筑波大学大学院博士課程

システム情報工学研究科修士論文

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

佐藤 竜也

(コンピュータサイエンス専攻)

指導教員　田中 二郎

2009年3月

概要

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 への入力操作と出力される画面を捉えながら、入出力それぞれに対応して実行されるソースコードを抽出する必要がある。しかし、操作と、それによって引き起こされる画面変化、ソースコード上に分散している実行部分を容易に結び付ける手段が存在しないことから、この理解作業を行うことが困難である。

本研究ではこれらの理解作業を支援するために 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、操作前後での実行画面のスナップショットと併せて表示する。このような表示を行うことで、一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い、Java の GUI プログラムを対象とした理解支援システム 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 Fisheye 表示を用いて拡大するズーミング表示を提供する。

さらに、作成したシステム ORCA を用いて被験者による評価実験を実施し、本手法の有効性を明らかにした。評価実験によって、本手法は GUI への操作を伴う機能の実装部分を抽出する際に特に効果的に働くことが示された。

目 次

図 目 次

表 目 次

第1章　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上で、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 プログラムには、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という特徴がある。その特徴から GUI プログラムを理解することはさらに困難となる。

以降ではまず GUI プログラムの特有の理解作業と既存ツールを用いて作業を行う上での問題点について述べる。その後、本研究で行う可視化の目的と貢献について述べる。

1.1　GUI プログラムの理解作業

上記の GUI プログラムの特徴から、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度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伴う。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 への入力及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わかる。例えば、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時、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捉えた上で、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 の役割毎に分けての実装される。さらに、GUI プログラムはオブジェクト指向に従っている。それ故、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。例えば、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ており、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。

複数イベントを伴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）を伴うものがある。複数イベントを伴う機能には、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通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。故に、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。

1.2　既存ツールを利用した GUI プログラム理解の問題点

プログラムの動的な挙動を理解するための手段として、デバッガ及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、先に挙げたような 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。ここで既存の動的解析ツールとその問題点について挙げる。

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。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として柏村ら [1] や久永ら [2] の研究がある。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[3] や Reiss らの研究 [4, 5] 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。

1.3 本研究の目的

本研究では、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 者を一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、理解の対象となるプログラムは Java によって書かれたものであるとした。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。

1.4 本研究の貢献

本研究では、GUI プログラムへの操作と画面情報を活用することで、特に操作に対して画面変化を伴うようなインタラクティブな機能の把握に適した可視化を行う。本手法では操作イベント毎に可視化の単位を区切るという表現を行うが、この表現が操作とソースコードの実行部分の対応付けに有用であることを評価実験により示した。

また本手法ではソースコード全体を一画面上に表示し、その上に色付けとエッジを重畳表示することでソースコード実行部分の可視化を行う。この表現がソースコード実行部分を抽出する上で有用であることを評価実験により実証した。

またソースコードの実行情報を関数呼び出しの単位毎に読み進められるように、ソースコード可視化表示の一部をズーミングする機能を備えている。評価実験の中で、この機能を用いれば実行されたソースコードを読み進めることが可能であることを示した。

以上の可視化の特徴により、従来のツールを用いた場合に生じていた GUI プログラム特有の理解作業の問題点を解決した。本手法を用いれば、GUI プログラムへの操作に基づいて理解作業を行うことができるようになる。本手法は特に、他人によって書かれた既存プログラムに対して初期理解を行う際に、理解対象となる機能の実装部分を特定する上で役立つと考えられる。

また本研究の可視化表現の中で、木構造データに対する 2 次元空間上への敷き詰め手法を提案した。本敷き詰め手法によって、従来手法では表示することが難しかった、木構造中の全ての要素がデータを持つ木構造に対しても可視化が行えるようになった。

1.5 論文の構成

本章では、GUI プログラムの理解作業と既存ツールを使用した場合の問題点について整理し、本研究の目的を示した。続いて 2 章では、関連研究を紹介する。3 章では、本章で示した GUI プログラムの理解作業を支援するための可視化手法の設計方針について述べる。4 章では、その方針に従って設計した可視化手法について述べる。5 章では、実装した理解支援システム ORCA について説明し、6 章ではその実装方法、7 章ではシステムの適応例について述べる。8 章では、ORCA の有用性を示すために行った評価実験について述べ、9 章ではその結果を元に議論する。最後に 10 章で、本研究についてまとめる。

第2章　関連研究

本研究と特に関連のある研究について以下に挙げる。まず、プログラム可視化の研究分野について説明する。その後、本研究と同様に GUI プログラムを対象としている動的情報の可視化の研究について述べる。さらに、本研究のソースコードに関する可視化表現手法と関連のあるプログラム可視化の研究について述べる。最後に、本研究と同様に Java プログラムを対象としている動的情報の可視化に研究について述べる。

2.1　プログラム可視化

ここではプログラム可視化について述べる。まず、大規模データや複雑な情報を人間のわかりやすい形で表示することを情報可視化といい、その 1 つの分野にソフトウェア可視化 [6] がある。ソフトウェア可視化は、プログラムの内部情報や、バージョン情報、プロジェクト情報といったソフトウェア開発に関わる情報の可視化全般のことである。プログラムの可視化はその中でもプログラムの内部情報を視覚的に表現するものを指す。プログラム可視化の目的はプログラムの理解やデバッグ等様々である。

プログラム可視化は静的情報の可視化と動的情報の可視化の 2 つに分類できる。静的情報の可視化では、プログラムの実行を行うことなくソースコードを解析した結果を可視化する。一方、動的情報の可視化では、プログラムの実行情報を元に可視化を行う。本研究はプログラムの動的情報の可視化に分類される。

以降は本研究と特に関連のあるプログラム可視化の研究を紹介する。

2.1.1　GUI プログラムを対象とした動的情報の可視化

柏村らは実行トレースの可視化システム ETV[7] を拡張し、GUI プログラムのデバッグを目的に特化した実行トレースの可視化システムを実装した。このシステムでは、GUI への入力操作の履歴、実行画面の変遷、実行されたソースコード行等を可視化する。このシステムでは操作時には GUI への操作情報のみを記録し、その後で記録した操作を元にプログラムの実行を自動再生して実行トレース情報を採取している。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図っている。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。そのため、ソースコード実行部分の抽出と理解のために時間がかかってしまう。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。これにより、ソースコード実行部分の抽出と理解のための作業コストの減少を図っている。

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 [2]。このシステムでは 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。我々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。そのため、このシステムでは複数イベントについて画面変化の流れの中から理解することは難しい。

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 [8]。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。本研究では一度に表示する実行情報の単位を操作毎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毎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ため、複数イベントの理解を行うことは難しい。本研究ではその問題に対処するために過去の可視化結果の履歴を再閲覧することを可能としている。

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 [9]。この手法では、実行画面のスナップショットを時系列順に並べて表示し、各実行画面に対しては矢印やラベルなどで表現された操作履歴の注釈を付加する。操作の様子はアニメーションで再生することが可能である。このような表示を行うことで、GUI への操作と実行画面の変遷を結びつけることができる。本研究とは、GUI への操作と画面遷移を結びつけている点で似ている。本研究とは、実行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。このシステムは、操作と画面を結びつけることに長けているが、それらをソースコード実行部分と対応付けることは難しい。

2.1.2 ソースコードに関する可視化表現手法

ソースコードの表現方法としては、図的表現とソースコードベースでの表現の 2 種類が考えられる。図的表現とは、ソースコードを図で抽象化したもので、UML がその代表的な例である。UML はオブジェクト指向によるプログラム設計図の統一記法である。UML では状況に応じて複数種類の図が用いられるが、そのうちプログラムの動的な側面を表現する図としては、オブジェクト図、シーケンス図、コラボレーション図がある。UML の各図はしばしばプログラム作成前の設計図として作成されるが、逆にプログラムの実行情報から UML の図を自動作成することもできるので、これらをプログラム理解に用いることができる。UML を元にした可視化の研究も試みられているが、それについては次小節で詳細に述べる。

一方、ソースコードベースでの表現とは、ソースコードそのものを活かした表示である。ソースコードベースの表現はプログラム可視化の分野でこれまで研究されてきており、その代

表的な研究として Seesoft[3] がある。Seesoft では、ソースコード全体を一画面上に縮小表示する。その際、ソースコードはファイル毎に区切って表示し、ソースコードの各行は 1 本の線として表現される。Seesoft では、ソースコードの更新履歴等といった関心事を、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。さらに Seesoft の 1 つの適応例である SeeSlice[10] では、動的なプログラムスライスの可視化を行っている。Seesoft のような表現を用いることで、ソースコード全体から関心事を把握することができる。本研究では、ソースコード全体をファイル毎に区切りソースコードの各行を縮小して表示するという、Seesoft とよく似た表現を用いている。このような表現を用いてプログラムの実行を可視化しているという点では SeeSlice と同じである。本研究ではさらに各ファイルをクラス階層に従って配置することで、よりオブジェクト指向言語に特化した可視化を行っている。

またソースコードベースでの表現に近い手法として、ソースコード全体の概略表示に対して色分けによる強調を行ってプログラムの内部情報を表現する手法について述べる。Ball と Eick らは、大規模なプログラム全体を、ソースコード全体の概略的な表現を用いて可視化するいくつかのスケーラブルな表示手法を提案した [11]。前述の Seesoft もこの中の研究の 1 つである。これらの表現手法は、ソースコードの更新履歴や更新差分、ソースコードの静的な特性、ソースコードのプロファイリング、プログラムスライス等の可視化に適応されている。

Reiss らによる JIVE[4] と JOVE[5] では、Java プログラムに対して動的な実行情報を可視化するためにソースコードの概略的な表現を用いる。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 [4] を行うために、1 つ 1 つのクラス (またはパッケージ) をボックスにより表現をし、実行状態をそれらのボックスを色付けすることで表現する。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 [5] を行う。JOVE の表示は対象プログラムのファイル毎に横に並んだいくつかの領域から成り、その領域はソースコード概略を表している。システムは、実行状態に応じてファイル領域のサイズを変えながら、ソースコード実行部分の色付けをする。これらの研究では、特に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、実行時の色付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、一画面上で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で本研究と異なる。

膨大なソースコードの実行情報を閲覧する手法としては、プログラムの構造に対してズーミングを行うことが有望であるが、その代表的な研究として SHriMP がある [12]。SHriMP はプログラム構造とソースコードを閲覧しながらインタラクティブにブラウジングすることができるシステムである。大きく複雑なプログラムを検索可能にするために、プログラム構造をネストグラフを使った入れ子構造で表す。例えば、パッケージの中にいくつかのファイルがあり、その中にクラスがあり、さらにその中にフィールドやメソッドがあるといった情報を入れ子構造として表現する。その入れ子構造をズーミングしながらブラウジングすることで、下の階層の構造やソースコードなどを閲覧することができる。本研究でも SHriMP と同様にソースコード閲覧に対するズーミングを行っている。本研究ではプログラムのクラス階層の木構造に着目したズーミングを用いて可視化と実行のトレースを行っている。

2.1.3 Java プログラムを対象とした動的実行情報の可視化

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。

Gestwicki と Czyz らによる研究 [13, 14] 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 のオブジェクト図及びシーケンス図を拡張した表現によって表示される。

Jeliot 3 では Java プログラムのデータフローやコントロールフローの様子を、UML によく似た記法を用いて可視化する [15]。実行の流れはアニメーションしながら提示する。ソースコードの解析にはオープンソースの Java インタプリタを用いており、可視化はソースコードを確認し実行しながら行うことができる。Jeliot 3 での可視化の目的は初心者の Java プログラミング学習である。

JAN はプログラム理解のための Java 実行アニメーションを提示するシステムである [16]。JAN では UML のオブジェクト図とシーケンス図をプログラム実行をトレースする形でアニメーション表示する。ここで使われるオブジェクト図は Java の Array や Collection などのデータ構造に合うように拡張されている。このシステムでは可視化のためにソースコードにアノテーションを付加することが必要である。

谷口らは Java の実行履歴から UML のシーケンス図を作成する手法を提案した [17]。この研究では、実行履歴からループなどの繰り返し処理部分を抽象的な表現に置き換えることで、圧縮された実行情報を提示している。

本小節で紹介した研究では、ソースコードの実行について把握することは可能であるが、GUI プログラムの特徴である操作や画面の情報は可視化しないため、操作や画面と実行されたソースコードを結びつけることが困難である。

2.2 本研究の位置付け

本研究で提案した可視化手法の各可視化要素は [1] をはじめとする動的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。

第3章　可視化の方針

GUI プログラムに特化した理解支援を行うための可視化の方針について説明する。我々は 1 章で挙げた GUI プログラムの理解に必要な 3 つの作業を支援するような可視化を行う。必要な理解作業を以下に再掲する。

理解作業 1 操作・実行されたソースコード・画面変化の対応付け

理解作業 2 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

理解作業 3 複数イベントを伴う機能の実装部分の把握

3.1　操作・画面変化・実行されたソースコードの対応関係の可視化表現

操作と、それによって引き起こされた画面変化、実行されたソースコードの 3 者の対応関係をしやすくするために、これら 3 つの情報を併せて可視化する。まず可視化の単位を操作毎に区切り、操作毎にまとまった実行情報として閲覧可能にする。可視化する実行情報は、操作イベント名、画面スクリーンショット、ソースコードの実行トレースをそれぞれ操作、画面変化、実行されたソースコードを表す情報として提示する。これにより、ユーザは操作が起こった時点での画面変化と実行されたソースコードを把握可能になる。

3.2　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可能にするために、ソースコード全体に対するソースコード実行部分の提示を行う。本手法では、2 章で紹介した図的表現とソースコードベースでの表現のうち、後者を用いてソースコード実行部分を可視化する。本研究が対象とするプログラム理解では、操作に応じて動的に実行されたソースコードを把握する必要があることから、よりソースコードに近い表現が適切であると考えたためである。さらに本手法では Seesoft と同様にソースコード全体を一画面上に表示する可視化表現を用いて、その全体表示の上にプログラムの実行情報を併せて表示する。このような表現を用いることで、ソースコードの実行部分をソースコード全体の概略の中から抽出できるようにする。

3.3　複数イベントの把握のための可視化結果に対する時系列アクセス

複数イベントの把握のために、任意の可視化結果を時系列の中から取得可能にする。複数イベントを把握する際には、時系列順に発行されたイベントの中から理解に必要なイベントを取り出し、そのイベントの実装について知るという手順で作業を行う。各イベントの実装を知るための可視化表現は、それぞれ 3.1、3.2 節で示した通りである。そのためこれらに加えて、必要なイベントを取り出すための手段を提供する。まず操作毎の可視化結果を履歴として保存する。可視化結果の履歴は時系列順に並べて提示して順次アクセス可能とする。時系列順に並べることで、操作イベントの発行順と画面変化のコンテキストの中から必要なイベントを選出することができるようになる。これにより、選出した任意イベントのみの可視化結果だけを閲覧していくことが可能になる。

第4章　可視化手法の設計

3 章で述べた方針に従って設計した可視化手法について述べる。

4.1　操作・画面変化・実行されたソースコードの対応関係の可視化表現

図 4.1 に操作・画面変化・実行されたソースコードの 3 者の対応関係を表現する可視化の模式図を示す。画面変化を表すスクリーンショットとして、GUI への操作が行われる直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を並べて表示する（図 4.1 下部）。以降、この並べて表示される 2 つの画面サムネイルをサムネイルペアと呼ぶことにする。サムネイルペアの上には、操作のトリガとなった関数名をイベント名として重畳表示する。サムネイルペアと併せて、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 4.1 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 への操作が行われると、その操作に対する実行情報を即座に可視化する。一通りの更新が終了した後は可視化結果を静的に閲覧することができるようにする。

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伴う機能の理解作業を支援する。

4.2　ソースコード実行部分の包括的表現

3.2 節で述べたように、本手法では Seesoft と同様にソースコード全体を一画面上に表示する。さらに、本手法でもまたソースコードをファイル毎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 4.2 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一画面上に敷き詰めて縮小表示する。

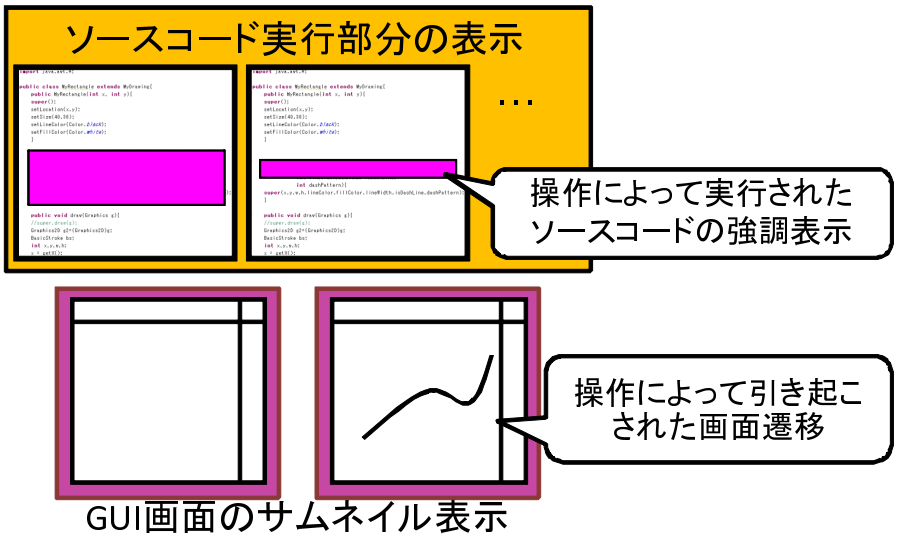


図 4.1: 操作・実行されたソースコード・画面変化の対応関係の可視化表現

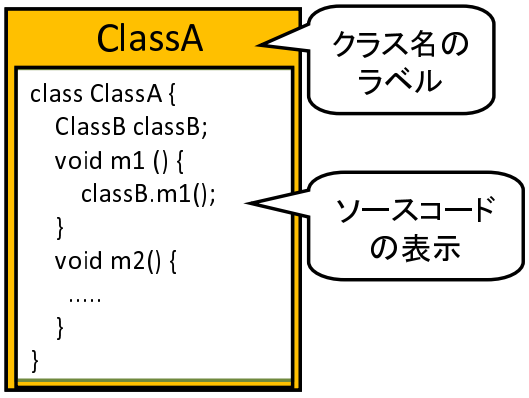


図 4.2: クラス定義の表現

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併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
- 関数呼び出し
- クラス階層

以下にこれらの各実行情報を提示する理由と表現方法について詳細に述べる。

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 4.3 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付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つ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 4.3 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さ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 部品やプログラム中で定義される描画対象は、継承を使って実装されることが多い。例えば、ドローイングツールでの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必要がある。

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一画面上に収めて表示する。そのため、表示領域内にすべてのクラス定義を木構造に従いながら敷き詰める必要がある。本研究では、上に親、下に子が来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

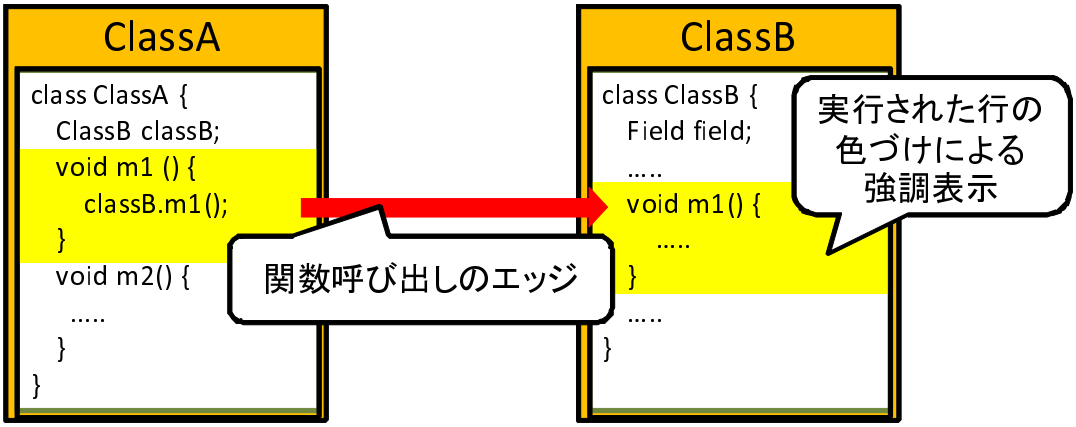


図 4.3: 動的解析情報の表現

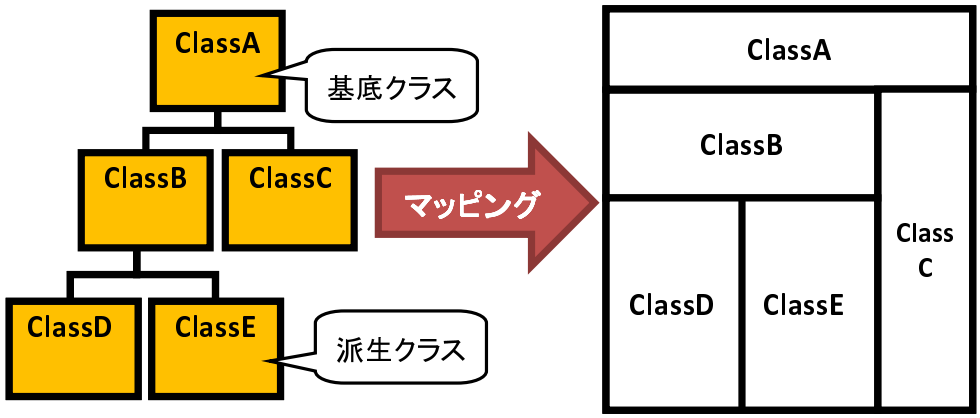


図 4.4: クラス階層の表現

き詰めを行う。図 4.4 に木構造に従ったクラス階層の表現を示す。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来るような木構造表現に従った割り当てを行う。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、図 4.5 に示すように、各クラス階層の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。

4.2.1 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一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

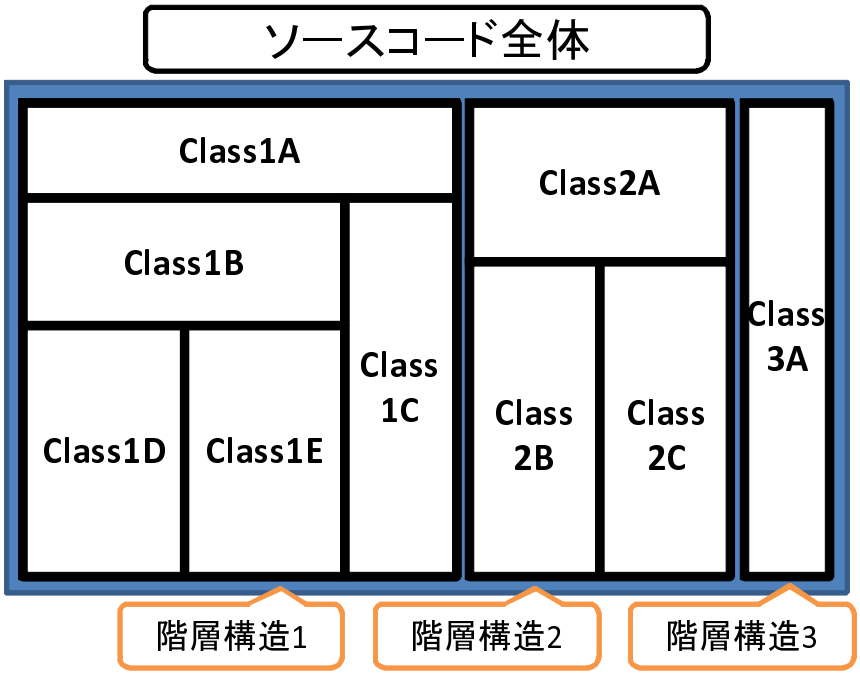


図 4.5: 複数のクラス階層構造があった場合のソースコード全体の表現

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づらくなる。

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。具体的には関数呼び出し毎に焦点を当てられるようにして、注目している関数呼び出しの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、ズーミング手法の一つである Fisheye 表示 [18] のうち木構造に特化したものを作成して用いる。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常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。今回作成した 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ことにした。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等が近いクラスにも高い重要度を割り当てるものとした。

以上を踏まえた上で、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 2 次元空間に敷き詰めて表示する手法である TreeMap[19] 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。

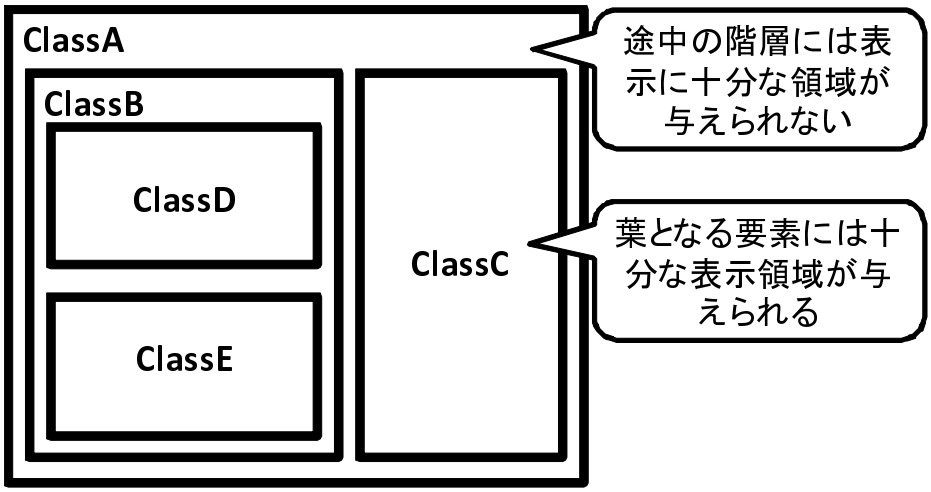


図 4.6: TreeMap を用いた場合のクラス階層表現

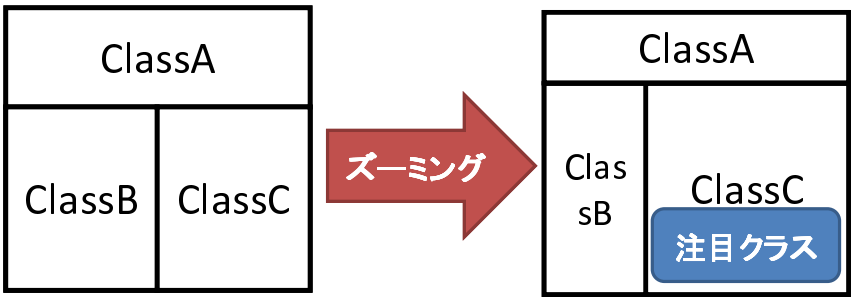


図 4.7: 重要度に基づくズーミング

TreeMap では木構造を入れ子構造とみなして、領域分割を再帰的に繰り返すことで、領域割り当てを行う。それ故、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（例えば、図 4.4 の左に示すクラス階層構造を単純な TreeMap で可視化すると図 4.6 のようになり、親である **ClassA** と **ClassB** には十分な領域が割り当てられない）。

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、下に子が来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 4.4 参照)。

さらに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 4.7 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り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** を重要度の高いクラスとみなすことで **ClassC** に対してより大きな表示領域が割り当てられている。

以上のようにクラス階層のズーミングを用いて各クラス定義に大きな領域を割り当てても、

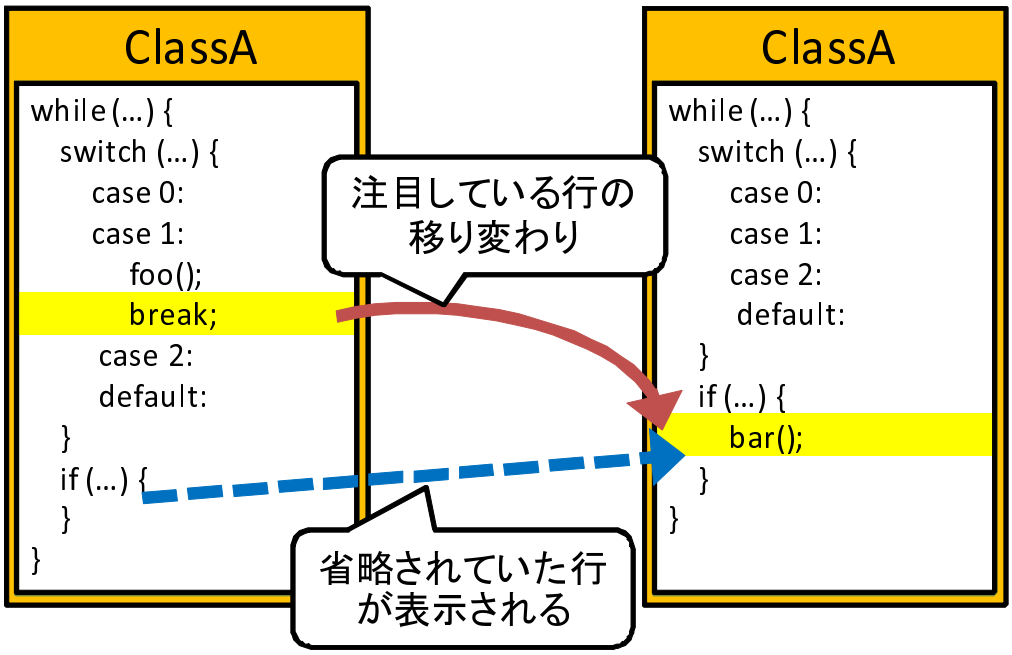


図 4.8: Furnus によるソースコードズーミング

1 つのクラスを表示する領域は限られている。そのため、クラスを定義するソースコード行が多いとき、領域内にすべての行を単純に納めようとすると文字フォントをかなり小さくしなければならない。これではソースコード可読性の問題を解決できない。

そこで各クラス定義のソースコードに対しても、Fisheye 表示を用いたズーミングを行う。すなわち、注目しているソースコード行付近の文字フォントを拡大しそれ以外の行の文字フォントを縮小する。

既に Furnas は [18] の中で、段付けされたプログラムの各行を木構造とみなした Fisheye 表示を行っている。この中では、現在注目している行の前後数行と、注目行からの距離は離れているがプログラムの制御構造を知る上で重要な行（例えば for 文，if 文， switch 文等）の同時表示を行う。その間にある重要でない行に関しては表示を行わない。この表示法の欠点は、注目行を移動した時の表示変化が激しいことである。図 4.8 はその表示例である。これは 1 ステップでの表示変化を表している。プログラム構造が複雑である場合には図左のように次ステップでの実行される行が表示されないことがある。このような場合、ステップを切り替えると図右のように表示されていなかった行が突然表示されるため、ユーザに大きな認知負荷を与えてしまう。

この問題を解決するために小池らは [20] の中でマルチスケーラブルフォントでの表示を提案した。この手法では複数の異なるサイズの文字フォントを用いてソースコードを表示する。各行のフォントサイズはその行が持つ重要度に従って決められる。この手法では基本的に表示が隠される行がないので、連続的変化を認識しやすい。

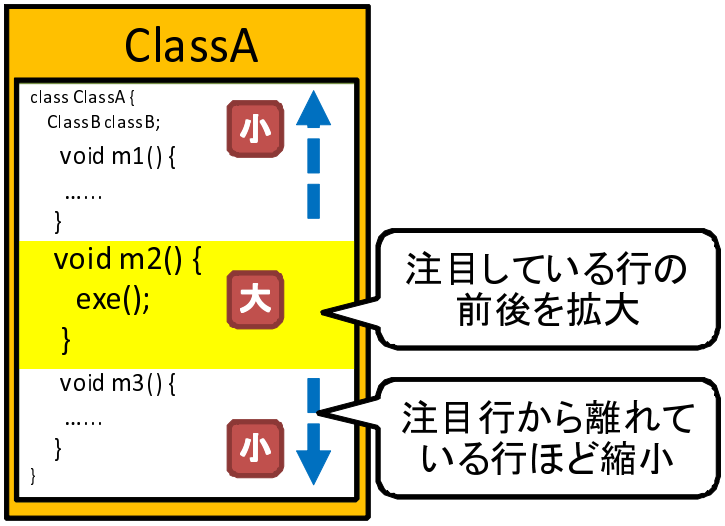


図 4.9: マルチスケーラブルフォントによるソースコードズーミング

本手法ではよりソースコードの流れが追いやすくなるように、Fisheye 表示に対して小池らの提案したマルチスケーラブルフォントによる表示手法を適応する。現在注目している行の前後行では重要度を高く、距離が離れるに従って行の重要度を低く設定し、重要度に従った異なるサイズの文字フォントを用いて表示を行う。図 4.9 にマルチスケーラブルフォントでのズーミング表示を示す。このように注目している行が最も大きなフォントで、距離が離れた行になるほど小さなフォントで表示が行われる。

第5章　GUIプログラムの理解支援システム ORCA

設計した可視化手法に基づいて実装した理解支援システム ORCA（Operation Reaction Code Analyzer）[21, 22, 23, 24] について述べる。

5.1　画面構成と情報提示方法

システムの概観を図 5.1 に示す。システムはコントロール部、グラフ表示部、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及び実行情報を可視化する（図 5.1 グラフ表示部参照）。このようにソースコード全体を一画面上に収まるように縮小して表示する。以降、グラフ表示部に表示される可視化結果のことをグラフ表示と呼ぶことにする。ここでソースコードに重畳表示される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 5.2 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 切り替え機能を用意した。図 5.3 は図 4.4 のクラス階層を実際にマッピングしている様子である。4.2 節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わかる。また各イベントの呼び出し開始点が確認できるように、開始点となる行の先頭位置には緑色の円弧を表示している。図 5.1 のグラフ表示部では **NetworkPanel** クラスのソースコード上に開始点が表示されている。

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 5.1 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桃色の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）が併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毎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 つの操作であるとみなして、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

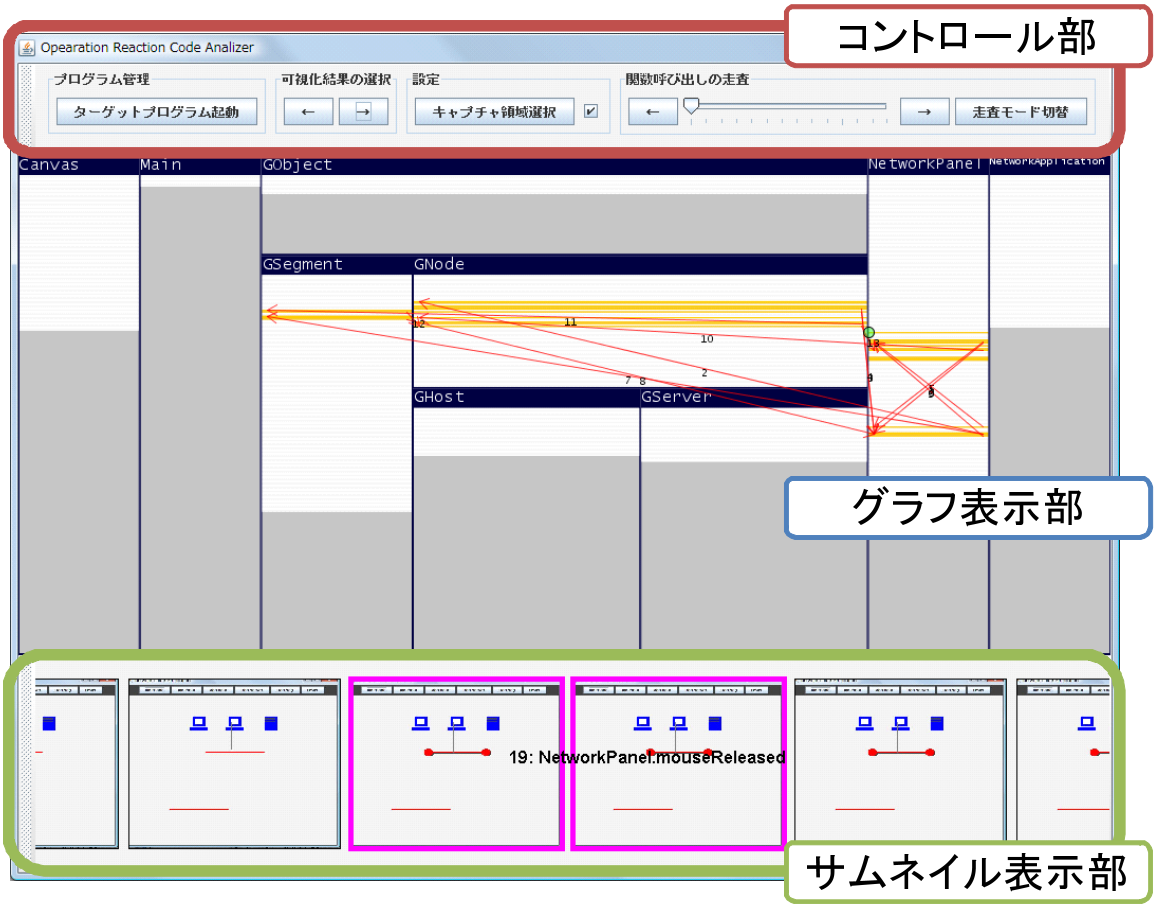


図 5.1: GUI プログラム理解支援システム ORCA

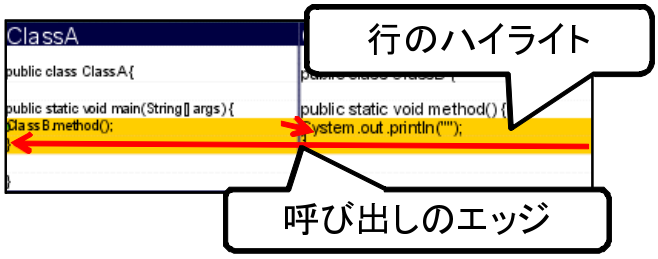


図 5.2: ORCA における呼び出しエッジ表示と強調表示

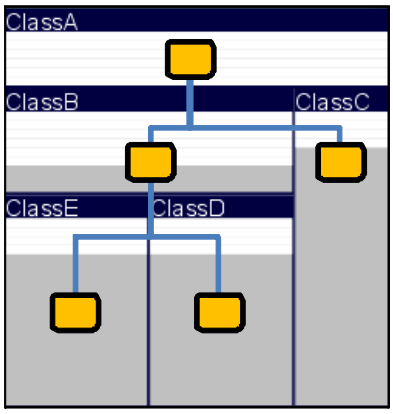


図 5.3: ORCA におけるクラス階層表示

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。

図 5.1 で示されるシステムの概観は図 5.4 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。

ユーザ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 5.1 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毎の関数呼び出しのエッジを辿る機能である。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。ORCA の操作方法の詳細については論文末尾の付録 2 を参照されたい。

5.2 関数走査機能

ORCA は、操作毎の可視化表示結果に対して、ソースコードの実行情報を関数呼び出し毎にソースコードを順々に辿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。この機能では、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。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可能になる。ソースコードの詳細表示方法には、4.2 節で述べたズーミング表示を用いる方法と関数呼び出しが起こった行付近のソースコード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 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

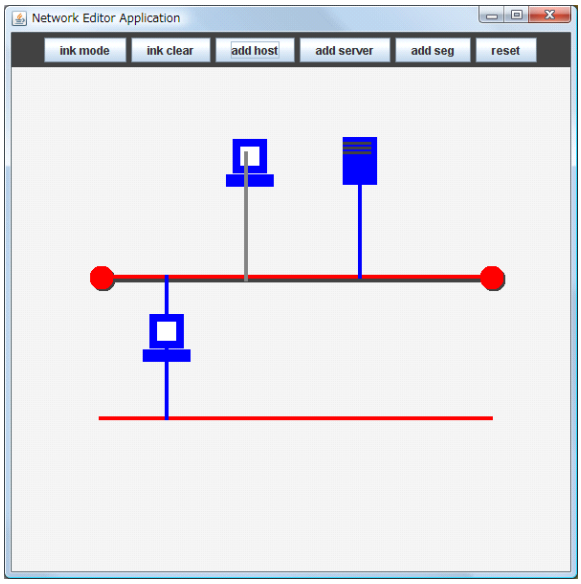


図 5.4: 対象とする GUI プログラム (ネットワーク構造図エディタ)

と呼ぶ。

5.2.1 ポップアップ走査

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である行付近のソースコードをポップアップとして表示する。ポップアップには、呼び出しが起こったクラス名とメソッド名も併せて表示する。ポップアップに表示されるソースコードは数行程度であるが、ポップアップ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 5.5 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。ポップアップ中での表現としてソースコードに対して、呼び出しがあった行には「>>>」がそれ以外の行には「>」を各行の先頭に付加する。

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が隠されてしまうことを軽減することができる。ポップアップ表示に、グラフ表示における強調表示の配色を反映させるのではなく、単色を用いた理由は、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辿ることができるというメリットがある。

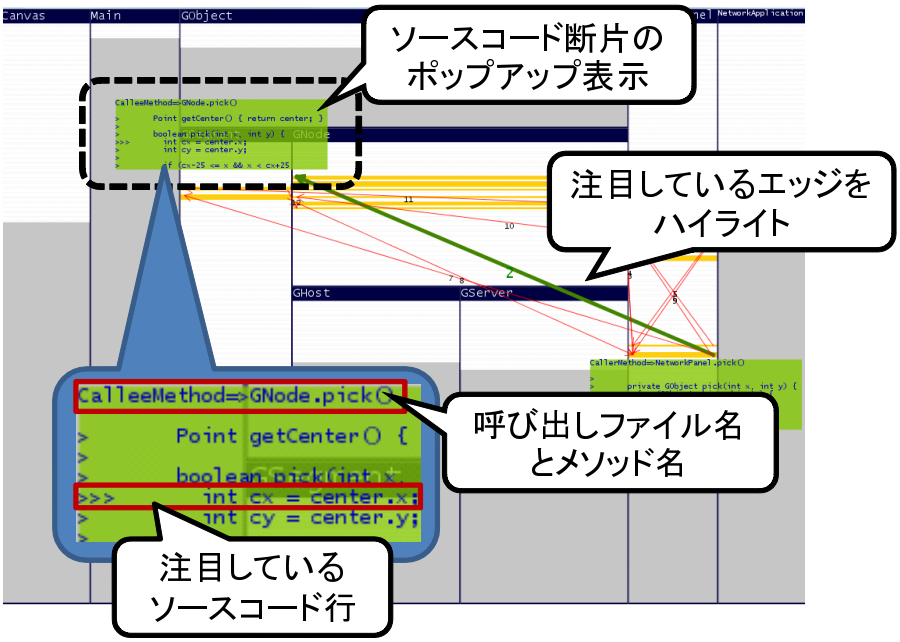


図 5.5: ポップアップ走査

5.2.2 ズーミング走査

ズーミング走査を行っている様子を図 5.6 に示す。今、**クラス A** の関数から**クラス B** の関数が呼び出されている。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。

図 5.7 は、順次、ズーミング走査を行ったときの例である。ここではクラスが **A** → **C** → **B** → **B** の順に実行されている様子を表している。それぞれの図中の **A**、**B**、**C** は実行しているクラスや行に応じて、表示領域の大きさが異なっている。例えば、シーン 1、シーン 2 では、**B** はその時点で注目している関数呼び出しには関与していない。そのため、重要度が低いクラスとみなされ、ソースコードが見えないほど縮小して表示されている。このようにズーミング表示では注目しているクラスや行に応じて拡大部分や拡大率が大きく異なっていることがわかる。ポップアップ走査では、関数呼び出しが起こった行にしか実行情報を付加しないためそれ以外の行の実行は確認できないが、ズーミング走査では実行された行に対して色付けによる強調表示を行うため、関数呼び出しが起こった行以外の行に対しても実行の有無について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。

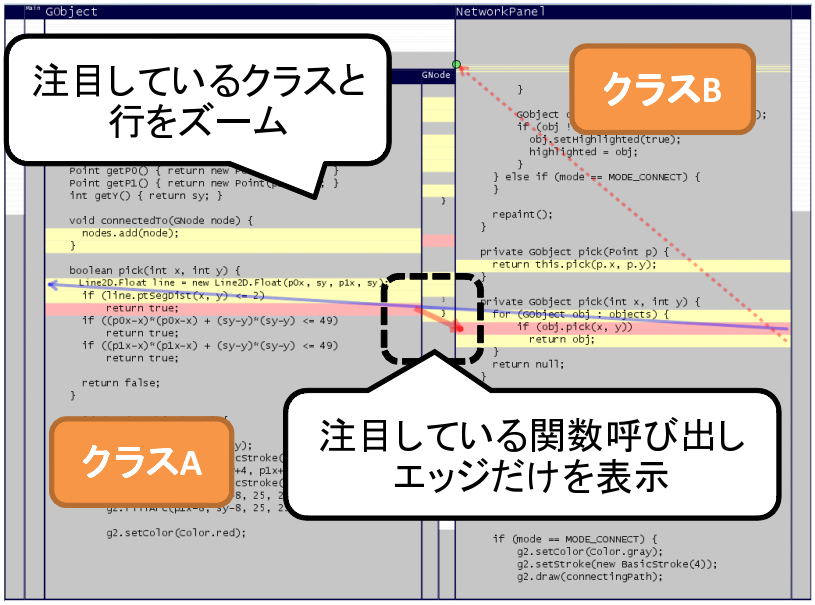


図 5.6: ズーミング走査

5.3 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 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 のデバッガを用いればプログラム実行時における変数の値等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取り掛かることができる。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情報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。

ORCA は Eclipse に統合されているため、ORCA の起動は Eclipse 上から行う。起動を行っている様子を図 5.8 に示す。Eclipse 上で Java プログラムプロジェクトの中から対象となるプログラムの main メソッドを持つクラスを選択し、ポップアップメニューから ORCA の起動メニューを選択することで、ORCA が起動される。

次に ORCA の表示と Eclipse との連動機能について述べる。この機能にはまず 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 (図 5.9 参照)。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

[1] http://www.eclipse.org/

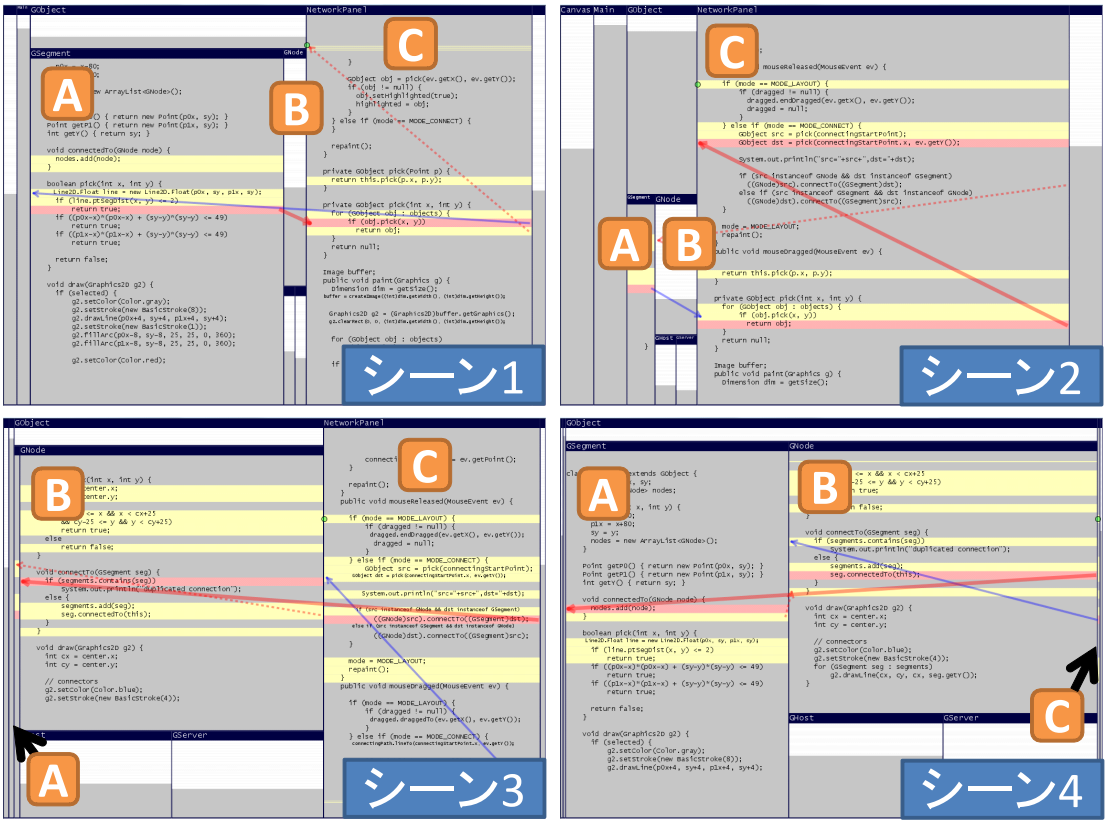


図 5.7: 連続したズーミング走査の様子

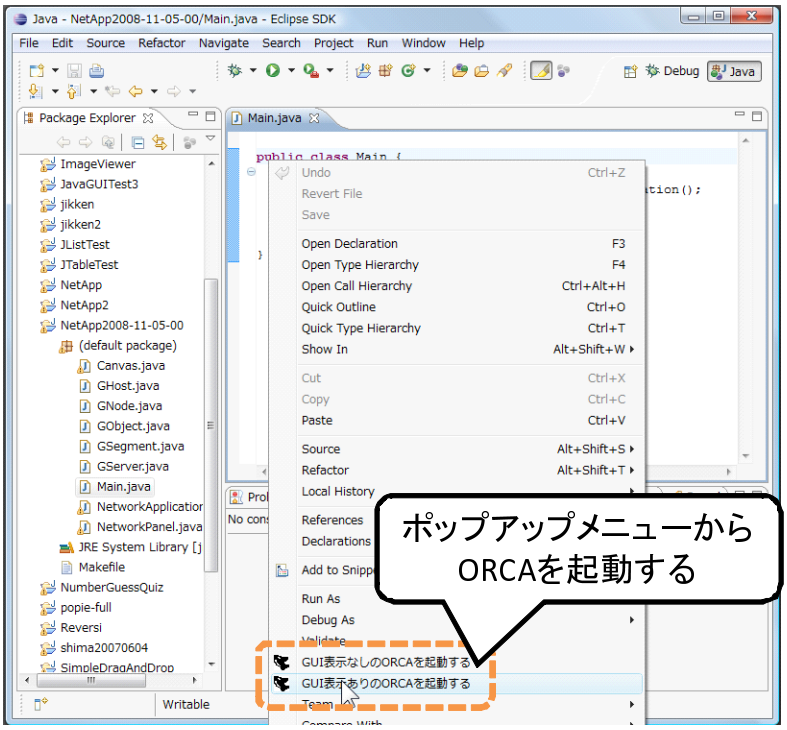


図 5.8: Eclipse からの ORCA の起動

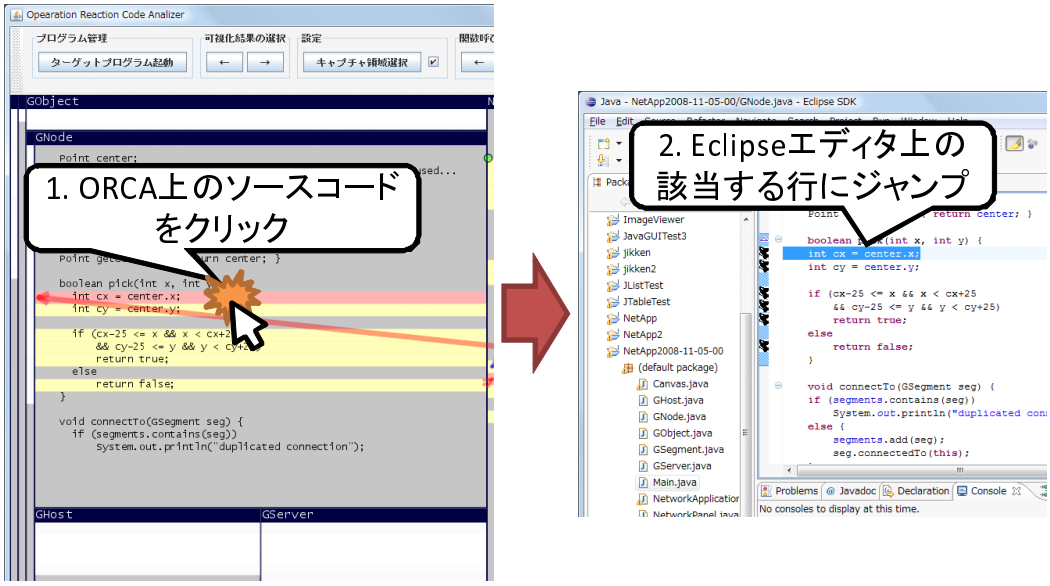


図 5.9: ORCA から Eclipse エディタ上へのソースコードジャンプ機能

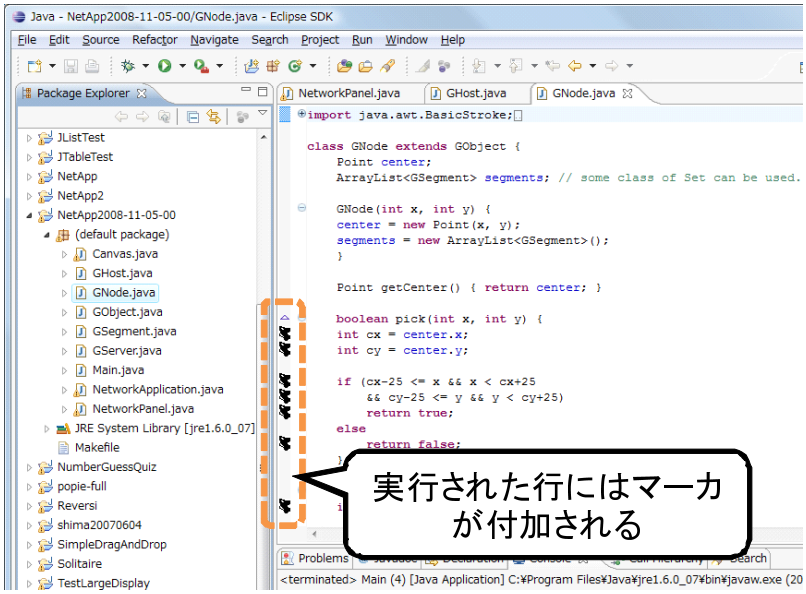


図 5.10: Eclipse エディタ上へのマーカ表示機能

る。図 5.10 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毎のソースコード実行箇所がわかる。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。なお、現在はこのマーカに対して Eclipse 側から操作を行うことは禁止しており、ORCA 上で可視化結果を選択して切り替えた時のみマーカの表示が切り替わる。

第6章　実装

ORCA では Java によって記述された GUI プログラムを理解の対象としているが、ORCA 自体もまた Java によって開発されている。まず ORCA のシステム構成と実装に用いた要素技術について述べる。その後、可視化表示を行う上でのアルゴリズムについて述べる。なお、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。

6.1　システム構成

図 6.1 に ORCA のシステム構成を示す。ここで角丸矩形は ORCA のプログラムの各役割部分を表し、その中の矩形は保持されるデータを表している。ORCA ではまずソースコードから GUI プログラムの静的情報を取得する。対象となる GUI プログラムが起動された後で、ユーザが GUI 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、その場で動的に GUI プログラムの実行トレースとスクリーンショットを取得する。可視化表示部ではこれらの取得した動的情報を元にソースコード実行情報と画面変化の様子を可視化する。

6.2　実装に用いた要素技術

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[25]（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[26] を使用した。

ここで動的情報をイベント毎に区切る方法について述べる。まず動的情報は以下のタイミングで取得する。

- 対象プログラム内で定義されたメソッドの呼び出し
- 上記のメソッド内でのソースコード行のステップ実行
- 上記のメソッドの終了

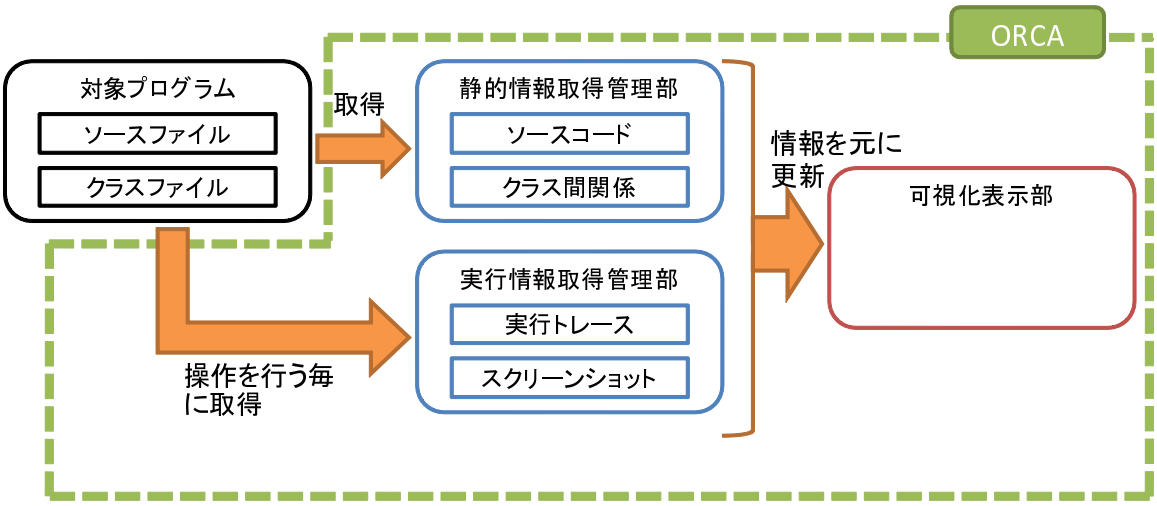


図 6.1: ORCA のシステム構成

メソッドは呼び出されると、その中で呼び出されるメソッドを階層的に辿り処理を行った後に、元のメソッド呼び出しに戻って来ることにより終了する。その特徴を利用して、ORCA でははじめのメソッド呼び出しからそのメソッドの終了までを 1 つの操作イベントの実行情報として扱うことにした。

画面サムネイルの取得に関しては Java の AWT パッケージの Robot クラスの機能を利用した。Robot クラスはネイティブなシステム入力イベントを生成することができる。本システムではその一部である画面キャプチャ機能を利用している。画面のキャプチャ領域の範囲は、標準ではスクリーン全体としているが、ユーザがキャプチャしたい画面の範囲を指定することも可能である。標準でスクリーン全体をキャプチャしている理由は、プログラムが別のアプリケーションによって引き起こされた画面変化や、ウィンドウの最大化、最小化といったイベントに対する処理に対しても対応できるという利点があるからである。

可視化表示のレンダリングには Piccolo.Java1.2[27] を利用した。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。

またプラグインの実装には、Eclipse のプラグイン開発環境（PDE）[28] を利用した。現在のプラグインは Eclipse のバージョン 3.3 に対応している。

6.3 ズーミング表示アルゴリズム

6.3.1 木構造敷き詰めアルゴリズム

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。

手順 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

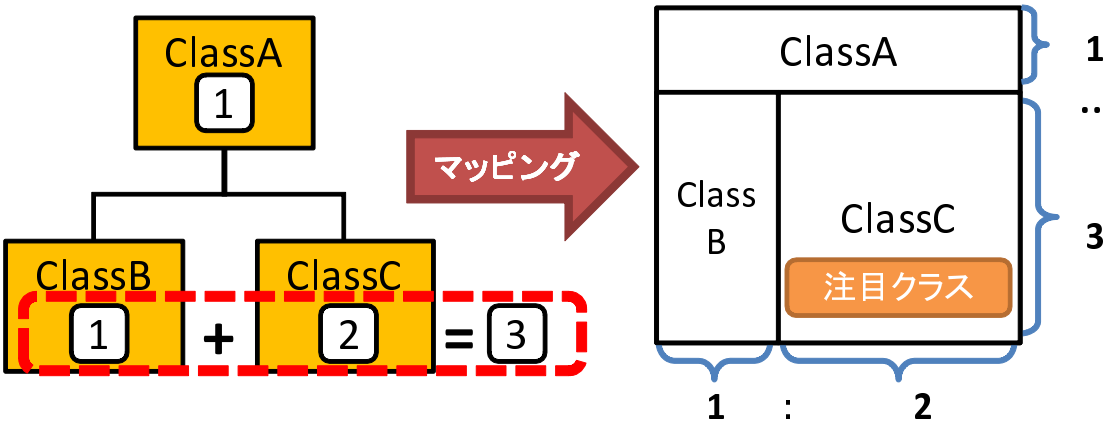


図 6.2: 木構造敷き詰めアルゴリズム

手順 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

手順 3. 以下は各子クラス部分木に対して、手順 1 と 2 を再帰的に繰り返す

図 6.2 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現在各クラスに割り当てられている重要度とする。**ClassC** の重要度がもっとも高いことから、このクラスが今最も注目したいクラスである。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。もしクラス階層が複数個あった場合には、初めに各クラス階層全体の重要度の比で領域を横に分割した後で、各クラス階層の木に対する領域の割り当てを行う。

ここで重要度の計算方法について述べる。Furnas の Fisheye 表示では木の潜在的重要度 (API) と焦点からの距離 (D) を用いた以下の式に基づき木の各ノードの重要度 (DOI) を決定する。

$$DOI = API - D \tag{6.1}$$

三末らは上記の Fisheye における重要度を視点、構造、意味の 3 属性によって決定づけることによって、さらに多様な要求に対応可能な手法を提案した [29]。本研究で扱うクラス間の関係は、単純なクラスの親子関係だけでなく、インタフェースの実装関係や関数呼び出し関係といった親子関係以外の要素を含んでいる。そこで我々の手法では、三末らの手法と同様に複数属性によって重要度を決定する。各クラスの重要度を求める計算式には以下を用いた。

$$I = W_s * F_s(I_s) + W_m * F_m(I_m) \tag{6.2}$$

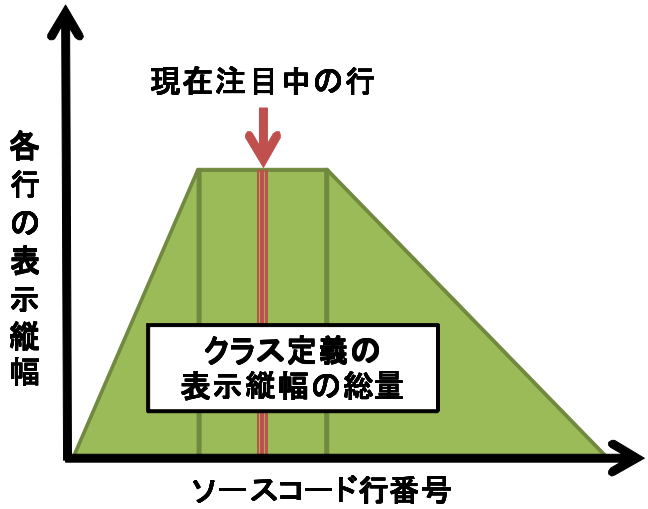


図 6.3: ソースコードのズーミングアルゴリズム

ここで I は重要度、I_s は構造的重要度、I_m は意味的重要度を表す。W_x と F_x はそれぞれの重要度の重み係数と関数を表す。なお各要素は 0〜1 の間で値をとり、W_s と W_m の間には $W_s + W_m = 1$ という関係がある。構造的重要度は、各クラスの親子関係による重要度を示す。ORCA では注目しているクラスがあった場合、親等の近いクラス (親クラスや子クラス) は注目すべきである情報であるという考えに基づき、構造的重要度を振り分けている。意味的重要度はクラスの親子関係とは切り離されたクラス間のつながりによる重要度を示す。現在の ORCA では特に関数呼び出しの順序関係を元に、各クラスの意味的重要度を振り分けている。

6.3.2 ソースコードのズーミングアルゴリズム

各クラス定義内のソースコードに対しては簡易的な Fisheye ズーミングを実装した。現在は各行の表示の縦幅を、図 6.3 に示すような注目行周辺を最大値とした線形関数に従った領域に分割する。もし 1 つのクラス表示内に注目行が 2 箇所あった場合には、各注目行周辺を最大値とした 2 こぶの線形関数に従って領域を分割する。各行の文字フォントサイズは領域に文字列が収まりきるように決定する。現在は、縦幅に収まりきるフォントサイズと、横幅に表示するソースコードの文字列が収まりきるフォントサイズを比較し、小さいほうのフォントサイズを表示に利用している。現在の ORCA の実装では、注目している行の縦幅が 12pt の文字を十分表示できるように線形関数の最大値を設定して領域分割を行っている。

第7章　適用事例

ORCA を用いたプログラム理解作業の適応事例について述べる。まずは GUI プログラムを理解する際の ORCA の利用手順について説明する。次に ORCA を用いたプログラム理解作業の具体例について紹介する。ここでは、プログラムの機能改良、機能追加、再利用という理解目的の異なる 3 つの例を挙げる。

7.1　ORCA を利用したプログラム理解の手順

ORCA を利用してプログラムを理解する際には以下のような手順で作業を行う。

1. ORCA を用いた理解の準備 ORCA を用いた理解作業を開始するために、まず ORCA と対象プログラムを起動する。ORCA を起動するためにはあらかじめ理解の対象となるプログラムのソースファイルを Eclipse のプロジェクトとしてインポートしておく必要がある。

1a. ORCA の起動 Eclipse のプロジェクトの中から対象プログラムの main 関数を含む Java ファイルを選択し、ポップアップメニューを展開し「ORCA の起動」メニューを選択すると、対象プログラムを可視化する ORCA が起動される。

1b. 対象プログラムの起動 起動された ORCA のコントロール部にある「対象プログラムを起動」ボタンを押下すると、対象プログラムが起動されて、main 関数から順にプログラムの実行が可視化される。

2. 理解対象の機能に関連するクラスの抽出と各クラスの役割と関連性の理解 まずは理解したい機能に関係のあるクラスだけを抽出して理解対象を絞る。その上で、それらのクラスの役割や相互関係を把握するための作業を行い理解を深める。

2a. 対象プログラムの GUI への操作入力 ORCA から起動された対象プログラムの GUI に対して、理解したい機能に関する操作を行うと、操作よって引き起こされたソースコードの実行と画面変化の様子が可視化される。

2b. 理解対象イベントの選出 操作に対する一連の可視化結果の中から、理解したい機能に関係のあるイベントの可視化結果だけを選出する。選出時には、操作名と画面変化の時系列に基づいて、理解の必要がある場面かどうかを判断する。例えば、図 7.1 ではサムネイル表示において、中心にある 3 枚の画面のみ表示内容が異なる。

もし、理解したい機能がこの表示を行っている間でのみ動作しているのであれば、表示内容が異なっている区間のイベントにおける可視化結果のみを選んで理解作業を進めればよい。表示する可視化結果は ORCA のコントロール部にある「可視化結果の選択」矢印ボタンを押下することで前後の時点のものに切り替えることができる。

2c. ソースコード実行部分と関数呼び出し関係・クラス階層表示の閲覧 選出したイベントによって実行されたクラスを把握するために、各クラスに色づけによって強調表示されたソースコード行があるかを確認する。色づけ部分を含んでいるクラスがあれば、それはこの機能に特に関連があるクラスとなる。次に実行されたクラス間の関連性について確かめる。まず関数呼び出しのエッジの結びつきからクラス間の依存関係を把握する。次にクラス階層表示からその親子関係を理解する。実行されたクラスの親クラスは実行されていなかった場合にも機能に関係している可能性が十分にある。

3. 機能の処理の流れの理解 抽出したクラスだけに対象を絞り関数を順に追うことで処理内容を詳細に理解する。その作業は以下の機能を理解したい情報や状況に応じて使い分けながら行う。

3a. ポップアップ走査機能の使用 ポップアップ走査を行い、関数呼び出しの順序を理解する。この機能は特に関数呼び出しの流れの概略を理解する上で役立つ。この機能を利用する際には、ORCA のコントロール部にある「関数呼び出しの走査」矢印ボタンを用いる。矢印ボタンを押下すると前後ステップにおける関数呼び出しに注目した表示に切り替わる。

3b. ズーミング走査の使用 ズーミング走査を行い、関数呼び出しを順に辿って処理内容を理解する。この機能は特に実際に実行されたソースコードを詳細に確認しながら理解を進める上で役立つ。この機能はポップアップ走査機能と同様の操作を行うことで利用できる。ズーミング走査とポップアップ走査は「関数呼び出しの走査」の「操作モード切替」ボタンの押下によって交互に切り替わる。

3c. Eclipse エディタ上での実行の確認 ORCA のグラフ表示部のソースコードをクリックすると対応する Eclipse 上のエディタが開かれてクリックした行に移動する。Eclipse エディタ上ではマーカ表示を閲覧することで、実行された行を確認することができる。これらの機能と Eclipse が提供する機能を相互に利用をすることで各クラスについての理解を深める。

実際に理解作業を行う場合には、これらの手順を踏む順序を多少前後させる必要や特定の手順を繰り返す必要がある。

図 7.1: 一連の可視化結果の中からのイベント選出

7.2 適応事例 1: ネットワーク構造図エディタの理解作業

7.2.1 理解対象と理解の目的

本節では理解作業の例として、図 5.4 に示したネットワーク構造図エディタを改良する目的で 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。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、その上でマウスリリースを行ったときにエッジが結ばれるように改良したい。これがこの機能を理解する目的である。

7.2.2 理解作業

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× n →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（例えばマウスリリース時に図 5.1a 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わかる。

この時点で機能に関わるクラスを絞ることができるので、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わかる（図 5.1a 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 のポップアップ及びズーミング走査、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、それぞれの間を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わかる。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わかる。

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わかったが、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、切り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辿ると、**GSegment** に描画関数が実装されていて、

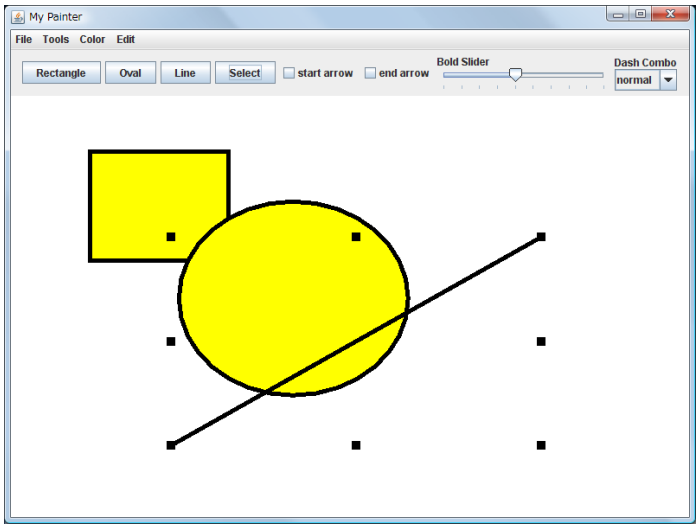


図 7.2: 対象とする GUI プログラム (ドローイングツール)

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わかる。

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わかった。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。

7.3 適応例 2: ドローイングツールの理解作業

7.3.1 理解対象と理解の目的

本節では図 7.2 のドローイングツールに対して、機能追加をする目的で ORCA を利用した場合の例を示す。このドローイングツールはボタンを押すことで図形の種類を選択し、キャンバス内をドラッグすることで図形を描画する機能を有する。

ここでは開発者がドローイングツールに新たな種類の図形を描画するために描画図形モード選択ボタンとその描画機能を追加する場合を考える。その際には、開発者は既存の図形描画機能であるそれらの実装方法を模倣して新たな図形描画機能の追加を行えばよい。そこで既存の図形描画機能を理解することを目的とする。

7.3.2 理解作業

ここではまず既存の図形描画機能として矩形描画機能から理解を行うことにする。

まず、理解対象の機能に関するクラスを抽出するために、ORCA によって起動された対象プログラムの GUI に対して矩形描画機能に関する操作を行う。ここでは、まず矩形モード選

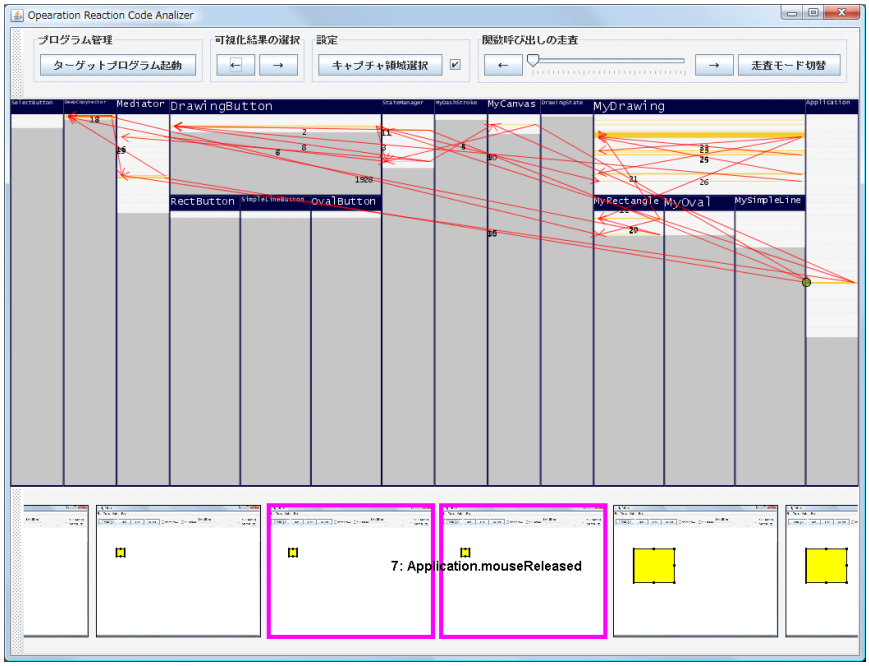


図 7.3: ドローイングツールの可視化結果

択ボタンを押下し、キャンバス内で、マウスをドラッグする。一連の操作の中で、ボタンプレス→マウスプレス→マウスドラッグ× n →マウスリリースの順にイベントが発行され、マウスイベントの間で再描画イベントが発行される。これらの操作それぞれに対する可視化結果のグラフ表示部を確認すると **Application**、**DeepCopyVector**、**DrawingButton**、**Mediator**、**MyCanvas**、**MyDrawing**、**MyRectangle**、**RectButton**、**StateManager** の計 9 クラスが実行されていたことがわかる（例として図 7.3 にマウスリリース時の可視化結果を示す）。

続いて抽出した各クラスの役割と関連性を確認する。関数呼び出しのエッジを確認すると、ボタンプレスイベントに関しては、**DrawingButton** がイベントハンドラを実装していて、その中から **StateManager** を呼び出していることがわかる。またマウス操作イベントに関しては **Application** がイベントハンドラを実装していることが把握できる。ORCA のクラス階層表示から、**RectButton** は **DrawingButton** の子クラスとなっていることがわかる。**DrawingButton** の子クラスにはその他に **SimpleLineButton**、**OvalButton** があることから、**DrawingButton** は描画図形のモード選択ボタンの抽象クラスを表わしていて、**RectButton** は矩形モード選択用の具象クラスを表していることが推測できる。**MyDrawing** と **MyRectangle** の間にも同様の関係があるため、こちらは描画図形オブジェクトそのものを表しているクラスであることが推測できる。

以上より、矩形描画機能の特有の実装部分は **RectButton** と **MyRectangle** 内で実装されていると推測し、各処理内容について解析を行っていく。まずはボタンプレスイベントに関して関数走査を用いて確認を行うと、図形モード選択ボタンはそれぞれのモードを状態とする State

パターン [30] として実装されており、ボタンプレスイベント発行時には押されたボタンが表す描画図形を状態として設定していることが把握できる。次にマウス操作イベントについて関数走査を用いて確認を行うと、まずマウスプレス時に矩形オブジェクト (すなわち **MyRectangle**) が新たに生成され、図形オブジェクトを管理する配列に追加されることがわかる。マウスドラッグ中にはマウスの座標に応じて矩形オブジェクトのサイズを変更し、リリース時には図形オブジェクトを管理する配列がコピーされて保存される処理が行われることが把握できる。なお、マウスドラッグとリリースに関しては、矩形描画に特化した実行部分 (**RectButton** と **MyRectangle** を区別して使用している部分) は無い。また再描画イベントを確認すると、描画は図形オブジェクトを管理する配列の要素に順次アクセスしながら、オブジェクトの描画関数が呼び出されていることが把握できる。

他の種類の図形描画機能を確認したところ、同様の処理がなされていることが把握できる。これらから、新たな図形の描画機能を追加するためには、描画図形モードの状態とボタンそのものを表す **DrawingButton** の具象クラス (**RectButton** の実装を模倣) と、描画図形オブジェクトを表す **MyDrawing** の具象クラス (**MyRectangle** の実装を模倣) を新たに実装すればよいことがわかる。

7.4 適応例 3: ドラッグ＆ドロッププログラムの理解作業

7.4.1 理解対象と理解の目的

本節ではドラッグ＆ドロップの基本機能を実装したプログラムを理解する例を示す。理解の対象となるプログラムは、図 7.4 に示すように画面は 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という基本的なドラッグ＆ドロップの機能を有している。プログラムの実装には、ドラッグ＆ドロップをサポートする AWT パッケージである java.awt.dnd パッケージを利用している。

ここで Java のドラッグ＆ドロップ機能の実装について言及する。Java のドラッグ＆ドロップの仕様[1] は複雑で、マウスイベントなどの一般的なイベントに比べて利用用途が限られている。そのため Web 上や書籍における Java の GUI プログラミングの解説においても、ドラッグ＆ドロップについて体系的に述べているものは少ない。以上の理由から、このパッケージを利用した経験のない開発者がいきなりドラッグ＆ドロップを用いたプログラムを実装することは難しい。

前提知識の無い API の利用方法を理解する有益な手段として、サンプルプログラムの実装を確認することが挙げられる。サンプルプログラムはサン・マイクロシステムの Java の公式サイトをはじめとした Web サイト上で多数見つけることができる。しかしサンプルプログラムの多くには、文書やコメントでの解説は付いていない。そのため理解時にはプログラムを実行し、ソースコードを追いながら、その挙動を把握する必要がある。

[1] http://java.sun.com/javase/ja/6/docs/ja/technotes/guides/dragndrop/index.html

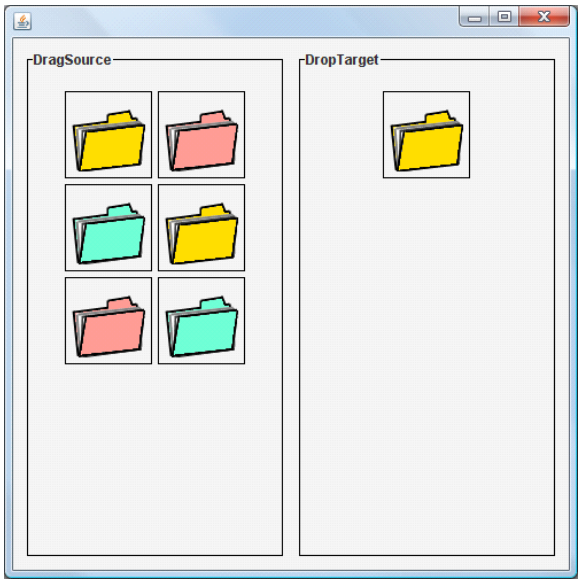


図 7.4: 対象とする GUI プログラム (ドラッグ＆ドロッププログラム)

今回の例で用いるプログラムは非常に小規模かつドラッグ＆ドロップの基本機能のみを実装していることから、サンプルプログラムと同等なプログラムであるとみなすことができる。そのためこの例では、未知のイベント体系であるドラッグ＆ドロップパッケージの利用方法を理解し、そのソースコードを再利用することを目的する。

7.4.2 理解作業

ドラッグ＆ドロップのイベント体系を理解するために、イベントがどのような順序で発行されていて、各イベントがどのような役割を担っているのかを把握する必要がある。特にドラッグ＆ドロップの処理は状況や操作に依存して変化するため、常にドラッグ＆ドロップ機能に関係のあるすべてのイベントの実行について注意を払わなければならない。

しかし、既存ツールを用いた場合には、イベントの発行順序を把握することが難しい。例えば、デバッガを用いた場合には、ブレークポイントの設定により実行を中断しながら行わなくてはならないため、ドラッグ操作を行うこと自体が困難である。デバッグコードを挟む方法ではすべてのイベントリスナに対してデバッグコードを挟むことで順序関係を把握することが可能であるが、ソースコード上からイベントリスナを探し出す作業やデバッグコードを記述する作業に大きなコストがかかる。

一方、ORCA では GUI への操作を行えば、実行されたイベントをすべて取得できる。そのため、他ツールのように繁雑な作業を行う必要や各イベントリスナにおける実行の有無について注意を払う必要はない。

以降では、実際に ORCA を利用した理解作業について述べる。ここではまずドラッグ＆ド

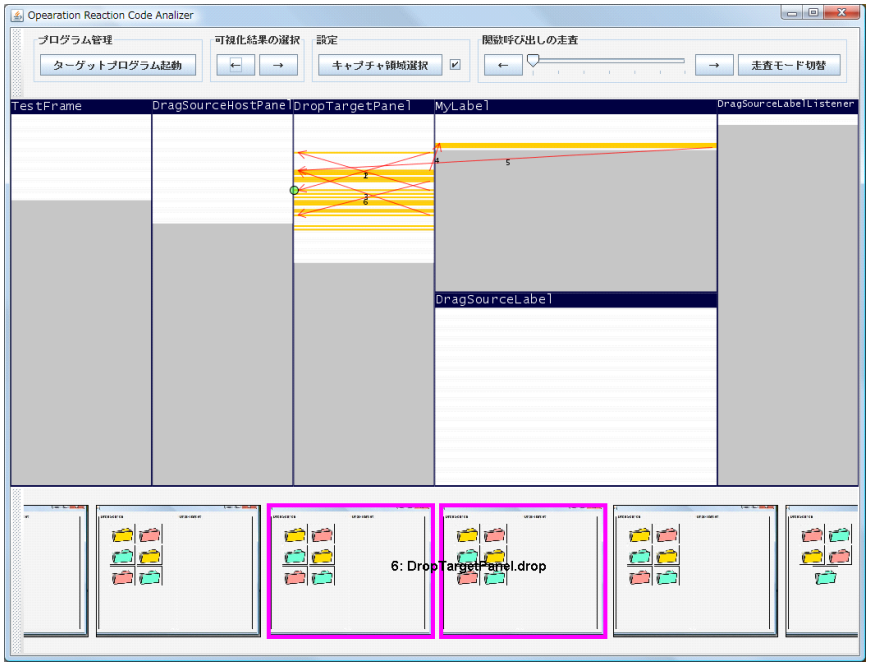


図 7.5: ドラッグ＆ドロッププログラムの可視化結果

ロップの機能のうちオブジェクトの移動処理の実装を把握することで、対象プログラムにおけるドラッグ＆ドロップ機能の概略を理解するものとする。

まずは理解の対象となるクラスの抽出作業を行う。オブジェクトの移動処理を行うために、対象プログラムの左領域から右領域に向かってオブジェクトをドラッグ＆ドロップする。操作を行うとそれに応じて、**DragSourceLabel.dragGestureRecognized()** → **DropTargetPanel.dragEnter()** → **DropTargetPanel.dragOver()** → **DropTargetPanel.drop()** → **DragSourcePanel.dragDropEnd()** という順序でイベントが実行され (リスナが呼び出され)、対応する実行が ORCA 上で可視化される（例としてイベント **DropTargetPanel.drop()** の可視化結果を図 7.5 に示す）。イベントごとの可視化結果を順に眺めて、実行された行の強調表示を確認すると、**dragEnter()**、**dragOver()** はそれぞれ空の実装になっていることがわかる。そのため、今回のサンプルプログラムにおいては **DragSourcePanel.dragGestureRecognized()**、**DropTargetPanel.drop()**、**DragSourcePanel.dragDropEnd()** の 3 つのイベント (リスナ) の働きに焦点を当てればよいことが理解できる。また、ORCA の強調表示から一連のイベントの中で実行されたクラスは **DragSourceHostPanel**、**DropTargetPanel**、**MyLabel**、**DragSourceLabel** の 4 クラスであったことが確認できる。このように ORCA を用いることでイベント発行の順序を理解しながら、理解が必要なクラスやイベントを抽出することができる。

次に、抽出したイベントとクラスに対してイベントの実行を順に辿っていくことで、ドラッグ＆ドロップ機能の理解を行う。**DragSourceLabel.dragGestureRecognized()** イベントを順に辿ると、このイベントではドラッグ開始時における Ctrl ボタン押下有無の判定や転送

するオブジェクトの加工といったドラッグの開始処理を行っていることが把握できる。次に **DropTargetListener.drop()** イベントを順に辿ると、このイベントでは右領域にドロップされたオブジェクトを追加していることが把握できる。最後に **DragSourcePanel.dragDropEnd()** イベントを順に辿ると、このイベントでは左領域から移動されたオブジェクトを削除していることが把握できる。実際には、これらの各クラスはドラッグ＆ドロップパッケージのイベントリスナを実装しているので、必要に応じて各クラスのソースコード上で、実装しているイベントリスナのインタフェースについて確認を行えば、ドラッグ＆ドロップパッケージについての理解をより深めることができる。

以上が移動処理の概略の理解であるが、コピー処理等についても同様の手順で行うことができる。このようにサンプルプログラムの実装について理解を行えば、サンプルコードを参考にフルスクラッチでのプログラミングすることやサンプルコードを再利用したプログラミングを行うことができる。

第8章　評価

我々の作成したシステムの有効性を検証するために被験者による評価実験を行った。実験を行うにあたって、プログラム可視化ツールの被験者による評価実験を行っている研究である SHriMP[12, 31, 32] や ClassBlueprint[33]、CARE[34] の実験設計や評価方法を参考にした。

8.1　実験目的

実験では、以下の仮説を実証することを目的とする。

仮説 1 ORCA は複数イベントを伴う機能の理解する上で有効に働く

仮説 2 ORCA は操作と実行されたソースコードの対応関係を把握する上で有効に働く

仮説 3 ORCA のソースコード実行部分の包括的な可視化表現は機能の実装部分の把握に有効に働く

仮説 4 ORCA の画面サムネイル表示は実行された状況を同定する際に有効に働く

8.2　実験方法

8.2.1　使用する理解支援ツール

実験では以下の 3 つのツールを比較対象とした。

- Eclipse
- ORCA（画面サムネイル表示無し）
- 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伴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併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無いものを用意した (図 8.1)。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。

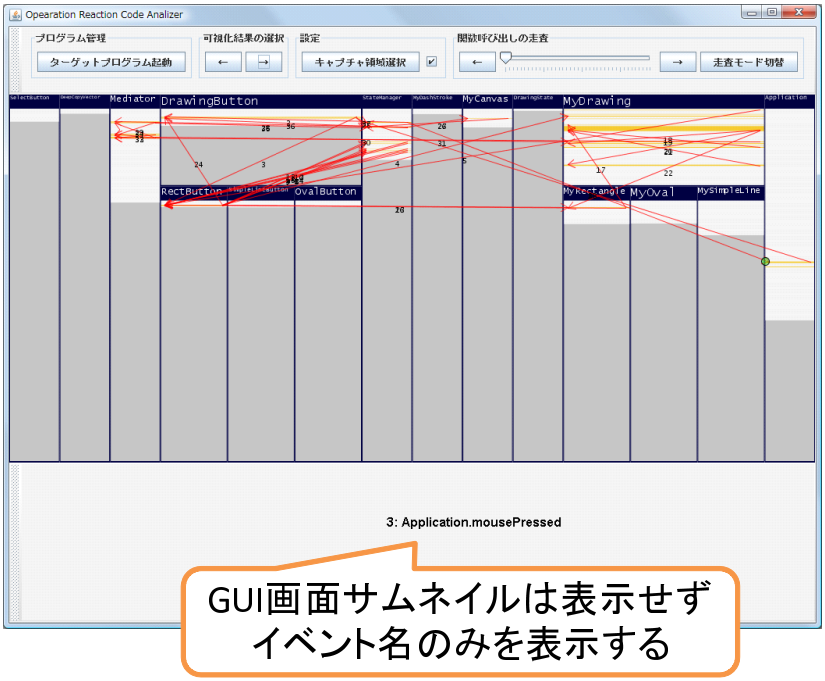


図 8.1: サムネイル表示無しの ORCA

8.2.2 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、被験者毎にこれらのプログラムと使用するツールの組み合わせを、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を表 8.1 に示す。今回の実験に使ったプログラムは 7 章の適応事例で用いたものであるため詳細についてはそちらを参照されたい。表 8.1 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。

8.2.3 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毎に用意した。

表 8.1: 実験に使用したプログラム

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	1445	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

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伴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伴うものであるため、設問全体の結果から仮説 1 を検証する。設問 1 は機能の実装部分を操作に結びつ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、仮説 2 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、仮説 3 についても併せて検証する。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 の結果から仮説 4 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、設問の曖昧さを無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繋いだ場合に、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時の描画処理と載せていない時の描画処理におけるフローの違いはどこか説明せよ。

他のプログラムについての設問や設問の詳細については論文末尾の付録 1 を参照されたい。

また、被験者には設問を解答するに当たっての諸注意として以下のような指示を与えた。

- 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（但し、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構わない、解答途中で別の問題に移ってもよい

8.2.4 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）

1 人の被験者に実験にかかる総時間は 3 時間程度である。

8.2.5 実験環境

実験にはデスクトップ PC（Intel（R）Core（TM）2 Duo 3.00GHz、メモリ 2GB、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 1680x1050、縦置き 20 インチ・解像度 1024x1028）を接続して用いた。

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予め依頼した。実験中の発話や操作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 ORCA 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上で実験に臨んでもらった。配布したマニュアルは論文末尾の付録 2 として添付したので、詳細についてはそちらを参照されたい。

8.2.6 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ーした。レクチャー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は図 8.2 に示すようなプログラムで、プログラムの規模は行数 344、

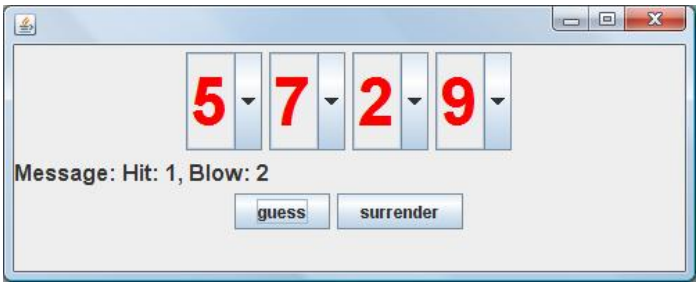


図 8.2: 対象とする GUI プログラム (数字当てゲームプログラム)

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。

8.2.7 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。普段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、被験者に ORCA の有効性を評価するアンケートに回答してもらった。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通りである。

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。

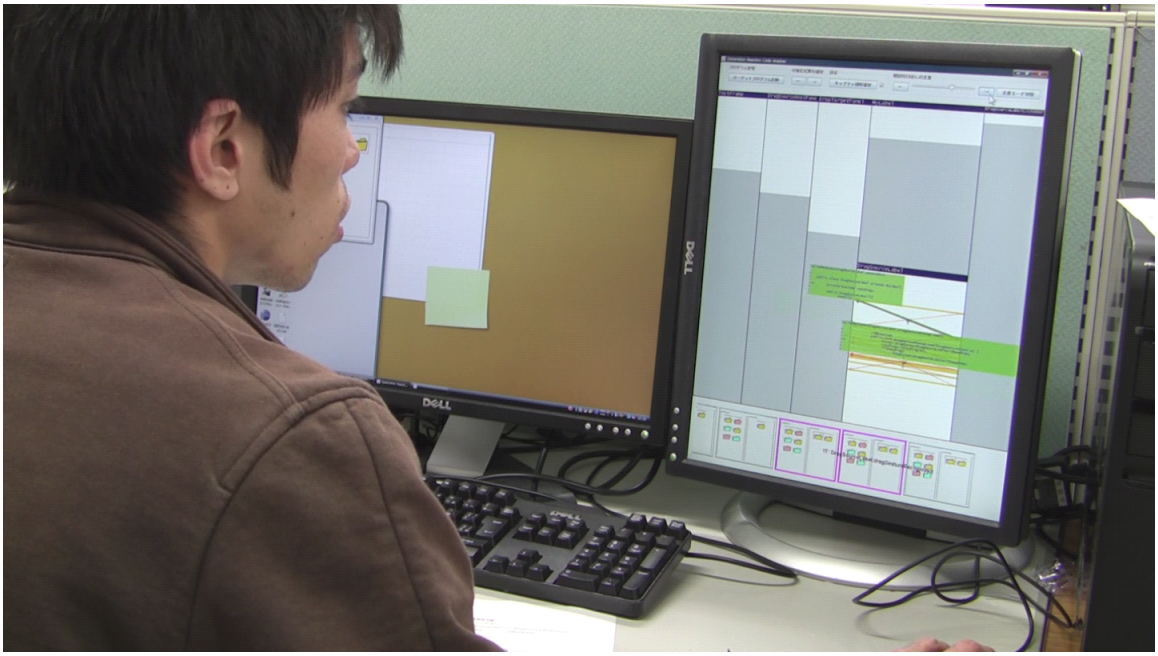


図 8.3: 実験の様子

8.3　実験結果

実験は 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 人を雇用して行った。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。図 8.3 に実験を行っている様子を示す。

8.3.1　設問解答の採点結果

各設問は予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 8.2 と図 8.4 に結果を示す。

8.3.2　アンケート結果

事後アンケートの結果を表 8.3 と図 8.5 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。

表 8.2: 実験の採点結果

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

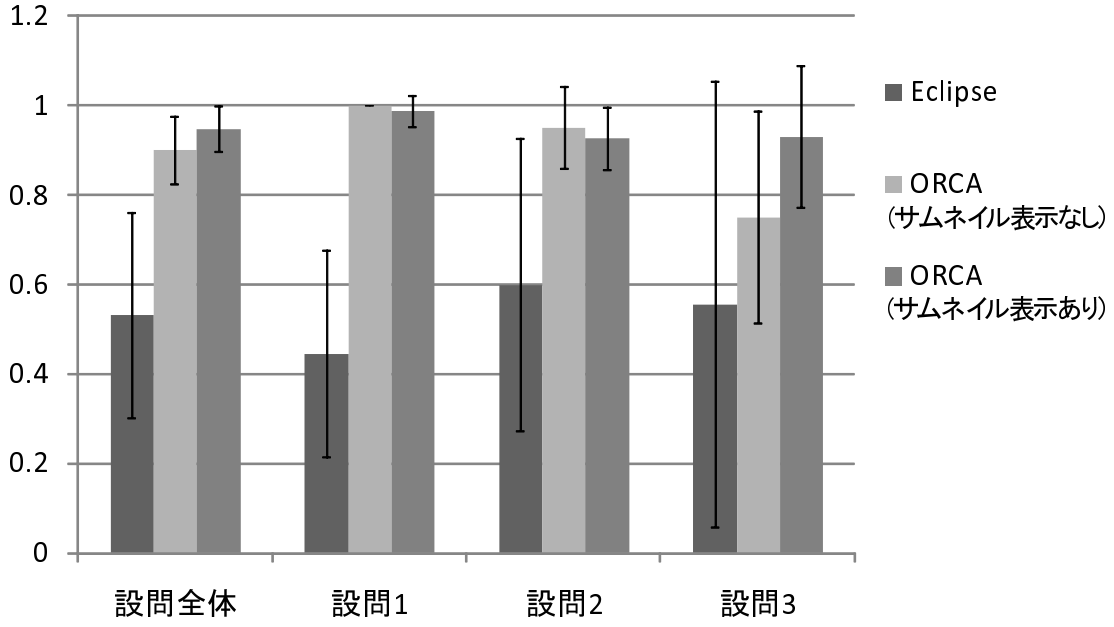


図 8.4: 実験の採点結果

表 8.3: アンケートの結果

アンケート項目	平均値	標準偏差
項目 1	3.56	0.53
項目 2	2.44	0.88
項目 3	3.56	0.73
項目 4	3.63	0.74
項目 5	3.50	0.76
項目 6	2.63	1.30
項目 7	3.56	0.53

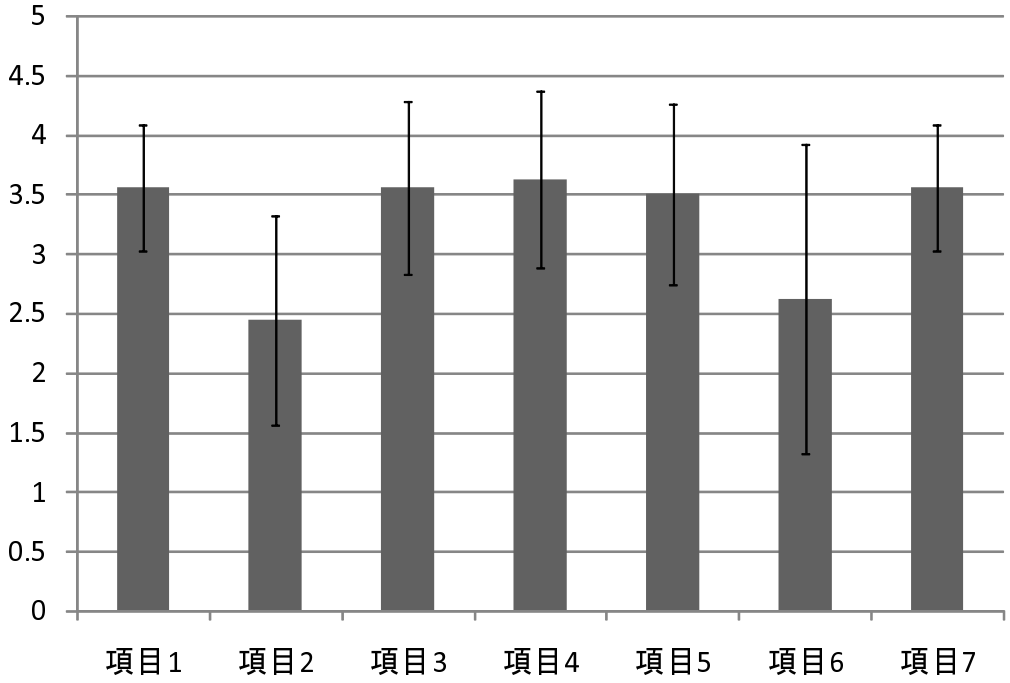


図 8.5: アンケートの結果

8.4 分析と考察

8.4.1 設問解答の採点結果の分析

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。設問全体ではツールによる効果は有意だった（$p < 0.01$）。Ryan 法による多重比較を行ったところ、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（$p < 0.01$）。一方、ORCA

の画面サムネイル表示があるものと無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、無しの場合には 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 についてはツールによる効果は有意であり（$p < 0.01$）、Ryan 法による多重分析を用いた結果、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つけを行い、実行されたソースコード部分を抽出する上で十分有効であることが実証できた（仮説 2、3 を実証）。

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無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無しの 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。仮に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。これらの事実及び、前述した Eclipse と ORCA の間での分散分析の結果から、ORCA は複数イベントを伴う機能の把握にも十分有効であることが実証できた（仮説 1 を実証）。

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、プログラム間でのスコアに有意差は見られなかった（$p = 0.699$）。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被験者全員が Eclipse を利用した経験があり、うち 6 人が時々利用すると答え、3 人が普段から利用すると答えた。そのため、今回の被験者は Eclipse の操作には慣れていたと言える。また 5 人が普段からデバッガを利用し、4 人が普段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上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見られなかった（$p = 0.677$）。また GUI プログラミングに関して、4 人が 1 年以内の経験があり、5 人が 3 年以上の経験があった。経験が多いグループと少ないグループの間でスコアに差があるかを確かめるために、2 つのグループに対する分散分析を行ったところ、統計的な有意差は見られなかった ($p = 0.906$)。

図 8.4 を見ると、Eclipse の結果には大きなばらつきがあることが確認できる。これは Eclipse では被験者の習熟度や被験者と設問との相性等によって解答の出来不出来が大きく変わってくることを示している。一方、ORCA では結果に Eclipse ほど大きなばらつきはない。これより、ORCA を用いれば被験者の経験や解答する設問に関わらず、高い水準で理解作業を行うことができると期待される。

8.4.2 アンケート結果の分析

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に関して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。

クラス階層表示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得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。

GUI 画面のサムネイル表示に関しては 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、“サムネイルが無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全くわからなかった”、“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つける上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わかりやすい” という意見があった。このことから、画面表示の切り替わりが早く肉眼では精密に遷移を辿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挙がった不満として “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。画面変化がない時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

画面変化が伴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、例えば、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。画面変化毎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、イベント毎に区切ることも行単位で処理を理解する時に役立つ。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毎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。

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挙がっており、表示方法に改善の余地があることがわかった。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” という意見も挙がった。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。以上を踏まえると、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、前後の呼び出しもわ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、関数呼び出しを辿りづらかった” という意見があった。そのため、“切替時におけるアニメーションをよりリッチにしてほしい” という要望が挙がった。

Eclipse エディタ上へのマーカ表示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、処理の流れが追いづらかった” 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上で十分に機能するということを表す結果であると捉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、色付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一つであると考えられる。

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” と言う意見が多かった。

第9章　議論

9.1　本手法を利用する利点

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、連続イベントを伴う機能において、GUI への各操作と実行されたソースコード部分を結びつける上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伴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。

次に ORCA のアニメーション処理に関する対応可能性について述べる。ORCA ではイベントをトリガにして可視化表示を区切っている。そのため単純に別スレッド上でアニメーション処理を実装しているような GUI プログラムでは、アニメーション処理とイベントによる処理は区別されずに可視化が行われる。このような場合には、アニメーション処理のみを抽出して理解を行うことはできない。しかし、アニメーション処理をタイマーイベント (例えば、javax.swing.Timer クラス等) を用いて実装した場合には、アニメーションを更新するごとにイベントが発行されるため、ORCA でもそれらのアニメーションイベントを取得することができる。このように実装されたアニメーション処理については ORCA を利用して有効に理解作業を行うことができる。

9.2　本手法の限界

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一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、リアルタイムでの可視化表示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。

次に、プログラムの実行情報取得におけるスケーラビリティについて述べる。現在は、対象プログラム内のソースコードにおける行単位での実行情報をすべて取得している。そのため、情報取得のコストが高く、計算機の性能によっては十分な実行速度が得られない場合がある。特にリアルタイムでの可視化に関して、GUI への操作と画面変化及びソースコード実行情報の表示にタイムラグが発生することがあり、その場合には理解を妨げてしまう恐れがある。その際にも可視化結果の静的なグラフを元に解析を行っていくことは可能だが、この問題への解決法としては、まずリアルタイムでの可視化を行わない前提で、スケーラブルなデバッガ [35] 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 フェーズに分ける解決法が挙げられる。

9.3 本研究とアスペクト指向プログラミングとの関連性

アスペクト指向プログラミング [36] 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一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[37] や AspectJ[38] 等のアスペクト指向用デバッガと併せて用いることも可能であると考えられる。アスペクト指向用デバッガでは、従来ソースコード内に直接記述する必要があったデバッグコードを、ソースコードを別々に記述することを可能とする。

9.4 本手法のマルチスレッド対応

現在の GUI プログラムはマルチスレッドによる並列化が行われている。例えば、アニメーションの描画処理はメイン処理とは別のスレッドにおいて制御されることが多い。そのため、描画に関わる機能を把握する際などには、ソースコードの実行がどのスレッドによって行われたのかを把握することが重要である。

このような背景から、我々は本研究とは別に、ORCA の表示をマルチスレッドによる実行情報も可視化できるように拡張し、並列プログラムに対する理解支援をするという試みを行った [39]。

マルチスレッドによる実行情報の表現について述べる。ORCA では、マルチスレッドによるプログラムの実行をスレッドごとの色分けによって表現する。そのため、上記 2 つの情報をスレッドごとに別々の色で表現する。色分けされたスレッドごとの表示の重ね合わせを行うことで、マルチスレッドによるプログラムの実行の様子を一画面上で表現する。図 9.1 では

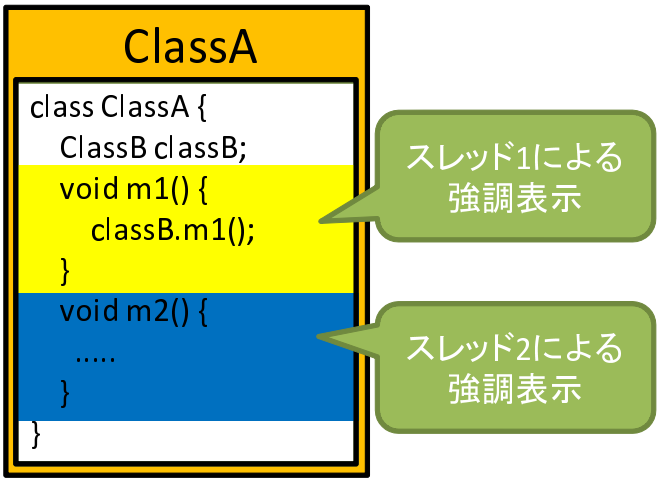


図 9.1: マルチスレッドによる実行の表現

スレッド 1 とスレッド 2 の 2 つのスレッドが実行されている。このように複数スレッドの実行部分が別々の色で表示される。

図 9.2 は実際に ORCA の画面で、マルチスレッドによる実行を可視化している様子である。このようにスレッド毎に別々の色により強調表示を行う。

9.5 木構造割り当て手法の他シーンへの応用

本研究で提案したクラス階層の 2 次元空間への敷き詰め手法は、木構造上の全ての要素に対して表示領域を割り当てるという特徴がある。本手法は、クラス階層の表現に限定されたものではなく、木構造を持つ情報全般に対して適応することが可能である。特に TreeMap では扱うことに問題のあった、途中の階層にある要素が表示情報を持つような木構造に対する領域割り当てに有効であると期待する。

我々は既に本手法をクラス階層以外の木構造への適応を試みている。まず、途中の階層に表示領域を割り当てる必要のない木構造に対する適応例としてディレクトリ構造の可視化を試みた。ディレクトリ構造は、フォルダとファイルの入れ子構造になっているため、途中の階層は多くの情報を持たない。そのため、TreeMap を適応した可視化を行うことができる。TreeMap を適応した場合では、葉以外の要素に対して表示領域を割り当てる必要がないので、本手法を適応した場合に比べて空間の利用効率は良い。一方、本手法を適応した場合には、TreeMap に比べて親子関係を把握しやすくなると考えられる。図 9.3 に本手法を用いてディレクトリを可視化している様子を示す。作成したプログラムではディレクトリとファイルのサイズに意味付けを行った。そのため、使用容量の大きいディレクトリほど、大きな表示領域を持つ。赤はディレクトリ、黄色はファイルを示す。各領域の左上にはファイル名が表示される。

途中の階層にも表示情報を含むような木構造に対する適応例として、閲覧履歴表示付き Web

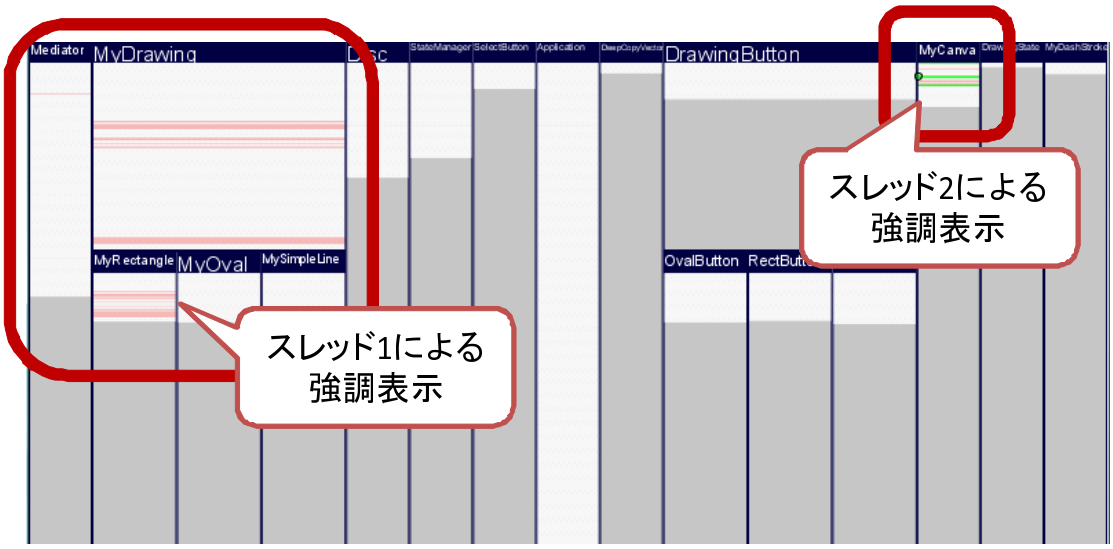


図 9.2: ORCA におけるマルチスレッド実行の表現

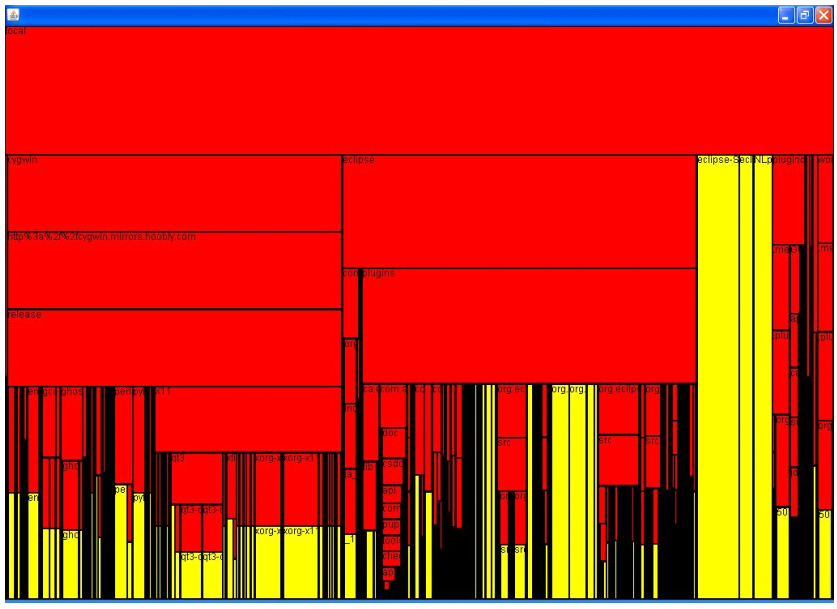


図 9.3: 提案した木構造割り当て手法を用いたディレクトリ構造の可視化

図 9.4: 提案した木構造割り当て手法を用いた閲覧履歴表示付き Web ブラウザ

ブラウザを紹介する。図 9.4 に作成したシステムの概観を示す。図のように過去の閲覧した Web ページをすべて一画面上に並べる形で表示する。

Web ページでのハイパーリンクを辿りながら行う閲覧は木構造として表現することができる。図 9.5 左に閲覧履歴の木構造の例を示す。図中の矢印はページの遷移を表し、逆方向への矢尻は遷移したページを閲覧後に元のページに戻ってきたことを表している。

そこで我々は Web ページの閲覧履歴をページ遷移の木構造に従って 2 次元空間に敷き詰めて表示するブラウザを開発した。図 9.5 右は図 9.5 左に示す Web 閲覧履歴の木構造を可視化している様子を表す。このように過去に閲覧したページを親と見なした木構造に従って Web 閲覧履歴を可視化する。同じページから別のページに遷移があった場合には、遷移先のページを兄弟として横並びにして提示する (図中の D、E、F)。本ブラウザでは、現在閲覧中のページにもっとも大きな重要度を与え、そのページに親等の近いページほど大きな重要度を割り当てている。本ブラウザを用いれば、過去の閲覧したページを確認しながら、Web ブラウジングを行うことができる。

このように本割り当て手法は木構造に従う様々な情報の可視化に利用できると期待している。今後は別の木構造情報に対しても適応を行い、手法の有効性を検証したい。

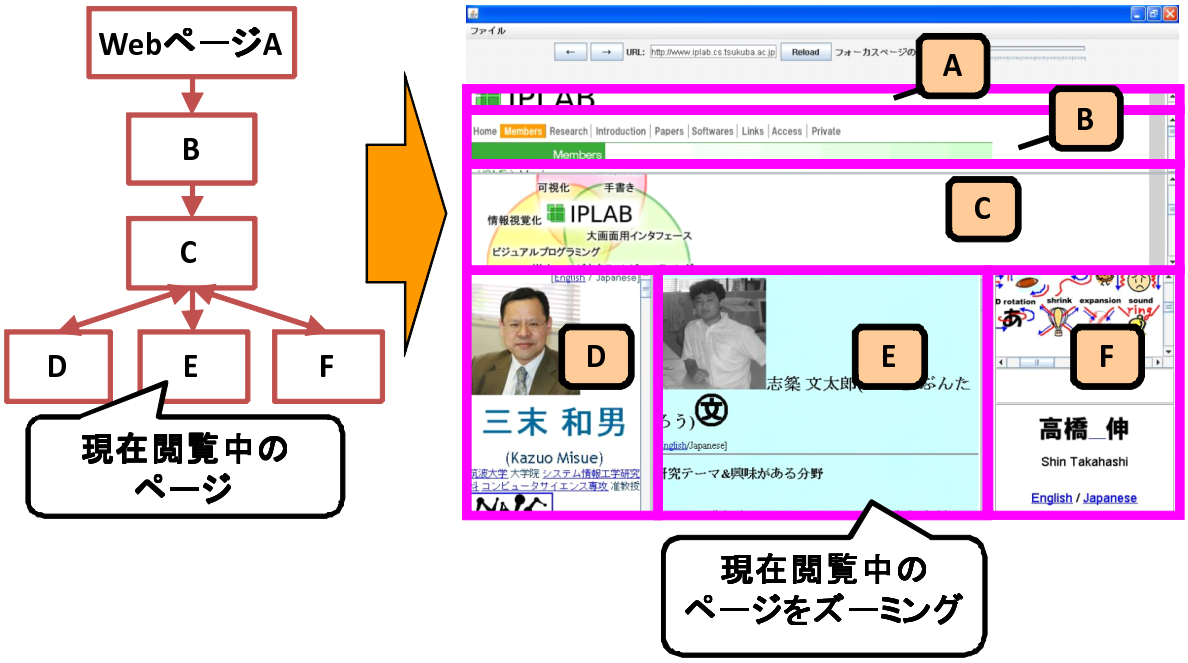


図 9.5: 閲覧履歴表示付き Web ブラウザで扱う木構造とその可視化

第10章　まとめ

本研究では、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を一画面上に表示する可視化手法を開発した。提案手法を用いることにより、GUI への入力操作と出力される画面を捉えながら、入出力それぞれに対応して実行されるソースコードを抽出するという GUI プログラム特有の理解作業の困難さが解消される。

さらに我々は提案手法に従った Java の GUI プログラム理解支援システム ORCA を作成し、ORCA を用いて本手法の有効性を検証する評価実験を実施した。評価実験によって、本手法が、GUI への連続的操作によって発行された複数イベントの中から特定イベントのみを選出し、各操作とその実装部分を抽出する際に特に効果的に働くことが示された。

また本研究の可視化表現の中で、木構造データに対する 2 次元空間上への敷き詰め手法を提案した。本敷き詰め手法によって、従来手法では表示することが難しかった、木構造中の全ての要素がデータを持つ木構造に対しても可視化が行えるようになった。

謝辞

本研究を行うにあたって、田中二郎教授には、指導教員という立場から多くのご指導をいただき、学外の研究発表の機会を与えていただきました。講師の志築文太郎先生には、研究の方針から論文の執筆に至るまで、丁寧できめ細かいご指導をいただきました。三末和男准教授、講師の高橋伸先生には、様々な視点からのご意見と研究発表に関する数多くのご助言をいただきました。ここに深く感謝いたします。

高度 IT 人材育成のための実践的ソフトウェア開発専修プログラムの担当講師である菊池純男教授、駒谷昇一教授にはコース活動を通して様々なご指導をいただきました。心より感謝いたします。

株式会社 NTT データには奨学金による経済的な支援していただくとともに研究発表と議論の場を提供していただきました。心から御礼申し上げます。

インタラクティブプログラミング研究室の皆様には研究活動と日常生活を通じて大変お世話になりました。特に WAVE チームの皆様には日常の議論やチームゼミを通じて多くのアイデアをいただきました。心より感謝いたします。

また高度 IT 人材育成のための実践的ソフトウェア開発専修プログラムのメンバの方々には、被験者実験に協力をしていただき、学内発表等を通して研究に関する多くの意見をいただきました。心より感謝いたします。

最後に、今日まで私を支えてくれた両親と家族、友人、学生生活の中でお世話になった全ての方々に心より感謝いたします。本当にありがとうございました。

参考文献

[1] 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. 7 pages.

[2] 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 pages.

[3] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft: A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, pp. 957–968, 1992.

[4] Steven P. Reiss. Visualizing java in action. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pp. 57–65, 2003.

[5] Steven P. Reiss and Manos Renieris. Jove: Java as it happens. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pp. 115–124, 2005.

[6] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[7] Minoru Terada. Etv: A program trace player for students. *SIGCSE Bulletin*, Vol. 37, No. 3, pp. 118–122, 2005.

[8] 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.

[9] 中村利雄, 五十嵐武夫. 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.

[10] Thomas Ball and Stephen G. Eick. Visualizing program slices. In *In IEEE/CS Symposium on Visual Languages*, pp. 288–295, 1994.

[11] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, Vol. 29, No. 4, pp. 33–43, 1996.

[12] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. Shrimp views: An interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pp. 520–521, 2002.

[13] Paul Gestwicki and Bharat Jayaraman. Interactive visualization of java programs. *IEEE CS International Symposium on Human-Centric Computing Languages and Environments*, p. 226, 2002.

[14] Paul Gestwicki and Bharat Jayaraman. Methodology and architecture of jive. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pp. 95–104, 2005.

[15] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pp. 373–376, 2004.

[16] Klaus P. Lohr and Andre Vratislavsky. Jan: Java animation for program understanding. *IEEE CS International Symposium on Human-Centric Computing Languages and Environments*, pp. 67–75, 2003.

[17] 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎. Java プログラムの実行履歴に基づくシーケンス図の作成. 第 11 回ソフトウェア工学の基礎ワークショップ, pp. 5–16, 2004.

[18] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 16–23, 1986.

[19] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphics*, Vol. 11, No. 1, pp. 92–99, 1992.

[20] Hideki Koike. Fractal views: A fractal-based method for controlling information display. *ACM Transactions on Information Systems*, Vol. 13, No. 3, pp. 305–323, 1995.

[21] 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. 6 pages.

[22] 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.

[23] Tatsuya Sato, Buntarou Shizuki, and Jiro Tanaka. Support for understanding GUI programs by visualizing execution traces synchronized with screen transitions. In *ICPC '08: Proceedings of The 16th IEEE International Conference on Program Comprehension*, pp. 272–275, 2008.

[24] 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. インタラクション 2007 論文集, pp. 179–180, 2007.

[25] Java Development Tools. http://www.eclipse.org/jdt/.

[26] Java Debug Interface. http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html.

[27] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering*, Vol. 30, No. 8, pp. 535–546, 2004.

[28] Plug in Development Environment. http://www.eclipse.org/pde/.

[29] 三末和男. 図的思考支援を目的とした図ドレッシングについて. 富士通情報研, 1994. 17 pages.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. オブジェクト指向における再利用のためのデザインパターン. ソフトバンククリエイティブ, 1995. 本位田 真一, 吉田 和樹 訳.

[31] M.-A.D. Storey, K. Wong, H.A. Mueller, P. Fong, D. Hooper, and K. Hopkins. On designing an experiment to evaluate a reverse engineering tool. *Working Conference on Reverse Engineering*, pp. 31–40, 1996.

[32] M.-A.D. Storey, K. Wong, and H.A. Muller. How do program understanding tools affect how programmers understand programs. *Working Conference on Reverse Engineering*, p. 12, 1997.

[33] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, Vol. 31, No. 1, pp. 75–90, 2005.

[34] Panagiotis K. Linos, Philippe Aubet, Laurent Dumas, Yann Helleboid, Patricia Lejeune, and Philippe Tulula. Visualizing program dependencies: An experimental study. Software: Practice & Experience, Vol. 24, No. 4, pp. 387–403, 1994.

[35] GUIllaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Notices*, Vol. 42, No. 10, pp. 535–552, 2007.

[36] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: In European Conference on Object-Oriented Programming*, pp. 220–242, 1997.

[37] Yoshiyuki Usui and Shigeru Chiba. Bugdel: An aspect-oriented debugging system. *APSEC '05: Asia-Pacific Software Engineering Conference*, pp. 790–795, 2005.

[38] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisword. An overview of aspectj. In *ECOOP '01: In European Conference on Object-Oriented Programming*, pp. 327–354, 2001.

[39] 佐藤竜也, 志築文太郎, 田中二郎. GUI を持つプログラムの理解支援のための可視化システムのマルチスレッド対応. 情報処理学会第 70 回大会 CD-ROM, 2008. 2 pages.

付録1. 評価実験で用いた設問・アンケート用紙

今回の実験について

この度は実験へのご協力ありがとうございます。今回、お願いする実験は「GUI プログラム理解支援ツールの評価」に関する実験です。GUI プログラムとは Graphical User Interface を有するプログラム全般を指します。これから指定する 3 つのツールを使って特定の GUI プログラムを理解していただきます。こちらから GUI プログラムに関する問題を出題するので、それぞれのツールを利用してプログラムを理解しながら問題に解答していただきます。設問は各 GUI プログラムにつき 3 問ありますが、どの順番に解いていただいても構いません。また解答には**1 プログラムにつき 30 分の制限時間**があるためすべての設問に解答するには時間が足りないかもしれませんが、**なるべく多くの設問に速く正確に解答することを目標としてください(部分点がつくのでわかったところまででもかまいません)。**また、すべての設問に手をつけてもらうために**各設問につき最低 5 分以上、問題を解く時間を設けてください(5 分以内に設問が解けた場合は別)。**また、**解答用紙への記述は制限時間内に終わらせてください。**実験終了後、ツールを利用しての感想を伺います。

実験に際してのお願い

- 実験中にはあなたが**考えていることを声に出しながら、作業**を行っていただけると助かります(プログラムの理解の仕方やツールの機能が我々の意図通りであるかを検証する調査等に使用します)。
  - 今から行おうとしている動作とその目的(～をするために～をする)
  - 作業を行っているうちにわかったこと(～機能では～をしていることがわかった)

- 作業をこなしている様子をビデオカメラでの撮影と音声の録音をさせてください。

  上記 2 点に関しては可能な範囲で構いませんので、ご協力よろしくお願いいたします。なお、今回の実験で収集した情報は個人が特定できないような形で、学内外での発表等で利用させていただきます。撮影した映像や解答などはそれ以外の目的では一切利用しません。

実験の手順と目安時間

1. 事前アンケート(10 分)
2. 各ツールの紹介と操作説明(20 分)
3. 練習問題の出題と操作説明(30 分)
4. 各ツールを利用したプログラム理解作業問題の出題・解答(30 分 * 3 ツール)
5. 事後アンケート(15 分)
6. インタビュー(10 分)

不明点等がありましたら、逐次実験者に質問してください。

実験事前アンケート用紙

あなたについて教えてください。

Q1. あなた自身について教えてください。

年齢 _____ 歳　　男・女

Q2. あなたのプログラミング経験を教えてください。

_____年

Q3. あなたの Java プログラミング経験を教えてください。

_____年

Q4. あなたの GUI プログラム経験を教えてください。

経験がある･経験がない　(経験がある場合: _____年､経験した言語等_____)

Q5. あなたが普段使用している開発環境を教えてください。

_____

Q6. あなたは Eclipse を使用したことがありますか？

1. 普段から使用する　　2. たまに使用する

3. ほとんど使用したことがない 4. まったく利用したことがない

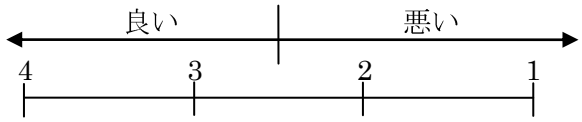
Q7. あなたはデバッガを使用しますか？

1. 普段から使用する　　2. たまに使用する

3. ほとんど使用したことがない 4. まったく利用したことがない

Q8. あなたの本日の体調はいかがですか？(数字に丸)

評価:



Q9. あなたの昨日の睡眠時間について教えてください。

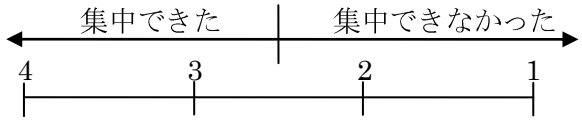
_____時間睡眠　　(備考(眠い、徹夜など):)

実験事後アンケート用紙

実験で使用したツール ORCA(GUI 画面のサムネイルがあったもの)の各機能について、プログラムの理解に役立ったかどうかご回答ください。

Q1. 実験には集中できたかどうか教えてください(数字に丸)。

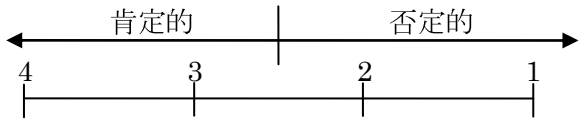
評価:



(理由:)

Q2. ターゲットプログラムに対する 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がプログラムを理解する上で役に立ったかどうか採点してください(数字に丸)。

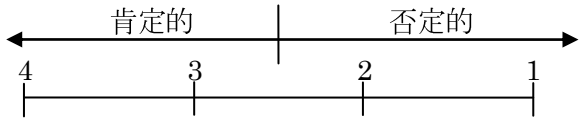
評価:



(理由:)

Q3. クラス階層表示がプログラムを理解する上で役に立ったかどうか採点してください(数字に丸)。

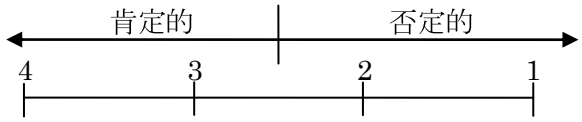
評価:



(理由:)

Q4. GUI 画面遷移のサムネイル表示がプログラムを理解する上で役に立ったかどうか採点してください(数字に丸)。

評価:

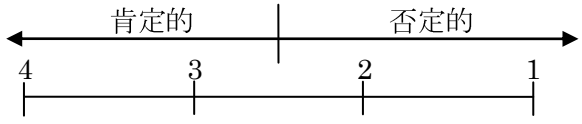


(理由:)

Q5. ポップアップビューを利用した関数の走査機能を使用しましたか？使用した場合にはこの機能がプログラムを理解する上で役に立ったかどうか採点してください(数字に丸)。

この機能を　　利用した・利用しなかった

評価:

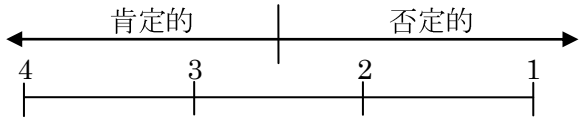


(理由:)

Q6. ズーミングビューを利用した関数の走査機能を使用しましたか？使用した場合にはこの機能がプログラムを理解する上で役に立ったかどうか採点してください(数字に丸)。

この機能を　　利用した・利用しなかった

評価:

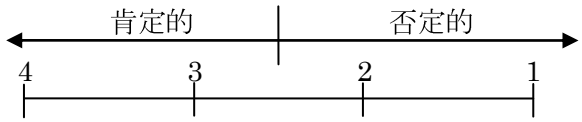


(理由:　　)

Q7. Eclipse エディタ上において、実行されたソースコード行にマーカを配置する機能を使用しましたか？使用した場合にはこの機能がプログラムを理解する上で役に立ったかどうか採点してください(数字に丸)。

この機能を　　利用した・利用しなかった

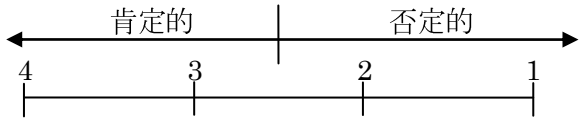
評価:



(理由:　　)

Q8. ORCA は GUI プログラムの理解に役立ったかどうか採点してください(数字に丸)。

評価:



(理由:　　)

Q9. 実際に ORCA があったら、プログラム理解作業に使いたいですか？

はい　・　いいえ (いいえの場合 理由:　　)

Q10. その他、ツールの改善要望や感想、プログラム理解作業に関してのご意見などを自由に記述してください。

X プログラム理解作業設問用紙

Q1. プログラムに対して以下の操作を行った際の実行行を含むファイル名をすべて列挙せよ。

Q2. プログラムに対して以下の操作をした際に実行されるメソッドの順序を答えよ。メソッドはクラス名.メソッド名()の形式で書くこと。

Q3. 以下の 2 つの処理におけるフローの違いはどこか説明せよ。(処理の分岐点を明らかにし、分岐後のそれぞれの処理について説明せよ。)

※ 解答するクラス、メソッド等は原則としてプログラムのプロジェクト内で定義されたファイルのみでよい。つまり、Java の標準の API で定義されているものに関しては記述しないこと。

ドローイングツールプログラム理解作業設問用紙

Q1. プログラムに対して以下の操作を行った際の実行行を含むファイル名をすべて列挙せよ。

(ア) Oval ボタンをクリック
(イ) キャンバスをマウスプレス→そのままマウスドラッグ→マウスリリース

操作時における前提条件:

- 何も図形が描画されていないこととする
- 描画図形の色や線の種類、太さは初期設定から変更しないものとする

Q2. プログラムに対して以下の操作をした際に実行されるメソッドの順序を答えよ。

(ア) 選択状態でない描画図形をクリック
ただし、図形を選択状態にするまでの処理を答えることとし、描画処理やヒストリの追加といった処理は含めないこととする。

操作時における前提条件:

- Select ボタンをクリック後であること
- 選択状態でない図形が 1 つ以上描画されていること

Q3. 以下の 2 つの処理におけるフローの違いはどこか説明せよ。(処理の分岐点を明らかにし、分岐後のそれぞれの処理について説明せよ。)

1. 実線での図形描画処理
2. 点線での図形描画処理

ただし、操作ではなく、描画処理の違いだけ答えることとする。

操作条件:

- 実線と点線の描画切り替え操作はコンボボックスの選択項目を変更した際に行われるが、
- また、描画する図形は矩形に限定する。

※ 解答するクラス、メソッド等は原則としてプログラムのプロジェクト内で定義されたファイルのみでよい。つまり、Java の標準の API で定義されているものに関しては記述しないこと。

ドラッグ＆ドロッププログラム理解作業設問用紙

Q1. プログラムに対して以下の操作を行った際の実行行を含むファイル名をすべて列挙せよ。

(ア) Ctrl ボタンを押しながら、DragSource 領域にあるオブジェクトをドラッグして、DropTarget 領域にドロップする(オブジェクトのコピーが行われる)

但し、答えには、ドラッグの認識から、ドロップの終了までのすべてのイベントについて記述すること。

Q2. プログラムに対して以下の操作をした際に実行されるメソッドの順序を答えよ。

(ア) DragSource 領域にあるオブジェクトをドラッグして、DragSource 領域にドロップする(オブジェクトのコピーも移動も行われない)

Q3. 以下の 2 つの処理におけるフローの違いはどこか説明せよ。(処理の分岐点を明らかにし、分岐後のそれぞれの処理について説明せよ。)

1. オブジェクトのコピー
2. オブジェクトの移動

ただし、ドラッグ開始時の認識の差ではなく、ドラッグ＆ドロップ終了時におけるオブジェクトが消える(移動)処理と消えない(コピー)処理自体の違いを述べること。

※ 解答するクラス、メソッド等は原則としてプログラムのプロジェクト内で定義されたファイルのみでよい。つまり、Java の標準の API で定義されているものに関しては記述しないこと。

(練習問題) 数字当てクイズプログラム理解作業設問用紙

Q1. プログラムに対して以下の操作を行った際の実行行を含むファイル名をすべて列挙せよ。

(ア) Guess ボタンをクリック

操作時における条件:

- 前提として、コンボボックスのに適当な値が設定されていて、値が重複していないこと。(正しい例: 「0123」、悪い例: 「1123」)
- 予測が正解していないこと。

Q2. プログラムに対して以下の操作をした際に実行されるメソッドの順序を答えよ。

(ア) Surrender ボタンをクリック

Q3. 以下の 2 つの処理におけるフローの違いはどこか説明せよ。(処理の分岐点を明らかにし、分岐後のそれぞれの処理について説明せよ。)

1. コンボボックスに正常な値が設定されている状態(「1234」など)で Guess ボタンをクリック
2. コンボボックスに不正な値が設定されている状態(「1222」など)で Guess ボタンをクリック

※ 解答するクラス、メソッド等は原則としてプログラムのプロジェクト内で定義されたファイルのみでよい。つまり、Java の標準の API で定義されているものに関しては記述しないこと。

ネットワーク図エディタ理解作業設問用紙

Q1. プログラムに対して以下の操作を行った際の実行行を含むファイル名をすべて列挙せよ。

(ア) コンピュータの形をしたアイコンに対してマウスの右ボタンをプレス→そのままマウスドラッグ→赤い線上でマウスの右ボタンをリリース

操作時における条件:

- 初期状態からオブジェクトの追加や削除を行っていないこと。
- 手書きメモ入力モードでないこと。

但し、マウス操作に関する処理で実行された行を含むファイルのみ答えることとし、描画処理で実行されたクラスは含まないこと。

Q2. プログラムに対して以下の操作をした際に実行されるメソッドの順序を答えよ。

(ア) add host ボタンをクリック

但し、ボタンの押下操作に関するメソッドの呼び出し順序のみを答え、描画処理での実行を含まないこと。

Q3. 以下の 2 つの処理におけるフローの違いはどこか説明せよ。(処理の分岐点を明らかにし、分岐後のそれぞれの処理について説明せよ。)

1. サーバのアイコン上にマウスカーソルが乗っていない時の図形描画処理
2. サーバのアイコン上にマウスカーソルが乗っている時の図形描画処理

ただし、操作ではなく、描画処理の違いだけ答えることとする。

操作時における条件:

- 前提として、サーバのアイコンは画面上に 1 つだけ表示されていること。
- 手書きメモ入力モードでないこと。

※ 解答するクラス、メソッド等は原則としてプログラムのプロジェクト内で定義されたファイルのみでよい。つまり、Java の標準の API や外部 JAR ファイル内で定義されているものに関しては記述しないこと。

X プログラム理解作業解答用紙

Q1. 実行されたファイル名にチェックしてください。

■ Card ■ CardPile ■ DeckPile □ DiscardPile

□ Solitare ■ SolitareMain □ SuitPile □ TablePile

Q2.

Solitare.mouseDown()→TablePile.select()┬→TablePile.empty()
├→TablePile.addCard()→Ⓐへ
└→CardPile.remove ()

Ⓐ→TablePile.select()→TablePile.empty()

Q3.

TablePile.select()内の
if (card.isOpen())
の分岐によって処理が分かれる

カードが開かれた状態でクリックした場合、if 文は true となり、
クリックされたカードを選択状態し、選択状態のカードを描画する処理を行う。

一方、カードが閉じられた状態でクリックした場合、if 文は false となり、
エラーメッセージを画面に表示する処理を行う。

ドローイングツールプログラム理解作業解答用紙

Q1. 実行されたファイル名にチェックしてください。

☐ Application	☐ DeepCopyVector	☐ DrawingButton	☐ DrawingState
☐ Mediator	☐ MyCanvas	☐ MyDashStroke	☐ MyDrawing
☐ MyOval	☐ MyRectangle	☐ MySimpleLine	☐ OvalButton
☐ RectButton	☐ SelectButton	☐ SimpleLineButton	☐ StateManager

Q2.

Q3.

ドラッグ＆ドロッププログラム理解作業解答用紙

Q1. 実行されたファイル名にチェックしてください。

☐ DragSourceHostPanel　☐ DragSourceLabel　☐ DragSourceLabelListener

☐ DropTargetPanel　☐ MyLabel　☐TestFrame

Q2.

Q3.

(練習問題) 数字当てクイズプログラム理解作業解答用紙

Q1. 実行されたファイル名にチェックしてください。

☐ Controls ☐ GuessActionListener ☐ Main ☐ NumberBox

☐ NumberBoxActionListener ☐ NumberGuessQuiz

☐ NumberGuessQuizWindow ☐ NumberSquence ☐ SurrenderActionListener

Q2.

Q3.

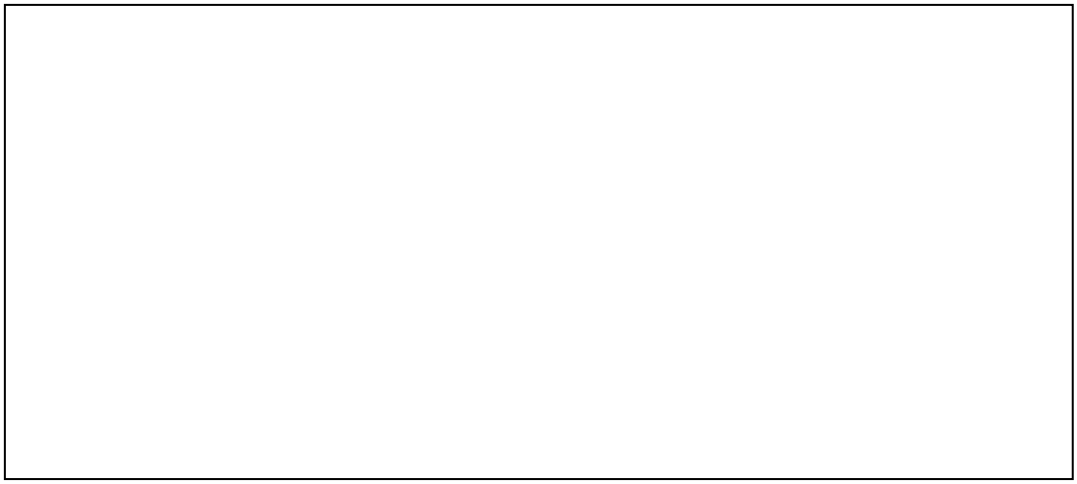
ネットワーク図エディタプログラム理解作業解答用紙

Q1. 実行されたファイル名にチェックしてください。

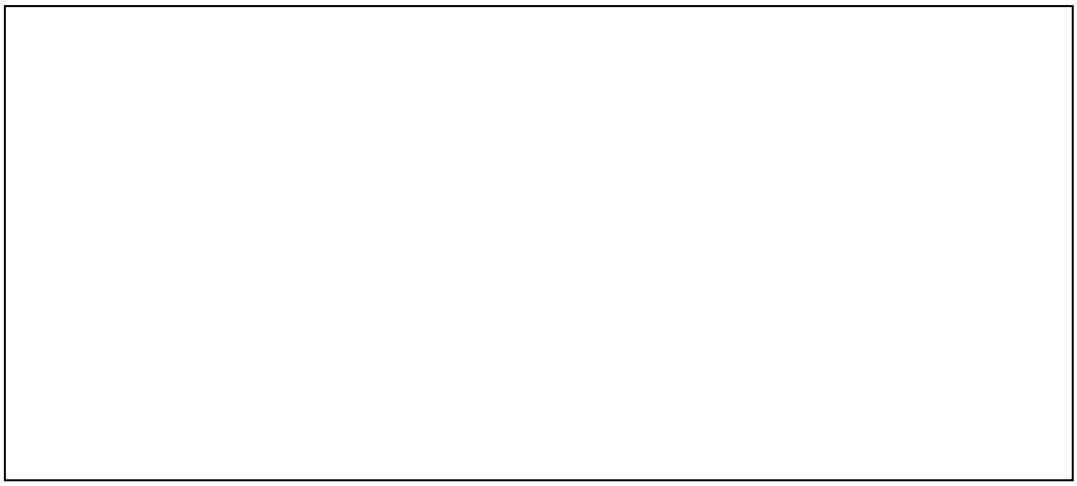
☐ Canvas ☐ GHost ☐ GNode ☐ GObject ☐ GSegment

☐ GServer ☐ Main ☐ NetworkApplication ☐ NetworkPanel

Q2.



Q3.



(練習問題) 数字当てクイズプログラム理解作業解答用紙(解答編)

Q1. 実行されたファイル名にチェックしてください。

■ Controls ■ GuessActionListener □ Main □ NumberBox
□ NumberBoxActionListener ■ NumberGuessQuiz
□ NumberGuessQuizWindow ■ NumberSquence □ SurrenderActionListener

Q2.

SurrenderActionListener.actionPerformed()→NumberGuessQuiz.surrender()
┬→NumberSequence.toString()
└→Controls.setMessage()

Q3.

NumberGuessQuiz.guess ()内の
if (hasSameNumber())の分岐によって処理が変わる

コンボボックスに正常な値が設定されている状態では if 文は false を返し、
else if 以降の正しい推測かどうかの判定、及び hit 数と blow 数の表示に移る。

一方、コンボボックスに不正な値が設定されている状態では if 文は true を返し、
エラーメッセージを表示する。

付録2. 評価実験で用いたORCAの操作マニュアル

GUI プログラム実行情報可視化ツール

利用マニュアル

Ver. 1.1

目次

1. はじめに

ORCA(Operation Reaction Code Analyzer)は、Java によって記述された GUI プログラムを対象としたプログラム理解支援ツールです。

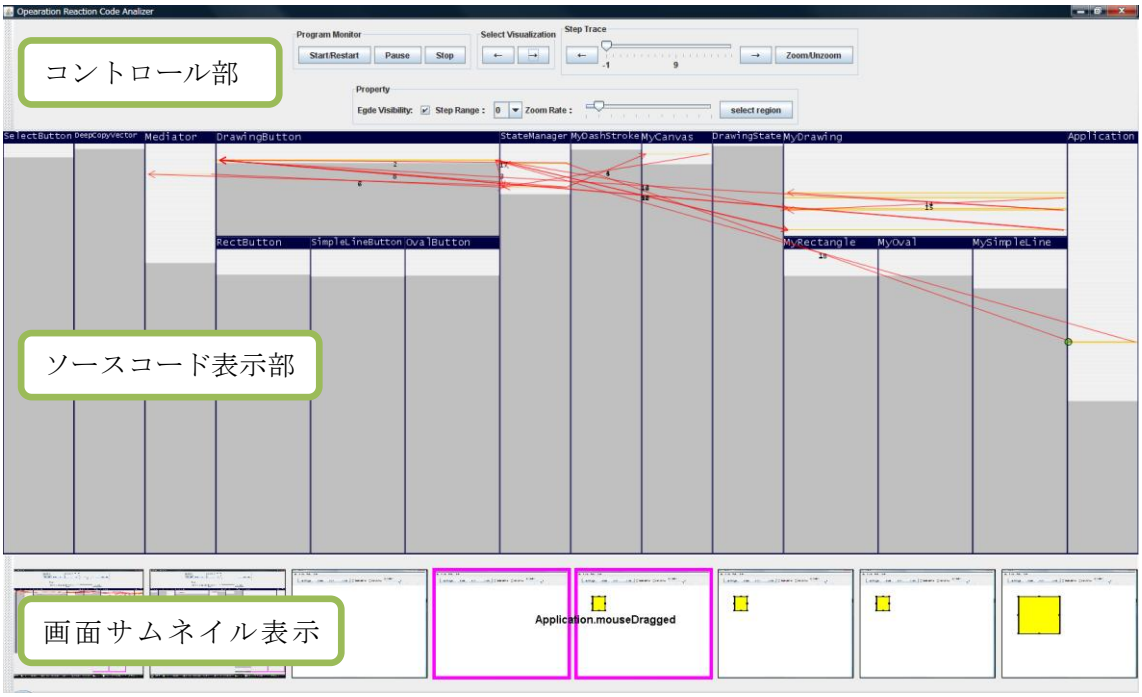
ORCA では対象となる GUI プログラム(以下、ターゲットプログラム)の実行を監視し、ターゲットプログラムへの操作に対するソースコード実行の流れをオンラインで可視化します。

ORCA は統合開発環境 Eclipse のプラグインとして実装されています。ORCA では「ソースコードの実行情報の表示」を始めとした Eclipse との連動機能を提供することで、開発者のプログラム理解作業の支援をより豊かなものにしています。

2. 機能概要

2.1. 概観

ORCA は「コントロール部」、「ソースコード表示部」、「画面サムネイル表示部」から構成されます。コントロール部ではターゲットプログラムの起動や可視化結果の切り替えといった ORCA の操作を行います。ソースコード表示部にはソースコード実行の可視化結果が表示されます。画面サムネイル表示部にはターゲットプログラムの画面サムネイルが時系列順に表示されます。

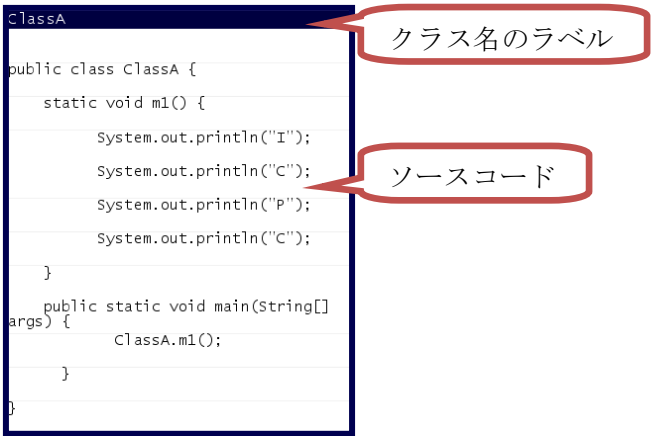


2.2. 実行の可視化表示

ORCA ではソースコード全体を一画面上に表示します。ソースコードの実行があった場合には実行の様子が重畳表示されます。ソースコードは以下の要素によって表現されます。

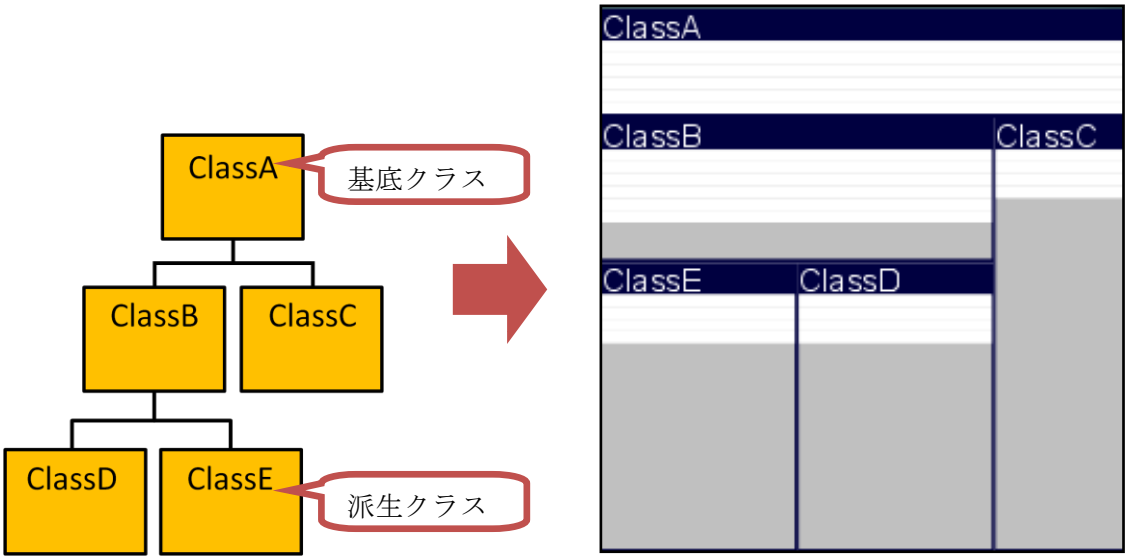
- **クラス表現**

ソースコードはクラス定義ごとに区切って表示します。各クラス表示の上部にはクラス名が表示され、その下にソースコードが縮小表示されます。



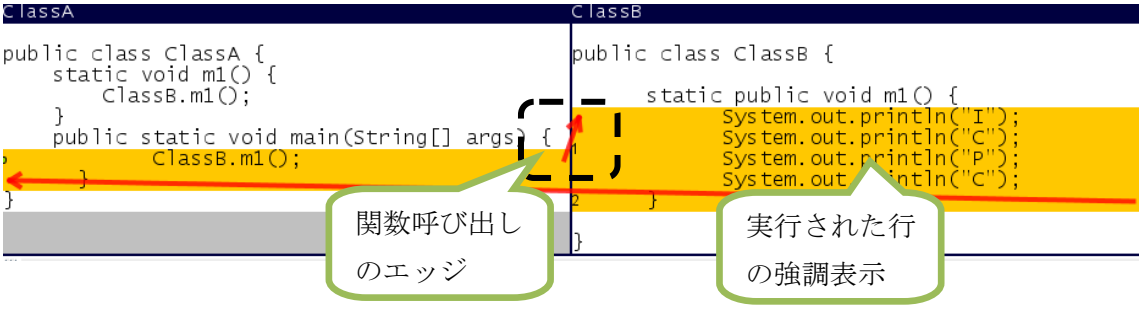
- **クラス階層**

各クラスはクラス階層に従って配置されます。クラス階層は下図のように、画面上部に基底クラスが下部に派生クラスが位置するように配置されます。



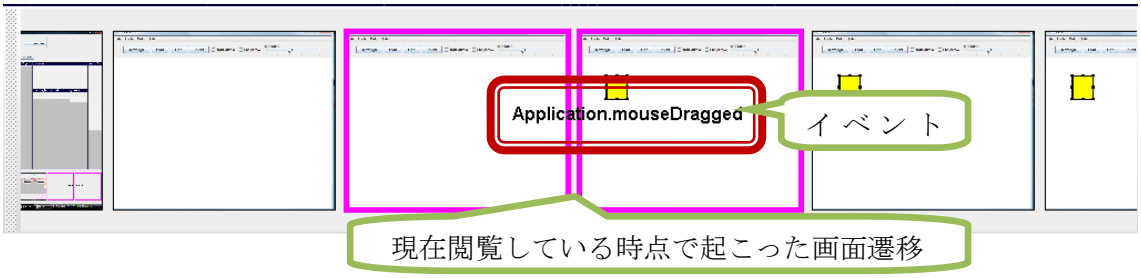
- **実行ソースコード行・関数呼び出し**

実行されたソースコード行は黄色で強調表示されます。関数呼び出しがあった場合には矢印によってエッジ付けされます。



- **画面サムネイル表示**

サムネイル表示部には、GUI プログラムを操作した際に発生した画面遷移を時系列順に並べてサムネイル表示する。枠線による強調がされている画面対が現在注目している画面遷移であることを示す。ソースコード表示部ではこの画面遷移が発生した際に実行されたソースコードの可視化結果を表示する。現在注目している画面遷移にはそのイベント名も付加表示する。



参考: イベント名について

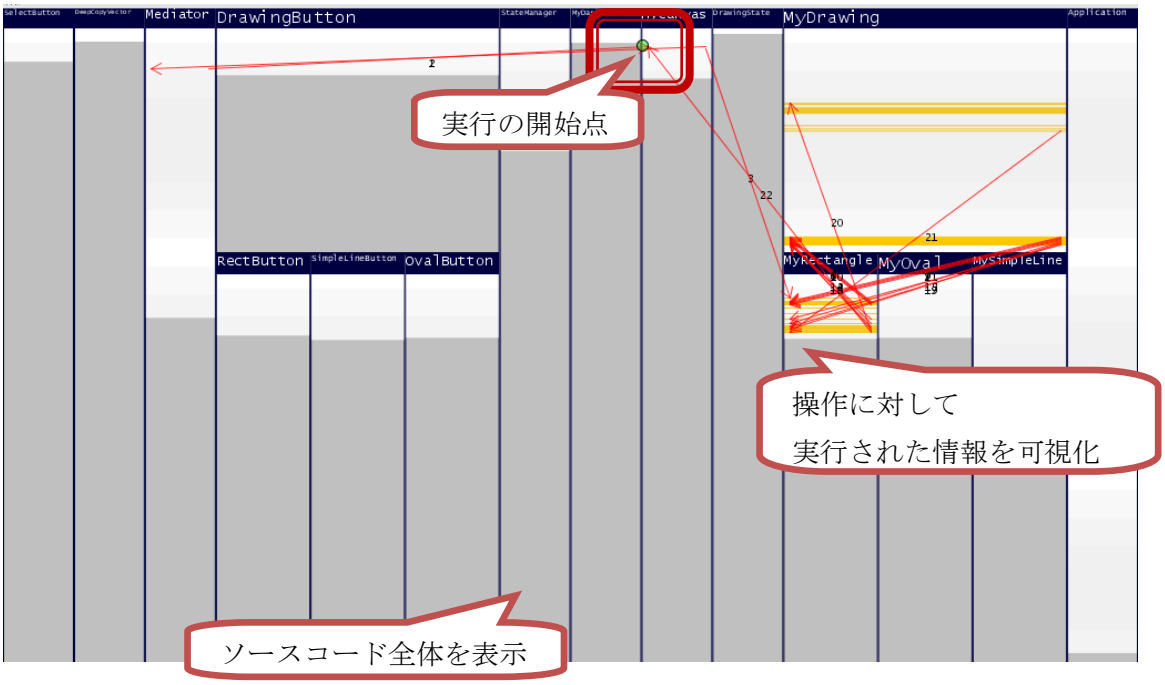
ORCA ではイベント名として、イベントのトリガとなったクラスとメソッド名を表示している。マウスなどの頻出する操作との対応付けは以下の通りである(Swing の場合)。

GUI への操作	表示されるイベント名の例
マウスプレス	XXX.mousePressed
マウスドラッグ中	XXX.mouseDraged
マウスリリース	XXX.mouseReleased
マウスクリック	XXX.mouseClicked
マウスを動かす	XXX.mouseMoved
ボタンなどをクリック	XXX.actionPerformed
描画処理(内部的な呼び出し)	XXX.paint or XXX.paintComponent

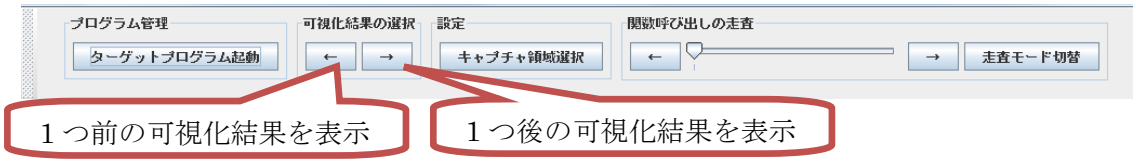
2.3. ツールの実行

コントロール部にある「ターゲットプログラム起動」ボタンを押下することで、ターゲットプログラムの実行の監視を開始します。

ターゲットプログラムが起動された後で、ターゲットプログラムに対して操作を行うと可視化結果がリアルタイムに更新されます。可視化結果は以下に示すようなソースコード全体の俯瞰表示として表示されます。俯瞰表示の各要素は 2.2 で述べたとおりです。



各可視化結果を再閲覧することができます。その際には、以下のようにコントロール部にある「可視化結果の選択」の矢印ボタンを押下して、可視化結果の切り替えを行います。

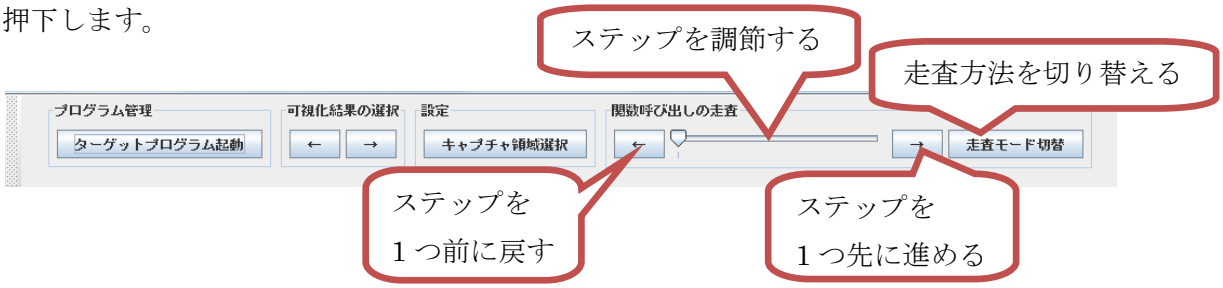


2.4. 関数呼び出しの走査機能

可視化結果に対して、関数呼び出しのエッジを順々に詳細表示しながらたどることができます。走査方法は「ポップアップビューを用いた走査」、「ズーミングビューを用いた走査」の 2 種類があります。

どちらの走査もコントロール部の「関数呼び出しの走査用ユーザインタフェース(下図参照)」を用いてコントロールします。スライダーバーをドラッグするか、矢印ボタンを押下することで、前後のエッジに遷移することができます。また、ソースコード表示部に表示されているエッジを直接クリックすることで、そのエッジの詳細表示にジャンプすることが可能です。

走査方法は相互に切り替えることが可能です。切替時には「走査モード切替」ボタンを押下します。

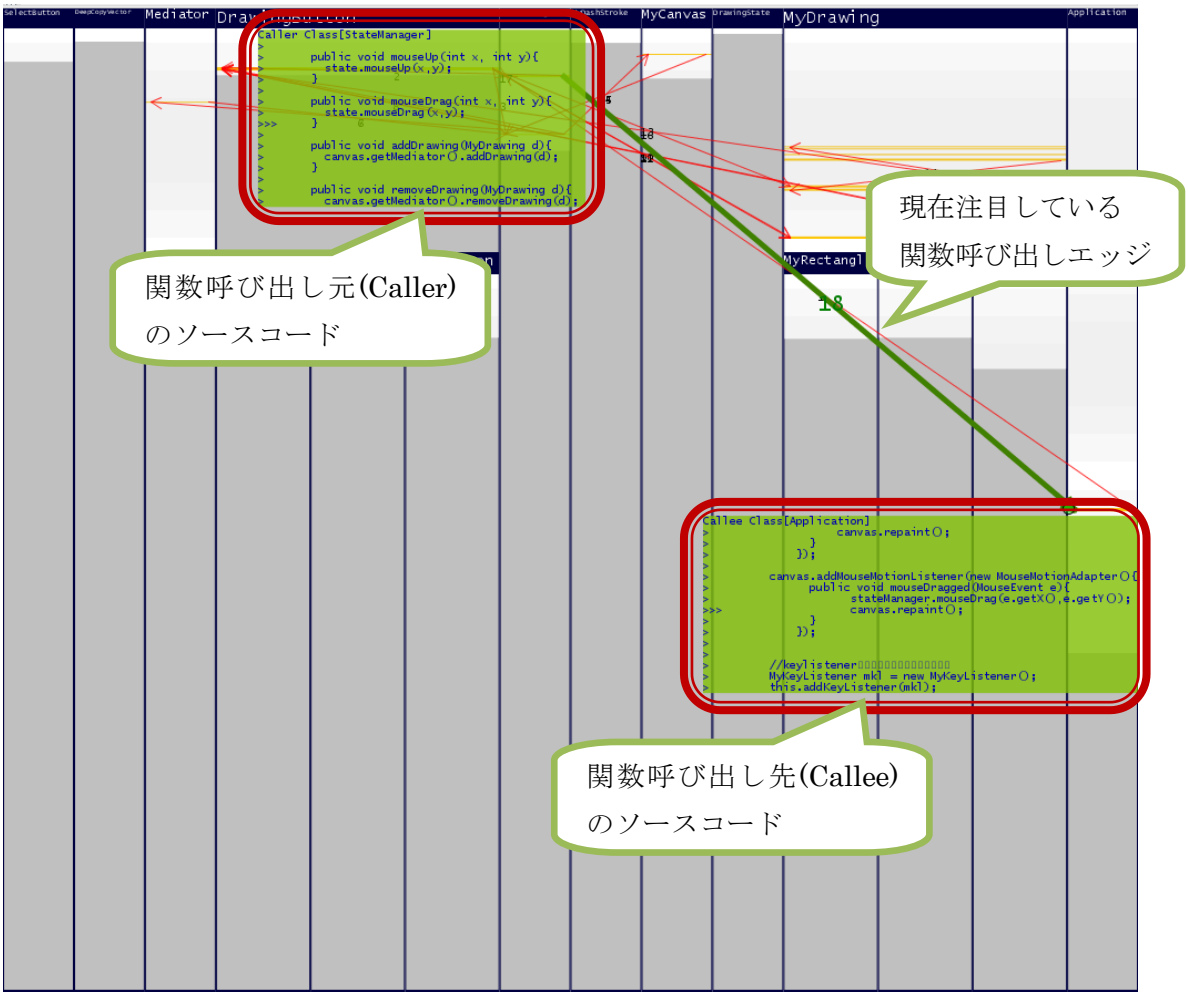


以下に各走査方法によるビューを示します。

- **ポップアップビューを用いた走査**

俯瞰表示上に関数呼び出しをポップアップ表示します。ポップアップ上部には呼び出し情報（呼び出し元なら Caller、先なら Callee)とクラス名および関数名を表示します。その下に関数呼び出し元と先のソースコード付近の数行を示します。ソースコードにはアノテーション「>」を付加します。関数呼び出し元と先のソースコード行にはアノテーション「>>>」を付加します。

ポップアップはドラッグで位置を移動させることが可能です。またマウスホイールで表示している行を上下に変更することができます。

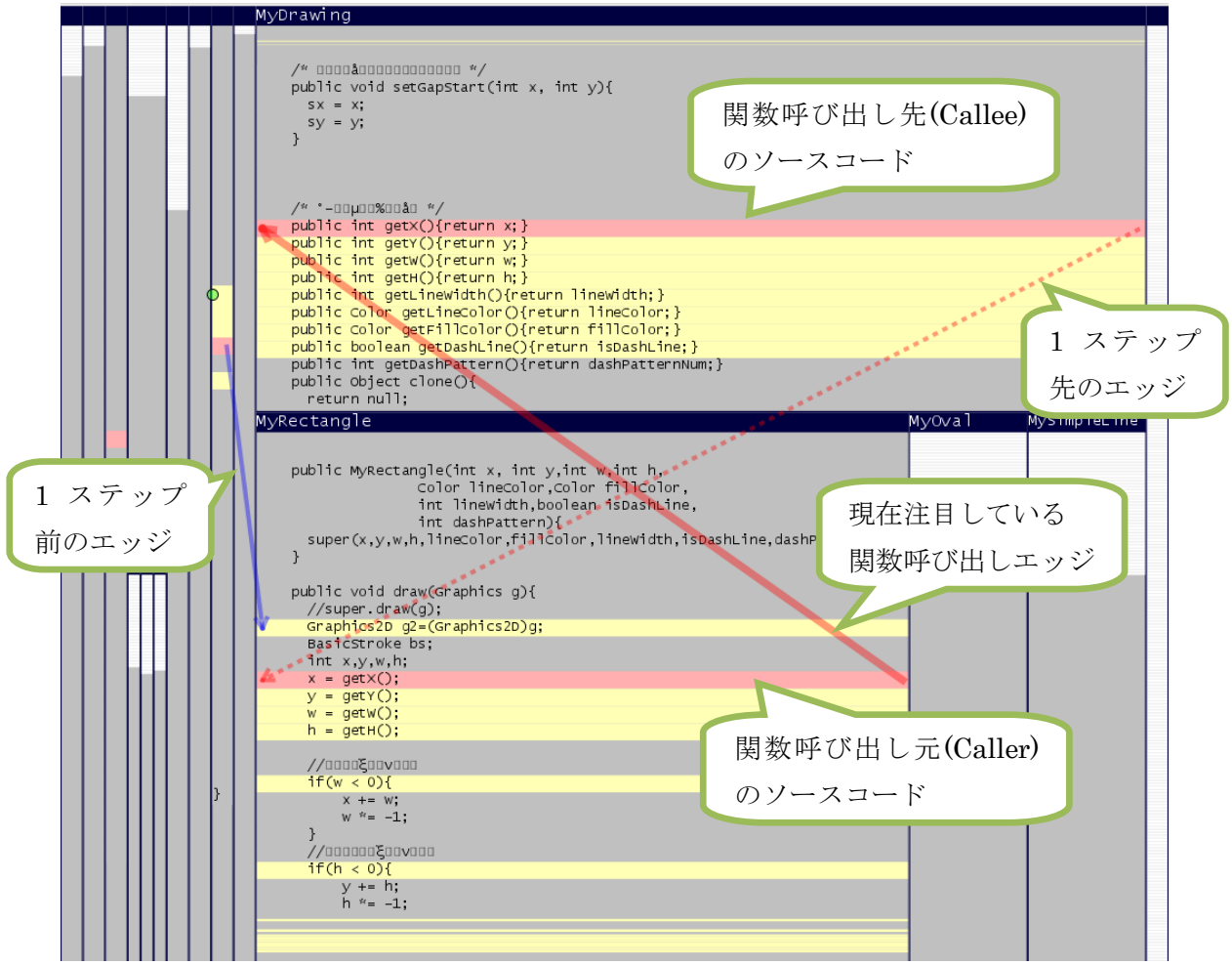


● **ズーミングビューを用いた走査**

俯瞰表示の概観を保ったまま、関数呼び出し元と先のクラス表示とソースコード行に対するズーミング表示をします。

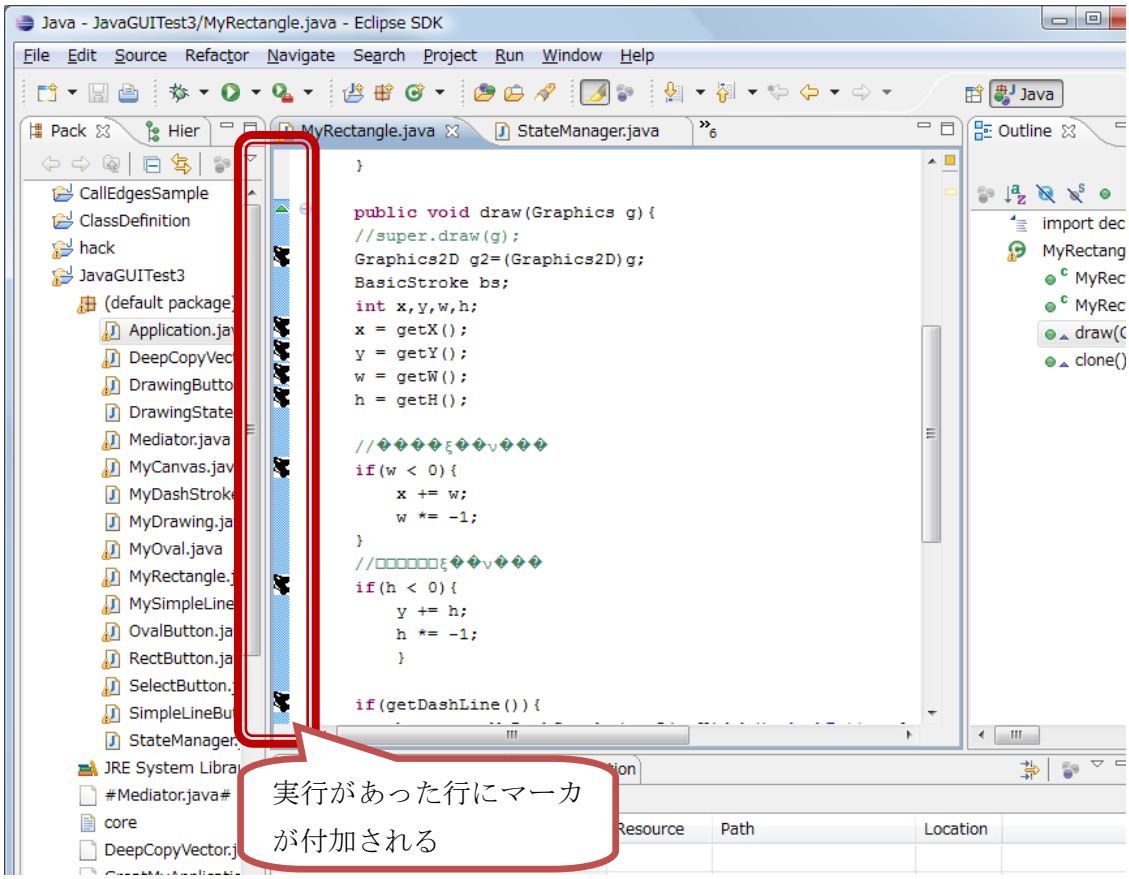
ズーミングビューでは関数呼び出し元と先のクラス表示とソースコード行の付近だけを拡大して、それ以外の部分を縮小して表示します。関数呼び出し元と先のソースコード行をピンク色で強調表示し、それ以外の実行ソースコード行を黄色で強調表示します。現在注目している関数呼び出しのエッジは赤色実線で表示します。さらに、1 ステップ前の関数呼び出し、1 ステップ先の関数呼び出しのエッジをそれぞれ青色実線、赤色破線で表示します。

ソースコード上でマウスホイールを操作することで、ソースコード上のズーミング焦点を上下に動かすことができます。



- **Eclipse エディタへのジャンプ**

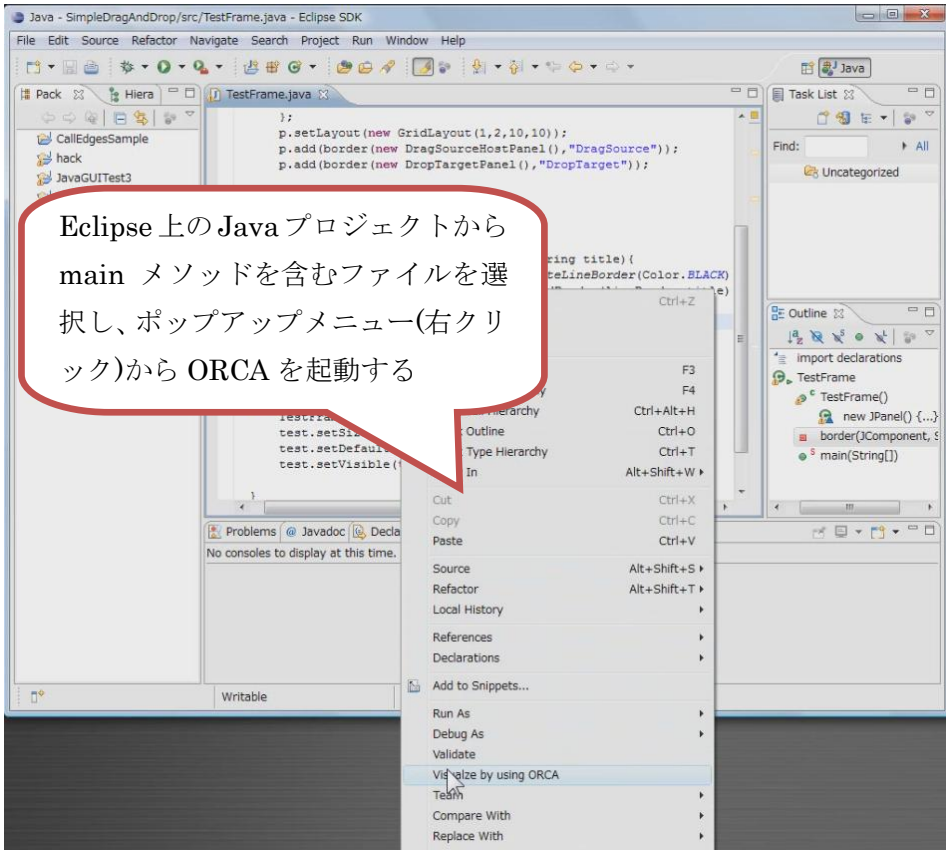
ORCA ビューのソースコード表示部のソースコードをクリックすると、Eclipse エディタ上の対応するソースコード行にジャンプすることができます。Eclipse エディタ上では実行されたソースコード行に対してマーカが配置されているので、Eclipse エディタ上でもソースコードの実行部分を把握することが可能です。



3. 操作手順

1. Eclipse にターゲットプログラムを含むプロジェクトをインポートします
2. プロジェクトからターゲットプログラムの main 関数を含むファイルを選択します
3. 右クリックでポップアップメニューを起動し「Visualize by using ORCA」を選択し、ORCA を起動します
4. コントロール部にある開始ボタンを押下し、ターゲットプログラムを起動します
5. (GUI 画面を取得する領域の指定を行います)
6. 起動されたターゲットプログラムの GUI に対して操作を行います
7. 操作に対してリアルタイムに更新される ORCA の可視化情報を閲覧します
8. 操作終了後はコントロール部にある可視化切り替えボタンを利用し、再閲覧したい可視化情報を選択します。
9. 再表示された可視化結果に対して詳細な閲覧を行います
   (ア) ポップアップビューを利用した関数呼び出しの走査
   (イ) ズーミングビューを利用した関数呼び出しの走査
   (ウ) Eclipse エディタ上でのソースコード実行の閲覧
   これらの詳細閲覧機能は相互に切り替えを行うことが可能です。
10. 手順 6、9 を繰り返し、理解作業を行います

※ **(補足 1)** Eclipse からの ORCA の起動の仕方



※ **(補足 2)** GUI 画面キャプチャ領域の設定

GUI 画面のキャプチャ領域を設定する場合、まずコントロール部にある「キャプチャ領域選択」ボタンを押下し、選択用フレームを起動します。起動されたフレーム内に表示された画面に対して、左上から右下に向かってドラッグを行い、キャプチャ領域を選択します。この際選択領域には赤い矩形領域が重畳表示されます。選択終了後にフレームの右上にある「x」ボタン(閉じるボタン)を押下します。以降は選択した領域がキャプチャされるようになります。