ソフトウェア理解支援のための多粒度ソフトウェアマップ

上原 伸介[†] 大須賀 俊憲[‡] 小林 隆志[‡]
金子 伸幸[‡] 山本 晋一郎[§] 阿草 清滋[‡]

[†]名古屋大学 工学部電気電子・情報工学科
[‡]名古屋大学 大学院情報科学研究科
[§]愛知県立大学 情報科学部

ソースコードレビューやデバッグにおいては，ソフトウェアの動作の大局的な流れと処理の詳細の双方を理解する必要がある．ソフトウェアの動作を理解する際には，関数の呼び出し関係を辿りながらその処理内容を理解する方法が一般的であるが，呼び出し関係が複雑である場合には単純に呼び出し関係を辿るだけでは大局的な流れと処理の詳細を理解することは困難である．本研究では，処理の詳細と大局的な流れの双方に対してその理解を支援する手法として，ソースコードを多粒度で表現し，対話的なナビゲーションによるソフトウェア理解支援を可能とするソフトウェアナビゲーションマップを生成する手法を提案する．さらに提案手法を実装したツールを用いて手法の有効性を議論する．

Multi Grained Software Map for Software Comprehension Support

NOBUYUKI UEHARA[†], TOSHINORI OSUKA[‡], TAKASHI KOBAYASI[‡],
NOBUYUKI KANEKO[‡], SHINICHIRO YAMAMOTO[§] and KIYOSHI AGUSA[‡]

[†] School of Engineering, Nagoya University
[‡] Graduate School of Information Science, Nagoya University
[§] Faculty of Information Science and Technology, Aichi Prefectural University

In source code review and/or debug phase, developers must understand both the big picture and details of the software behavior. To comprehend behavior of software, developers roughly trace function call for getting a big picture view, and then read each source codes of caller and callee functions back and forth. However it is difficult to understand both details and the whole of target software at the same time.In this paper, we propose a method for supporting software comprehension that forms "software navigation map". It allows developers to view details and the whole at the same time by changing granularity depending on the developers' focus. We also introduce a tool based on our method and discuss the usability of our method.

1. はじめに

ソフトウェアの大規模化や複雑化は，ソフトウェアの理解を困難にする．その対策として，単純にソースコードを提示せず，様々な情報を追加，整理してソフトウェアの理解を支援する手法が提案されてきた [1]．大規模化が進み，さらに他人のつくったソフトウェアや異なる環境，時代においてつくられたソフトウェアも理解しなければならない機会が増えた現在，ソフトウェア理解支援は非常に重要である．

ソフトウェアは多くの構成要素からなり，その基本単位は関数やメソッド等の手続きである．ソースコードレビューやデバッグ作業の際には，処理の大まかな流れを辿ることで，大局的な流れを把握し，その後，処理の流れに沿って詳細に手続きの呼び出し関係を辿り，ある手続きからその手続きが呼び出している別の手続きに注意を移すという作業を繰り返す機会が多い．呼び出し関係が複雑である場合，この作業は大きな負担となる．また，オブジェクト指向言語の場合は，大規模化だけではなく，抽象化やデザインパターンの利用により，処理の委譲や，多態による動的束縛などが多用されるため，追跡の負担が増すことがある．

これまでにメソッドの呼び出し関係の追跡する支援手法として，呼び出し関係のグラフであるコールグラフを生成し支援する手法 [2] や，メソッド呼び出しをその定義と置き換えて表示することで追跡を容易にす

る手法 [3] 等が提案されているが，大局的な流れと，処理の詳細の双方を効率的に理解するための支援には至っていない．

本研究の目的は，ソースコードを解析することで得られる情報を元に，ソフトウェアの構成要素を効果的に可視化することで，呼び出し関係を追跡しながら，ソフトウェアの大局的な流れと処理の詳細の双方を理解するための支援を行うことにある．

既存のソフトウェア理解支援手法を，その対象とする情報の粒度によって細粒度なものと粗粒度なものに分類することが出来る．細粒度な手法では，処理の内容や呼び出しの際の文脈を詳しく知ることができるが，手続き間をまたがる処理全体の流れが見えにくい．逆に粗粒度な手法では，複数の手続きがどうつながり処理全体の流れとなっているのかを知ることができるが，個々の手続きの詳しい内容はわからないという問題がある．

本研究では，ソフトウェアの動作を理解する際，開発者の注意が次にどこに移るかを考慮することで，ソフトウェア各構成要素の適切な表示粒度を推測し，その粒度でソースコードを表示する手法を提案する．

提案手法では，オブジェクト指向言語を対象とし，ソースコードをプログラム依存グラフ (PDG: Program Dependence Graph)，クラス階層，メソッド呼び出し関係などの構成要素間の詳細な関係を用いた細粒度のグラフ構造として扱う．グラフ構造はソフトウェアナビゲーションマップとして，可視化され，開発者がソースコードを辿る際のグラフ構造上での位置を考慮し，候補の提示や，各構成要素が適切な粒度となるよう，多粒度での可視化を行う．

以下ではまず，既存のソフトウェア理解支援手法の問題点を考察し，3. でソフトウェアの各構成要素を異なる適切な粒度で表示するソフトウェアナビゲーションマップを提案する．さらに，4. で提案手法によるソフトウェア理解支援ツールの実装について報告し，手法の評価を行なう．

2. 関連研究

ソフトウェア理解のための支援手法は数多く提案されている．

処理の流れを理解するための手法としては，メソッド呼び出しの関係に着目した方法が一般的である．コールグラフは手続き間の呼び出し関係をそのまま図にしたものであり，複雑な呼び出し関係を俯瞰できる．さらに，シーケンス図を用いると，コールグラフと同じ名前や呼び出し関係といった情報に加え，呼び出しの順序関係や関わったオブジェクトの生存期間等から，重要な呼び出しの見当を付けることができる．このため，ソースコードからシーケンス図を自動生成する研究 [4] や，そのようにして機械的に生成され，巨大になってしまうシーケンス図を，見やすくする研究 [5,6] がある．

これらの粗粒度な手法では，処理の概要を表現できるが，各手続きの詳細が見えなくなるため，1) 重要な処理を見過ごしてしまう，2) 最初に辿っていたはずの文脈とは異なる文脈での処理の流れにそれて行ってしまうといったことが起こり得る．例えばコールグラフやシーケンス図のみを用いて呼び出し関係を辿る場合，気付かない内に，今興味の無いエラー処理やエスケープ処理に迷い込んでいる可能性がある．さらに，その事自体に気付くことにも，どこで経路を誤ったのかを知ることにも，結局図を見るだけではなくソースコードを読む必要がある可能性がある．

一方，細粒度な情報を扱うための支援手法として，プログラムスライシングを用いる方法が古くより提案されている [7]．プログラムスライシングの技法を用いることで，興味対象に関係の無いソースコードを表示せず，効率的に処理を理解することが可能となる．しかしながら，細粒度な手法では，処理の詳細が表示できる一方で，各手続きが分断されて見えるため，1) どういった流れで現在の手続きに達したのか，2) これからどういった流れで進む，または既知の部分に戻るのか，が理解しづらいという問題がある．この問題に対して，メソッド呼び出しをそのメソッドの定義部分と置き換えて表示することで，前後の状況がわからなくなることを防ぐ手法 [3] も提案されているが，複雑な呼び出しには対応できないという問題がある．

また，近年では，ソースコードを閲覧している利用者に対してその閲覧履歴を保存し，次に見るべき場所の提示（ナビゲーション）を行う研究 [8,9] がなされている．これらの手法では利用者のファイル閲覧履歴からファイル間の関連付けを自動的に構築し，利用者が表示しているファイルに対する関連度に応じて，次に見るべきファイルの候補に加えたり，表示しないようにすることで，理解を支援する．これらの支援では，過去に多くの人が辿った呼び出し関係であっても，今自分が興味を持っている文脈においては重要ではない可能性があり，さらにソースコード間の関係を考慮していないため，なぜそのソースコードが提示されたかを理解することが難しい可能性がある．

3. ソフトウェアナビゲーションマップ

3.1 アプローチ

既存の理解支援手法の問題点は，それらが単一の粒度でのみソフトウェアを扱うために生じる．そこで本手法では，ソフトウェアを多粒度で表現し，PDG を用いたナビゲーションにより，現在の利用者の状況に応じて対話的に粒度を変化させる手法として，「ソフトウェアナビゲーションマップ」を提案する．

3.2 多粒度なソフトウェア表示

多粒度なソフトウェア表示とは，ソフトウェアの各構成要素によって，ソースコードレベルの細粒度な表示と，名前や他の部分との関係のみの粗粒度な表示とが混在する表示方法である．利用者にとって重要である構成要素を細粒度に表示し，それ以外の構成要素を粗粒度に表示することで，細粒度な手法と粗粒度な手法の長所の両立を狙う．

図 1 は，複雑な呼び出し関係の追跡を多粒度な視点で行う場合のイメージである．利用者が着目すべき重要な手続きを細粒度に表示し，それ以外の手続きを素粒度に表示する．細粒度と粗粒度な表示を混在させることにより，呼び出し関係の前後の文脈を忘れずに，手続きの内容を読むことができる．多粒度な表示は，抽象化や委譲によって，実質的な処理の記述に達するまでに多くの手続きを辿らなければならない場合により効果的である．

3.3 ナビゲーション

呼び出し関係を追いかける場合，利用者にとって重要な要素とは，前節で触れたナビゲーションの研究と同じく「次見るべき場所」であると考えられる．ただしこれらの研究で行われているナビゲーションでは，利用者の表示履歴を収集する必要がある．表示履歴をファイル単位より細かな粒度で自動収集することは困難であるため，現在の利用者はこの if 節に注目している，というような細粒度な対話性を目指す本手法の目的には向かない．

そこで本手法では PDG を用いたナビゲーションを行う．PDG と利用者の状況から，利用者が次に見るべき場所を推測し，その場所を細粒度に，それ以外の場所を粗粒度に表示する．PDG を用いることで，利用者の履歴を収集するよりも細粒度なナビゲーションが可能になる．

3.4 ソフトウェアナビゲーションマップの定義

ソフトウェアナビゲーションマップは，ソフトウェアの各構成要素をノードとし，そのつながりをエッジとしたグラフである．

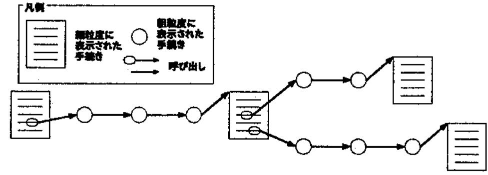

図 1 多粒度な視点で呼び出し関係を辿るイメージ

3.4.1 ソフトウェアの構成要素を表すノード

ソフトウェアナビゲーションマップは，ノードを様々な粒度で表示できなければならない．例えば Java 言語のソフトウェアの場合，具体的には粗い順に以下のような粒度がある．

- ドメイン
- パッケージ
- ファイル
- 型（クラス，インターフェース）
- 手続き（メソッド，コンストラクタ，static 節）
- 手続きの内容（文，式等）

ドメインとは本研究独自の単位で，複数のパッケージからなる最も粗い粒度である．そのパッケージの提供組織 (nagoya-u.ac.jp 等) をパッケージ名から推測し利用する．

3.4.2 呼び出し関係を表すエッジ

呼び出し関係を表すエッジは以下の 4 つである．

- 呼び出し（式から手続きへの関係）
- オーバーライド（メソッドからメソッドへの関係）
- 継承（クラスからクラスへのエッジ）
- 実装（クラスからインターフェースへのエッジ）

メソッドがオーバーライドされている場合，実行時に実際に行われる処理はオーバーライドしているメソッドである．そのため，メソッドのオーバーライドを辿る必要がある場合があり，メソッドの呼び出しの関係の他にメソッドのオーバーライドもエッジとする．継承，実装はオーバーライドを辿る際の補助になると考えエッジに加えた．

3.4.3 ナビゲーションに必要な情報

ソフトウェアの各構成要素を表すノードとそのつながりを表すエッジの他に，ナビゲーションに必要な情報として「手続きの重要度」と「エッジの重要度」を保持する．これらの情報からナビゲーションを行う手法については 3.6 節で述べる．

3.5 生成手法

ソフトウェアナビゲーションマップを生成するために，ソフトウェアの各部分の階層構造，呼び出し，オー

バーライドを解析する他に，ナビゲーションに必要な情報として手続きとエッジの重要度をPDGから計算する．

3.5.1 プログラム依存グラフ

ナビゲーションに必要な情報の計算に，本手法ではPDGを用いる．これは以下のような理由による．

- ソフトウェアの理解を目的として広く使用されていること [10]
- ソースコードのみから生成できること
- 細粒度であること

PDGはソースコード中の依存性をグラフとして表したものであり，ソースコードの一部分（行，文，式等）がノード，ノードの間の依存関係がエッジとして表される．依存関係には制御依存とデータ依存がある．例えばif文の条件式からはif節とelse節へ向かう制御依存があり，変数への代入文からはその変数を参照する式へ向かうデータ依存がある．

3.5.2 手続きの重要度

手続きの重要度は，以下の3つの考えに基づいて計算する．

(1) 複雑な手続きは重要である
(2) フィールドにアクセスする手続きは重要である
(3) 利用者が注目している手続きと同じフィールドにアクセスする手続きは重要である

(1)は，複雑な手続きにはそれだけ多くの処理が記述されており，利用者が探している処理が含まれている可能性が高いためである．また，getter，setterに代表されるように，単純過ぎる手続きは名前から処理全てを想像できる場合が多い．更に，単純過ぎる手続きは抽象メソッド，委譲メソッド等の，ソースコードを表示したい機会が少ない手続きである可能性が高い．抽象メソッドであれば実装，委譲メソッドであれば委譲先が本当に必要な情報である．手続きの複雑さは，PDGのエッジ数と相関関係にあると考えられる．

(2)は，フィールドにアクセスする手続きは，オブジェクトの状態を変え，以降のオブジェクトの利用に影響を与える可能性があるからである．

(3)は，(2)と同じ理由に加え，同じフィールドにアクセスする手続き間には，依存性がある可能性がより高いからである．ただしこの情報は対話的に変化するため，必要に応じてその都度PDGから計算する．

このことから，以下のように手続きの重要度を定義する．ただし，$\alpha_1 \sim \alpha_3$は正の比例定数とする．

手続きの重要度 = PDGのエッジ数 $\times \alpha_1$
+アクセスするフィールドの数 $\times \alpha_2$
(以下は注目している手続きがわかっている場合)
+共にアクセスするフィールドの数 $\times \alpha_3$

3.5.3 エッジの重要度

エッジには呼び出しとオーバーライドの2種類が必要であるが，この2つでは重要度の計算方法が異なる．

呼び出しの重要度は，以下の4つの考えに基づいて計算する．

(1) 引数，フィールドに依存している呼び出しは重要である
(2) 戻り値，フィールドが依存している呼び出しは重要である
(3) 利用者が注目している呼び出しがわかっている場合，PDG上でその呼び出しに近い呼び出しも重要である
(4) 利用者が注目している呼び出しがわかっている場合，その呼び出しと同じ発生条件である呼び出しも重要である

(1)は，引数に依存している呼び出しは呼び出し元に，フィールドに依存している呼び出しはオブジェクトの状態に依存するため，いずれも利用者がこれまでにたどってきた呼び出しに依存している可能性が高いからである．

(2)は，戻り値が依存している呼び出しは呼び出し元が，フィールドが依存している呼び出しはオブジェクトの状態が依存するため，いずれも利用者がこれまでにたどってきた呼び出しが依存している可能性が高いからである．

(3)は，PDG上で近い呼び出しは，より直接的に依存している呼び出しだからである．

(4)は，利用者は今，その発生条件での処理の流れを追いかけている可能性が高いからである．

ただし(3)，(4)の情報は対話的に変化するため，必要に応じてその都度PDGから計算する．

このことから，以下のように呼び出しの重要度を定義する．ただし，$\alpha_1 \sim \alpha_6$は正の比例定数とする．

呼び出しの重要度 =
呼び出しが依存している引数の数 $\times \alpha_1$
+呼び出しが依存しているフィールドの数 $\times \alpha_2$
+呼び出しに依存している戻り値の数 (0 or 1) $\times \alpha_3$
+呼び出しに依存しているフィールドの数 $\times \alpha_4$
(以下は注目している呼び出しがわかっている場合)
+PDG上の距離 $\times (-\alpha_5)$
+共通している発生条件の数 $\times \alpha_6$

オーバーライドの重要度は，抽象メソッドのオーバーライドは重要である，という考えに基づいて計算する．抽象メソッドの場合，通常のメソッドとは異なり実行時には必ずその抽象メソッドをオーバーライド

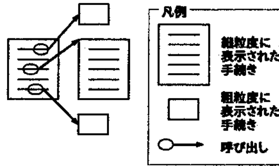

図 2　多粒度なソフトウェア表示

する別のメソッドが実行されるからである．

このことから，オーバーライドであるエッジの重要度は，オーバーライドされるメソッドが抽象メソッドである場合とそれ以外の場合で異なる定数とする．

3.6　ソフトウェアナビゲーションマップを用いた理解支援

手続きの重要度とエッジの重要度から実際にナビゲーションを行い，ソフトウェアを多粒度に表示する方法について述べる．

3.6.1　多粒度なソフトウェア表示

利用全体の流れは以下のとおりである．

(1)　利用者が，1 つの手続きを指定する．

(2)　利用者の指定から，次に利用者が指定する手続きを推測する．

(3)　利用者の指定した手続きと次に利用者が指定すると推測された手続きを細粒度に，それ以外の部分を粗粒度に表示する（図 2）．

3.6.2　ナビゲーションの実現

以下の流れで利用者が指定する手続きを推測する．

(1)　利用者がこれまで指定していた手続きと新しく指定した手続きの 2 つに注目する．

(2)　注目している手続きの中から，その重要度に応じて，いくつかのエッジを選ぶ．

(3)　エッジそれぞれについて，その先の手続きに注目し直す．

(4)　2〜3 を繰り返し，重要な手続きに到達したらその手続きを次に利用者が指定する手続きと推測する．何回までエッジをたどるかは，それまでにたどってきたエッジの重要度に応じる．

利用者が現在注目している手続きにおいて重要である呼び出しについてその先を調べていき，重要な手続きに到達したところでそこを次の注目箇所の候補にするという作業を半自動化できる．

4.　ソフトウェアナビゲーションマップ生成ツールの実装

提案手法に基づくソフトウェアナビゲーションマップを実際に生成し，それを表示してソフトウェア理解支援を行うツールを実装した．

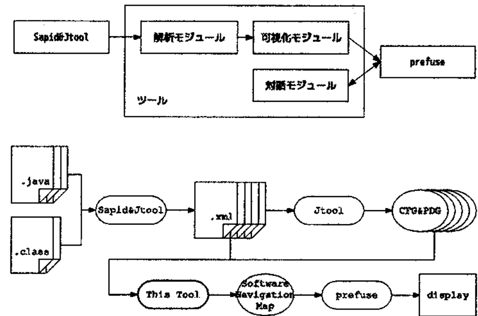

図 3　ツールの構成と入力と出力

4.1　ツールの構成

本ツールは Java 言語 1.4 で記述されたソフトウェアを対象とし，解析モジュール，可視化モジュール，対話モジュールからなる（図 3）．

解析モジュールでは，Java ソースコードの静的解析に Sapid [11] と Sapid/XML Tool Platform “Jtool” [12] を利用している．Sapid は Java ソースコードの静的で細粒度な解析及びマークアップを行い，JX-model という形式の XML 文書を出力する．Jtool は JX-model をさらに依存解析し，同様に XML 文書として出力する．

また，可視化モジュールでは，Visualization toolkit “prefuse” [13] を利用している．prefuse は表，ツリー，グラフ等のグラフィカルな表示，それに対するユーザの操作によるインタラクティビティ等をサポートする．本ツールはソフトウェアナビゲーションマップを prefuse で表示可能なグラフとして生成し，prefuse を利用して表示する．内容粒度で表現されているノードはソースコードとして表示される．ドメインから手続きまでの粒度で表現されているノードは，アイコン，または子孫要素を囲む枠として表示される．いずれの場合にも，パッケージ名，クラス名等の名前が併せて表示される．また，手続き粒度のノードには，名前に加えて重要度も表示される．アイコンはそれぞれ，地球の絵がドメイン，箱の絵がパッケージ，紙の絵がファイル，C と書かれた丸の絵が型，歯車の絵が手続きを表す．

対話モジュールでは，prefuse の機能を利用したパンやノードのドラッグのような基本的な操作の他に，呼び出し式をクリックすることでナビゲーションが行える．ナビゲーションが行われるとその結果に従い，各ノードの粒度，配置，可視／不可視が決定される．ま

た，ノードをクリックすることで明示的に粒度を変更することもできる．

4.2 ツールが行うナビゲーション

本ツールは，最初に手続きを1つ指定して起動する．指定された手続きはフォーカスを持ち，内容粒度で表示される．また，内容の内，呼び出し式は強調表示される．強調表示された呼び出し式をクリックすることで，新しい手続きが指定される．

新しい手続きが指定されると，その手続きの表示も内容粒度になり，また新しい手続きを指定できる．また，その手続き内の呼び出し全てについて次指定すべきか否かの推測が行われ，次指定すべきと判断されたものは利用者の次の指定を待たずこの段階で既に内容粒度で表示される．

これまでに指定された手続きと今指定している手続き以外は，メソッドを表すアイコンやクラスを表すアイコンといった形で，できる限り粗粒度に表示される．

推測の際には，指定された手続きからの呼び出しを1回だけたどり，たどった先の手続きの重要度が閾値を超え，かつ最大であるものを選ぶ．

4.3 実装上の制限

現時点では，本ツールには以下のような制限がある．

- エッジの重要度の計算を実装しておらず，全てのエッジが同じ重要度として扱われる．
- フィールド，static節，デフォルトコンストラクタは取得できない．
- ソースファイルではなくクラスファイルから取得した部分には，ソースコードが存在しないため，内容粒度の情報が存在しない．
- 解析ライブラリの制限により，catch節の内容は解析しない．

4.4 ツールの評価

前節で実装したツールによって，実際にソースコードの可視化を行い，提案手法の有用性を議論する．

4.4.1 評価方法

サンプルとしてあるソフトウェアのソースコードを本ツールによって可視化し，ある処理のシーケンス図を作成した．また，Eclipseの標準的な機能のみを使って同じシーケンス図を作成し，作業にかかった時間を比較した．

今回評価に用いたソフトウェアは，10個のソースファイルからなるCommandパターンのサンプルプログラムである．シーケンス図を作成した処理は，ユーザの入力によって，まず直線を描画するコマンドが実行された後，次にアンドゥコマンドが実行され描画コマンドが取り消されるという処理である（図4）．この

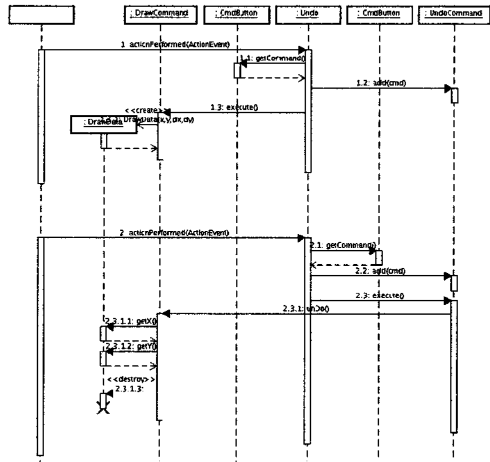

図4 作成したシーケンス図

処理の記述には，汎化による多態が利用されている．

ユーザの入力は，描画コマンドであってもアンドゥコマンドであっても，`Undo#actionPerformed(ActionEvent)`の呼び出しによってこのプログラムに通知される．よって今回は，`Undo#actionPerformed(ActionEvent)`からの呼び出し関係を辿る．

4.4.2 結果

シーケンス図の作成にかかった時間は，本ツールを利用した場合が8分，Eclipseを利用した場合が10分であった．

本ツールの実行時の様子を図5,6を示す．なお，この実行例では，フォーカスメソッドと直接つながっていない要素や標準ライブラリの要素を表示しないオプション機能をオンにしている場合がある．

図5は，本ツールによって`Undo#actionPerformed(ActionEvent)`が呼び出されてから`Command#execute()`が呼び出されるまでの処理の流れが表現されている画面である．手続き名の右に表示されている数字が，その手続きの重要度である．図5では，左上に内容粒度で表示されている`Undo#actionPerformed(ActionEvent)`は標準ライブラリを除いて3つのメソッドを呼び出しており，その内右下の`UndoCommand#add(Command)`の呼び出しがもっとも複雑であることから，`UndoCommand#add(Command)`が内容粒度で表示されている．

図6は，本ツールによって`execute()`から先の処理の流れが表現されている画面である．図6から，上段中央やや右の`Command`インター

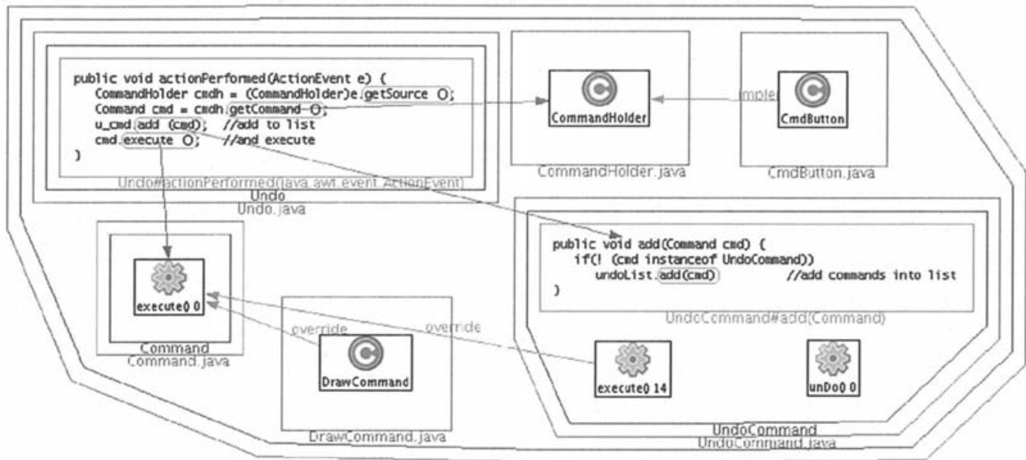

図 5　actionPerformed から execute までの処理

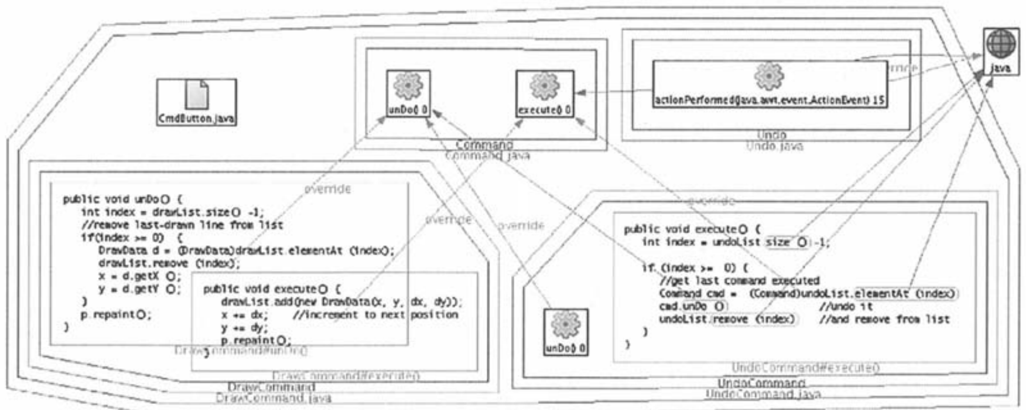

図 6　execute から先の処理

フェースの execute() は 2 つのクラスで実装され，その内右下に内容粒度で表示されている UndoCommand クラスでの実装は，上段中央やや左の Command#unDo() を呼び出していることがわかる．また，その Command#unDo() は，下段中央やや右の UndoCommand#unDo() の重要度が 0 であることから，左下に内容粒度で表示されている DrawCommand での実装が現在実質的に唯一の実装であることがわかる．

4.4.3 考　　察

本ツールを利用することで，シーケンス図を短時間で作成できた．これは，実装したツールにはいくつかの制約があったものの、メソッド間，特にファイル間をまたぐ追跡のコストの削減が有効であったものと考える．

一般的なエディタでこのサンプルのような処理をたどる場合には，多くのファイルを開き，その間をジャンプする必要がある．エディタで開かれていないファイルや隠されたファイルは，本ツールでは粗粒度に表示されている要素に当たる．本ツールは，そのような要素についても，呼び出しやオーバーライドによって関係していればそれらの関係を表示して，他の情報を隠す事無く要素間のつながりを表示している．これによって，要素間をまたぐ追跡のコストを小さくして

いる．

このことから，提案手法は呼び出し関係の追跡におけるソフトウェア理解支援手法として有用であると言える．

5. おわりに

5.1 まとめ

本稿では，多粒度なソースコード表示とPDGを用いたナビゲーションによるソフトウェア理解支援手法を提案し，提案手法を実現するためにソフトウェアナビゲーションマップを定義した．また，実際に提案手法によるソフトウェア理解支援を行うツールを実装した．実装したツールにより実際にソフトウェアの可視化を行い，提案手法の有用性を議論した．

5.2 今後の課題

今後の課題として，まず，フィールドの付加が必要である．本研究で提案したモデルでは，フィールドをソフトウェアナビゲーションマップのノードに含めていないが，フィールドに関する情報は有用である．実際にメソッドを辿る際には，呼び出し関係の他にフィールドを介したメソッド間のつながりを辿ることも多いからである．ソフトウェアナビゲーションマップのノードとしてフィールドを追加し，エッジとしてフィールドとメソッドのつながりを追加することで，フィールドを介したつながりもソフトウェアナビゲーションマップ上で辿ることができるようになる．

また，各構成要素に関する新しい情報の付加が必要である．以下に挙げる情報は，本研究で実装したツールでは取得していないが，重要度の計算においても，また利用者にとっても重要な情報となり得る．

- コンストラクタ，メソッド，抽象メソッドの区別
- アクセス修飾子
- 頻繁に呼び出されているかどうか
- javadocコメント，アノテーション

表示粒度の調整方法に関してもさらなる検討が必要である．本研究で実装したツールでは，次に見るべきであると判断したメソッドは最も細粒度に表示され，それ以外の要素はできる限り粗粒度に表示される．このどちらかではなく，重要度に応じて粒度をより細かくに調整することで，それぞれのノードをより適切な粒度で表示できるようになると考える

謝辞

本研究を進めるにあたり熱心に議論して頂いた阿草研究室の皆様に感謝致します．本研究の一部は，文部科学省リーディングプロジェクト基盤ソフトウェアの統合開発e-Society高信頼WebWareの生成技術および文部科学省科学技術研究費 基盤研究(B)課題番号17300006，同若手研究(B)課題番号19700023の助成による．

参考文献

1) von Mayrhauser, A. and Vans, A. M.: Program Comprehension During Software Maintenance and Evolution, *IEEE Computer*, Vol.28, No.9, pp.44–55 (1995).
2) Grove, D., DeFouw, G., Dean, J. and Chambers, C.: Call graph construction in object-oriented languages, *Proc. OOPSLA'97*, pp.108–124 (1997).
3) Desmond, M., Storey, M.-A. and Exton, C.: Fluid Source Code Views for Just In-Time Comprehension, *Proc. SPLAT2006* (2006).
4) Rountev, A., Volgin, O. and Reddoch, M.: Static control-flow analysis for reverse engineering of UML sequence diagrams, *Proc. PASTE '05*, pp.96–102 (2005).
5) Sharp, R. and Rountev, A.: Interactive Exploration of UML Sequence Diagrams, *ACM Trans. Softw. Eng. Methodol.*, Vol.16, No.2, pp.1–6 (2005).
6) 小林隆志，堅田淳也，鹿内将志，佐伯元司：プログラムスライシングを用いたJava実行系列からの部分シーケンス図生成手法，FOSE2004論文集，pp.17–28 (2004).
7) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence graphs, *Proc. PLDI'88*, pp.35–46 (1988).
8) Singer, J., Elves, R. and Storey, M.-A.: NavTracks: Supporting Navigation in Software Maintenance, *Proc. ICSM'05*, pp.325–334 (2005).
9) DeLine, R., Czerwinski, M. and Robertson, G.: Easing Program Comprehension by Sharing Navigation Data, *Proc. VLHCC'05*, pp.241–248 (2005).
10) Horwitz, S. and Reps, T.: The use of program dependence graphs in software engineering, *Proc. ICSE'92*, pp.392–411 (1992).
11) 福安直樹，山本晋一郎，阿草清滋：細粒度ソフトウェア・リポジトリに基づいたCASEツール・プラットフォームSapid，情報処理学会論文誌，Vol.39, No.6, pp.1990–1998 (1998).
12) Maruyama, K. and Yamamoto, S.: A CASE Tool Platform Using an XML Representation of Java Source Code, *Proc. SCAM2004*, pp.158–167 (2004).
13) Heer, J., Card, S. K. and Landay, J. A.: prefuse: a toolkit for interactive information visualization, *Proc. CHI'05*, pp.421–430 (2005).