは実行トレースの可視化システム ETV[18] を拡張し，GUI プログラムのデバッグを目的に特化した可視化システムを実装した[2]．このシステムでは，GUI への入力操作の履歴，画面の変化，実行されたソースコード行等を可視化する．このシステムでは，本研究のように操作時に実行情報の取得を行うのではなく，操作時には GUI への操作のみを記録する．その後，記録した操作をもとにプログラムの実行を自動再生してトレースを採取する．これにより，実行情報の記録時におけるオーバヘッドの軽減を行っている．このシステムはデバッグを目的としているため，各ファイルごとに実行情報を閲覧するビューと，行を単位とした実行のトレース機能を提供する．一方，本研究は理解を目的としているため，ソースコード全体に対する包括的な実行部分を閲覧するビューと，関数呼び出しを単位とした実行のトレース機能を提供している．

久永らは GUI プログラムにおいて GUI 部品のインスタンスが木構造状に配置されることに着目したプログラム理解支援ツールを開発した[3]．このツールでは GUI 部品の木構造を 3D 表示したビューを用いて関数の呼び出しを可視化することで理解を支援する．GUI の表示を活用する点，関数の呼び出しを可視化するという点では本研究と同じである．一方，本研究は実行情報の把握により重点を置いた可視化を行っており，GUI 部品そのものではなく GUI 画面の変化に着目しているという点で異なる．

丹野はゲームプログラムのようなリアルタイム性の高いプログラムを対象としたデバッガを開発した[19]．このデバッガは我々の手法と同様にプログラムの実行を停止させることなくリアルタイムに実行されたソースコード行の強調表示を行う．このデバッガの大きな特徴は，強調色をグラデーション表示することで実行経路をより把握しやすくしている点である．本研究では 1 度に表示する実行情報の単位を操作ごとにしているのに対して，このデバッガではゲームのループ処理の切れ目となるフレームごとにしているという違いがある．このデバッガではリアルタイムな可視化に特化しているため実行情報の履歴は一定時間しか残さないが，本研究では過去の可視化履歴を再閲覧することができる．

中村らは GUI プログラムの操作履歴の可視化手法を提案した[20]．この手法では，画面のスナップショットを時系列順に並べて表示し，各画面に対しては矢印やラベル等で表現された操作履歴の注釈を付加する．このシステムでは，プログラムの実行を巻き戻すことが可能であるため，操作の様子をアニメーションで再生することもできる．このような表示を行うことで，GUI への操作と画面の変化を結び付けることができる．本研究とは，GUI への操作と画面変化を結び付けている点で似ている．本研究とは，画面に対して注釈を付けている点，ソースコードの実行情報を取得せず GUI 部分の可視化にのみ特化している点で異なる．

次に Java の動的実行情報の可視化システムの中で本研究と特に関連のある研究について述べる．

Reiss らによる JIVE[5] と JOVE[6] では，Java プログラムに対する動的な実行情報を可視化する．これらの研究においては，動的実行情報を扱う際のオーバヘッドを抑えながら，プログラムを理解するための可視化をすることを 1 つの目標としている．JIVE ではパッケージ・クラスレベルでの理解を目的とした可視化を行うために，1 つ 1 つのクラス（またはパッケージ）を矩形として表現し，それらを色づけすることで実行状態を表現する．一方，JOVE ではステートメントレベルでの理解のための可視化を行う．JOVE の表示は，対象プログラムのファイルに対応したいくつかの領域からなり，その領域はソースコード概略を表している．システムは，実行状態に応じてそれらの領域の大きさを変えながら，実行されたソースコード部分の色づけをする．これらの研究では，マルチスレッドプログラムに対する実行情報を可視化することにも主眼を置いており，実行時の色づけには実行スレッド情報も含まれている．これらの研究では，プログラム全体に対する実行部分を提示しているという点で本研究と似ている．一方，ソースコードの詳細なレベルでの理解を支援していない点や GUI 画面情報を可視化に利用しない点で本研究と異なる．

Gestwicki らによる研究[21),22)] では，オブジェクト指向におけるオブジェクト間の関係に着目した可視化を行う．その際に実行状態は UML のオブジェクト図およびシーケンス図を拡張した表現によって表示される．これらの研究では，GUI プログラムに特化していない

ために，操作や画面と実行されたソースコード部分の可視化の間で同期をとることが困難である．

本研究で提案した可視化手法の各可視化要素は文献 2) をはじめとする動的解析情報の可視化手法に近いが，操作・実行されたソースコード・画面変化の対応と実行されたソースコード部分の包括的な表示を行うことで，GUI プログラムの理解に特化した支援を行っている．

最後にアスペクト指向プログラミング[1] と本研究との関連性について述べる．グラフ表示部において提供されるソースコード行の強調表示は，GUI への操作を実装を抽出して表示する．この抽出されたコードはアスペクト指向プログラミングにおけるある 1 つのアスペクトを表していると考えられる．また本可視化手法は Bugdel[23] や AspectJ[24] 等のアスペクト指向用デバッガとあわせて用いることも可能であると考えられる．

10. 議　　論

本システムを利用する利点

評価実験の結果から，提案手法が GUI プログラムの理解作業に有効であることを示すことができた．特に GUI への操作と実行されたソースコード部分を結び付けるうえで本手法は効果的に働く．

また本システムでは可視化履歴を保存し，時系列順のアクセスを可能とするため，開発者がソースコードの実行トレースをリプレイすることも支援する．そのため本手法は，GUI プログラムのデバッグ用途でも活用できると考えられる．同一のイベントの反復的発行をともなう処理のデバッグ等，既存デバッガでは解析が行いにくい状況において本手法が特に有効であると考えられる．

本可視化手法は Java による GUI プログラムを対象としている．そのため，手法はオブジェクト指向に基づいているが，Java のみに限定されるような仕様はない．よって，本手法は他のオブジェクト指向言語によって記述された GUI プログラムに対しても流用することができると期待している．

本システムの利用形態

我々の可視化では，プログラムの操作に対する即座的な閲覧と，操作履歴を巻き戻しての再閲覧の両方を可能とするが，我々は後者を閲覧しながら理解作業を行うことを主として想定している．前者は操作にあわせてその場で可視化結果が更新されるため，可視化結果が示す操作場面を想起しやすいというメリットがある．しかし一方で，その結果は複数イベントが入力された場合等には，すぐに次のイベントの可視化が行われてしまうため，間のイベントの可視化結果を閲覧することは困難である．後者は一連の操作履歴を再閲覧するため，間のイベントの可視化結果についても確認が可能である．操作履歴を巻き戻しながら各可視化結果が指し示す場面を想起する際に，画面サムネイル表示の一連の画面変化の様子が効果を発揮すると考えている．我々が即座的な可視化を行っている意図としては，より操作場面と可視化結果を対応させやすくすることに加えて，可視化が逐次更新される様子から操作の間に発行されるイベント数や実行の流れが把握できることがあげられる．

被験者実験において，ORCA の利用方法についてビデオ分析を行ったところ，被験者の大多数はひととおりプログラムに対して操作を行ったうえで，操作履歴を巻き戻して再閲覧しながら理解作業を進めていた．また，即座的な可視化によって結果が更新される様子については，行った操作に対しておよそ何個分のイベントについての可視化結果が発行されたのかを確かめるために利用していた．このように，被験者の利用形態は我々の意図どおりであり，意図した利用形態でシステムを用いれば理解作業が可能であることが確認できた．

本システムの限界

現在の手法ではスケーラビリティ面でのいくつかの限界がある．

まず，可視化情報の表示に関するスケーラビリティについて述べる．本手法ではソースコードをクラス階層に従って 1 画面上に収めて表示する．木構造敷き詰めアルゴリズムでは，クラス階層が大規模になった場合や複雑になったときには，縦長や横長の領域が増えてしまい，十分な領域を割り当てることができないことがある．次に関数呼び出し表現について言及する．巨大なコールグラフや狭い範囲におけるコールグラフの表示を行ったとき，それらの呼び出し関係や順序関係を把握することは難しくなる．本システムではこれらの表示におけるスケーラビリティの問題を軽減するために，ズーミング表示機能を提供している．グラフ表示上での可視化情報を見るだけでは理解が難しい場合には，プログラムの操作に対する即座的な可視化表示での閲覧やズーミング走査を行うことで，それなりに対処は可能である．

また，本システムでは，8 章で計測したような実行時間におけるオーバヘッドが生じる．これらのオーバヘッドが操作への影響を及ぼす場合や，描画の厳密な更新のタイミングを知る必要がある場合等には，オーバヘッドの影響を軽減しなければならない．1 つの解決策としては，現在の実装のようにプログラムの操作に対する即座的な可視化を行うのではなく，まず実行情報の取得だけを行い，そのあとで可視化を行うように，実行方式を変更することが考えられる．ORCA による可視化は元々，実行を巻き戻して再閲覧しながら，理解を行

うことを想定しているため，この変更による支障はない．さらにパフォーマンスを向上させるための策としては，スケーラブルなデバッガ[25] を用いる等をして取得データ自体を減らす解決法や，柏村らの手法と同様に実行情報の取得を 2 フェーズに分けてしまう解決法を，ORCA に適応することがあげられる．

11. 結　　論

本論文では，GUI プログラムの理解作業を支援するために，GUI への操作と，操作によって引き起こされたソースコード実行部分と画面変化の 3 者を 1 画面上に表示する可視化手法について述べた．本手法は，GUI への操作をともなう機能を理解する際に特に効果的に働く．その効果は，今回行った被験者による評価実験の結果から確かめることができた．今後は，Eclipse の相互連動機能を増やしながら，理解作業と実行作業の包含的な支援を行っていきたい．

参 考 文 献

1) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *ECOOP '97: In European Conference on Object-Oriented Programming*, pp.220–242 (1997).
2) 柏村俊太朗，丸山一貴，寺田　実：Java プログラムを対象とする GUI 操作記録・再生型デバッグシステム，第 67 回情報処理学会プログラミング研究会 (2008).
3) 久永賢司，柴山悦哉，高橋　伸：GUI プログラムの理解を支援するツールの構築，日本ソフトウェア科学会第 17 回大会 CD-ROM (2000).
4) Eick, S.G., Steffen, J.L. and Sumner, Jr., E.E.: Seesoft: A Tool for Visualizing Line Oriented Software Statistics, *IEEE Trans. Softw. Eng.*, Vol.18, No.11, pp.957–968 (1992).
5) Reiss, S.P.: Visualizing Java in Action, *SoftVis '03: Proc. 2003 ACM symposium on Software visualization*, pp.57–65 (2003).
6) Reiss, S.P. and Renieris, M.: Jove: Java as it Happens, *SoftVis '05: Proc. 2005 ACM symposium on Software visualization*, pp.115–124 (2005).
7) Sato, T., Shizuki, B. and Tanaka, J.: Support for Understanding GUI Programs by Visualizing Execution Traces Synchronized with Screen Transitions, *ICPC '08: Proc. 16th IEEE International Conference on Program Comprehension*, pp.272–275 (2008).
8) 佐藤竜也，志築文太郎，田中二郎：実行の可視化システムと連動した統合開発環境による GUI ベースプログラムの理解支援，第 15 回インタラクティブシステムとソフトウェアに関するワークショップ論文集，pp.25–30 (2007).
9) 佐藤竜也，志築文太郎，田中二郎：GUI プログラムの理解支援のための可視化システム，日本ソフトウェア科学会第 24 回大会 CD-ROM (2007).
10) Furnas, G.W.: Generalized Fisheye Views, *CHI '86: Proc. SIGCHI conference on Human factors in computing systems*, pp.16–23 (1986).
11) Shneiderman, B.: Tree Visualization with Tree-Maps: 2-d Space-Filling Approach, *ACM Trans. Graphics*, Vol.11, No.1, pp.92–99 (1992).
12) Bederson, B.B., Grosjean, J. and Meyer, J.: Toolkit Design for Interactive Structured Graphics, *IEEE Trans. Softw. Eng.*, Vol.30, No.8, pp.535–546 (2004).
13) Storey, M.-A., Best, C., Michaud, J., Rayside, D., Litoiu, M. and Musen, M.: SHriMP Views: An Interactive Environment for Information Visualization and Navigation, *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pp.520–521 (2002).
14) Storey, M.-A., Wong, K., Mueller, H., Fong, P., Hooper, D. and Hopkins, K.: On Designing an Experiment to Evaluate a Reverse Engineering Tool, *Working Conference on Reverse Engineering*, p.31 (1996).
15) Storey, M.-A., Wong, K. and Muller, H.: How Do Program Understanding Tools Affect How Programmers Understand Programs, *Working Conference on Reverse Engineering*, p.12 (1997).
16) Ducasse, S. and Lanza, M.: The Class Blueprint: Visually Supporting the Understanding of Classes, *IEEE Trans. Softw. Eng.*, Vol.31, No.1, pp.75–90 (2005).
17) Linos, P.K., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. and Tulula, P.: Visualizing program dependencies: An Experimental Study, *Software: Practice & Experience*, Vol.24, No.4, pp.387–403 (1994).
18) Terada, M.: ETV: A Program Trace Player for Students, *SIGCSE Bulletin*, Vol.37, No.3, pp.118–122 (2005).
19) 丹野治門：ゲームプログラムに適したリアルタイム性の高いデバッガの提案と実装，情報処理学会論文誌：プログラミング，Vol.1, No.2, pp.42–56 (2008).
20) 中村利雄，五十嵐健夫：インタラクティブプログラムのための操作履歴視覚化手法，第 15 回インタラクティブシステムとソフトウェアに関するワークショップ論文集，pp.31–34 (2007).
21) Gestwicki, P. and Jayaraman, B.: Interactive Visualization of Java Programs, *IEEE CS International Symposium on Human-Centric Computing Languages and Environments*, p.226 (2002).
22) Gestwicki, P. and Jayaraman, B.: Methodology and Architecture of JIVE, *SoftVis '05: Proc. 2005 ACM symposium on Software visualization*, pp.95–104 (2005).
23) Usui, Y. and Chiba, S.: Bugdel: An Aspect-Oriented Debugging System, *APSEC '05: Asia-Pacific Software Engineering Conference*, pp.790–795 (2005).
24) Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Grisword, W.G.:

An Overview of AspectJ, *ECOOP '01: In European Conference on Object-Oriented Programming*, pp.327–354 (2001).
25) Pothier, G., Tanter, E. and Piquer, J.: Scalable omniscient debugging, *SIGPLAN Notices*, Vol.42, No.10, pp.535–552 (2007).

(平成 20 年 12 月 19 日受付)
(平成 21 年 3 月 23 日採録)

佐藤　竜也（正会員）

1984 年生．2005 年木更津工業高等専門学校情報工学科卒業．2007 年筑波大学第三学群情報学類卒業．2009 年筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程修了．現在，株式会社日立製作所中央研究所に勤務．情報の可視化，ヒューマンインタフェースに興味を持つ．日本ソフトウェア科学会，IEEE 各会員．

志築文太郎（正会員）

1971 年生．1994 年東京工業大学理学部情報科学科卒業．2000 年同大学大学院情報理工学研究科数理・計算科学専攻博士課程単位取得退学．博士（理学）．現在，筑波大学大学院システム情報工学研究科講師．ヒューマンインタフェースに関する研究に興味を持つ．日本ソフトウェア科学会，ACM，IEEE Computer Society，電子情報通信学会，ヒューマンインタフェース学会各会員．

田中　二郎（正会員）

1975 年東京大学理学部卒業．1977 年同大学大学院理学系研究科修士課程修了．1984 年米国ユタ大学大学院計算機科学科博士課程修了．Ph.D. in Computer Science．1984 年から 1992 年に富士通（株）に勤務，その間 1985 年から 1988 年に（財）新世代コンピュータ技術開発機構に出向，第五世代コンピュータ核言語の技術開発に従事する．1993 年から筑波大学に勤務．現在，筑波大学大学院システム情報工学研究科教授．研究分野としてはプログラミング言語やヒューマンインタフェースに興味を持つ．ACM，IEEE Computer Society，電子情報通信学会，人工知能学会，日本ソフトウェア科学会各会員．2007 年から 2009 年まで本学会理事．