平成18年度

筑波大学第三学群情報学類

卒業研究論文

題目 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

主専攻　　情報科学主専攻

著者　　佐藤 竜也

指導教員　田中 二郎　志築 文太郎　三末 和男　高橋 伸

要　旨

本研究では、GUI を持つプログラムに機能追加をする際に必要となるプログラムの理解を支援するために、GUI が持つ特徴を活かしたプログラム可視化システムを作成した。

本システムは GUI を持つプログラムの理解するための重要な要素である GUI への操作のプログラム全体に対する実装部分、クラス階層、関数呼び出し階層を効果的に見せるための可視化を行う。また対象プログラムの GUI への操作に対してインタラクティブに反応することで、GUI への操作と可視化情報との対応付けを容易にする。本システムによって機能追加のために必要なプログラム情報を取得できるため、ソースコードを読む作業のコストは軽減し、作業効率を向上させることが可能となる。

目 次

第 1 章　序論 **1**
1.1　プログラムを理解するための要素 1
1.2　プログラム理解のための作業に関する問題 2
1.2.1　一般的なツールを利用した場合の問題 2
1.2.2　既存のプログラム可視化システムを利用した場合の問題 4
1.3　本研究で目指す可視化 4

第 2 章　関連研究 **6**

第 3 章　GUI を持つプログラムの可視化 **8**
3.1　可視化の概要 8
3.2　概観表示 11
3.3　ズーミング機能を用いた詳細表示 12
3.3.1　クラスのズーミング方法 13
3.3.2　ソースコードのズーミング方法 15
3.4　対象プログラムの GUI を含む画面状態遷移のサムネイル表示 16

第 4 章　実装 **18**
4.1　Java Debug Interface(JDI) の概要 18
4.2　実行時情報取得部の実装 19
4.3　可視化ビュー部の実装 20

第 5 章　利用シナリオ (本システムを用いたプログラム理解の方法) **26**
5.1　GUI を持つプログラムを理解するための要素を得る方法 26
5.2　具体的な利用シナリオ 26

第 6 章　今後の展望 **39**

第 7 章　結論 **40**

謝辞 **41**

参考文献 **42**

図 目 次

3.1 クラス表示 8
3.2 クラスの親子関係のツリー表示 9
3.3 ソースコード全体の表示 10
3.4 関数呼び出しのエッジ 10
3.5 概観表示での完成した静的グラフ 11
3.6 ズーミングを用いた詳細表示 12
3.7 木構造の表示領域マッピング 14
3.8 Fisheye ビューによるクラス表示 14
3.9 Fisheye ビューを用いたソースコード表示 (Furnas) 15
3.10 ソースコードのズーミング表示 16
3.11 サムネイル表示 17

4.1 システム構成図 19
4.2 システムのスナップショット 20
4.3 グラフ表示部 (概観表示) 22
4.4 グラフ表示部 (詳細表示) 22
4.5 ソースコードの各行の領域決定法 23
4.6 ソースコードズーミングのサンプル 24
4.7 コントロール部 25
4.8 サムネイル表示部 25

5.1 サンプルプログラム 27
5.2 サンプルプログラムのクラス関係図 27
5.3 システムの起動時 28
5.4 対象プログラム起動時 29
5.5 矩形ボタン選択時 30
5.6 図形描画のためのマウス操作時 32
5.7 マウスプレス時の詳細表示 37

第1章　序論

他人の書いたプログラムに機能を追加する場合、開発者はそのプログラムを理解するために、実行の外的振る舞いや関数呼び出し、クラス階層などの要素を把握する必要がある。ソースコードのみからこの把握を行う場合、プログラムの規模が大きくなるとプログラムの理解に必要な部分を即座に見出すことは難しい。

また、現在開発されるソフトウェアの多くは GUI(Graphical User Interface) を持つ。GUI はマウスやキーボードを用いた操作に応じて処理を行い視覚的なフィードバックを返す特徴を持つ。そこで本研究ではこの特徴を活かしたプログラム可視化手法を提案することで GUI を持つプログラムの理解を支援し、機能追加時にソースコードを読む作業のコストを最小化し作業効率を向上させることを目的とする。

1.1　プログラムを理解するための要素

GUI を持つプログラムに機能追加を行う際に、開発者がそのプログラムを理解しなければならないことがある。このとき、開発者はソースコードを閲覧しながら、以下に挙げるプログラムの理解すべき重要な要素を把握する。

1. 各操作のプログラム全体に対する実装部分
2. 関数呼び出し階層
3. クラス階層

ある機能があったとき、その機能にまつわる操作を実装している部分が、機能実装部分となる。ここで操作とは、GUI に対するマウスやキーボードによる入力のことを示す。例えば、ドローツールの図形描画機能があるとき、マウスやキーボードを用いた図形の種類の選択や色の変更、キャンバス上への描画などがこの機能にまつわる操作である。その操作が行われたときの処理内容が書かれているソースコード部分が、機能実装部分となる。

ここで追加する機能は大きく以下のように分けられる。

1. 既存機能を修正して実現する機能
2. 既存機能に類似した機能
3. 既存機能を基にしない新規機能

ここでドローツールについて、それぞれの項目の例を挙げる。第 1 項目の例として、図形の包含領域の判定方法の変更を挙げる。現時点では図形の領域は図形を含む矩形領域で設定されている。このとき直線に関しては包含領域を矩形ではなく直線に合わせたものにしたいとする。そのような場合には直線描画機能を修正することになる。第 2 項目の例として、描画図形の種類の追加を挙げる。これは既存図形の描画を模倣することで実現できる。そのため、これは既存機能の修正と言い換えることもできる。第 3 項目の例として、描画の Undo/Redo 機能を挙げる。現在の図形の保存の方法といった既存機能がこの機能に影響してくるものの、ベースとなる機能は無いため、開発者は新たに機能を作ることになる。

既存機能を修正することで実現する機能の場合には、既存機能にまつわるすべての操作の実装部分を把握する必要がある。つまり要素 1 の把握の繰り返しこそが機能の実装自体の把握となる。既存機能に類似した機能を追加する場合、機能はその既存機能を模倣することで実現可能である。この場合には模倣する既存機能にまつわる操作について同様の作業を行えばよい。既存機能とは別に新規機能を追加する場合でも、既存のプログラムに追加する以上、他の機能に影響する部分は出てくる。他の機能の実装部分を理解し、影響部分を把握しなければならないため、要素 1 の把握は必要となる。

それらの実装部分は、複数のクラスに点在していることが多い。そのとき、実装部分はまったく別々に呼び出されるのではなく、互いに呼び出しあっている。それらの呼び出しは開発者の操作によって呼び出された関数に始まって、次々と関数が呼び出されることになる。例えば、図形描画が行われる際には、操作によって呼び出された関数に始まり、図形オブジェクトの生成、キャンバスへの図形の追加といった関数が呼び出される。機能がどのような形で実装されているのかを理解するためには、その呼び出しの結びつきを知るために要素 2 の把握が必要となる。

オブジェクト指向が主流となった現在では、多くの場面でクラス継承が用いられる。GUI を持つプログラムの機能に関しても例外ではなく、操作に対する一連の関数呼び出しで使用されるクラスも何らかのクラスを継承したものである可能性がある。例えば、機能の雛形となるようなクラスとそれに対して具体的な肉付けを行った継承クラスがあるとき、継承クラスは元のクラスの関数を使用あるいはオーバライドする場合が出てくる。そのようなときには要素 3 を把握しなければならない。ときには継承前の親クラスを修正の対象とする場合もあり、要素 3 はそれを判断するための重要な材料となる。例えばドローツールでは、図形の共通の属性や動作を持った図形抽象クラスを各図形クラスが継承する。この場合、各図形クラスは抽象クラスの図形描画関数をオーバライドして、各図形の描画を設定する。このような仕組みを見つけ出すには要素 3 を把握することが必要である。

1.2 プログラム理解のための作業に関する問題

1.2.1 一般的なツールを利用した場合の問題

上記の要素を把握するために開発者はまず、そのプログラムを実行する。そしてプログラムに対して操作をすることで外的な振る舞いを確認する。その後はソースコードを閲覧しなが

ら、関数の呼び出し階層やクラス階層の要素を把握して、プログラムの構造を理解することになる。大規模なプログラムが多くなってきている昨今では、ソースコードのみからこの把握を行うと、膨大な作業コストを消費することになる。また機能追加を目的とする場合、ソースコードの修正を必要とする部分はプログラムのごく一部分であることが多い。しかしソースコードのみからの探索では、理解に必要な部分を即座に見出すことは難しい。

現在では多くの場面で統合開発環境 (IDE) が使われる。IDE とはエディタ、コンパイラ、デバッガなど、プログラミングに必要なツールが一つのインターフェースで統合して扱える環境の総称である。例えば、Eclipse は Sun Micro System が提供する Java を対象としたフリーの IDE である [17]。Eclipse は豊富な開発ツールを備えているため、広く使われている。このような IDE ではクラス階層や関数の呼び出し階層を静的に閲覧することができる。またデバッガが備えられているため、プログラムのデバッグを行うことで、上記の作業を行うことも可能である。

またソフトウェア開発のためのモデリング言語として、統一モデリング言語 (UML, Unified Modeling Language) がある [15]。10 種類を超える図を必要に応じて使い分けるこの言語はモデリング言語としては最も普及している。本来、UML は設計 (モデリング) の段階で使用される。UML の各図はその用途に応じて、プログラムの特徴を忠実に表している。特にクラス図、オブジェクト図、シーケンス図は、クラスや関数呼び出しといった実装に近い部分を表現する。つまり UML 図はソースコードを見ることなしにプログラムの構造理解のためにも使用することができる。現在では、ソースコードからそのプログラムの UML 図を導きだすツールも登場している [16]。

これらツールを組み合わせて使えば、ソースコードのみからの単純な検索に比べ作業コストは減少する。しかし本来これらのツールはプログラムの理解を目的に作られたものではないため、理解に必要な情報を十分には把握しづらい。例えば、UML の複数の図を使ってプログラム全体の把握を行うことはできるが、機能を実装している部分を見つけ出すのは難しい。デバッガを使いブレークポイントを決める場合には、ある程度プログラム全体の流れを知っておく必要がある。本研究では機能追加を目的としている。そのような場合には、プログラム全体についての理解することはそれほど重要ではない。

また、GUI を持つプログラムを理解の対象とする場合には、それ特有の問題が発生する。GUI は視覚的なフィードバックを返すという特徴を持つ。つまり、グラフィカルな表示がプログラムを理解するための一つの要素でもある。しかし、上記のツールはグラフィカルな表示とは切り離されて行われる。ドローツールでは描画されている図形の形や色といったグラフィカルな情報が意味をなしている。それらはソースコード上では色や形状を表す変数などで表現されている。デバッガなどを使った場合には、それらの変数情報を取得することはできる。しかし、それに付け加えてその図形の表示自体を与えたほうが、よりプログラムの理解が進むものと思われる。

さらに、GUI を持つプログラムはマウスやキーボードによる操作に応じて、動的にプログラムを実行し、インタラクティブに状態を変えるという特徴を持つ。しかし、これらを使った作業はプログラムの実行とは切り離されて行われる。プログラムへの操作と作業によるプ

ログラム実行情報取得が別々に行われるので、後でそれぞれを対応付ける必要がある。しかし操作と情報取得の間で時間差が生じるため、開発者はいつ、どのような操作を行ったかを忘れてしまい、操作とプログラムの実行情報を結びつけにくい。そのため、これらの特徴を活かしたシステムを作ることで、作業コストの大きな改善が図れる。

1.2.2 既存のプログラム可視化システムを利用した場合の問題

プログラム理解のための作業に関する問題の解決策の一つとしてプログラム可視化の研究が多くなされてきた [9]。これはプログラムに対して視覚的な表示をすることによって、プログラムの理解やデバッグをするために役立つデータを効果的に見せるというものである。可視化された情報を見ることによって、前節で紹介したツールよりも、容易にプログラム理解のための要素を把握することができる。プログラムの可視化手法はソースコードそのものを表示に利用するコード可視化手法や、プログラム内のデータを何らかの形で表示するデータ可視化手法などに分類され、古くからこれらについての研究がなされてきた。

一方で、GUI を持つプログラムを対象に絞った可視化の研究はさほど行われていない。つまり、プログラム可視化の分野では、GUI を持つプログラムとコンソールプログラムとを区別して扱うことは少ない。しかし、前節で述べたように GUI には特徴があり、それを活かした可視化を行うことで理解するための作業コストをより少なくすることができる。例えば、多くの可視化システムではプログラムの実行と可視化が別々に行われる。あらかじめ実行トレース情報をとっておくことによって、既知の情報だけを元に可視化が行える。これによって情報が整理された表示や高速な表示を行うことが可能となる。しかし、このような手法は GUI を持つプログラムへの操作と可視化情報との対応付けがしづらくなってしまう。

また、これらの可視化システムでは GUI の画面表示自体は利用しないが、これもプログラムを理解するための情報として価値がある。

つまり、既存の可視化システムでは、一般ツールを用いた際のプログラム理解における問題は解決しているものの、GUI の特徴を活かしきれていない。これらを改善することでより GUI を持つプログラムに適した可視化が行えると考えられる。

本研究と特に関連のある研究については次章でくわしく説明する。

1.3 本研究で目指す可視化

GUI を持つプログラムに対する理解を支援するためには、前節までに述べた事柄を踏まえた可視化を行う必要がある。そこで本研究では、以下のような特徴を持った可視化を行う

可視化する要素

以下に示す GUI を持つプログラムに機能追加を行う場合の重要な要素を効果的に視覚化する。

1. 各操作のプログラム全体に対する実装部分
2. 関数呼び出し階層
3. クラス階層

GUI 表示との同期

プログラムへの操作と可視化が別々に行われることによる対応付けのしづらさを解決するために、我々は GUI 実行画面の操作に対してインタラクティブに反応する可視化システムを構築する。また、操作と GUI 表示画面との対応付けが容易に行えるシステムを目指す。

以上のような可視化手法を行うことで GUI を持つプログラムを理解するためのコストの最小化を目指す。

第2章　関連研究

ここで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。

寺田らによるプログラム紙芝居はデバッガを紙芝居メタファを用いて拡張したものある [2, 3]。ソースコード上のプログラムの実行が紙芝居風に再生され、プログラム紙芝居のシートの重なり合いから、関数の呼び出し階層を連続的にたどることを可能にする。関数の呼び出し階層を可視化するという点では本研究と同じである。なお、このシステムはあらかじめ実行トレースをしておく必要がある。さらにプログラムの表示自体は使用しない。そのため、GUI には使用しづらい。

久永らは GUI を持つ Java プログラムの理解支援ツールを作成した [4]。このシステムでは特に GUI の見た目に注目している。プログラム上では GUI 部品のインスタンスは木構造状に配置される。その木構造を可視化するために GUI を 3D 表示した上で、関数の呼び出しを見せる。GUI の表示を利用する点、関数の呼び出し階層を可視化するという点では本研究と同じである。しかし、このシステムもあらかじめ実行トレースをしておく必要がある。そのため、プログラムへの操作と可視化が別々に行われる。

Storey らによる SHriMP は Java のプログラム構造とソースコードとドキュメント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 [5, 6, 7]。大きくて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。その入れ子をズーム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。このシステムでは静的な関係として関数呼び出し階層とクラス階層を見せる。またプログラムの全体構造を表示することも可能だが、各操作ごとの全体に対する実装部分までは指定することができない。このシステムは静的な関係しか表していない。しかし GUI を持つプログラムの場合では、操作に対する動的な実行結果も重要な役割を果たす。その点では GUI を持つプログラムを対象とすると扱いづらい。

AT & T で開発された Seesoft は、大規模なプログラムに関する全体情報の把握を目的としたシステムで、プログラムの各行は色の付いた 1 本の線として表示される [1]。この色はソースコードコードの更新履歴といった統計データを表し、ユーザはプログラムの概略を把握するとともに、プログラムの履歴情報を獲得することができる。そのため、プログラム全体に対しての実装部分の把握には有用である。しかしその他の重要な要素を把握することは難しい。

Lohr らによる 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 [10]。これはソースコードに付加されたアノテーションから、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。プログラム実行をトレースしながら、関数呼び出しを見せる点では本研究と同じである。しかし、このシステムもあらかじめ実行トレースをしておくことや、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点で本研究と異なる。

　このように既存の可視化手法では、本研究で特に注目している GUI の特徴についてはあまり考慮されてはいない。

第3章　GUIを持つプログラムの可視化

本研究では1章で挙げた要素1〜3を効果的に可視化する。要素1を見せるためにプログラム全体のソースコードとそれに対する実行行の強調表示、要素2を見せるために関数呼び出しを表すエッジの表示、要素3を見せるためにクラスの親子関係のツリー表示をそれぞれ行う。また、対象プログラムのGUI表示との同期を行う。そのためにGUIへの操作に対してインタラクティブに反応する可視化表示を行い、GUI表示変化の様子をその可視化結果に付加する。

3.1　可視化の概要

プログラム理解のためにはソースコードを直接見る必要がある。そのため、本システムではソースコードそのものを表示に利用する。その際にクラス間の関係を明確にするため、ソースコードは各クラス定義ごとに切り分ける。図3.1にクラス表示の例を示す。このようにクラ

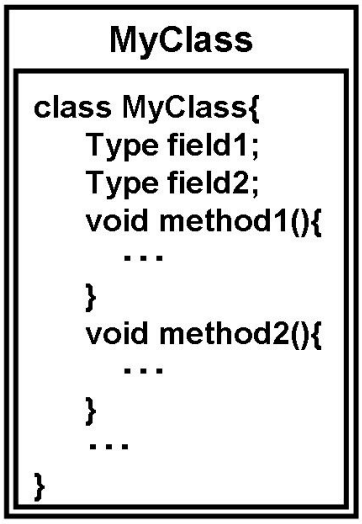


図3.1: クラス表示

スのソースコードがクラス名でラベル付けされて表示される。また、本システムでは前章で

挙げた重要な要素を可視化する。各操作のプログラム全体に対する実装部分を把握するために、ソースコード全体を表示する。これは図 3.1 に示されたクラス表示が画面上にすべてのクラスについて表示されるということである。このソースコード全体の表示に対して、プログラムの実行された行だけを強調表示する。これを見ることで、全体に対して操作に関しての実装部分がどこであるのかを把握することができる。各クラス表示は、単純に配置するのではなく、親子関係を表す木構造状に配置する。これによってクラス階層を可視化する。図 3.2 に親子関係のツリーを示す。このようにクラスは親クラスをルートノードとした木構造状に配置される。キャンバスはソースコード全体を表示するためにクラスを親子関係木構造ご

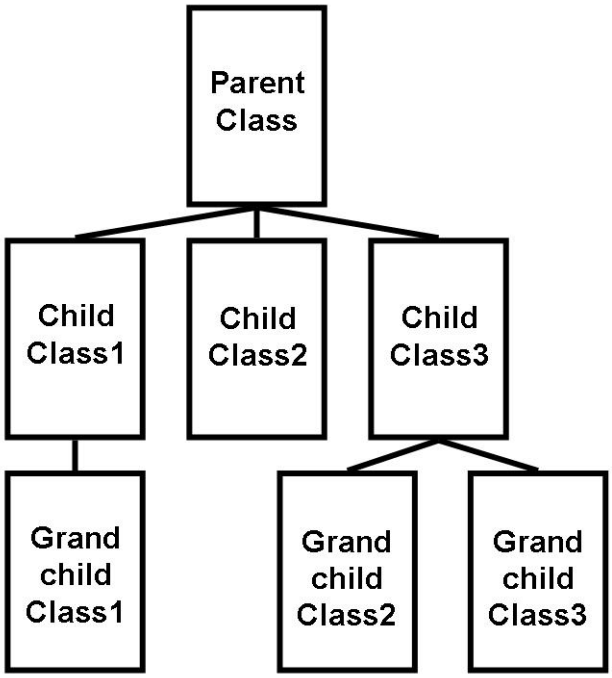


図 3.2: クラスの親子関係のツリー表示

とにまとめて配置する。図 3.3 にソースコードの全体表示の例を示す。矩形に囲まれた部分がキャンバスであり、その中に納まるように各クラスが配置される。また、関数の呼び出し階層を見せるために、呼び出された関数間をエッジでつなぐ。図 3.4 はその例である。各クラスの実行行が別色で強調表示され、クラス間の関数呼び出しがエッジでつながれている。以上が可視化情報を構成する主な要素となる。

システムは対象となる GUI を持つプログラムも起動する。これは開発者が対象プログラムの GUI を操作するのと同時に可視化を行うためである。開発者が GUI を操作するとその場でインタラクティブに反応して可視化表示を行う。これによって開発者は GUI への操作と可視化情報を対応付けながら閲覧することができる。

GUI への操作は、クリック、ドラッグなどのマウス操作、あるいはキーボード入力などであ

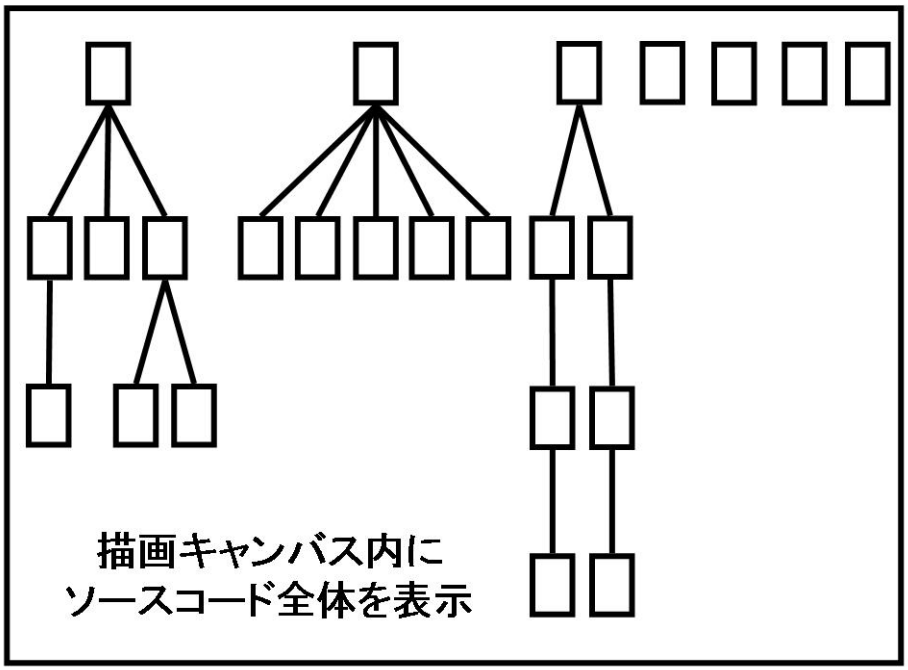


図 3.3: ソースコード全体の表示

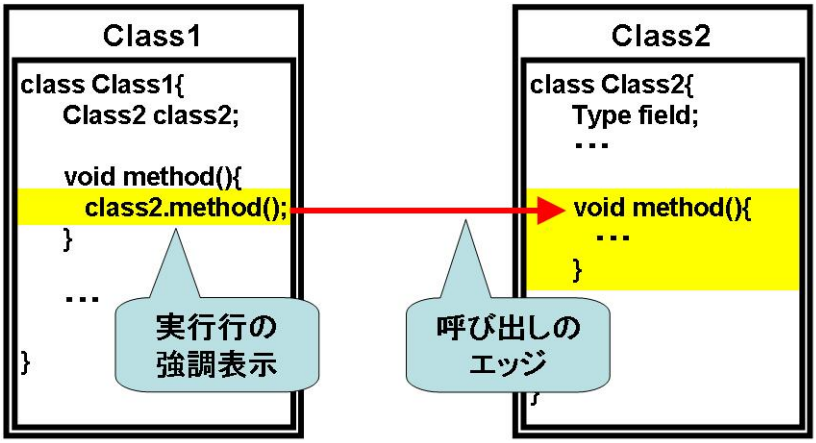


図 3.4: 関数呼び出しのエッジ

る。GUI プログラム上では、これらの操作が行われると、それに対応したプログラムが実行される。GUI が備えている機能というのは、このような操作の集合によって構成される。つまり機能についての実装を理解するためには、その機能上での各操作に応じたプログラム実行部分を理解すればよい。そこで我々は、操作に対するプログラム実行を 1 つの単位とした可視化を行う。

つまりシステムは、開発者による対象プログラムの GUI への操作を監視し、その操作に応じて実行されたプログラムを可視化する。

以上のような可視化表示を行うが、その際に、全体を均一に眺める概観表示だけでなく、ある時点の実行に注目する詳細表示も用意した。これはプログラム実行を詳細に把握するための機能である。この機能はソースコード全体の表示を損ねることなく、注目箇所にズームして表示を行うものでる。概観表示、詳細表示については以降の節で詳しく説明する。

3.2 概観表示

概観表示は前節で述べた表示をそのまま行う。先に述べたように、システム上で開発者は起動した対象プログラムの GUI を操作することができる。システムは GUI への操作に反応して、インタラクティブに表示を更新する。そのときシステムは随時、操作によって実行されたソースコード行を強調表示し関数呼び出しをエッジで結び順序付けを行う。

可視化情報の更新は、プログラムの実行に応じて徐々に更新される。操作によるプログラム実行が終了すると、その操作に対するグラフ表示が完成し、それ以降静的に閲覧することができる。図 3.5 に完成した静的グラフの例を示す。このように遠目から眺めるようにプログラムの実行を見ることで、開発者は各操作のプログラム全体に対する実装部分を知ることができる。ここで図中の緑色のマーカはプログラムの実行開始点を表している。

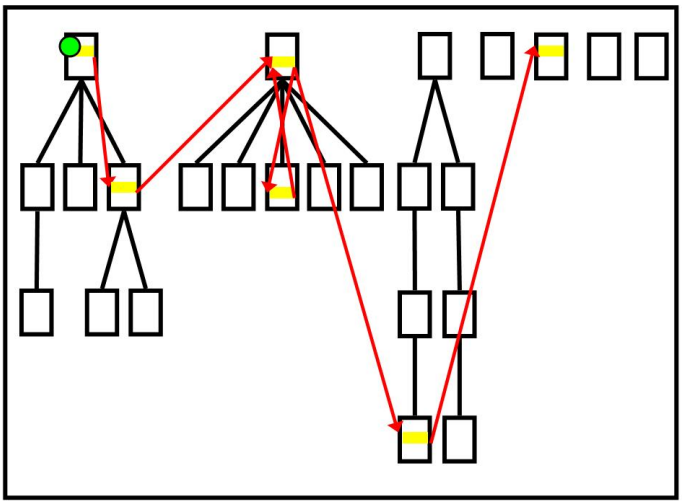


図 3.5: 概観表示での完成した静的グラフ

3.3　ズーミング機能を用いた詳細表示

操作ごとのプログラム実行を詳細に把握するには、静的グラフを構成するエッジを順に追えばよい。そこで我々は静的グラフの注目している部分のソースコードが見えるように拡大しながら、エッジを順に追う詳細表示を実装した。注目部分は見ている時点の実行クラスとその前後のクラス間の関数呼び出しであると考え、そのクラスと関数呼び出しのエッジを拡大表示する。またそれ以外は注目していない情報とみなし、縮小表示する。エッジに関しても透過処理等を施して目立たなくさせる。エッジはその時点の以前と以後の呼び出しであるかがわかるようにそれぞれ別の色で描画される。このように開発者は注目部分だけを取り出しながら、プログラムの実行の様子を追うことが可能となる。

また表示されるソースコード自体に関して言えば、実行行が注目部分となる。そのため、クラス表示に関しても実行行に焦点を当てたズーミングを行う。

ズーミングはソースコード全体を表示しながら行われる。そのため、どのようなズーミングの手法をとるのかが、重要になってくる。以降の小節でクラス、ソースコードそれぞれに対するズーミング手法について述べる。図 3.6 に詳細表示の例を示す。中心で大きく表示されているのが、注目しているクラスである。現在の実行行は赤の実線、これから実行される行は赤の点線で表示されている。また、既に実行されている行は別の色である青い線で表示される。

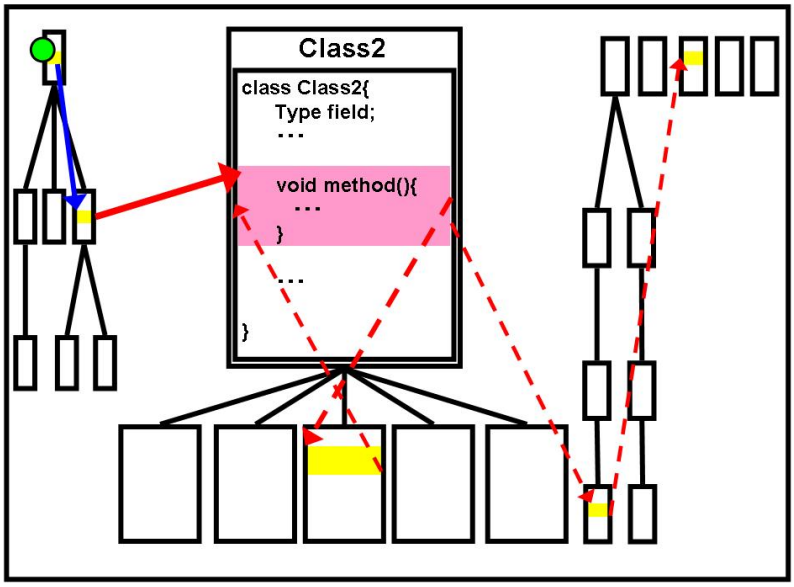


図 3.6: ズーミングを用いた詳細表示

3.3.1 クラスのズーミング方法

本研究ではクラスのズーミングの際の拡大の指標として、Furnas によって提案された Fisheye ビューを利用する [18]。この手法は木として表現できる情報構造に利用できる Focus+Context 手法の手法の 1 つである。今、クラスは親子関係の木構造で表現されている。さらにそれらの木は一つのキャンバス上に配置されるため、これも 1 種の木構造とみなすことができる。この手法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。

$$DOI = API - D$$

ここで潜在的重要度は、木のルートから単調減少する値として定義される。各ノードの重要度を計算して、重要度にしたがってクラスを拡大表示する。一般に、重要度が一定の閾値を下回るものを表示しないという方法もある [18]。しかしソースコード全体を常に表示するほうが、開発者の理解につながると考えたため、本システムでは用いないものとした。つまり、これによって常にすべてのソースコードを眺めながら、注目部分の実行状態を知ることができる。

また、三末は上記の Fisheye の重要度を視点、構造、意味の 3 つの属性によって決定づけることで、さらに多様な要求に対応可能な手法を提案した [19]。プログラムには親子関係や呼び出し階層などの様々な属性があり、これらはプログラムを理解する上での重要な要素である。そのため本研究ではこちらの手法も利用する。一般にプログラムの実行中においては、親子関係はあるクラスから見たとき、親クラスの構造のほうがより重要は情報を含んでいる。例えば、オーバライドされた関数が呼び出された場合には、その親で定義されているオーバライド前の関数定義は特に見たい情報の一つである。そこで、親子関係を構造属性として与えることとする。また、関数呼び出しの順序も重要な情報を含んでいる。例えば、今の視点である注目部分の前後に呼び出された関数定義はプログラムを理解する上で必要な情報である。そこで、関数呼び出しの順序関係も重要度を決める属性の一つとする。

ここでプログラムの木構造に関する表示領域割り当てについて説明する。図 3.7 は木構造のマッピングの例である。図 3.7 左図は、対象プログラムのすべてのクラスを表している。これらの木構造のそれぞれの構成要素の描画領域はキャンバス上に割り当てられる。概観表示時には注目する焦点は特に定まらない。そのため各ノードの重要度は潜在的重要度よってのみ決まる。図 3.7 右図は、概観表示時の木構造の描画領域である。左図の木構造は右図ような領域にマッピングされる。各ツリーはノードについては潜在的重要度に従って領域が確保される。そのため、ルートノードに近いものほど大きな領域が割り当てられることになる。ここで図 3.7 のノード 10 に焦点を当てる (右図中の赤い領域に注目する)。そのときのズーミングの結果を図 3.8 に示す。このとき、重要度に従ってクラスを拡大表示するために、重要度が高いノードほど大きな描画領域を確保するようにする。図 3.8 から、焦点に近いほど大きな領域が確保されていることがわかる。ここで、焦点が当てられているクラスの先祖のクラス群の領域のほうが子孫のクラス群の描画領域よりも大きくなっている。これは先に述べた親子関係の構造属性を反映させた結果である。

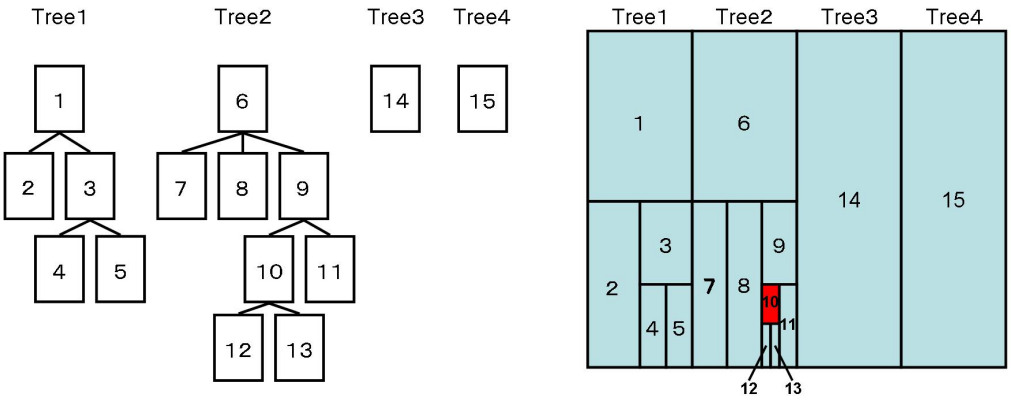


図 3.7: 木構造の表示領域マッピング

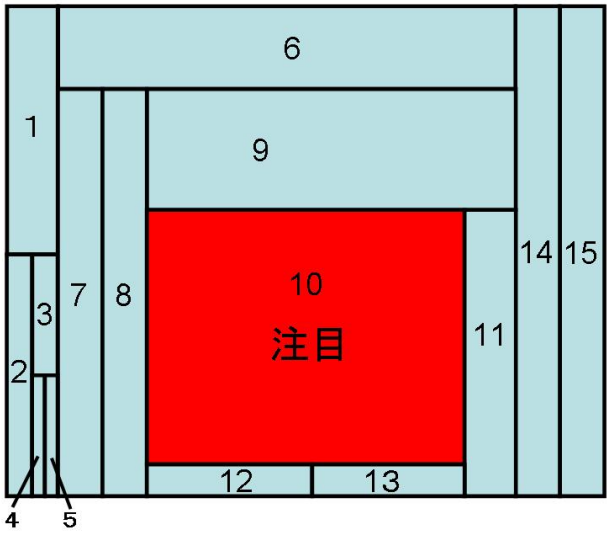


図 3.8: Fisheye ビューによるクラス表示

3.3.2 ソースコードのズーミング方法

1 つのクラスが巨大になると、そのソースコードの行数は増えてしまう。一方、1 つのクラスを表示する領域は限られている。可視化の方法上、ソースコードのすべての行が常に表示されていなければならない。領域内にすべての行を単純に収めるとすると文字のフォントをかなり小さくしなくてはならないため、可読性が損なわれてしまう。そこで各クラスパネルごとにも、ソースコードに関する Fisheye ビューを行う。すでに Furnas は [18] の中で、段付けされたプログラムを木構造とみなして、Fisheye ビューを適用している。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）の同時表示を行い、それ以外の行の表示は省略する。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。特に、現在表示されていない行へ移動した時のユーザに与える認知負荷が大きいことが実験的使用の結果わかってきた。図 3.9 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のように次ステップでの実行行が省略されてしまう。このような表示変化はユーザに大きな認知負荷を与えてしまう。

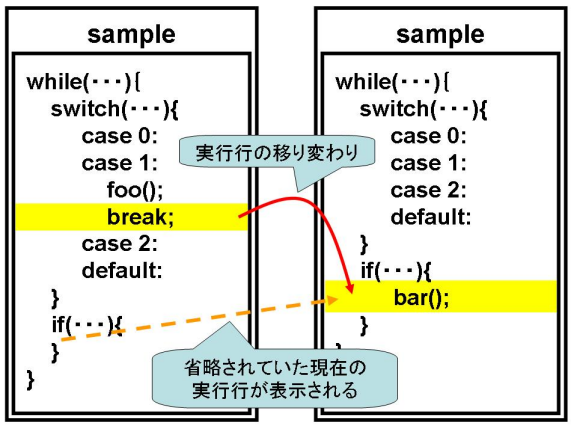


図 3.9: Fisheye ビューを用いたソースコード表示 (Furnas)

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モードでの表示を提案した。各行はその重要度に対応する大きさのフォントで表示される。基本的に省略される行がないので、連続的変化を認識しやすい。

本システムでは単純に現在実行している行の前後行では重要度を高く、距離が離れた行ほど重要度を低く設定し、マルチスケーラブルフォントモードでの表示を行う。図 3.10 にマルチスケーラブルフォントモードでのズーミング表示を示す。このように実行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。

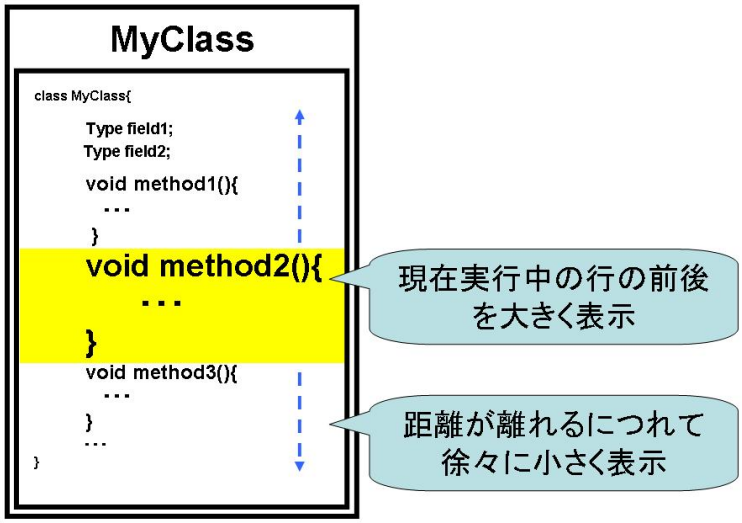


図 3.10: ソースコードのズーミング表示

3.4 対象プログラムの GUI を含む画面状態遷移のサムネイル表示

機能について理解するためには、操作に対する可視化情報の集合を見る必要がある。そのため、可視化情報は操作のたびに破棄するのではなく、常に保存しておきたい。しかしながら、単純に保存しておき、以前のものを閲覧可能にするだけでは、その情報がどの時点のものだったのかがわからなくなってしまう。これでは、プログラムへの操作と可視化表示の時間差の問題を解決したことにはならない。その解決策として操作のイベント名を取得して可視化情報にラベル付けすることは、有効な手段であるが、プログラムの状態によって処理結果が異なる場合が考えられるので、操作の識別には不十分である。一方、GUI を持つプログラムの状態の変化は、多くの場合表示画面の変化として現れる。例えば、ドローツールならば描画した図形、エディタならば入力された文字が表示される。以上の点を踏まえて我々は、プログラムの状態の変化を保存しておくために、各時点での表示画面を保存する。操作前と後の表示画面を加えて可視化情報のラベルとして利用する。これによって、開発者はその可視化情報が示している操作を想起することができる。

ここで保存しておく画面表示の内容について言及する。GUI を持つプログラム自体が持つすべてのウィンドウあるいはアクティブなウィンドウだけを保存しておくことは有効な手段の一つである。しかし、ウィンドウが隠される自体が操作になる場合や、ウィンドウの位置が処理の内容に関わる場合などには有効に働かない。またプログラムが外部プログラムに対して機能する場合も少なくない。そこで我々はプログラムが動いているディスプレイ全体の表示内容を保存することにする。隠されているウィンドウなどは見れなくなるが、上記のような場合にもある程度有効に働く。なにより、開発者が見ていた画面そのものなので、可視

化情報が示している操作の想起には大いに役立つと思われる。

保存した画面を閲覧しやすくし、なおかつ操作に対する画面の変化を把握しやすくするために、保存した画面は操作前と後のものをそれぞれ対にしてサムネイル表示する。開発者は操作に対応するサムネイル表示と可視化情報を同時に閲覧することができる。図 3.11 は、サムネイル表示のイメージである。このように操作前と後の表示画面が表示される。

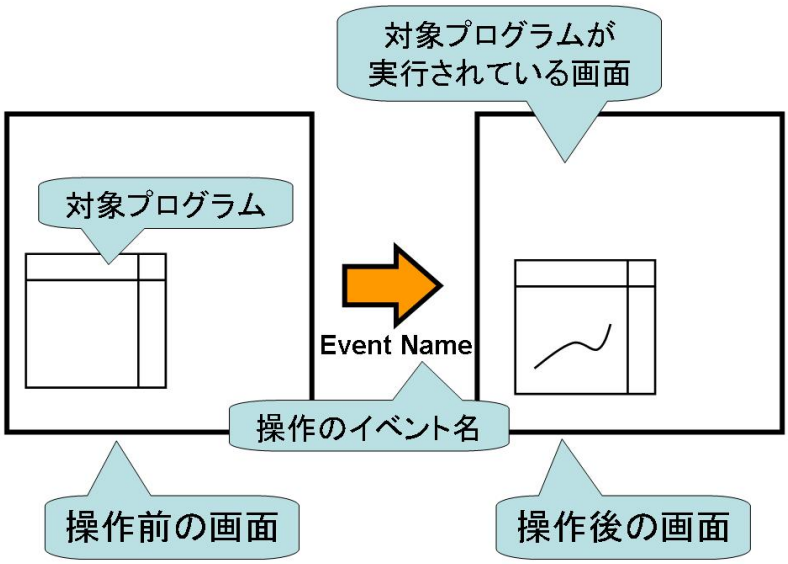


図 3.11: サムネイル表示

第4章　実装

本研究では Java を対象言語にした。これは Java が現在よく使われる言語だからである。Java プログラムの動作を解析するためにはソースコードとクラスファイルから、メソッドやフィールドといったクラス構成、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。特に動的情報はトレースとして実行時に取得する。クラス構成などの静的な情報の取得には Java の ReflectionAPI を用いる。ReflectionAPI は、クラスおよびオブジェクトについてのリフレクション情報を取得するためのインタフェースで、任意のクラスのメソッドやフィールドを実行時にアクセスできる仕組みである。動的情報を取得するためには JDI(Java Debug Interface)API を利用する [12]。JDI では JavaVM の実行を明示的に制御し、各クラスの情報を監視することができる。このようにして可視化対象となる GUI を持つプログラムを監視して、必要な情報を取得する。

ビューは取得した情報を元に実行の様子を可視化する。前章で述べた詳細表示を実現するために、ズーミング機能を行う枠組みが必要となる。ズーミング機能の実装には Piccolo.Java 1.2 を利用する。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 [13, 14]。図 4.1 にシステム構成図を示す。モデル部が可視化に必要な情報を管理しており、対象プログラムから直接ソースコード情報を取得する。対象プログラムのプログラムが実行されると実行時情報トレーサがその実行情報を取得して、モデル部のキューに情報を追加する。ビューはモデル部の変化に応じて、モデル部が持つ情報を元に可視化を行う。

4.1　Java Debug Interface(JDI) の概要

JDI は、Java Platform Debugger Architecture (JPDA) が構成するインタフェースの一つであり、デスクトップシステムの開発環境でデバッガを使用することを前提として設計されている。JDI は、VM の動作状態にアクセスする必要のあるデバッガやそれと同様の機能を持つシステムに対して有用な情報を提供する。JDI は、VM の動作状態、クラス、配列、インタフェース、プリミティブ型、およびそれらの型のインスタンスへの内省的なアクセスを提供する。また JDI は、VM の実行を明示的に制御できる。スレッドの中断および再開を行なう機能、ブレークポイントやウォッチポイントを設定する機能も備えており、その他に、例外の通知、クラスのロード、スレッドの作成などを行える。さらに、中断されたスレッド状態、ローカル変数、スタックバックトレースなどを調べることもできる。

JDI を使用したアプリケーション (以下、デバッガアプリケーション) とデバッグ中の VM(以下、ターゲット VM) との通信は、それぞれのプロセス間で通信する形式で実現されている。

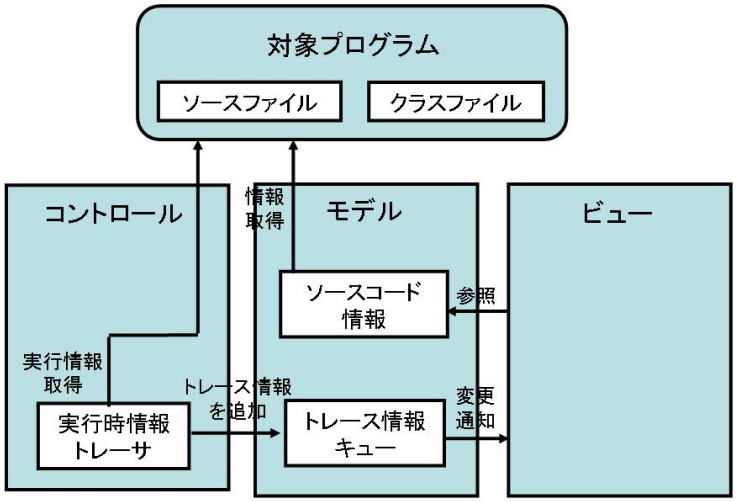


図 4.1: システム構成図

デバッガアプリケーションとターゲット VM の間に接続が確立される際には、一方がサーバとして動作し、接続を待機する。その後、もう一方がリスナーに接続され、接続が確立される。 これらの接続は、デバッガアプリケーションまたはターゲット VM のどちらかをサーバとして動作させることが可能であるが、本システムでは前者をサーバとしている。

ターゲット VM からの情報はイベントとして取得することができる。デバッガアプリケーション側必要なイベントを要求する。ターゲット VM のプログラムを実行中にイベントが発生すると、それらはイベントキューに保持する。要求されたイベントをキューの中から取り出し、デバッガアプリケーション側で情報を利用する。

4.2 実行時情報取得部の実装

情報取得は以下のイベント発生時に行われる。

- 対象プログラム内で定義されたメソッドの呼び出しイベント
- 上記のメソッド内でのステップ実行イベント
- 上記のメソッドの終了イベント

この取得方法では、メソッド呼び出しが起こるたびに呼び出し階層は深くなっていくが、実行が進むと再び元のメソッドの実行に戻ってくる。そのため、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 つの操作で実行されたメソッドとする。イベントに関する情報として以下を利用する。

- 実行中のクラス名
- 実行中のメソッド名
- 実行行番号

このような情報を操作ごとに逐一実行情報キューに登録する。可視化ビューはこのキューの情報を元に表示を行う。

4.3　可視化ビュー部の実装

ビュー部分は取得した情報から、実行の様子を可視化する。ビュー部分は以下の 3 つの要素から構成される。

- グラフ表示部
- サムネイル表示部
- コントロール部

図 4.2 に実装した可視化ビューのスナップショットを示す。

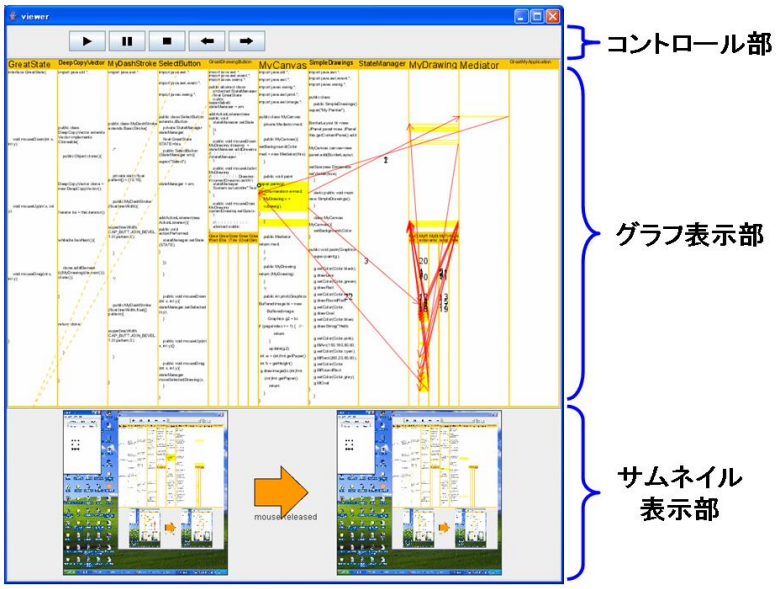


図 4.2: システムのスナップショット

グラフ表示部では、先に述べた概観表示と詳細表示を行う。概観表示でのクラス表示や親子関係ツリー、実行に対する強調表示とエッジは図 4.3 に示すように表示される。実行の開始

点を明確にするために、図中に示すようなマーカを表示する。ここで Java のインタフェースへの対処について説明する。まず、インタフェースを継承したインタフェースはクラス同士の継承と同様の extends によって行われるため、常に単一継承となる。このような場合にはインタフェースの親子関係ツリーを表示する。一方、クラスがインタフェースを実装する、つまり implements による継承の場合には、しばしば多重継承が起こるため木構造では表現しきれないという問題がある。しかし Java のインタフェースはそもそも継承というよりも機能の追加という用途で扱われる概念である。そのため、本システムではインタフェースの実装は木構造の中で表現するのではなく、クラスとインタフェース同士のエッジによって表現する。図中におけるオレンジ色の点線がそのエッジである。詳細表示の様子を図 4.4 に示す。現在の実行クラスを焦点とした Fisheye 表示を行っている。

本システムでは、DOI に従った厳密な Fisheye ズーミングを行うのではなく、簡易的な Fisheye ズーミングを実装した。まず各ツリーの領域決定方法を述べる。キャンバス全体の横幅を $width$、注目ツリーの拡大率を $zoom_p$(0.0〜1.0)、ツリーの個数を $numtrees$ とすると、注目しているツリーの横幅は $width * zoom_p$、それ以外のツリーの横幅は $width * (1 - zoom_p)/(numtrees - 1)$ で求められる。さらに前回の注目部分があった場合にはそのツリーの拡大率を $zoom_s$(0.0〜1.0, $zoom_p + zoom_s < 1.0$) とすると、そのツリーの横幅は $width * zoom_s$、それ以外のツリーの横幅は $width * (1 - zoom_p - zoom_s)/(numtrees - 2)$ で求められる。注目するクラスを持つツリーのクラス領域決定法も同様に計算される。縦幅、横幅共に注目クラスが拡大率 $zoom_c$(0.0〜1.0) で拡大表示されるように計算している。ここで、直前の実行クラスの重要度も高く設定されている。また、呼び出しの順序を明確にするため、エッジを実行時点に応じた別々の色で表現している。ここでは、青い線が以前の呼び出し、赤い線が現在を含めた以降の呼び出しとなる。このとき点線はこれから呼び出される呼び出しを表す。現在の呼び出しから時間的に離れるのに従って、より細く、より色の薄い線で表す。現在は経験的な値として、拡大率をそれぞれ $zoom_p = 0.50$、$zoom_s = 0.30$、$zoom_c = 0.65$ に設定している。

次にソースコードのズーミング方法を述べる。こちらも簡易的な Fisheye ズーミングを実装した。現在は各行の表示の縦幅を、図 4.5 に示すような注目行を最大値とした線形関数に従って領域を決定する。各行のフォントは領域に収まるように選定するため、縦幅と横幅のうち小さいほうの領域に合わせる。しかしながら、現在のディスプレイの解像度では表示可能なフォントサイズには下限がある。そのためソースコードの行が多い場合には、表示されない可能性が出てきてしまう。それを防ぐために、実行行の縦幅の下限を 20、表示フォントの下限を 6 にそれぞれ設定している。図 4.6 にソースコードズーミング例を示す。これらはすべて同一のクラスである。このように注目点が最も大きな領域で表示される。

コントロール部は、本システム自体を操作する制御部分であり、以下の操作を行うユーザインタフェースを持つ。

- 対象プログラムの制御
  - 起動・再開
  - 中断

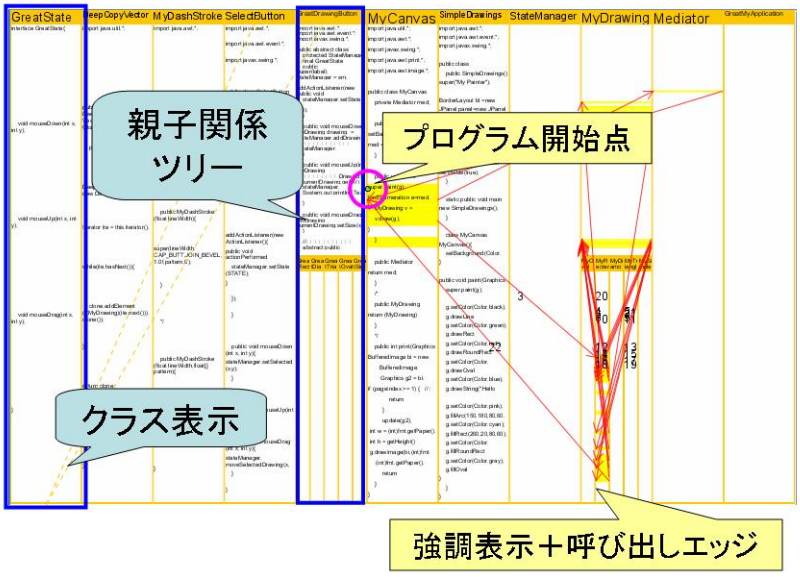


図 4.3: グラフ表示部 (概観表示)

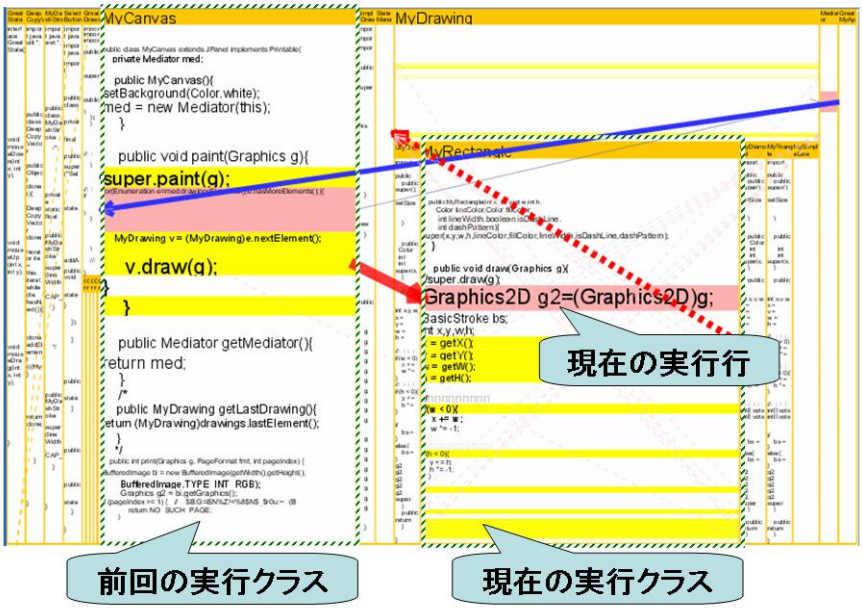


図 4.4: グラフ表示部 (詳細表示)

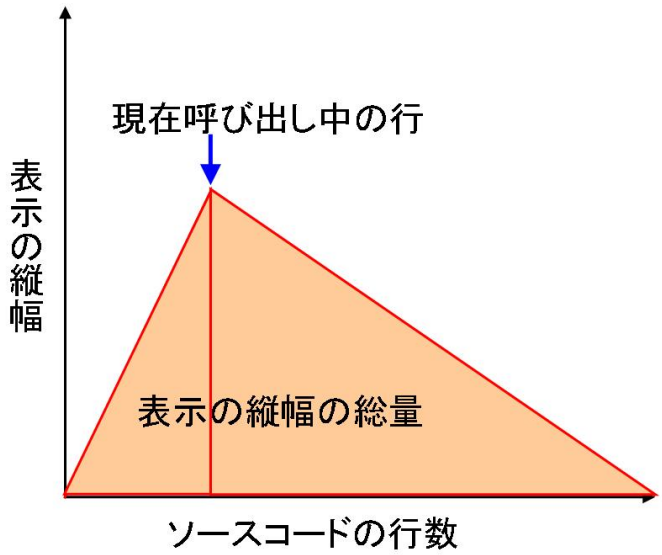


図 4.5: ソースコードの各行の領域決定法

  - 終了

- 詳細表示の制御 (トレースのステップ制御)
  - 進む
  - 戻る

コントロール部は図 4.7 に示すように実装した。図の注釈で示しているように、上記の操作をそれぞれのボタンに割り当てている。

サムネイル表示部には、操作前と後での対象プログラムの GUI の画面遷移がサムネイルで表示される (図 4.8)。サムネイルは対象プログラムへの操作が行われるたびに取得した画面情報によって更新される。現在の実装では、このサムネイルは選択することが可能で、左の画面を選択すると前、右の画面を選択すると後の操作の実行の静的グラフに切り替えることが可能である。なお、サムネイルの元となる画面のスナップショットの取得には Java の Robot クラスを利用した。Robot はマウスやキーボード制御をネイティブなシステム入力イベントとして生成することができる。その他にスクリーンから読み取るピクセルイメージを作成することも可能である。これはプログラム外の領域についてもイメージを読み取ることができる。

図 4.6: ソースコードズーミングのサンプル

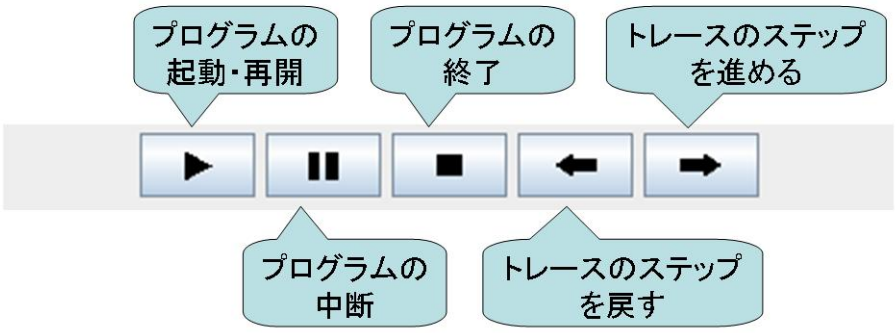


図 4.7: コントロール部

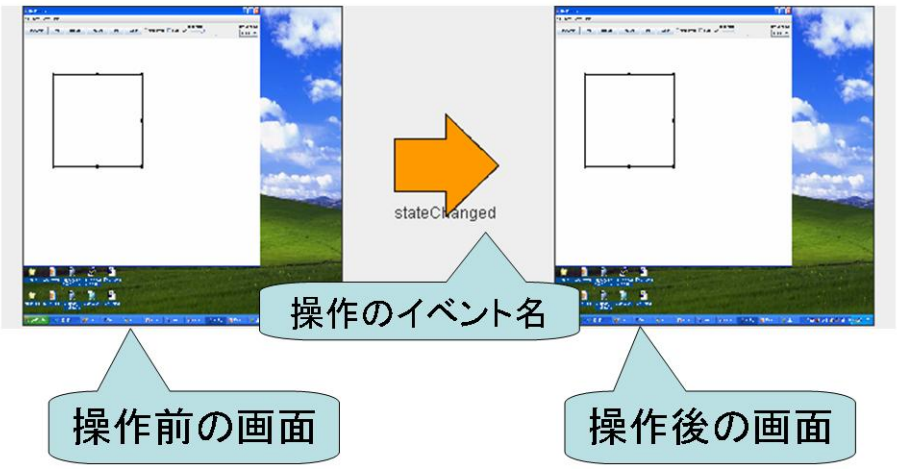


図 4.8: サムネイル表示部

第5章　利用シナリオ(本システムを用いたプログラム理解の方法)

5.1　GUIを持つプログラムを理解するための要素を得る方法

開発者は以下の手順で本システムを使用する。なお、本システムを利用するためには対象プログラムのソースファイルとクラスファイルの両方が必要となる。

1. 対象プログラムの実行コマンドを引数として与えて、本システムを起動する
2. システムはプログラムのソースコード全体とクラス階層を可視化する
3. 次に対象プログラムに対して、可視化したい操作を行う
4. システムは操作に対するメソッド呼び出しを可視化する
5. 表示されたメソッド呼び出しの静的グラフに対して、詳細表示をしながらステップトレースを行う

このように、操作のプログラム全体に対する実装部分を見ながら、関数呼び出し階層とクラス階層の理解を行う。

5.2　具体的な利用シナリオ

サンプルプログラムとして簡単なGUIドローツールプログラムを用意した。図5.1にそのサンプルプログラムを示す。ドローツールは図5.1に示すようにボタンを押すことで描画する図形の種類を選択し、キャンバス内でドラッグ&ドロップを行うことで図形を描画するというごく一般的なものである。また、メニューやコンボボックスといったGUI部品を操作することで、図形の色や線の幅といったパラメータを変更することができる。現在描画できる図形は矩形、楕円、直線であり、モード選択ボタンはそれぞれRectangleボタン、Ovalボタン、Lineボタンである。キャンバスへの描画後の図形オブジェクトを選択し、移動や色の変更などを行うためのモードとして図形選択モードがある。Selectボタンを押すことでこのモードに切り替わる。図5.2にこのプログラム全体のクラス関係図を示す。

ここではシナリオ例として、開発者がドロー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。その場合には、既存の図形のモード選択ボタンの機能と描画の実装を模倣すればよい。そのような場面では、この

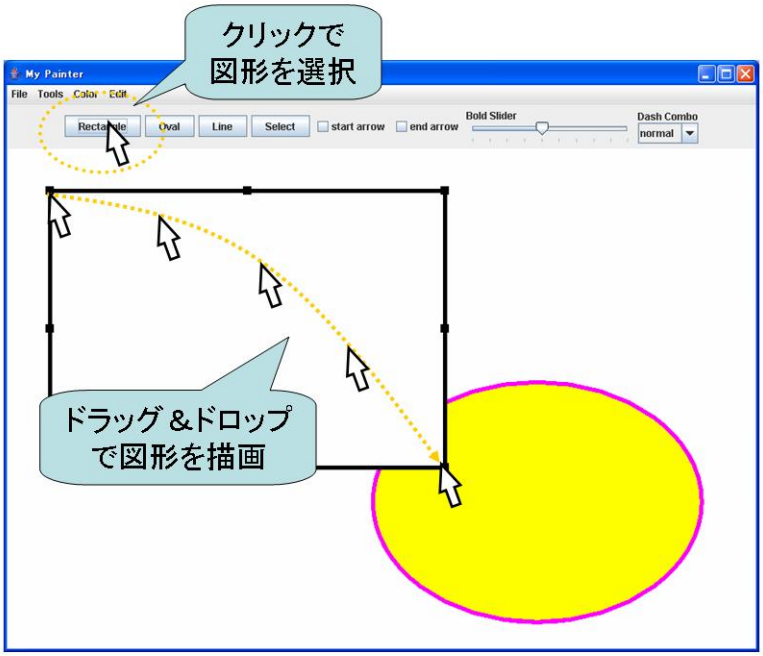


図 5.1: サンプルプログラム

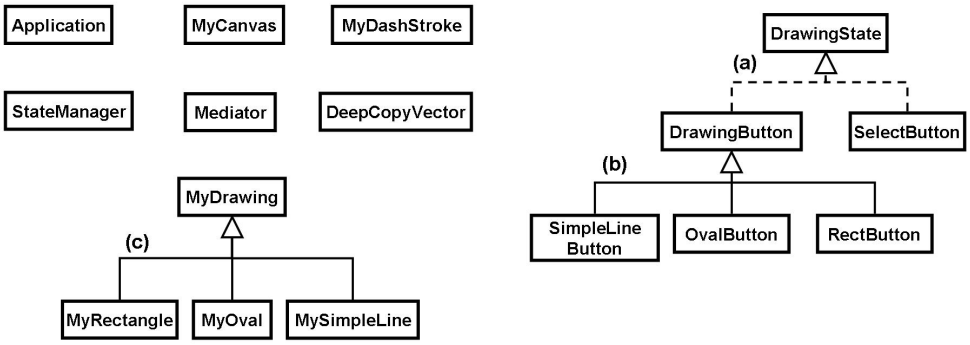


図 5.2: サンプルプログラムのクラス関係図

プログラムを理解の対象として本システムを利用すればよい。開発者は本システムを使って、描画図形モードの選択と描画の操作を行う。その後、詳細な分析を行うことで理解すべき要素を把握していく。図に本システムの起動時のスナップショットを示す。開発者はまずクラス

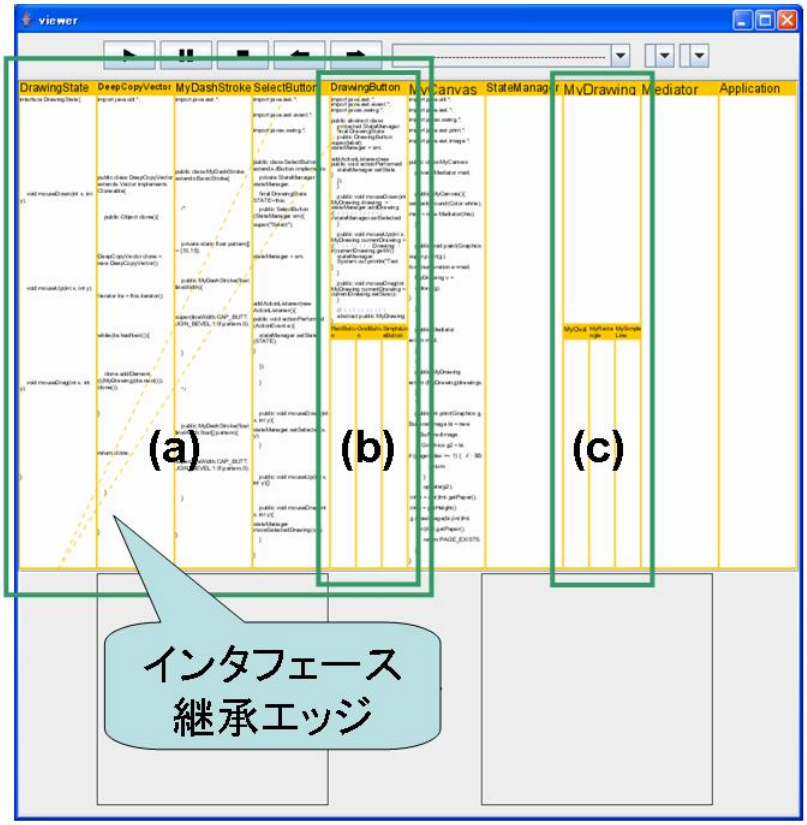


図 5.3: システムの起動時

表示部の静的に各クラスの親子関係を含めたクラス関係図を確認する。ここで、図 5.2 のクラス関係図と比較してみると、親子関係は忠実にツリー化されている。特に関係 (a)〜(c) に注目して欲しい。一見、DrawingState インタフェースからの継承関係が表現されていないように見えるが、これはインタフェースによる継承なので、木構造ではなくエッジによる表現で示されている。開発者はこれを見ることで、クラス階層を一目で把握することができる。

次に、コントロール部のプログラム起動ボタンを押して、対象プログラムを立ち上げる。すると、プログラムの実行の様子がグラフ表示として可視化される。図 5.4 は対象プログラム起動時の静的グラフである。対象プログラムが立ち上がった後は、対象プログラムの GUI を操作しながら、可視化を行っていく。今回の機能追加を達成するためには、既存の図形描画機能を行う操作を行い、その実装について理解すればよい。ここでは、既存の矩形描画機能をその対象とする。まずは、マウスで Rectangle ボタンをクリックし、矩形描画モードに切り

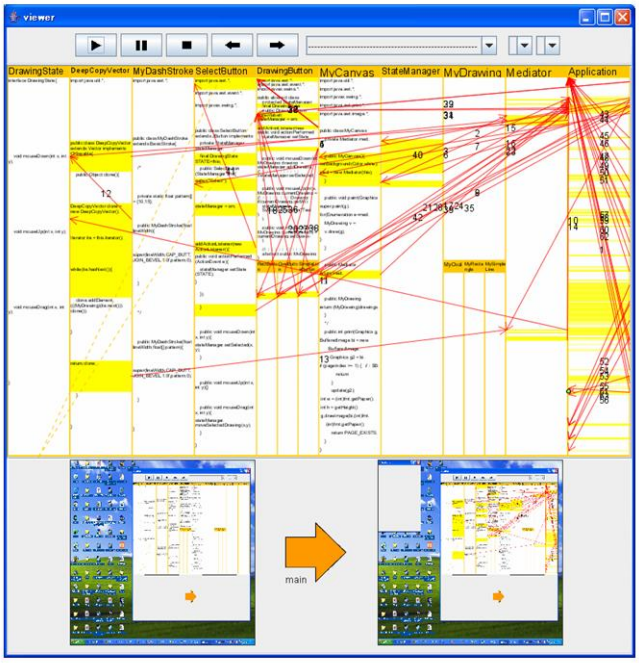


図 5.4: 対象プログラム起動時

替える。すると、図 5.5 に示す矩形モード選択操作に対する静的グラフが完成する。ここでサムネイル表示部に注目すると、操作後で Rectangle ボタンが選択状態になっている。サムネイル表示と静的グラフを見ることで、後で見たときに Rectangle ボタンが選択したときの可視化結果であることがわかる。静的グラフを眺めると、ソースコード全体に対して DrawingButton クラスと StateManager クラスのごく一部分が実行されていることがわかる。それを確認した後に、引き続き操作を行う。次にキャンバス上でマウスをドラッグ＆ドロップして矩形を描画する。すると、図 5.6 のようにマウスプレス、ドラッグ、リリース時の操作がそれぞれ行われ、その間で画面の再描画が行われる様子が、可視化表示として次々と現れる。 モード選択、マウス操作、再描画のすべての可視化結果を眺めると Application クラス、DrawingButton クラス、RectButton クラス、StateManager クラス、MyDrawing クラス、MyRectangle クラス、Mediator クラス、MyCanvas クラス、DeepCopyVector クラスが使用されていることがわかる。ここで、インタフェースの関連エッジから DrawingButton クラスが DrawingState インタフェースを実装していることがわかるので、DrawingState インタフェースもこの機能に関連している可能性がある。つまり、矩形描画機能の全体に対する実装部分はこれらのクラス内の実行部分ということになる。このようにして、全体に対する機能の実装部分を把握する。ここまでで、矩形描画機能に関する基本的な操作を行った。

次に上記で行った各操作の可視化結果について詳細に分析する。そのためにトレースをステップさせながら、詳細表示を行っていく。操作と可視化情報の対応付けは、サムネイル表示

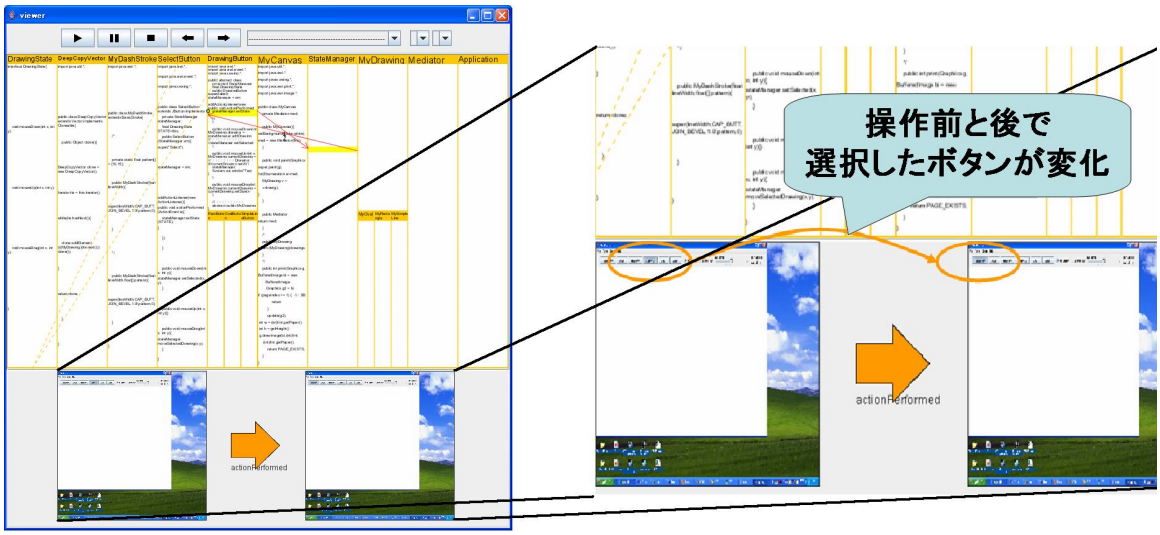
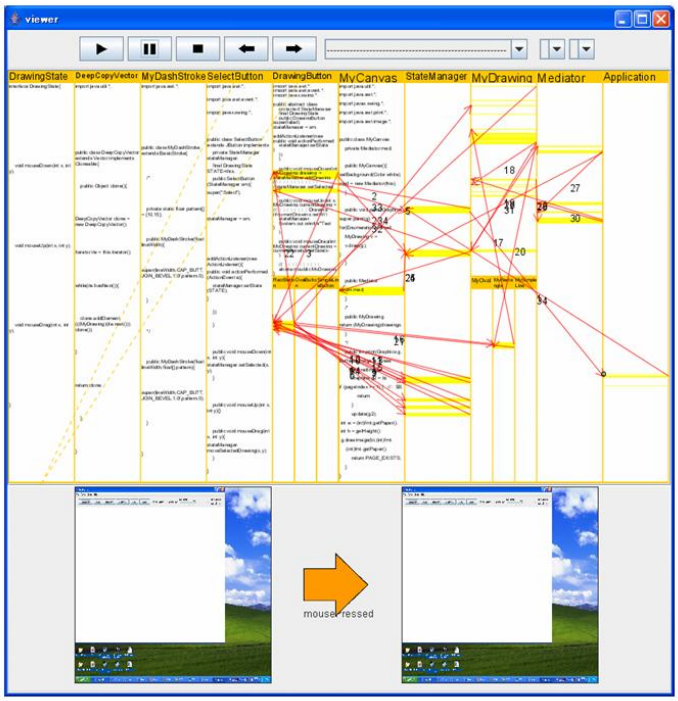


図 5.5: 矩形ボタン選択時

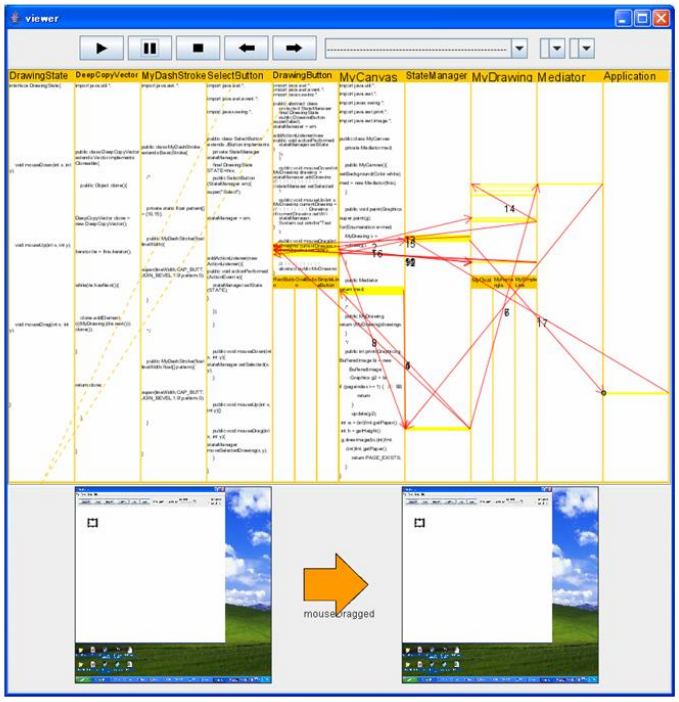
部に示されるサムネイルとイベントのメソッド名を元に行う。これはすべての操作について行うが、ここでは例としてマウスプレスに関して詳細表示を行っていく。ここで概観表示からわかる情報はマウスプレス時には、Application クラス、DrawingButton クラス、RectButton クラス、StateManager クラス、MyDrawing クラス、MyRectangle クラス、Mediator クラス、MyCanvas クラスが呼ばれるということ、プログラムの開始点は Application クラスであることである。図 5.7 に示すように、詳細表示で関数呼び出しをたどっていく。 基本的にはピンク色で示されている現在の実行行と現在の呼び出しを結ぶ赤実線のエッジを見ればよい。理解の用途に応じて次ステップでの呼び出しを結ぶ赤点線や前のステップでの呼び出しを結ぶ青線のエッジ、親子関係のツリーを見ながら、実装の様子を理解する。

ここで詳細表示でエッジをたどると、まず Application クラスでマウスイベントを取得しそれに対する実行として StateManager.mouseDown() が呼ばれる (a)。次に DrawingButton.mouseDown() が呼ばれ、DrawingButton 内で定義される MyDrawing 型を返す関数 createDrawing() が呼ばれる (b)。しかし次に実行されるのは RectButton 内の createDrawing() である (c)。ここで実行される createDrawing() は RectButton でオーバライドされた関数であることがわかる。この中で MyRectangle クラスのインスタンスが生成されるが、これは MyDrawing クラスの子クラスであることは親子関係のツリーから理解することができる。引き続き、エッジをたどると MyRectangle 型インスタンスの生成に必要な引数として StateManager が管理する状態を取得している (d)。しかし StateManager 自体には状態変化を与えていない。それらの情報を元に MyRectangle 型のインスタンスが生成されるが、このとき MyDrawing クラスのコンストラクタも使用している (e, f)。生成されたインスタンスは StateManager.addDrawing() の中でキャン

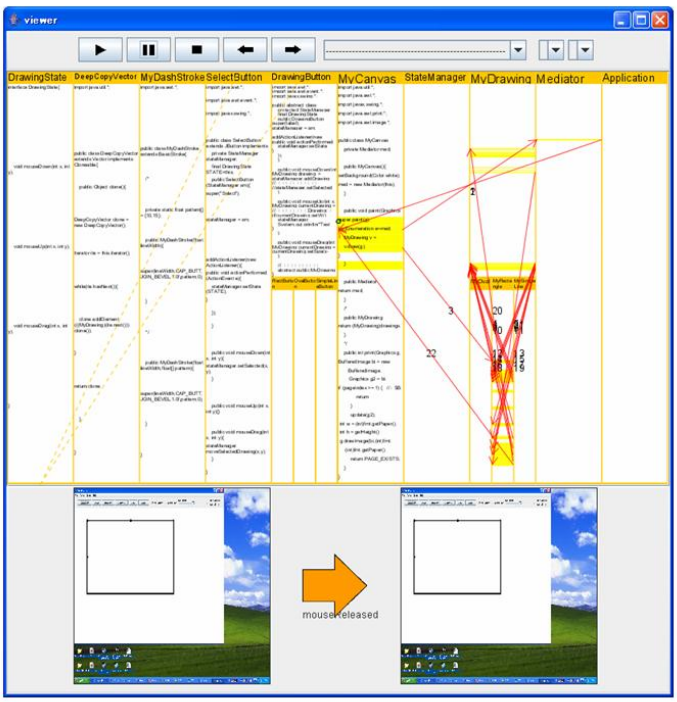
a)


viewer
DrawingState
DeepCopyVector
MyDashStroke
SelectButton
DrawingButton
MyCanvas
StateManager
MyDrawing
Mediator
Application
mousePressed

b)


viewer
DrawingState
DeepCopyVector
MyDashStroke
SelectButton
DrawingButton
MyCanvas
StateManager
MyDrawing
Mediator
Application
mouseDragged

c)



d)

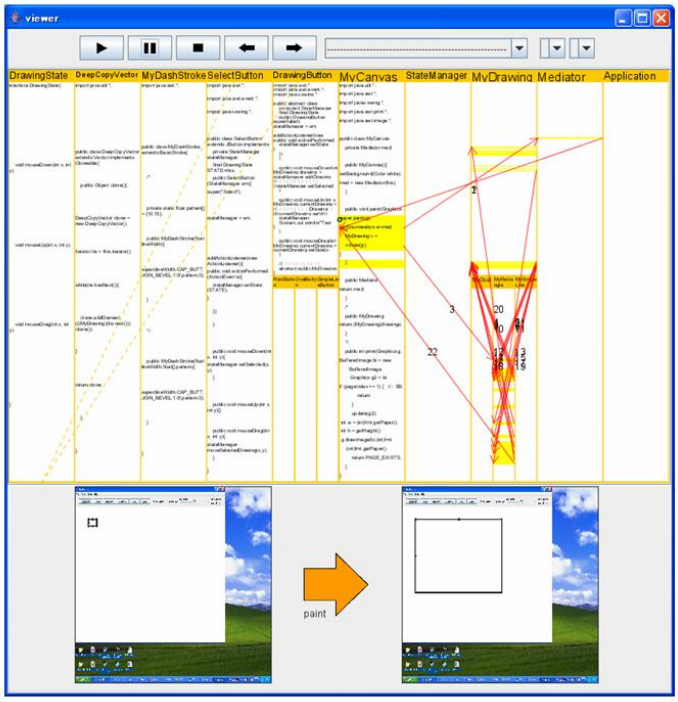
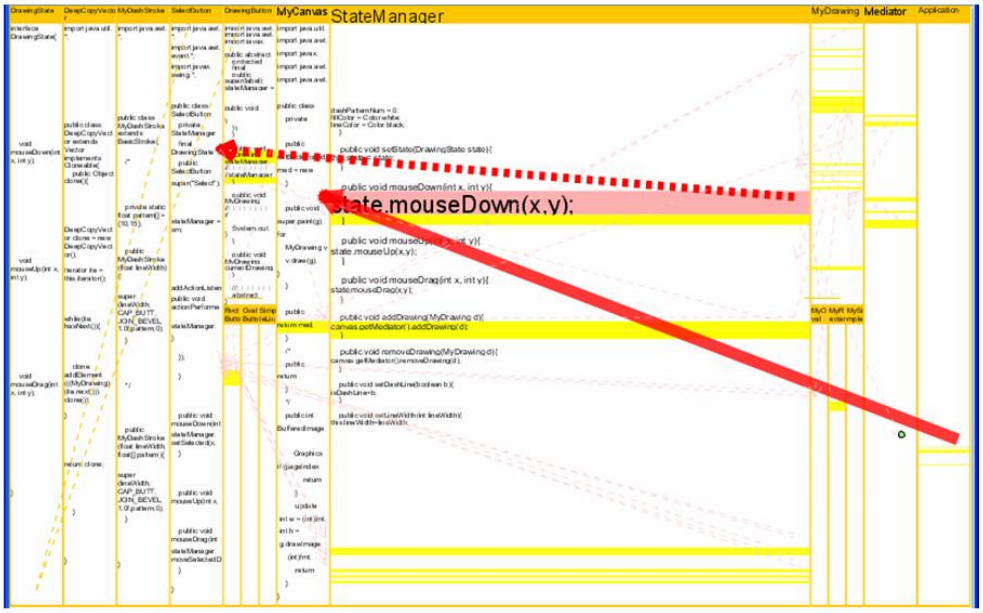
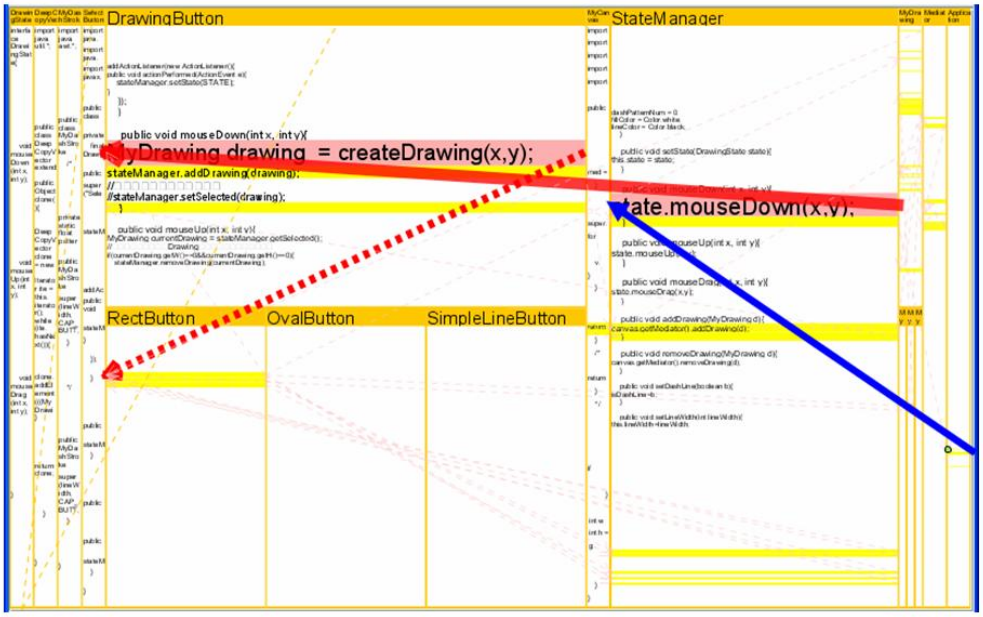


図 5.6: 図形描画のためのマウス操作時

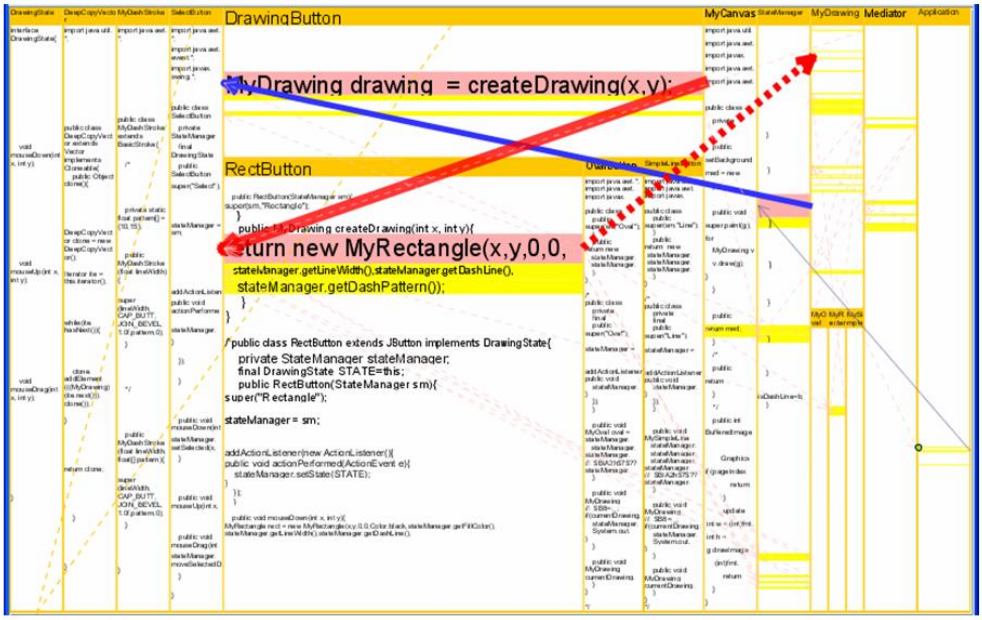
a)


MyCanvas
StateManager
MyDrawing
Mediator
Application
State.mouseDown(x,y);

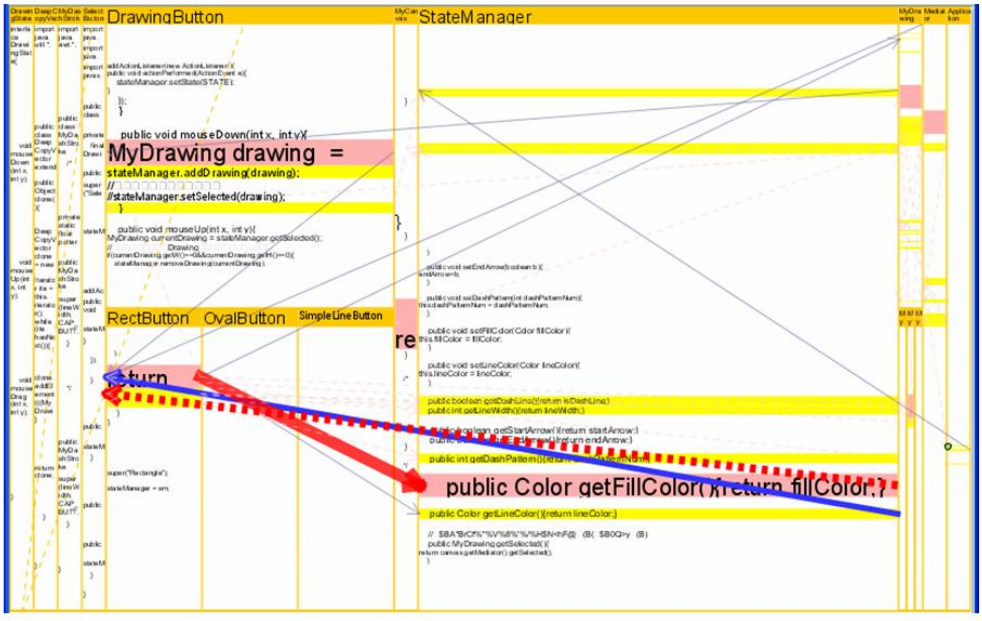
b)


DrawingButton
StateManager
public void mouseDown(int x, int y){
yDrawing drawing = createDrawing(x,y);
stateManager.addDrawing(drawing);
//stateManager.setSelected(drawing);
tate.mouseDown(x,y);
RectButton
OvalButton
SimpleLineButton

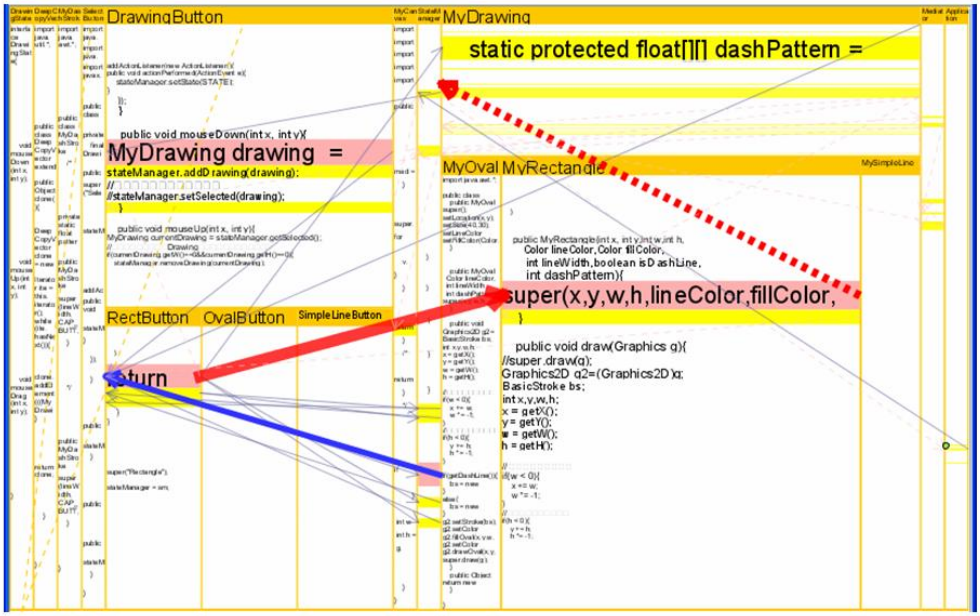
c)


DrawingButton
MyDrawing drawing = createDrawing(x,y);
RectButton
return new MyRectangle(x,y,0,0,
stateManager.getLineWidth(),stateManager.getDashLine(),
stateManager.getDashPattern());
public class RectButton extends JButton implements DrawingState{
private StateManager stateManager;
final DrawingState STATE=this;
public RectButton(StateManager sm){
super("Rectangle");
stateManager = sm;
addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e){
stateManager.setState(STATE);
}
});
}
public void mouseDown(int x, int y){
MyRectangle rect = new MyRectangle(x,y,0,0,Color.black,stateManager.getFillColor(),
stateManager.getLineWidth(),stateManager.getDashLine());
MyCanvas
StateManager
MyDrawing
Mediator
Application

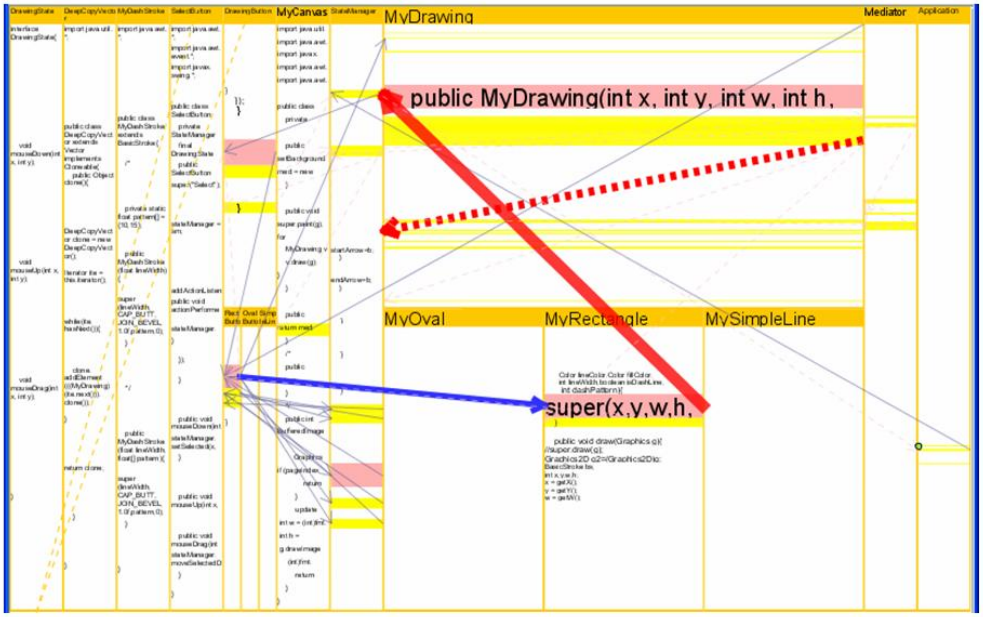
d)


DrawingButton
StateManager
public void mouseDown(int x, int y){
MyDrawing drawing =
stateManager.addDrawing(drawing);
//stateManager.setSelected(drawing);
}
public void mouseUp(int x, int y){
MyDrawing currentDrawing = stateManager.getSelected();
RectButton
OvalButton
SimpleLineButton
return
public Color getFillColor(){return fillColor;}
public Color getLineColor(){return lineColor;}

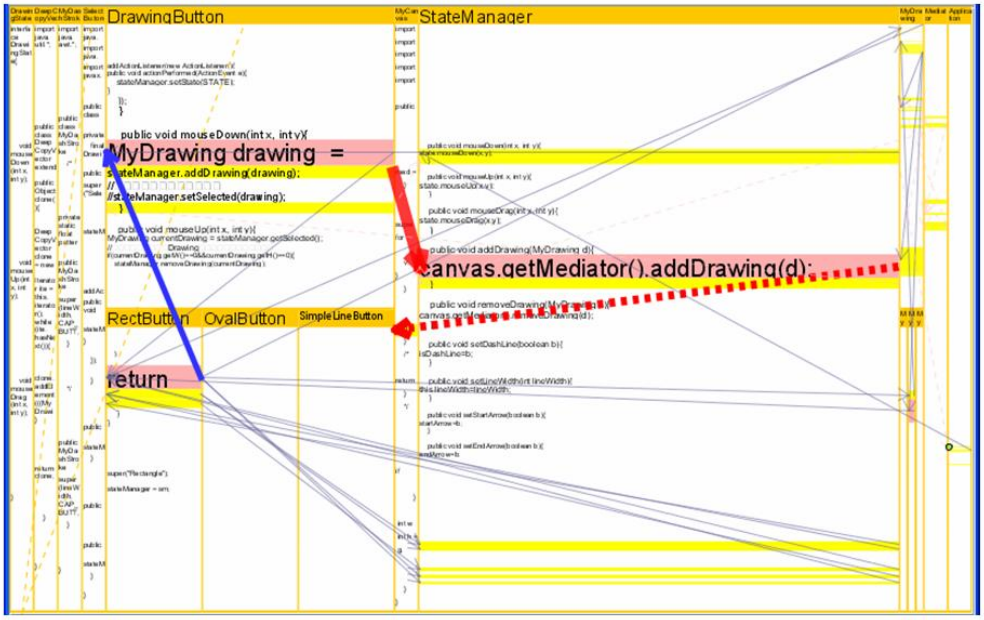
e)


DrawingButton
MyDrawing
static protected float[][] dashPattern =
public void mouseDown(int x, int y){
MyDrawing drawing =
stateManager.addDrawing(drawing);
//stateManager.setSelected(drawing);
public void mouseUp(int x, int y){
MyOval
MyRectangle
MySimpleLine
public MyRectangle(int x, int y,int w,int h,
Color lineColor, Color fillColor,
int lineWidth, boolean isDashLine,
int dashPattern){
super(x,y,w,h,lineColor,fillColor,
public void draw(Graphics g){
//super.draw(g);
Graphics2D g2=(Graphics2D)g;
BasicStroke bs;
int x,y,w,h;
x = getX();
y = getY();
w = getW();
h = getH();
RectButton
OvalButton
SimpleLineButton
return

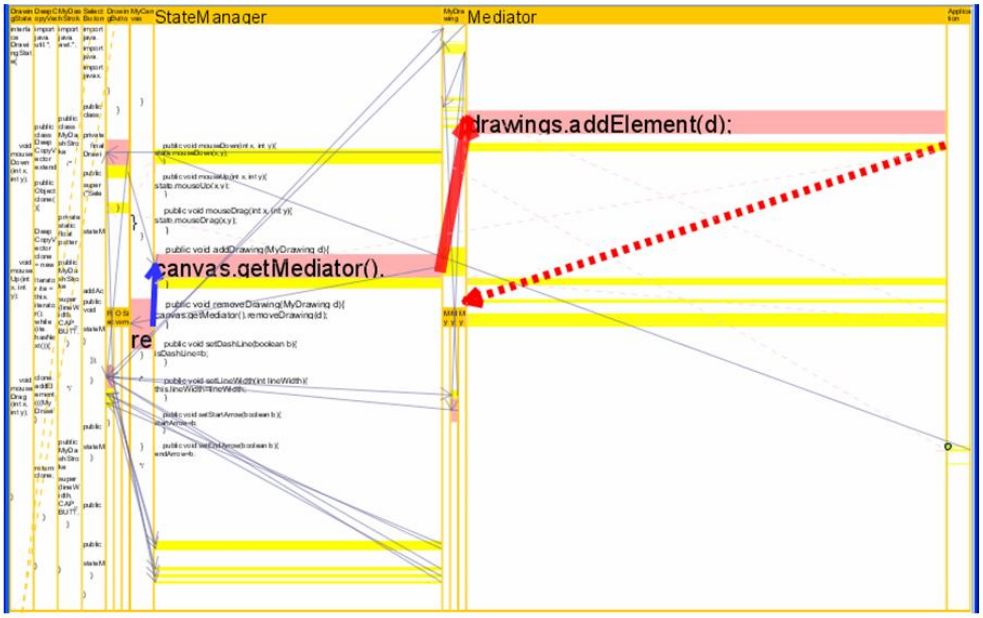
f)


MyDrawing
Mediator
public MyDrawing(int x, int y, int w, int h,
MyOval
MyRectangle
MySimpleLine
super(x,y,w,h,

g)


DrawingButton
StateManager
public void mouseDown(int x, int y){
MyDrawing drawing =
stateManager.addDrawing(drawing);
//stateManager.setSelected(drawing);
public void addDrawing(MyDrawing d){
canvas.getMediator().addDrawing(d);
RectButton
OvalButton
SimpleLineButton
return

h)


StateManager
Mediator
drawings.addElement(d);
canvas.getMediator().

i)

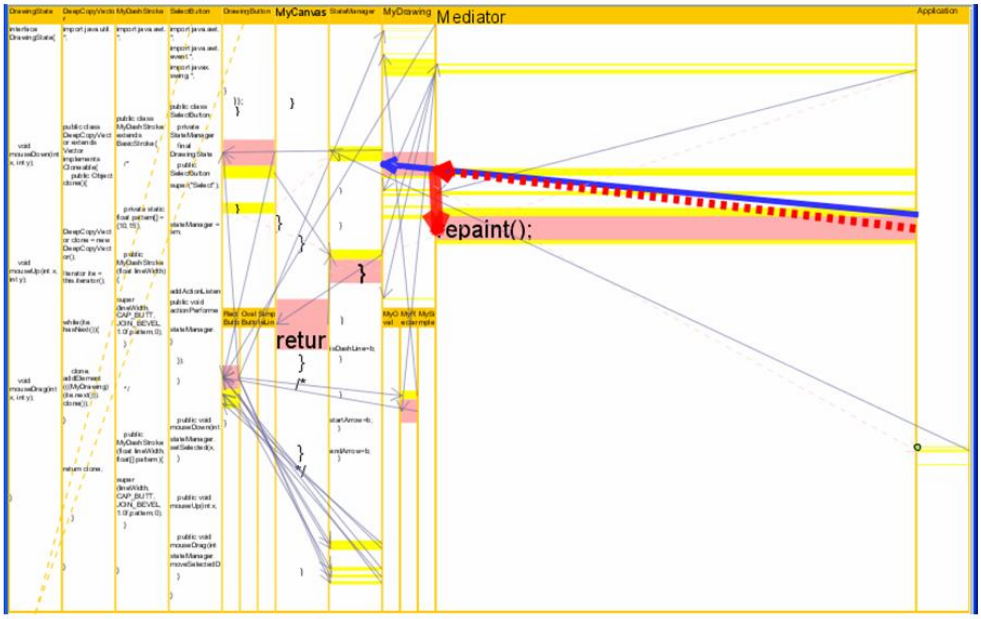


図 5.7: マウスプレス時の詳細表示

バス上への追加を行っている (g, h)。最後に、キャンバスの再描画が行われる (i)。このようにして詳細表示でエッジをたどることで、関数呼び出し階層を中心とした実行情報を取得する。ここでわかる重要な情報は RectangleButton クラスで DrawingButton クラス内で定義された関数 createDrawing() がオーバライドされているということ、そして MyRectangle クラスのインスタンス生成時に親クラスである MyDrawing クラスのコンストラクタが呼び出されているということである。これをすべての操作について行うことで、機能が使用しているクラス、使用しているメソッドと実行の流れを把握することができる。最後にこれらのクラスの実体を生成している部分を main 実行時の静的グラフから見つけ出せばよい。マウスプレスの例と同様に他の機能について作業を行うと、これらの操作に関して修正すべきクラスは MyRectangle クラス、RectButton クラス、Application クラスの高々3 つのクラスであることがわかる。厳密にはクラス名やメソッド名といった情報も理解に利用したため、ある程度憶測を含んでいる。しかし、現在のプログラミングにおいて、クラス名などの命名は大切な規則であるため、有益な情報を含んでいるはずである。そのため利用しない手はないが、時にその名前と役割が異なる場合もある。このような要因のために、それだけでは不安な開発者は矩形描画機能の他の描画機能についても同様の作業を行うことをお勧めする。そうすることにより、各描画機能での共通部分や相違点がさらに浮き彫りになる。しかしながら、今回はあくまでも矩形描画機能ついての把握のみで十分であると考え、それ以外の種類の描画機能については作業を行わない。

ここまでで、矩形描画の大まかな実装を知ることができた。もちろん、色や線の太さなどのパラメータ変化など、細かな他機能との兼ね合いもこの後知る必要がある。その際にも同様の操作をすればよい。開発者は既に機能のために使用するクラスに目星がついている状態である。つまり、概観表示を見て、それらクラスを使用していない操作については把握する必要が無い。そのため、この作業にはそれまでの作業ほど時間をとられないはずである。

このようにして要素を把握した後は再利用すべき実装部分や継承すべきクラスなどを知ることができている状態にある。後は作業で得ることのできた情報を元に矩形描画を模倣し実装を行えばよい。このようにして本システムを用いることで機能追加に必要な要素を理解することができるため、速やかに機能の実装に取り掛かることが可能となる。この作業は、1 章で述べた一般的に使用されているツールを使用した場合に比べ、より短い時間で行えると期待している。

第6章　今後の展望

現在は操作に対するプログラム全体の実装部分を提示しているが、機能追加のために本当に得たい情報は機能に対する実装部分である。すでに述べた通り、機能はそれにまつわる操作の集合であるから、各操作に対する静的グラフをすべてマージしたグラフは機能に対するプログラム全体の実装部分を表すことになる。また、類似した機能間での違いを見つけるためにはそれらの操作に対する静的グラフの差分が役に立つ。これらのことから、静的グラフの統合あるいは差分を表示する機能があるとさらに理解に効果がある。

詳細表示でのズーミング方法にもまだまだ改善の余地がある。

ソースコードのズーミングでは、各行の領域は線形関数に基づいて計算されている。そのため現状では行数が増えると現在の実行行以外の行の閲覧がしづらい。その解決策としては、ある 1 点の近傍を大きく、その点から離れるにつれて小さくなるような非線形関数を導入することが考えられる。

また現在のクラスのズーミングは簡易的な Fisheye ズーミングによって実装されている。マルチスレッドのそれぞれに対する視点属性、呼び出し順序や頻度などの意味属性などを重要度として与え、さらに厳密な Fisheye ズーミングをすることでさらに柔軟な可視化が可能となると思われる。

各時点でのオブジェクトの状態やその遷移なども、プログラムを理解する上で有用な情報の一つである。これらを表示も付加することでさらなる作業効率の向上が期待できる。

今後はこれらの課題を解決しさらに快適な可視化システムを目指していく。

第7章　結論

本稿では、既存ツールや可視化システムを利用して GUI を持つプログラムの理解する際の問題点を述べた。次に問題解決の手法として GUI の特徴を活かした可視化システムの提案と実装を行い、システムの利用例を述べた。本システムによって GUI を持つプログラムに機能追加を行う際のプログラム理解を支援することが可能になる。

今後はシステムの評価と改善を行い、理解支援のためにさらに快適な可視化システムを目指して行きたい。

謝辞

本論文の執筆にあたり、指導教員である田中二郎教授、志築文太郎講師をはじめ、三末和男助教授、高橋伸講師には多くのご助言やご指導をいただきました。心より感謝申し上げます。

また、田中研究室の皆様にも大変お世話になりました。特に WAVE チームの皆様にはチームゼミだけでなく日常的にご意見を頂きました。ここに深く感謝いたします。

最後に日頃より私を支えてくれました家族や友人たちに心より感謝いたします。

参考文献

[1] Stephen G. Eick and Joseph L. Steffen and E. Summer : Seesoft - A Tool For Visualizing Line Oriented Software Statics, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 18, No. 11, pp. 957–968 (1992).

[2] 寺田実：電子紙芝居 Electoronic Paper-Slide Show, 日本ソフトウェア科学会 WISS'98, pp. 181–186 (1998).

[3] 首藤達生, 寺田実: プログラム実行履歴情報を用いたソースコード読解支援システム, 日本ソフトウェア科学会 WISS2000, pp. 245–246 (2000).

[4] 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, 日本ソフトウエア科学会第 17 回 (2000 年度) 大会 (2000).

[5] Storey, M.-A., C. Best, and J. Michaud : SHriMP views - an environment for exploring Java programs, IEEE International Workshop on Program Comprehension '2001 (2001).

[6] Storey, M.-A., C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen : SHriMP views - an interactive environment for information visualization and navigation, Proceedings of CHI 2002. Conference Extended Abstracts on Human Factors in Computer Systems, pp. 520-521 (2002).

[7] Storey, M.-A. : ShriMP views - an interactive environment for exploring multiple hierarchical views of a Java program, ICSE 2001 Workshop on Software Visualization (2001).

[8] J. Seemann, J. Wolff von Gudenberg: Visualization of Differences between Versions of Object-Oriented Software, 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98), pp.201-204 (1998).

[9] John Stasko, John Domingue, Marc H. Brown, Blaine A. Price(editors): Software Visualization-Programming As a Multimedia Experience, MIT Press (1998).

[10] Klaus-Peter Lohr and Andre Vratislavsky: JAN-Java Animation for Program Understanding, 2003 IEEE Symposium on Human Centric Computing Languages and Environments, pp. 67–75 (2003).

[11] R.Oechsle, Th. Schmitt: Javavis - Automatic program visualization with object and sequence diagrams using the Java Debug Interface, Software Visualization - International Seminar, pp. 176-190 (2002).

[12] JDI: http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jpda/jdi/index.html/.

[13] Piccolo: http://www.cs.umd.edu/hcil/piccolo/.

[14] B. B., Grosjean, J.Meyer: Toolkit Design for Interactive Structured Graphics Bederson, IEEE Transactions on Software Engineering, pp. 535-546 (2004).

[15] UML: http://www.uml.org/.

[16] IBM Rational Software Architect: http://www-06.ibm.com/jp/software/rational/products/design/rsa/

[17] Eclipse: http://www.eclipse.org/.

[18] Furnas, G.W.: Generalized Fisheye Views, ACM SIGCHI'86 Conf. on Human Factors in Computing Systems, pp.16-23 (1986).

[19] 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).

[20] H. Koike: Fractal Views - A Fractal-Based Method for Controlling Information Display, ACM Trans. on Information Systems, Vol. 13, No. 3, pp.305-323 (1995).

[21] 楓基靖, 林晃一郎, 樋田洋明, 吉村陵二, 真野芳久: Java プログラムの種々の実行時情報の効果的取得方法,『アカデミア』数理情報編,Vol. 6, pp.31-38 (2005).